

TexisCore: Text Information Relational Database Server

Version 7.9.1 User's Guide

Thunderstone Software
Expansion Programs International, Inc.

May 25, 2020

Contents

| | | |
|----------|--|-----------|
| I | Using Taxis | 3 |
| 1 | Preface | 5 |
| 1.1 | Acknowledgment | 5 |
| 1.2 | Documentation Caveat | 5 |
| 1.3 | Documentation Structure | 6 |
| 2 | Introduction | 7 |
| 2.1 | Taxis: Thunderstone's Text Information Server | 7 |
| 2.1.1 | Features Unique to Taxis | 8 |
| 2.1.2 | Specifications | 9 |
| 2.2 | Taxis as a Relational Database Management System | 9 |
| 2.3 | Relational Database Background | 9 |
| 2.4 | Support of SQL | 12 |
| 2.5 | Case Example: Acme Industrial Online Corporate Library | 13 |
| 3 | Table Definition | 15 |
| 3.1 | Creating the Resume Table | 15 |
| 3.2 | Inserting Data into the Resume Table | 17 |
| 3.3 | Defining a Table | 18 |
| 3.4 | Removing a Table | 22 |
| 4 | A First Look at Queries | 23 |
| 4.1 | Retrieving From the Entire Table | 24 |
| 4.1.1 | Retrieving All the Columns | 25 |

| | | |
|----------|---|-----------|
| 4.2 | Retrieving a Subset of Rows: Simple Conditions | 25 |
| 4.3 | Retrieving a Subset of Rows: Compound Conditions | 27 |
| 4.3.1 | Retrieval Using the AND Operator | 28 |
| 4.3.2 | Retrieval Using the OR Operator | 29 |
| 4.3.3 | Retrieval Using Both AND and OR Operators | 30 |
| 4.3.4 | Retrieval Using Parentheses | 30 |
| 4.3.5 | Logical Operator NOT | 31 |
| 4.4 | Additional Comparison Operators | 32 |
| 4.4.1 | Range and Geographical Searches Using BETWEEN | 32 |
| 4.4.2 | Set-like Searches Using IN, SUBSET and INTERSECT | 35 |
| 4.4.3 | Search Condition Using LIKE | 40 |
| 4.4.4 | Relevance Ranking Using LIKER and LIKEP | 42 |
| 4.4.5 | Query searching using LIKEIN | 44 |
| 4.4.6 | Search Condition Using MATCHES | 44 |
| 4.5 | Sorting Your Results | 44 |
| 5 | Queries Involving Calculated Values | 47 |
| 5.1 | Arithmetic Calculations | 47 |
| 5.2 | Manipulating Information By Date | 49 |
| 5.3 | Summarizing Values: GROUP BY Clause and Aggregate Functions | 52 |
| 5.4 | Groups With Conditions: HAVING Clause | 55 |
| 5.5 | Server functions | 56 |
| 5.5.1 | File functions | 57 |
| 5.5.2 | String Functions | 60 |
| 5.5.3 | Math functions | 66 |
| 5.5.4 | Date functions | 66 |
| 5.5.5 | Bit manipulation functions | 67 |
| 5.5.6 | Internet/IP address functions | 68 |
| 5.5.7 | Geographical coordinate functions | 71 |
| 5.5.8 | JSON functions | 79 |
| 5.5.9 | Other Functions | 83 |

| | | |
|----------|---|------------|
| 6 | Advanced Queries | 97 |
| 6.1 | Retrieving Data From Multiple Tables | 97 |
| 6.2 | Nesting Queries | 103 |
| 6.3 | Forming Complex Queries | 107 |
| 6.3.1 | Joining a Table to Itself | 107 |
| 6.3.2 | Correlated Subqueries | 108 |
| 6.3.3 | Subquery Using EXISTS | 110 |
| 6.4 | Virtual Fields | 113 |
| 6.5 | Column Aliasing | 113 |
| 7 | Intelligent Text Search Queries | 115 |
| 7.1 | Types of Text Query | 115 |
| 7.2 | Definition of Metamorph Terms | 116 |
| 7.3 | Adjusting Linguistic Controls | 117 |
| 7.4 | Constructing a Metamorph Query | 117 |
| 7.4.1 | Keyword Search | 117 |
| 7.4.2 | Refining a Query | 118 |
| 7.4.3 | Adjusting Proximity Range by Specifying Delimiters | 119 |
| 7.4.4 | Using Set Logic to Weight Search Items | 122 |
| 7.5 | Other Metamorph Features | 130 |
| 8 | Indexing for Increased Performance | 131 |
| 8.1 | When and How to Index | 131 |
| 8.2 | Creating An Index | 132 |
| 8.3 | Creating a Unique Index | 133 |
| 8.4 | Creating a Metamorph Index | 135 |
| 8.4.1 | Metamorph Index Types: Inverted vs. Compact vs. Counter | 136 |
| 8.4.2 | Metamorph Index Capabilities and Limitations | 137 |
| 8.5 | Using LIKE3 for Index Only Search (No Post-Search) | 137 |
| 8.6 | Creating an Inverted Index | 138 |
| 8.7 | Index Options | 138 |

| | | |
|-----------|--|------------|
| 8.8 | Dropping an Index | 140 |
| 9 | Keeping the Database Current | 141 |
| 9.1 | Adding New Records | 141 |
| 9.1.1 | Inserting One Row at a Time | 141 |
| 9.1.2 | Inserting Text | 143 |
| 9.1.3 | Inserting Multiple Rows at a Time | 144 |
| 9.2 | Updating Records | 145 |
| 9.3 | Making a Taxis Owned File | 146 |
| 9.4 | Deleting Records | 147 |
| 10 | Security | 151 |
| 10.1 | Creating Users and Logging In | 151 |
| 10.2 | Granting Privileges | 152 |
| 10.3 | Removing Privileges | 154 |
| 11 | Administration of the Database | 157 |
| 11.1 | System Catalog | 157 |
| 11.2 | Optimization | 161 |
| 11.2.1 | Table Compaction | 161 |
| 11.2.2 | Index Maintenance | 162 |
| 11.3 | Reserved Words | 164 |
| 11.4 | Server Properties | 165 |
| 11.4.1 | Search and optimization parameters | 165 |
| 11.4.2 | Metamorph parameters | 171 |
| 11.4.3 | Rank knobs | 173 |
| 11.4.4 | Other ranking properties | 173 |
| 11.4.5 | Indexing properties | 174 |
| 11.4.6 | Locking properties | 181 |
| 11.4.7 | Miscellaneous Properties | 182 |

| | | |
|-----------|---|------------|
| II | Metamorph Intelligent Query Language | 187 |
| 1 | Metamorph: The Program Inside the Program | 189 |
| 1.1 | Background | 189 |
| 1.2 | Original Design Mandate | 189 |
| 1.3 | Tokenization and Inverted Files | 190 |
| 1.4 | Metamorph Query Language Highlights | 192 |
| 1.5 | Your Basic Metamorph Search | 193 |
| 1.6 | More Complex Query Syntax | 194 |
| 2 | Hits, Intersections and Sets | 197 |
| 2.1 | Definition and Background | 197 |
| 2.2 | Basic Operation | 198 |
| 2.3 | Intersections and Search Logic | 198 |
| 2.4 | Intersection Quantity Specification Further Explained | 199 |
| 2.5 | How '+' and '-' Logic Affects Intersection Settings | 200 |
| 2.6 | Modulating Set Size | 201 |
| 2.6.1 | Limiting Sets to Roots Only | 201 |
| 2.6.2 | Expanding Sets by Use of See References | 203 |
| 2.7 | Modulating Set Content | 203 |
| 2.8 | Modulating Hit Size by Adjusting Delimiters | 204 |
| 2.8.1 | Discussion | 204 |
| 2.8.2 | Delimiter Syntax Summary | 206 |
| 2.8.3 | Rules of Delimiter Syntax | 206 |
| 3 | Thesaurus Customization | 209 |
| 3.1 | Specializing Knowledge | 209 |
| 3.2 | Equivalence File Explanation | 209 |
| 3.3 | Editing Concept Sets With Backref | 210 |
| 3.4 | Toggling Equiv Expansion On or Off | 212 |
| 3.5 | Creating a User Equivalence File By Hand | 214 |
| 3.6 | User Equivalence File Format | 214 |

| | | |
|----------|---|------------|
| 3.7 | Back Referencing | 215 |
| 3.8 | See Referencing | 216 |
| 3.9 | Word Classes and Parts of Speech | 216 |
| 3.10 | Rules of Precedence and Syntax | 217 |
| 3.11 | Specialized User Equivalence Files | 219 |
| 4 | Tailoring Metamorph's Linguistics | 221 |
| 4.1 | Editing Special Word Lists | 221 |
| 4.2 | Question Parsing | 222 |
| 4.3 | Morpheme Processing | 223 |
| 4.3.1 | Morpheme Stripping Routine | 223 |
| 4.3.2 | Suffix List Editing Notes | 225 |
| 4.3.3 | Toggling Prefix/Suffix Processing On/Off | 225 |
| 4.3.4 | Toggling Morpheme Rebuild Step On/Off | 225 |
| 4.3.5 | Setting Minimum Word Length | 226 |
| 4.4 | Warning to Linguistic Fiddlers | 226 |
| 4.5 | APICP Metamorph Control Parameters Summary | 226 |
| 5 | Types of Searches in Metamorph | 229 |
| 5.1 | Multiple Search Algorithms | 229 |
| 5.2 | The Natural Language Question | 230 |
| 5.3 | Single/Multiple Items, Keywords, Phrases, Wildcard Patterns (*) | 231 |
| 5.4 | Intelligent Intersection Search | 232 |
| 5.5 | String Search (Literal Pattern Match) | 233 |
| 5.6 | Fixed and Variable Length Regular Expressions (REX) | 234 |
| 5.6.1 | REX Description | 234 |
| 5.6.2 | Examples | 235 |
| 5.6.3 | Examples of Some Useful REX Expressions | 235 |
| 5.7 | Misspellings, Typos, and Pattern Approximations (XPM) | 236 |
| 5.7.1 | Searching for Approximations | 236 |
| 5.7.2 | XPM Rules of Syntax | 236 |

| | | |
|------------|--|------------|
| 5.8 | Numeric Quantities Entered as Text (NPM) | 237 |
| 5.9 | Designating Logic Operators (+) and (-) | 239 |
| 6 | REX On Its Own | 241 |
| 6.1 | REX, The Regular Expression Pattern Matcher | 241 |
| 6.2 | REX Program Syntax and Options | 243 |
| 6.3 | REX Expression Syntax | 244 |
| 6.4 | REX Repetition Operators | 245 |
| 6.5 | RE2 Syntax | 246 |
| 6.6 | REX Replacement Strings | 246 |
| 6.7 | REX Caveats and Commentary | 247 |
| 6.8 | Other REX Notes | 249 |
| III | Metamorph API Tools | 251 |
| 1 | Metamorph Application Program Interface Overview | 253 |
| 1.1 | Technical Description | 255 |
| 1.2 | Some Search Examples and Explanations | 256 |
| 1.3 | Potential Applications | 258 |
| 2 | The Programmers' Interface Overview | 259 |
| 2.1 | The Metamorph 3 API Package | 262 |
| 3 | Metamorph 3 API functions | 265 |
| 3.0.1 | openapicp, closeapicp - Metamorph API control parameters interface | 266 |
| 3.1 | APICP Variable Definitions | 267 |
| 3.2 | Application Notes | 269 |
| 3.2.1 | openmmapi, closemmapi - Metamorph API initialization and cleanup | 270 |
| 3.2.2 | setmmapi - Metamorph API reinitialization | 271 |
| 3.2.3 | getmmapi - Metamorph API search routine | 272 |
| 3.2.4 | infommapi - Metamorph API hit information | 274 |
| 3.2.5 | rdmmapi - synchronized read | 277 |

| | | |
|----------|---|------------|
| 3.2.6 | putmsg - handle a formatted message mmsgfh - FILE pointer for output mmsgfname - filename for output | 279 |
| 3.2.7 | putmsg() extensions: fixmmsgfh, closemmsgfh | 283 |
| 3.2.8 | putmsg() replacement | 284 |
| 3.3 | Query processing and Equivalence lookup | 286 |
| 3.4 | Equivalence editing callbacks | 291 |
| 3.5 | User equivalence file maintenance | 294 |
| 3.6 | Edit commands: | 295 |
| 3.7 | Reprogramming the Within Operator | 296 |
| 4 | Low level Pattern Matchers | 299 |
| 4.0.1 | Low level Pattern Matching functions | 300 |
| 5 | Windows Addendum | 305 |
| 5.0.1 | setmmsgfname - set the name of the message file | 306 |
| 5.0.2 | setmmsgfh - set the message file handle | 307 |
| 5.0.3 | setmmsg - set custom message handler function | 308 |
| V | Taxis Programs | 311 |
| 0.0.4 | tsql - Taxis Interactive SQL | 313 |
| 1 | The Taxis Client API | 315 |
| 1.1 | Overview | 315 |
| 1.1.1 | Taxis Client Functions | 316 |
| 1.1.2 | FLDLST - Struct containing results | 317 |
| 1.1.3 | n_getindexcount() - Hit information | 319 |
| 1.1.4 | taxis_open(), taxis_dup(), taxis_close() - SQL interface | 320 |
| 1.1.5 | taxis_prepare(), taxis_execute(), taxis_preexec() - SQL interface | 321 |
| 1.1.6 | taxis_fetch() - SQL interface | 324 |
| 1.1.7 | n_settaxisparam(), taxis_param(), n_resetparamtx() - SQL interface | 325 |
| 1.1.8 | taxis_flush() - SQL interface | 327 |
| 1.1.9 | taxis_flush_scroll() - SQL interface | 328 |

| | | |
|-------|---|-----|
| 1.2 | Modifying the server | 329 |
| 1.2.1 | adduserfuncs - Adding to the server | 330 |
| 1.2.2 | dbaddtype - Add a datatype | 331 |
| 1.2.3 | foaddfuncs - Add functions | 332 |
| 1.2.4 | fosetop - Add an operator | 334 |

List of Tables

| | | |
|------|--|-----|
| 2.1 | Specifications | 10 |
| 2.2 | Relational Database Terminology | 11 |
| 3.1 | Valid and Invalid Column Names | 19 |
| 3.2 | Data Types Used in Taxis | 19 |
| 4.1 | Comparison Operators Supported in Taxis | 26 |
| 4.2 | Logical Operator AND | 28 |
| 4.3 | Logical Operator OR | 29 |
| 5.1 | Arithmetic Operators Supported in Taxis | 48 |
| 5.2 | Taxis Aggregate Function Names | 53 |
| 7.1 | Metamorph delimiter abbreviations | 121 |
| 11.1 | Overview of System Catalog Tables in Taxis | 157 |

Part I

Using Taxis

Chapter 1

Preface

1.1 Acknowledgment

We would like to thank Ray Ageloff of the University of Rhode Island for his “*Primer on SQL*”, published by Times Mirror/Mosby College Publishing, Copyright (c) 1988. We found this to be the very best available text on Structured Query Language (SQL), and have drawn heavily upon it in our explanations of SQL, in a way which can be easily and practically understood by the end user.

1.2 Documentation Caveat

As Taxis is a breakthrough development, some of the more advanced features that have been planned for Taxis have not yet been fully implemented. Our goal with this manual has been to document both the features that are currently working, as well as those features that are planned. If a feature is shown as not currently implemented, or you don't see the capabilities you need, please call us to determine our current development timetable.

We thank you for your understanding, and encourage you to contact our technical support personnel with any questions that may arise. We are available 10 a.m. to 6 p.m. Eastern Time, Monday - Friday, by mail or Web, as follows:

*Thunderstone Software - EPI, Inc.
14837 Detroit Ave. PMB #303
Lakewood, Ohio 44107
Web: <http://www.thunderstone.com>*

1.3 Documentation Structure

There are three distinct sections to this manual. Where you start will depend on your needs. Chapters 2 to 11 of the book deal with the abilities of Taxis and provide an introduction to SQL. These chapters also note the differences between Taxis and a typical SQL database. These chapters can be skimmed by a person knowledgeable about SQL to note the differences with standard databases. In particular you may want to read about Metamorph queries in Chapter 7, and the section on `LIKE` and friends.

Part V, Chapter 1 explains how to use the Taxis server from within a C program. This chapter should be read by anyone intending to write a program to access Taxis.

Part V, Chapter ?? describes the programs that come with Taxis, including the daemon and ad-hoc query tool. This chapter is divided into three sections. These are the example programs, which are complete examples of how to program with Taxis, the programs required to use Taxis, and finally some maintenance and administration tools that are useful.

Chapter 2

Introduction

2.1 Taxis: Thunderstone's Text Information Server

What is it?

Taxis is a relational database server that specializes in managing textual information. It has many of the same abilities as products like Oracle(r), Informix(r), and Sybase(r) with one key difference: it can intelligently search and manage databases that have natural language text in them, which is something they can't do.

Why is that different?

Most other products are capable of storing a limited field that contains text, and usually that information is limited to being less than 500 characters long. And when it comes to finding something in that field you're limited to being able to search for a single word or string of characters.

In Taxis you can store text of any size, and you're able to query that information in natural language for just about anything you can imagine. We took our powerful Metamorph concept based text engine and built a specialized relational database server around it so that you have the best of both worlds.

What can we do with it?

Look at your current info-system and for the occurrence of anything that contains natural language information. This includes things like: e-mail, personnel records, research reports, memos, faxes, product descriptions, and word processing documents. Now imagine being able to collect and perform queries against these items as if they were a traditional database.

If you look closely enough you'll probably discover that about half the information that is resident within the organization has natural language or text as one of its most significant attributes. And chances are that there is little or no ability to manage and query these information resources based on their content. Traditional databases are fine as long as you are just adding up numbers or manipulating inventories, but

people use language to communicate and no product except Taxis can provide real access to the “natural language” components.

2.1.1 Features Unique to Taxis

Before we give you the specifications, we need to explain some features that are unique to Taxis.

Zero Latency Insert

When a record is added or updated within a Taxis table it is available for retrieval immediately. With any other product you would have to wait until the record was “indexed” before it would become available. The reason for this follows.

Variable Sized Records

Most databases allocate disk space in fixed size blocks that each contain a fixed number of records. The space allocated to each record is the maximum size that that record definition allows.

For example, let’s say you have a table that contains a 6 digit fixed length part number, and a variable length part description that could be up to 1000 characters long. Under the definition above, a 10,000 record database would require at least 10,060,000 bytes of disk space.

But let’s look a little closer at the facts behind our table. Our *maximum* part description is 1000 bytes, but that’s only because we have a few parts with really long winded descriptions; most of our part descriptions have an average length of about 100 characters. This is where Taxis comes in; it only stores what you use, not what you might use. This means that our table would only require about 1 MB of disk space instead of 10 Mbytes.

But wait, it doesn’t stop there! We have another hat trick. Because we store our records in this manner we also remove the limitation of having to specify the maximum length of a variable length field. In Taxis any variable field can contain up to a Gigabyte. (Not that we recommend 1 Gig fields.)

Indirect Fields

Indirect fields are byte fields that exist as real files within the file system. This field type is usually used when you are creating a database that is managing a collection of files on the server (like word processing files for instance). They can also be used when the 1 Gig limitation of fields is too small.

You may use indirect fields to point to your files anywhere on your file system or you may let Taxis manage them under the database.

Since files may contain any amount of any kind of data indirect fields may be used to store arbitrarily large binary objects. These Binary Large Objects are often called BLOBs in other RDBMSes.

However in Taxis the `indirect` type is distinct from `blob/blobz`. While each `indirect` field is a separate external file, all of a table’s `blob/blobz` fields are stored together in one `.blb` file adjacent to

the `.tbl` file. Thus, `indirect` is better suited to externally-managed files, or data in which nearly every row's field value is very large. The `blob` (or compressed `blobz`, available in Taxis Version 8 and later) type is better suited to data that may often be either large or small, or which Taxis can manage more easily (e.g. faster access, and automatically track changes for index updates).

Variable Length Index Keys

Traditional database systems allocate their indexes in fixed blocks, and so do we; but we were faced with a problem. Typical English language contains words of extremely variant length, and we wanted to minimize the overhead of storing these words in an index. Traditional Btrees have fixed length keys, so we invented a variable length key Btree in order to minimize our overhead while not limiting the maximum length of a key.

Why all the variable stuff?

Taxis stands for Text Information Server, and text databases are really different in nature to the content of most standard databases. Have a look at the word processing files that are on your machine. The vast majority are probably relatively small, but then there's an occasional whopper. The same is true for the content of most documents: their size and content are extremely variable.

Taxis is optimized for two things: Query time and variable sized data.

2.1.2 Specifications

2.2 Taxis as a Relational Database Management System

Taxis is a database management system (DBMS) which follows the relational database model, while including methods for addressing the inclusion of large quantities of narrative full text. Taxis provides a method for managing and manipulating an organization's shared data, where intelligent text retrieval is harnessed as a qualifying action for selecting the desired information.

Taxis serves as an "intelligent agent" between the database and the people seeking data from the database, providing an environment where it is convenient and efficient to retrieve information from and store data in the database. Taxis provides for the definition of the database and for data storage. Through security, backup and recovery, and other services, Taxis protects the stored data.

At the same time Taxis provides methods for integrating advanced full text retrieval techniques and object manipulation with the more traditional roles performed by the RDBMS (relational database management system).

2.3 Relational Database Background

Taxis, as most recent DBMSs, is based on the relational data model. The fundamental organizational structure for data in the relational model is the relation. A *relation* is a two-dimensional table made up of

Table 2.1: Specifications

| Feature | Texis Specs |
|-------------------------------|---|
| Multiple Servers per machine | Yes |
| Multiple Databases per server | Yes |
| Tables per database | 10,000 |
| Max table size | 2 gigabytes (2^{31}) on 32-bit systems, 9 exabytes (2^{63}) on 64-bit systems |
| Rows per table | 1 billion |
| Columns per table | Unlimited |
| Indexes per table | Unlimited |
| Max field size | 1 gigabyte |
| Max field name | 32 characters |
| Max tables per query | 400 |
| User password security | Yes |
| Group password security | Yes |
| Index types | Btree, Inverted, Text, Text inverted |
| Max index key size | 8192 |
| Standard Data Types | bool, [var]byte, dbyte, qbyte, [var]char, short, int, long, [u]int64, float, double, time, indirect, counter, strlst, blob[z] |
| Max user defined data types | 64 |

rows and columns. Each relation, also called a table, stores data about *entities*. These entities are objects or events on which an organization chooses to collect data. Patients, doctors, services, and insurance carriers are examples of entities.

The columns in a relation represent characteristics (*attributes*, *fields*, or *data items*) of an entity, such as patient identification number, patient name, address, etc. The rows (called *tuples* in relational jargon) in the relation represent specific occurrences (or records) of a patient, doctor, insurance group number, service rendered, etc. Each row consists of a sequence of values, one for each column in the table.

In addition, each row (or record) in a table must be unique. The *primary key* of a relation is the attribute or attributes whose value uniquely identifies a specific row in a relation. For example, a Patient identification number (ID) is normally used as a primary key for accessing a patient's hospital records. A Customer ID number can be the primary key in a business.

Over the years, many different sets of terms have been used interchangeably when discussing the relational model. Table 2.2 lists these terms and shows their relationship.

Figure 2.1 illustrates two relations. The first one depicts patients and the second represents outstanding patient invoices. A row in the PATIENT relation represents a particular patient, while a row in the INVOICE relation represents a patient invoice. Thus, a relation provides a structure for storing data about some entity within the organization. In fact, a database in the relational model consists of several relations, each representing a different entity.

An important characteristic of the relational model is that records stored in one table can be related to

Table 2.2: Relational Database Terminology

| Relational Model Literature | Relational DBMS Products | File Processing |
|-----------------------------|--------------------------|-----------------|
| Relation | Table | File |
| Tuple | Row | Record |
| Attribute | Column | Field |

Figure 2.1: PATIENT and INVOICE Relations

a. PATIENT Relation

| PATIENT ID | PATIENT NAME | ADDRESS | CITY | STATE |
|------------|--------------|----------------|------------|-------|
| 107 | Pryor | 1 Ninigret Ave | Quonsett | RI |
| 111 | Margolis | 3 Chester Ave | Westerley | RI |
| 112 | Frazier | 7 Conch Rd | New London | CT |
| 123 | Chen | 163 Namcock Rd | Attleboro | MA |
| 128 | Steckert | 14 Homestead | Norwich | CT |

b. INVOICE Relation

| INVOICE NO | DATE | AMOUNT | PATIENT ID |
|------------|----------|--------|------------|
| 71115 | 11/01/92 | 255.00 | 112 |
| 71116 | 11/03/92 | 121.25 | 123 |
| 71117 | 11/08/92 | 325.00 | 111 |
| 71118 | 11/08/92 | 48.50 | 112 |
| 71119 | 11/10/92 | 88.00 | 107 |
| 71120 | 11/12/92 | 245.40 | 111 |
| 71121 | 11/15/92 | 150.00 | 112 |
| 71122 | 11/17/92 | 412.00 | 128 |
| 71123 | 11/22/92 | 150.00 | 112 |

records stored in other tables by matching common data values from the different tables. Thus data in different relations can be tied together, or integrated. For example, in Figure 2.1, invoice 71115 in the INVOICE relation is related to Patient 112, Frazier, in the Patient relation because they both have the same patient ID. Invoices 71118, 71121, and 71123 are also related to Patient 112.

A database in the relational model is made up of a collection of interrelated relations. Each relation represents data (to the users of the database) as a two-dimensional table. The terms *relation* and *table* are interchangeable. For the remainder of the text, the term *table* will be used when referring to a relation.

Access to data in the database is accomplished in two ways. The first way is by writing application programs written in procedural languages such as C that add, modify, delete, and retrieve data from the database. These functions are performed by issuing requests to the DBMS. The second method of accessing data is accomplished by issuing commands, or queries, in a fourth-generation language (4GL) directly to the DBMS to find certain data. This language is called a *query language*, which is a nonprocedural language characterized by high-level English-like commands such as UPDATE, DELETE, SELECT, etc. Structured Query Language (SQL, also pronounced “Sequel”) is an example of a nonprocedural query language.

2.4 Support of SQL

As more corporate data processing centers use SQL, more vendors are offering relational database products based on the SQL language.

In 1986, the American National Standards Institute (ANSI) approved SQL as the standard relational database language. SQL is now the standard query language for relational database management systems.

Taxis supports the SQL query language. Any program capable of issuing SQL commands can interface with Taxis, to accomplish the database management, access, and retrieval functions.

For example, Microsoft ACCESS provides a means for creating a GUI (*graphical user interface*) front end for a database. Using icons in a point and click fashion familiar to the user, one can maneuver through the database options where queries are created and issued to the database. While the user does not see the form of the query, the ACCESS program is translating them to SQL. These queries can be passed to and implemented in a more powerful fashion by Taxis, where the results are passed back to the user via the Windows ACCESS application.

For any application written in C, an embedded SQL processor allows the C Programmer to use Taxis within his or her application.

Taxis is a SQL driven relational database server that merges the functionality of METAMORPH, our concept based text retrieval engine with a DB2-like database. The prime differences to other systems are in the LIKE statement and in the allowable size of text fields.

This manual will explain SQL as the query language used in an enhanced manner by Taxis, so that users will be able to write queries accessing data from a database.

2.5 Case Example: Acme Industrial Online Corporate Library

To provide a frame of reference to show the concepts and syntax of SQL for use by Taxis, we will use the example of Acme Industrial's Online Corporate Library. It is the job of the corporate librarian to make selectively accessible to Management, Personnel, Marketing, and Research & Development (R&D), the full text content of management, personnel, marketing, and R&D reports, both in tabulated and full text form.

Many entities and their related functions are involved. While a researcher in R&D requires a conceptual search and full text study of all work that has been done similar to her own project, the Technology Manager may be interested in hours spent by which staff, on what projects, and to what final results in encapsulated form. The Marketing Director will want to keep track of finished reports on subjects of interest, while having access to promotional budget information to plan the focus of the ad campaign over the next two quarters.

The Corporate Librarian must be able to supply concise short form and expanded long form information on demand to those who request it, while maintaining discretionary security. Therefore a mix of fielded and full text information must be available and easy to manipulate and turn into generated report content.

It may even be that each department wishes to create their own front end application program which defines the way in which they conduct their daily business while accessing this information. But where the information is shared, the online library database is common to each and must be managed as such.

All the daily activities of Acme Industrial create the need for recording and storing vast amounts of data. These activities affect the Online Corporate Library System in numerous ways. Data concerning transactions and daily events must be captured in order to keep the data in the system accurate. The system must have the capability to answer unplanned, one-time-only queries in addition to preplanned queries.

Taxis is the SQL Relational Database Server which has the horsepower to manage this main repository of information.

This introductory chapter has introduced you to several concepts and terms related to relational database management systems. In addition we have provided the background case of Acme Industrial's Online Corporate Library System that will be used in examples throughout the text. In the next chapter you will learn how to define and remove tables for use by Taxis.

Chapter 3

Table Definition

Texis permits users to define, access, and manipulate data stored in a database. This chapter describes how a table is defined and deleted. In addition, you will be shown an example of how data is loaded into a table.

3.1 Creating the Resume Table

One of the functions of the Librarian is to maintain a resume database for Personnel, for potentially qualified staff for jobs as they open up. Therefore one of the tables in the Acme Online Corporate Library System is the RESUME table. This table is created by issuing the CREATE TABLE command.

If you enter the following:

```
CREATE TABLE RESUME
( RES_ID CHAR(5) ,
  RNAME CHAR(15) ,
  JOB CHAR(15) ,
  EDUC CHAR(60) ,
  EXP VARCHAR(2000)
);
```

SQL statements as passed to Texis can be entered on one or more lines. Indenting is recommended to improve readability, but it is not required.

The CREATE TABLE command is entered interactively at a terminal, or as embedded in an application program. Note that the list of column definitions is enclosed in parentheses and that each column definition is separated from the next column definition by a comma. In all examples in this text, each SQL statement is shown in uppercase letters to help you identify what is to be entered. However, in most cases you actually can enter the statement in either upper or lowercase.

The first line in the CREATE TABLE statement identifies the name of the table: RESUME. The next five lines define the five columns that make up the RESUME table. The data types chosen to define each column are explained further on in this chapter.

Figure 3.1: RESUME Table after the CREATE TABLE Command

| RES_ID | RNAME | JOB | EDUC | EXP |
|---|-------|-----|------|-----|
| (No data is stored in the table at the time it is created.) | | | | |

1. The first column, named RES_ID, stores the resume's identification number (ID). Five characters are allowed for a Resume ID, following Acme internal naming conventions of a letter followed by up to 4 other characters; e.g., 'R243' or 'R-376'.
2. The second column, named RNAME, stores the name of the resume's job applicant. No name longer than 15 characters can be stored in this column.
3. The third column, named JOB, stores the job or jobs the person is applying for. A maximum of 15 characters is allowed for this column.
4. The fourth column, named EDUC, stores a brief description of the applicant's education. A maximum of 60 characters is allowed for this column. Note: One could choose to define EDUC with VARCHAR rather than CHAR, so that a full educational description could be entered without regard to waste of allocated space.
5. The fifth column, named EXP, stores the full text description of the applicant's job experience as included in the resume. You have two choices for the text field:
 - (a) You can store the entire description in the Taxis table. This is useful for short descriptive lines, for abstracts of one or more paragraphs, or for short reports of one to two pages as depicts the usual resume. Data type would be defined as a variable length character VARCHAR(x) where X indicates the suggested number of characters.
 - (b) You can store filenames in the Taxis table. In this case Taxis would use the filename to direct it to the text of the actual file. Data type would be defined as INDIRECT.

In our EXP text column for the RESUME table we have chosen to store the full text in the Taxis table, as concept searches of this column are part of almost every resume search request. If we only occasionally referred to the full text content, we might prefer to store filenames which would point to the full text only when necessary.

Tables defined with the CREATE TABLE command are referred to as *base tables*. The table definition is automatically stored in a data dictionary referred to as the *system catalog*. This catalog is made up of various tables that store descriptive and statistical information related to the database. The catalog can be accessed to retrieve information about the contents and structure of the database. The system catalog is discussed in more detail in Chapter 11.

As shown in Figure 3.1, the CREATE TABLE command results in an empty table.

Figure 3.2: RESUME Table with One Row Inserted

| RES_ID | RNAME | JOB | EDUC | EXP |
|--------|----------------|--------------|------------|------------------------|
| R323 | Perkins, Alice | Snr Engineer | M.B.A. ... | Presently employed ... |

3.2 Inserting Data into the Resume Table

Once the table has been created, and before any data can be retrieved, data must be added to the table using the `INSERT` command. The first row is added to the `RESUME` table as follows.

If you enter:

```
INSERT INTO RESUME
VALUES ('R323','Perkins, Alice','Snr Engineer',
       'M.B.A. 1984 George Washington Univ',
       'Presently employed at ...') ;
```

Syntax Notes:

- Columns defined as `CHAR` (character) and `VARCHAR` (variable length character) have values enclosed in single quotes.
- Parentheses must be placed around the set of data values.
- Each data value is separated by a comma.
- A long full text column such as job experience, would be loaded by a program function rather than manually typed in.

In the above statement, one row of data was stored in the `RESUME` table. Figure 3.2 shows the `RESUME` table after the first record has been added.

To add the second row into the `RESUME` table, you enter the `INSERT` command again.

If you enter

```
INSERT INTO RESUME
VALUES ('R421','Smith, James','Jr Analyst',
       'B.A. 1982 Radford University'
       'Experience has been in ...') ;
```

Figure 3.3 shows the contents of the `RESUME` table after two rows have been added.

Additional `INSERT` commands are used to enter the `RESUME` data, as was illustrated in Figure 3.3. A more complete description of the `INSERT` command appears in Chapter 9.

Figure 3.3: RESUME Table with Two Rows Inserted

| RES_ID | RNAME | JOB | EDUC | EXP |
|--------|----------------|--------------|------------|-------------------------|
| R323 | Perkins, Alice | Snr Engineer | M.B.A. ... | Presently employed ... |
| R421 | Smith, James | Jr Analyst | B.A. ... | Experience has been ... |

3.3 Defining a Table

As illustrated in the creation of the RESUME table, tables are created in Taxis when you specify their structure and characteristics by executing a `CREATE TABLE` command.

The form of this command is:

```
CREATE TABLE [table-type] table-name
  (column-name1 data-type
    [, column-name2 data-type] ... ) ;
```

Syntax Notes: A SQL statement may contain optional clauses or keywords. These optional parts are included in the statement only if needed. Any clause within brackets ‘[xxx]’ indicates an optional clause.

Command Discussion

The `CREATE TABLE` command gives the name of the table, the name of each column in the table, and the type of data placed in each column. It can also indicate whether null values are permitted in columns.

Table Type: When creating a table you can optionally specify a table type. A standard database table will be created if no type is specified.

Specifying a `RAM` table will create a table that only exists in memory for the current database connection. The table is not added to the system catalog, and is not visible to other database connections. It can be used as a temporary working table in an application. Within Vortex a `<sqlcp cache close>` or switching databases may remove the temporary table.

A `BTREE` table creates a table that is inherently indexed by the fields in the order listed. You can not create other indexes on a `BTREE` table. This can be useful for key-lookup tables that have a lot of small rows.

Table Names: Each table in Taxis is assigned a name. A table name can have up to 18 characters (case is significant). The first character must be a letter, but the remaining characters can include numbers, letters, and the underscore (`_`) character. Table names may not be the same as SQL keywords or data types. For example, `RESUME`, `BUDGET93`, and `PROD_TEST` are all valid table names. On MSDOS based systems table names must be unique regardless of case in the first 8 characters.

Table 3.1: Valid and Invalid Column Names

| Valid Column Names | Invalid Column Names | Reason Invalid |
|--------------------|----------------------|----------------------------------|
| EMP_NBR | EMP-NBR | Hyphen is not allowed. |
| EMP_NBR | EMP.NBR | Period is not allowed. |
| COST1 | COST_IN_\$ | \$ is not allowed. |
| COST_PER_MILE | COST PER MILE | Spaces are not allowed. |
| SALES1991 | 1991SALES | Name cannot start with a number. |
| Where | WHERE | Can not be SQL keyword. |
| Date | DATE | Can not be SQL data type. |

Table 3.2: Data Types Used in Taxis

| Type of Data | Taxis Syntax | Example | Data Value |
|--------------|-------------------|-------------------|----------------------|
| Character | CHAR(length) | CHAR(10) | SMITH |
| Character | CHARACTER(length) | CHAR(25) | 10 Newman Rd |
| Byte | BYTE(length) | BYTE(2) | DE23 |
| Numeric | LONG | LONG | 657899932 |
| Numeric | INTEGER | INTEGER | 657899932 |
| Numeric | SMALLINT | SMALLINT | -432 |
| Numeric | FLOAT | FLOAT | 8.413E-04 |
| Numeric | DOUBLE | DOUBLE | 2.873654219543E+100 |
| Numeric | UNSIGNED INTEGER | UNSIGNED INTEGER | 4000000000 |
| Numeric | UNSIGNED SMALLINT | UNSIGNED SMALLINT | 60000 |
| Date/Time | DATE | DATE | 719283474 |
| Text | VARCHAR(length) | VARCHAR(200) | "The subject of ..." |
| Text | INDIRECT | INDIRECT | Filename |
| Counter | COUNTER | COUNTER | 2e6cb55800000019 |
| String list | STRLST | STRLST | apple,orange,peach, |

Column Names: A column stores data on one attribute. In our example, we have attributes such as Resume ID, job sought, education, and experience. Each column within a table has a unique name and may consist of up to 18 characters (case is significant). The first character must be a letter and the remaining characters may consist of letters, numbers, and the underscore (_) character. No blank spaces are allowed in the column name. Table names may not be the same as SQL keywords or data types. Table 3.1 shows examples of valid and invalid column names.

Data Types: Each column within a table can store only one type of data. For example, a column of names represents *character* data, a column storing units sold represents *integer* data, and a column of file dates represents *time* data. In Taxis, each column name defined in the `CREATE TABLE` statement has a data type declared with it. These data types include *character*, *byte*, *integer*, *smallint*, *float*, *double*, *date*, *varchar*, *counter*, *strlst*, and *indirect*. Table 3.2 illustrates the general format for each data type. A description of each of the Data Types listed in Table 3.2 follows.

- CHAR(length):** Used to store character data, such as names, job titles, addresses, etc. Length represents the maximum number of characters that can be stored in this column. CHAR can hold the value of any ASCII characters 1-127. Unless you want to limit the size of the field absolutely you should in general use VARCHAR instead as it is more flexible.
- CHARACTER(length):** Same as CHAR, used to store character data, an alternate supported syntax. As with CHAR, length represents the maximum number of characters that can be stored in this column.
- BYTE:** Similar to CHAR but with significant differences, BYTE is used to store any unsigned (non-negative) ASCII values from 0-255. Specifying BYTE indicates each is a one byte quantity. A byte would be used where you want to store a small number less than 255 such as age, or perhaps a flag. A VARBYTE can also be used where the length of specified characters is variable rather than fixed, where you are storing arbitrary binary data.
- LONG:** Used to store large whole numbers; i.e., those without a fractional part, such as population, units sold, sales in dollars. The range of long values will depend on the platform you are using. For most platforms it is identical to INTEGER.
- INTEGER:** Used to store large whole numbers where you want to ensure a 32-bit storage unit. The largest integer value is +2147483647. The smallest integer value is -2147483648.
- UNSIGNED INTEGER:** Used for similar purposes as INTEGER when you know the number will never be less than zero. It also extends the maximum value from 2,147,483,647 to 4,294,967,295. This is synonymous with DWORD.
- SMALLINT:** Used to store small whole numbers that require few digits; for example, age, weight, temperature. The largest value is +32,767. The smallest value is -32,768.
- UNSIGNED SMALLINT:** Can store positive numbers in the range from 0 to 65,535. Can be used in many of the same places as SMALLINT.
- INT64:** Used to store large whole numbers when a 64-bit quantity must be assured (LONG size varies by platform). Value range is -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807. Added in version 6.
- UINT64:** Similar to INT64, but unsigned. Value range is 0 through 18,446,744,073,709,551,616. Added in version 6.
- FLOAT:** Used to store real numbers where numerical precision is important. Very large or very small numbers expressed in scientific notation (E notation).
- DOUBLE:** Used to hold large floating point numbers. Having the characteristics of a FLOAT, its precision is greater and would be used where numerical precision is the most important requirement.
- DATE:** Used to store time measured in integer seconds since 00:00:00 Jan. 1 1970, GMT (Greenwich mean time). When entered in this fashion the format is an integer representing an absolute number of seconds; e.g., 719283474. The DATE data type is used to avoid confusions stemming from multi-sourced information originating from different time zone notations. This data type is entered by a program function rather than manually, and would generally be converted to calendar time before being shown to the user. DATES may also be entered as strings representing a date/time format such as '1994-03-05 3:00pm'
- VARCHAR(length):** Used to store text field information entirely in Taxis. The specified length is offered as a suggestion only, as this data type can hold an unlimited number of characters. In the

example in Table 3.2, there may be a short description of the text, or a relatively small abstract which is stored in the field of the column itself.

BLOB: Used to store text, graphic images, audio, and so on, where the object is not stored in the table itself, but is indirectly held in a BLOB field. BLOB stands for Binary Large Object, and can be used to store the content of many fields or small files at once, eliminating the need for opening and closing many files while performing a search. BLOB is used when having a specific filename is not desired. The BLOB is created and managed at a system level. The total data held for all BLOBs in a table is limited by the filesystem. The BLOB file is not accessed unless the data in it is needed. This will improve the performance of queries that do not need to access the data. This can also be useful if you are creating a METAMORPH INVERTED index, and do not allow post processing, and do not display the actual contents of the record, as the data will not be accessed at all, and can be removed. This should only be done with extreme caution.

BLOBZ: Similar to BLOB fields, except that each BLOBZ's data is compressed before storing on disk, and is decompressed upon reading from disk. The compression/decompression is done internally. Alternatively, it can be handled externally via the executables specified by the `Blob Compress EXE` and `Blob Uncompress EXE` commands in the `[Taxis]` section of the `taxis.ini` configuration file. External compression allows custom compression types to be deployed – perhaps better than the `gzip` format supported internally by Taxis – but at a speed penalty due to the overhead of running the executables. The BLOBZ type is only supported in Taxis Version 8 and later.

INDIRECT: Used to store filenames which point to data stored in some other location. Most frequently an INDIRECT column would point to files containing quantities of full text. Only one filename may be stored in an INDIRECT field. The filenames can be inserted with SQL by specifying the filename as a string, or through a program, which might generate the files to store the data. The choice of storing text or filenames only in Taxis will depend on what you plan to do with the files, and also how big they are. INDIRECT can be used to point to images or other objects as well as text, although currently only text files can be meaningfully indexed.

COUNTER: This field holds an 8 byte value, which can be made unique across all tables in the database. To insert a counter value in SQL you can use the `COUNTER` keyword in the insert clause. A counter is made up of two fields, a time, and a sequence number. This allows the field to be compared with times, to find all records inserted before a particular time for example.

STRLST: A string list is used to hold a number of different strings. The strings are delimited by a user defined character in the input string. The delimiter character is printed as the last character in the result string when a `strlst` value is converted to a `varchar` result string (this aids conversion back to `strlst` when the `varchar to strlst sep` setting, p. 184, is “`lastchar`”). This type is most useful when combined with an application which needs lists of strings, and set-like operators such as `IN`, `SUBSET` or `INTERSECT`. Other operators are generally undefined for `strlst`, though in Taxis version 7 and later equality (“`=`” comparison etc.) is defined to be monolithic string-compare of the entire list; equality of `strlst` and `varchar` is the same, treating the `varchar` as a one-item `strlst` (if non-empty) or empty `strlst` (if empty).

One large difference in Taxis over other database management systems is in the range of data types it supports. While the traditional fixed length forms of `CHAR`, `INTEGER`, `FLOAT` and so on are used, there is a corresponding variable length data type which can be used when appropriate, such as is represented in `VARCHAR`.

The length following CHAR, as in `CHAR(100)`, indicates that 100 is the maximum number of allowed characters. Each record with such a data type defined will have a size of 100 characters, regardless of whether 3 characters, 57 characters, or even a NULL value is entered. The length following VARCHAR, as in `VARCHAR(100)`, indicates that 100 characters is a suggested length. If an entry of 350 characters is required in this field, VARCHAR can make allowances to handle it.

The 100 character suggestion in this case is used for memory allocation, rather than field length limitation. Therefore a VARCHAR length should be entered as the average, rather than the largest size for that field. Entering an extremely large length to accommodate one or two unusual entries would impair the handling of memory for normal operations.

The sophisticated aspects of database design involving choice and use of data types towards performance and optimization of table manipulation are addressed in more depth in Chapter 11, *Administration of the Database*.

The order in which the columns are listed in the `CREATE TABLE` command is the order in which the column names will appear in the table.

3.4 Removing a Table

When a table is no longer needed, it is deleted with the `DROP TABLE` command. The format of this command is:

```
DROP TABLE table-name ;
```

The information about the indicated table is removed from the system catalog tables that Taxis maintains on all tables in the database. In effect, you can no longer access, add, modify, or delete data stored in the table. From the user's viewpoint, the table definition and the data stored in the table have been eliminated.

Indirect files referenced within the dropped table are not deleted unless they are Taxis managed indirects under the database. So if you have indirects pointing to your own word processor files, they won't be lost when the table is dropped.

For example, if the RESUME table becomes no longer needed, you can delete this table. If you enter the following:

```
DROP TABLE RESUME;
```

This chapter has covered the creation and dropping of tables in Taxis. You were also shown how to insert data into a table. In the next chapter, you will begin to learn how to query the database, the most important feature of Taxis in differentiating its operation from other database management systems.

Chapter 4

A First Look at Queries

Taxis uses a query language that gives users access to data stored in a relational database. The data manipulation component of this language enables a user to:

- Write queries to retrieve information from the database.
- Modify existing data in the database.
- Add new data to the database.
- Delete data from the database.

In this and the next two chapters, we will review the query capabilities of Taxis. In Chapter 9, we will study the update, insert, and delete features of the language.

After the tables have been created and loaded with data, you can answer requests for information from a database without the help of professional programmers. You write a question, also called a query, that consists of a single statement explaining what the user wants to accomplish. Based on this query, the computer retrieves the results and displays them. In this chapter you will study some of the simpler ways to form queries.

In Taxis, you retrieve data from tables using the `SELECT` statement, which consists of one or more `SELECT-verbFROM`-`WHERE` blocks. The structure of this statement, in its simplest form, consists of one block containing three clauses: `SELECT`, `FROM`, and `WHERE`. The form of this statement follows:

```
SELECT  column-name1 [, column-name2] ...  
FROM    table-name  
[WHERE  search-condition] ;
```

Syntax Notes:

- The “...” above indicates additional column names can be added.
- Brackets ‘[]’ surrounding a clause means the clause is optional.

Command Discussion

SELECT: The `SELECT` clause lists the column names that you want displayed in answer to the query.

FROM: The `FROM` clause indicates the table of data “FROM” which you want to retrieve information.

WHERE: The `WHERE` clause is used to screen the rows you want to retrieve, based on some criteria, or search condition, that you specify. This clause is optional, and, if omitted, all rows from the table are retrieved.

4.1 Retrieving From the Entire Table

For this example, we will use a `REPORT` table, into which has been loaded reports submitted by all departments, by title, author, and reference filename. A three character department code is used, defined in long form in another `DEPARTMENT` table.

To retrieve the columns you want displayed, indicate the column names after the keyword `SELECT`. The order in which the column names appear after the `SELECT` clause is the order in which these columns will be displayed.

Example: Let’s retrieve a list of all report titles.

If you enter the statement:

```
SELECT  TITLE
FROM    REPORT ;
```

The result displayed on the screen will be:

```
TITLE
Innovations in Disappearing Ink
Disappearing Ink Promotional Campaign
Advertising Budget for 4Q 92
Improvements in Round Widgets
Target Market for Colored Paperclips
Ink Color Panorama
Departmental Meeting Schedule
```

The column name is automatically used as the column heading.

The first line in the `SELECT` statement indicates the column name `TITLE` is to be displayed. The second line indicates that `TITLE` is found in the `REPORT` table.

Example: If you want to display report titles, authors, and department, you must specify that information in the `SELECT` clause.

If you enter the statement:

```
SELECT  TITLE, AUTHOR, DEPT
FROM    REPORT ;
```

where each column name is separated from the next by a comma, and columns are displayed in the order you specify in the `SELECT` clause, the result displayed on the screen will be:

| TITLE | AUTHOR | DEPT |
|---------------------------------------|------------------|------|
| Innovations in Disappearing Ink | Jackson, Herbert | RND |
| Disappearing Ink Promotional Campaign | Sanchez, Carla | MKT |
| Advertising Budget for 4Q 92 | Price, Stella | FIN |
| Improvements in Round Widgets | Smith, Roberta | RND |
| Target Market for Colored Paperclips | Aster, John A. | MKT |
| Ink Color Panorama | Jackson, Herbert | RND |
| Departmental Meeting Schedule | Barrington, Kyle | MGT |

4.1.1 Retrieving All the Columns

You don't need to know the column names to select data from a table. By placing an asterisk (*) in the `SELECT` clause, all columns of the table identified in the `FROM` clause will be displayed. This is an alternative to listing all the column names in the `SELECT` clause.

Example: Let's look at all the data stored in the `REPORT` table.

If you enter the statement

```
SELECT  *
FROM    REPORT ;
```

the result displayed on the screen will be

| TITLE | AUTHOR | DEPT | FILENAME |
|------------------------------|------------------|------|---------------------|
| ... Disappearing Ink | Jackson, Herbert | RND | /docs/rnd/ink.txt |
| ... Ink Promotional Campaign | Sanchez, Carla | MKT | /docs/mkt/promo.rpt |
| ... Budget for 4Q 92 | Price, Stella | FIN | /docs/ad/4q.rpt |
| ... Round Widgets | Smith, Roberta | RND | /docs/rnd/widg.txt |
| ... Paperclips | Aster, John A. | MKT | /docs/mkt/clip.rpt |
| ... Color Panorama | Jackson, Herbert | RND | /docs/rnd/color.txt |
| ... Meeting Schedule | Barrington, Kyle | MGT | /docs/mgt/when.rpt |

4.2 Retrieving a Subset of Rows: Simple Conditions

Often you don't want to retrieve all the rows in a table but want only the rows that satisfy one or more conditions. In this case, you would include the `WHERE` clause in the `SELECT` statement to retrieve a portion, or subset, of the rows in a table.

Table 4.1: Comparison Operators Supported in Taxis

| Type of Comparison | Taxis Symbol |
|--------------------------|--------------|
| Equal to | = |
| Less than | < |
| Less than or equal to | <= |
| Greater than | > |
| Greater than or equal to | >= |
| Not equal to | <> or != |

A *search condition* expresses the logic by which the computer determines which rows of the table are retrieved and which are ignored. The search condition has many variations. A simple search condition is formed with a *conditional expression*, which specifies a comparison between two values. It has the following format:

expression comparison operator expression

The expressions in the conditional expression are usually a column name or a constant. The comparison operator indicates a mathematical comparison such as less than, greater than, equal to, etc. Table 4.1 shows the comparison operators allowed in Taxis.

Example: Let's say there is a DEPARTMENT table which has listed in it the department code, the long form department name, the department head, the division to which the department belongs, and the annual department budget. The conditional expression to find departments with a budget above \$25,000 can be written:

BUDGET > 25000

In this case BUDGET is being compared to a numeric constant.

The conditional expression to find all departments in the Product Division is written:

DIV = 'PROD'

Character constants, sometimes called character strings, are enclosed in single quotes. The conditional expression can compare numeric values to one another or string values to one another as just shown.

Each row in the indicated table is evaluated, or tested, separately based on the condition in the WHERE clause. For each row, the evaluation of the conditional expression is either true or false. When a condition is true, a row is retrieved; when the condition is false, the row is not retrieved. For example, if a department has a \$35,000 budget, then the conditional expression "BUDGET > 25000" is true and the row is included in the query result. However, if the department had a budget of \$15,000, then the result of the conditional expression "BUDGET > 25000" is false and the row is not retrieved.

Example: Let's develop a list of all departments, in long form, in the Product Division.

Enter the statement:

```
SELECT  DNAME
FROM    DEPARTMENT
WHERE   DIV = 'PROD' ;
```

'PROD' is the search condition, and as a character string must be enclosed in quotes.

The result displayed will be:

| |
|------------------------------|
| DNAME |
| Research and Development |
| Manufacturing |
| Customer Support and Service |
| Product Marketing and Sales |

In the WHERE clause, the condition "DIV must equal PROD" results in the retrieval of the name of each department in the Product Division. As only DNAME, the long form departmental name, was requested in the SELECT statement, a list of department names is all that is shown.

Example: Let's develop a list of all departments with a budget above \$25,000.

Enter the statement:

```
SELECT  DNAME, BUDGET
FROM    DEPARTMENT
WHERE   BUDGET > 25000 ;
```

Note that numeric values, as 25000, are not enclosed in quotes.

The result displayed will be:

| | |
|-------------------------------------|--------|
| DNAME | BUDGET |
| Finance and Accounting | 26000 |
| Corporate Legal Support | 28000 |
| Research and Development | 27500 |
| Manufacturing | 32000 |
| Strategic Planning and Intelligence | 28500 |

4.3 Retrieving a Subset of Rows: Compound Conditions

The conditions illustrated in the previous section are called simple conditions because each involves a single comparison. It is also possible to develop more complex conditions involving two or more conditional expressions. You combine conditions using the logical operators AND, OR, or NOT to connect conditional expressions. When two or more conditions are combined by logical operators, the conditional expression is

Table 4.2: Logical Operator AND

| | Values for DIV | Values for BUDGET | Condition1 DIV='CORP' | Condition2 BUDGET<12000 | Yields | Row Result |
|---|-------------------|----------------------|--------------------------|----------------------------|--------|---------------|
| 1 | CORP | 10500 | True | True | True | Retrieved |
| 2 | CORP | 28000 | True | False | False | Not retrieved |
| 3 | PROD | 11000 | False | True | False | Not retrieved |
| 4 | PROD | 27500 | False | False | False | Not retrieved |

called a *compound condition*. For example, you may want a list of departments from the Product Division only with budgets under \$20,000.

The form of the compound condition is:

```
conditional    logical    conditional    logical    conditional
expression1    operator    expression2    operator    expression3
```

As with simple conditional expressions, the evaluation of a compound condition is either true or false, with true resulting in retrieval of a row and false resulting in no retrieval.

4.3.1 Retrieval Using the AND Operator

When AND is used to connect two conditions, each conditional expression must be true for the condition to be true and the row retrieved. If any condition within a compound condition is false, the compound condition is false and the row is not selected.

For example, if you want to retrieve the records of Corporate Division Departments with a budget under \$10,000 you can write the following compound condition:

```
DIV = 'CORP' AND BUDGET < 12000
```

In this example, AND is the logical operator.

Table 4.2 illustrates the four possible cases that can occur with the logical operator AND for the compound condition just described.

Example: Based on the above, let's develop a list of departments in the Corporate Division with a budget under \$12,000.

If you enter the statement:

```
SELECT  DNAME, DIV, BUDGET
FROM    DEPARTMENT
WHERE   DIV = 'CORP' AND BUDGET < 12000 ;
```

the result displayed will be:

Table 4.3: Logical Operator OR

| | Values for DIV | Values for BUDGET | Condition1 DIV='PROD' | Condition2 BUDGET>=28000 | Yields | Row Result |
|---|-------------------|----------------------|--------------------------|-----------------------------|--------|---------------|
| 1 | PROD | 32000 | True | True | True | Retrieved |
| 2 | PROD | 27500 | True | False | True | Retrieved |
| 3 | CORP | 28000 | False | True | True | Retrieved |
| 4 | CORP | 10500 | False | False | False | Not retrieved |

| | | |
|--------------------------|------|--------|
| DNAME | DIV | BUDGET |
| Supplies and Procurement | CORP | 10500 |

4.3.2 Retrieval Using the OR Operator

When OR is used to connect two or more conditions, the compound condition is true if any condition is true, and the row is then retrieved. However, if all of the conditional expressions are false, then the row is not selected.

For example, suppose management is interested in any Product Division department OR any department with a budget of \$28,000 or greater. This compound condition can be written as follows:

```
DIV = 'PROD' OR BUDGET >= 28000
```

In this case OR is the logical operator used.

Table 4.3 illustrates the four possible cases that can occur with the logical operator OR for the example just given.

Example: Based on the above, let's develop a list of departments for management review, which are either in the Product Division or which have budgets of \$28,000 or greater.

If you enter the statement:

```
SELECT  DNAME, DIV, BUDGET
FROM    DEPARTMENT
WHERE   DIV = 'PROD' OR BUDGET >= 28000 ;
```

the result displayed will be:

| DNAME | DIV | BUDGET |
|-------------------------------------|------|--------|
| Corporate Legal Support | CORP | 28000 |
| Research and Development | PROD | 27500 |
| Manufacturing | PROD | 32000 |
| Customer Support and Service | PROD | 11000 |
| Product Marketing and Sales | PROD | 25000 |
| Strategic Planning and Intelligence | INFO | 28500 |

4.3.3 Retrieval Using Both AND and OR Operators

Compound conditions can include both AND and OR logical operators.

Example: If you enter the query:

```
SELECT  DNAME, DIV, BUDGET
FROM    DEPARTMENT
WHERE   DIV = 'CORP'   AND  BUDGET < 12000  OR  DIV = 'PROD' ;
```

the result displayed will be:

| DNAME | DIV | BUDGET |
|------------------------------|------|--------|
| Supplies and Procurement | CORP | 10500 |
| Research and Development | PROD | 27500 |
| Manufacturing | PROD | 32000 |
| Customer Support and Service | PROD | 11000 |
| Product Marketing and Sales | PROD | 25000 |

When you have a combination of AND and OR operators, the AND operators are evaluated first; then the OR operators are evaluated. Therefore, in the above query, rows from the DEPARTMENT table are retrieved if they satisfy at least one of the following conditions:

1. The department is in the Corporate Division with a budget under \$12,000.
2. The department is in the Product Division.

4.3.4 Retrieval Using Parentheses

Parentheses may be used within a compound condition to clarify or change the order in which the condition is evaluated. A condition within parentheses is evaluated before conditions outside the parentheses.

Example: Retrieve the department name, division name, and budget of all departments who have a budget of less than \$12,000, and who are either in the Corporate or the Product Division.

If you enter the query:

```

SELECT  DNAME, DIV, BUDGET
FROM    DEPARTMENT
WHERE   BUDGET < 12000
        AND   (DIV = 'CORP' OR DIV = 'PROD') ;

```

the result displayed will be:

| DNAME | DIV | BUDGET |
|------------------------------|------|--------|
| Supplies and Procurement | CORP | 10500 |
| Customer Support and Service | PROD | 11000 |

This query retrieves rows from the DEPARTMENT table that satisfy both of the following conditions:

1. The department has a budget of under \$12,000.
2. The department is in either the Corporate Division or the Product Division.

4.3.5 Logical Operator NOT

The logical operator NOT allows the user to express conditions that are best expressed in a negative way. In essence, it reverses the logical value of a condition on which it operates. That is, it accepts all rows except those that satisfy the condition. You write the conditional expression with the keyword NOT preceding the condition:

```
WHERE NOT condition
```

The condition can be a simple condition or a condition containing ANDs and ORs. The compound condition using NOT is true if the condition following NOT is false; and the compound condition is false if the condition following NOT is true.

For example, suppose you are looking for all departments who are not in the Corporate Division. You can write the conditional expression:

```
NOT (DIV = 'CORP')
```

Parentheses are optional but are included to improve readability of the condition.

If a department is in the Product Division, the program evaluates the condition in the following manner:

| Evaluation Process | Comments |
|-------------------------------|--|
| Step 1: NOT (DIV = 'CORP') | Original condition. |
| Step 2: NOT ('PROD' = 'CORP') | Substitute PROD for DIV. |
| Step 3: NOT (false) | Since PROD does not equal CORP, the condition DIV = 'CORP' is false. |
| Step 4: true | NOT changes false to true, the row is retrieved. |

NOT is typically used with logical operators such as IN, BETWEEN, LIKE, etc., which will be covered in a later section.

In the query condition NOT (DIV = 'CORP'), you are more likely to write the condition as follows:

```
WHERE DIV != 'CORP'
```

In this query the '!=' operator is used to show that DIV must not be equal to CORP.

Example: The NOT operator can be used with more than one expression. List all departments except those in the Corporate Division or those in the Product Division.

Enter the statement:

```
SELECT  DNAME, DIV
FROM    DEPARTMENT
WHERE   NOT (DIV = 'CORP' OR DIV = 'PROD') ;
```

Note that NOT precedes the entire condition.

The result displayed will be:

| DNAME | DIV |
|-------------------------------------|------|
| Information Systems Management | INFO |
| Corporate Library | INFO |
| Strategic Planning and Intelligence | INFO |

This statement retrieves the department and division name for all departments which are not Corporate or Product, revealing a division not yet retrieved in the previous searches, the Information Division.

4.4 Additional Comparison Operators

Texas has several special comparison operators for use with search conditions. These operators are indicated by the keywords BETWEEN, IN, SUBSET, INTERSECT, LIKE, LIKER, LIKEP and LIKE3, LIKEIN.

4.4.1 Range and Geographical Searches Using BETWEEN

The BETWEEN operator allows you to select rows of data in a given column if data in a given column contain values within a range. The general form of this operator is:

```
expression [NOT] BETWEEN lower value AND upper value
```

The condition is true if the expression is greater than or equal to the lower value and less than or equal to the upper value. If the NOT operator is used, the row is retrieved if the expression is less than the lower value or greater than the upper value.

Example: Let's find all departments whose budgets are between \$15,000 and \$25,000.

If you enter the statement:

```
SELECT  DNAME, BUDGET
FROM    DEPARTMENT
WHERE   BUDGET BETWEEN 15000 AND 25000 ;
```

the result displayed will be:

| DNAME | BUDGET |
|--------------------------------|--------|
| Product Marketing and Sales | 25000 |
| Corporate Library | 18500 |
| Information Systems Management | 22500 |

The name of each department whose budget is between \$15,000 and \$25,000 is retrieved. The limits include any budget of \$15,000 and of \$25,000; thus the Product Marketing and Sales Department with a budget matching the upper limit has been included.

The AND logical operator can also be used to form a query that selects values from a range. A query similar to the last example would look like the following.

If you enter the following statement:

```
SELECT  DNAME, BUDGET
FROM    DEPARTMENT
WHERE   BUDGET >= 15000 AND BUDGET <= 25000 ;
```

the result displayed will still be:

| DNAME | BUDGET |
|--------------------------------|--------|
| Product Marketing and Sales | 25000 |
| Corporate Library | 18500 |
| Information Systems Management | 22500 |

Notice that the results are identical to the output in example where BETWEEN was used in the WHERE clause.

The BETWEEN operator can be modified with the logical operator NOT so that rows outside a range will be selected.

Example: List the names of all departments who do not have a budget in the range of \$15,000 to \$25,000.

If you enter the statement:

```

SELECT  DNAME, BUDGET
FROM    DEPARTMENT
WHERE   BUDGET NOT BETWEEN 15000 AND 25000 ;

```

the result displayed will be:

| DNAME | BUDGET |
|-------------------------------------|--------|
| Corporate Legal Support | 28000 |
| Supplies and Procurement | 10500 |
| Customer Support and Service | 11000 |
| Manufacturing | 32000 |
| Research and Development | 27500 |
| Strategic Planning and Intelligence | 28500 |

This statement retrieves the names of all departments with budgets lower than \$15,000 or higher than \$25,000.

Geographical Searches with BETWEEN

A second form of BETWEEN is used for doing geographical searches. In this form the operator is used as:

```
location [NOT] BETWEEN (corner1, corner2)
```

(The parentheses are significant, and distinguish the special two-dimensional geographical form of BETWEEN from the normal one-dimensional range search.) The `location`, `corner1` and `corner2` values all represent single geographical (latitude/longitude) points – “geocode” values. This form of the BETWEEN operator will be true for all `location` points that are within (or on) the rectangular box defined by diagonally-opposite corners `corner1` and `corner2`.

The left-side `location` must be a `long` value. It is a geographically-encoded (“geocode”) value, returned from the SQL function `latlon2geocode()` or the Vortex function `<geo2code>`. Typically `location` is a `long` geocode column in a table representing the physical location of a row’s data.

The right-side `corner1` and `corner2` points define diagonally-opposite corners of the bounding box.¹ They are typically also `long` geocode values. However, in version 5.01.1194651000 20071109 and later, they may each be a single `varchar` (text) value containing a space- or comma-separated latitude/longitude pair, which will automatically be converted to geocode format. E.g.:

```
location BETWEEN ('40N 80W', '41N 81W')
```

In version 6.00.1298946000 20110228 and later, the bounding box may be computed inline from coordinates with `latlon2geocodebox()`; e.g. for a 0.5-degree “radius” bounding box centered on 40.5N, 80.5W:

¹Prior to version 5.01.1194489000 20071107, the box had to be specified as “lower-left”, “upper-right” (i.e. SW, NE) only. In that version (and later), other diagonal combinations (i.e. NW, SE) are supported.

```
location BETWEEN (select latlon2geocodebox(40.5, -80.5, 0.5))
```

When used in conjunction with a regular index on the `expression` column, the `BETWEEN` operator can greatly speed up geographical searches, as it reduces a two-dimensional `AND` search (with its potentially large merge or post-process) into a single-dimensional, all-index operation.

4.4.2 Set-like Searches Using `IN`, `SUBSET` and `INTERSECT`

The `IN`, `SUBSET` and `INTERSECT` operators can be used for set-like searches on multi-value type fields such as `strlst`. For example, to find rows where a query term is present in a `strlst` column, use `IN`. To find rows where a `strlst` column contains *any* of a list of query terms, use `INTERSECT` to find the set intersection of the row and the query set. To find rows where *all* query terms must be present in the row, use `SUBSET`.

Searches Using `IN`

The `IN` operator is used to select rows that match one of several listed values. In Taxis version 7 and later it behaves similar to the `SUBSET` operator (p. 37), i.e. it is true if all left-side value(s) are also present on the right-side. (See below for version 6 and earlier differences.)

The format of this operator is:

```
expression [NOT] IN (value1, value2, value3 ...)
```

Value1, value2, and so on indicates a list of values. Enclose the entire list in parentheses. Separate items in the list by commas.

Example: Let's list all departments in either the Corporate, Product, or Information divisions.

Enter the statement:

```
SELECT  DNAME, DIV
FROM    DEPARTMENT
WHERE   DIV IN ('CORP', 'PROD', 'INFO') ;
```

The row is retrieved if a department's division is in the set of divisions.

The result displayed will be:

| | |
|-------------------------------------|------|
| DNAME | DIV |
| Management and Administration | CORP |
| Finance and Accounting | CORP |
| Corporate Legal Support | CORP |
| Supplies and Procurement | CORP |
| Recruitment and Personnel | CORP |
| Research and Development | PROD |
| Manufacturing | PROD |
| Customer Support and Service | PROD |
| Product Marketing and Sales | PROD |
| Information Systems Management | INFO |
| Corporate Library | INFO |
| Strategic Planning and Intelligence | INFO |

A semantically equivalent (but usually less efficient) query can be formed using the logical operator `OR`. It looks like the following:

```
SELECT  DNAME, DIV
FROM    DEPARTMENT
WHERE   DIV = 'CORP'   OR  DIV = 'PROD'   OR  DIV = 'INFO' ;
```

The right-side of the `IN` operator may also be a `strlst` table column, in which case for each row, the left-side value is compared against each individual `strlst` item for that row. Parentheses are not needed in this case:

```
SELECT  UserName
FROM    Users
WHERE   'Administrator' IN GroupMembership;
```

In the above example, the `GroupMembership` column is of type `strlst`, and contains the list of groups that each user (row) is a member of. The query will thus return all `UserNames` that are members of the “Administrator” group.

The left-side of an `IN` operator may also be multi-value (e.g. a `strlst` parameter), in which case *all* the left-side values must be present on the right-side (if `inmode` is “subset”). The behavior of multi-value types other than `strlst` (on either side of `IN`) is currently undefined and thus such types should not be used.

The `IN` operator can be modified with the logical operator `NOT` (note however that an index cannot be used to optimize such a query).

Example: List all departments which are not in either the Corporate or the Information divisions.

Enter the statement:

```
SELECT  DNAME, DIV
FROM    DEPARTMENT
WHERE   DIV NOT IN ('CORP', 'INFO') ;
```


The result displayed will be:

| | |
|------------------------------|------|
| DNAME | DIV |
| Research and Development | PROD |
| Manufacturing | PROD |
| Customer Support and Service | PROD |
| Product Marketing and Sales | PROD |

Note that `IN` differs from `SUBSET` and `INTERSECT` in the interpretation of empty `varchar` values: for `IN` they are single-item empty-string sets. See p. 39 for details, as well as for other behaviors that `IN`, `SUBSET` and `INTERSECT` share in common.

IN with Version 6 Or Earlier In Taxis version 6 (or `compatibilityversion` 6) and earlier, `IN` behaved much like the `INTERSECT` operator (p. 38) instead of `SUBSET`, i.e. it was true if *any* left-side value was present on the right-side. This behavior can be restored with the `inmode SQL` property, p. 186 (or the `compatibilityversion` property, p. 170). Note however that with a single non-empty left-side value, there is no difference, as intersection and subset then behave the same.

Additionally, `IN` with version 6 and earlier did not always utilize indexes (e.g. if the table column was on the right-side), and had other quirks.

Searches Using `SUBSET`

The `SUBSET` operator allows subset queries, and is typically used with multi-value (i.e. `strlst`) fields that are treated as sets. It is true if the left-side is a subset of the right-side, i.e. if there are no values on the left-side that are missing from the right-side. Duplicates count, i.e. they must match one-to-one from left side to right.

For example, suppose the table `Users` contains one row per user (`UserName`), and has a `strlst` column `GroupMembership` that lists all the groups that row's user is a member of. To find all users that are members of groups "Management", "Sales" and "Marketing", a `SUBSET` query can be used:

```
SELECT UserName
FROM   Users
WHERE  ('Management', 'Sales', 'Marketing')
      IS SUBSET OF GroupMembership;
```

(Syntactically, `SUBSET` is always used as part of the phrase `IS SUBSET OF`, as it is only valid in `WHERE` clauses.) The above query will return the users that are members of all three groups – including any users that may also be members of additional groups.

Note that `SUBSET` is not commutative, i.e. if the left- and right-sides are reversed, the meaning is changed (unlike e.g. `INTERSECT`). If A is a subset of B, then B is *not* necessarily a subset of A; B is a subset of A if and only if both sets contain the same values. E.g. this query:

```

SELECT UserName
FROM   Users
WHERE  GroupMembership
      IS SUBSET OF ('Management', 'Sales', 'Marketing');

```

while merely the reversed version of the earlier query, behaves differently: it would list the users whose are in zero or more of the Management, Sales or Marketing groups – *and* are not in any other groups.

In set logic the empty set is a subset of any set; thus if there are *no* values on the left-side, SUBSET is true no matter what the right-side value(s) are. Note that SUBSET interprets an empty varchar value as empty-set, not single-item empty-string set set (as IN does). See p. 39 for details, as well as additional behaviors that IN, SUBSET and INTERSECT share in common. SUBSET was added in Taxis version 7.

Index Usage by SUBSET A SUBSET query can often utilize a regular (B-tree) index to increase performance. Generally the index should be created with `indexvalues` set to `splitstrlst` (the default), as this enables individual values of `strlsts` to be accessed as needed. There are some limitations and caveats for SUBSET and indexes however:

- **Empty parameter, `strlst` column (either side):**

Queries with empty-set parameters (i.e. zero-item `strlst`, or empty varchar) and a `strlst` column cannot use an `indexvalues=splitstrlst` index, regardless of which side of SUBSET the parameter and column are on. An index with `indexvalues=all` can be used however. It may be created in addition to the normal `indexvalues=splitstrlst` index, and the Taxis optimizer will choose the appropriate one at search time.

- **Empty `strlst` column left-side, non-empty parameter right-side:**

With a `strlst` column on the left-side, and a non-empty parameter on the right, empty rows will not be returned if an index is used – even though they properly match (as empty set is a subset of any set).

These caveats are due to limitations in `indexvalues=strlst` indexes; see p. 175 for more information.

Searches Using INTERSECT

The INTERSECT operator allows set-intersection queries, typically on multi-value (i.e. `strlst`) values. It returns the intersection of the left and right sides, i.e. the “set” (`strlst`) of all values that are present on both sides. Duplicates are significant, i.e. they must match one-to-one to be included in the intersection.

For example, suppose the table `Users` contains one row per user (`UserName`), and has a `strlst` column `GroupMembership` that lists all the groups that row’s user is a member of. To find all users that are members of groups “Management”, “Sales” or “Marketing”, an INTERSECT query can be used:

```

SELECT UserName
FROM   Users
WHERE  GroupMembership INTERSECT
      ('Management', 'Sales', 'Marketing') IS NOT EMPTY;

```

This will return users where the intersection of a user's `GroupMembership` with the three named groups is not empty (i.e. contains at least one value). Thus, users that are members of any of the three named groups are returned. The phrase `IS NOT EMPTY` must be added immediately after, both to turn the expression into a true/false condition suitable for a `WHERE` clause, and to allow an index to be used to resolve the query. (The phrase `IS EMPTY` is also permitted, for negation. However indexes cannot be used to resolve such queries.)

`INTERSECT` may also be used in a `SELECT` clause, to return the actual intersection set itself, rather than be used as a true/false condition. For example, given the same `Users` table above, to find each user's membership amongst just the three named groups, this query may be used:

```
SELECT UserName, GroupMembership INTERSECT
      ('Management', 'Sales', 'Marketing') AS SubMembership
FROM   Users;
```

This will return the membership of each user (`SubMembership`) in just the three named groups, as a `strlst`. If a user is not a member of any of the three groups, `SubMembership` will be empty. If a user is a member of some other group(s), they will not be named in `SubMembership`.

Note that unlike `SUBSET`, `INTERSECT` is commutative, i.e. reversing the left- and right-sides does not change its meaning. (The “=” equals operator is also commutative, for example: `x = y` has the same meaning as `y = x`.) Also note that `INTERSECT` interprets an empty `varchar` value as empty-set, not single-item empty-string set (as `IN` does). See p. 39 for details, as well as additional behaviors that `IN`, `SUBSET` and `INTERSECT` share in common. `INTERSECT` was added in Taxis version 7.

Index Usage by `INTERSECT` An `INTERSECT` query can utilize a regular (B-tree) index to increase performance. The index should be created with `indexvalues` set to `splitstrlst` (the default), as this enables individual values of `strlsts` to be accessed as needed.

`IN`, `SUBSET`, `INTERSECT` Commonality

The `IN`, `SUBSET` and `INTERSECT` operators, being set-like, share certain behaviors in common in Taxis version 7 (or `compatibilityversion 7`) and later:

A `varchar` value on either side of these operators is treated as a single-item `strlst` set – regardless of the current `varchartostrlstsep` setting. This aids usage of `IN`/`SUBSET`/`INTERSECT` in Vortex when `arrayconvert` is active for parameters: it provides consistent results whether the Vortex variable is single- or multi-value. A single `varchar` value will not be unexpectedly (and incorrectly) split into multiple values using its last character as a separator.

However, the operators differ on interpretation of *empty* `varchar` values. With `IN`, an empty `varchar` value is considered a single-item empty-string set, because `IN` is most often used with single-value (i.e. non-set-like) parameters. This makes the clause “`WHERE varcharColumn IN ('red', 'green', 'blue')`” only return “red”, “green” or “blue” `varcharColumn` values – not empty-string values too, as `SUBSET` would. This empty-string interpretation difference is the one way in which `IN` differs from `SUBSET` (and `INTERSECT`, if `inmode` is `intersect`).

With `SUBSET/INTERSECT` however, an empty `varchar` value is considered an empty set, because `SUBSET/INTERSECT` are more clearly set-like operators where both operands are sets, and an empty string is more likely to be intended to mean “empty set”. This is also more consistent with `convert()` and `INSERT` behavior: an empty string converted or inserted into a `strlst` value becomes an empty `strlst`, not a one-item (empty-string) `strlst`.

The current (or indexed) `stringcomparemode` setting value is used during `IN/SUBSET/INTERSECT` operations; thus case-insensitive comparisons can be accomplished by modifying the setting. At search time, the Taxis optimizer will choose the index whose `stringcomparemode` setting is closest to the current value.

Caveat: `IN/SUBSET/INTERSECT` behavior with multi-value types other than `strlst` is currently undefined and should be avoided. Single-value types other than `varchar` have limited support currently; it is recommended that only `varchar` (and `strlst`) types be used.

See also p. 37 for version 6 and earlier `IN` issues.

4.4.3 Search Condition Using `LIKE`

In most SQL applications, a column value that contains character values can be matched to a pattern of characters for the purpose of retrieving one or more rows from a table. This is often referred to as *pattern matching*. Pattern matching is useful when a user cannot be specific about the data to be retrieved. For instance:

- You’re not sure if someone’s last name is Robinson, Robertson, or Robbins. You search using the pattern “Rob”.
- You want a list of all employees who live on Newman Avenue, Road or Street. You search using the pattern “Newman”.
- You want a list of all employees whose name ends in “man”, such as Waterman, Spellman, or Herman. You search using the pattern “man”.

The `LIKE` operator is used in the `WHERE` clause to enable you to retrieve records that have a partial match with a column value. The `LIKE` operator has the following format:

```
WHERE column-name LIKE 'pattern'
```

In Taxis the capabilities of the `LIKE` clause have been exponentially increased through implementation of all features of the Metamorph search engine. Rather than the limited single item string search allowed in traditional SQL applications, Taxis allows any valid Metamorph query to be substituted for the `'pattern'` following `LIKE`.

Therefore, in addition to traditional string searches, text fields can be searched with all of Metamorph’s pattern matchers to find concepts, phrases, variable expressions, approximations, and numeric quantities expressed as text. These queries can contain multiple search items combining calls to different Metamorph pattern matchers. Intersections of such items can be located in proximity to one another within defined text units such as sentences, paragraphs, or the whole record.

It is this integration of Metamorph through the `LIKE` clause which brings together intelligent full text searching with relational database technology. For instance, within the confines of the Taxis relational database, you can also issue queries to find the following:

- All Research and Development reports covering conceptually similar research done on a field of interest. For example, a request for all research done concerning “red lenses” could discover a report about “rose colored glasses”.
- All strategic information reports concerning marketing campaigns over a certain dollar amount. For example, such a request for marketing information about wheels could reveal a “sales” campaign where “twenty-five thousand dollars” was allocated to promote “tires”.
- An employee whose name sounds like Shuler who helps fix computer problems. For example, a query for approximately Shuler and computers could find Elaine “Schuller” who works in “data processing”. And since you are querying a relational database, you could also pull up her phone extension and call for help.

Full use of the Metamorph query language is discussed in depth in Chapter 7. In this section we will concentrate on simple examples to illustrate how the `LIKE` clause can be used to further qualify `WHERE`.

Command Discussion

- The column name following the `WHERE` clause must contain character values; otherwise, the `LIKE` operator cannot be used.
- The `LIKE` operator compares the value in the specified column with the pattern, as inserted in single quotes following `LIKE`. A row is retrieved if a match occurs.
- You can put any Metamorph query in quotes (`' query '`) in place of a fixed length string, although you would need to escape a literal `'` with another `'` by typing `' '`, if you want the character `'` to be part of the query.
- The “pattern” inside single quotes following `LIKE` will be interpreted exactly as Metamorph would interpret such a query on its query line, in any Metamorph application (with the only exception being that a single quote or apostrophe must be escaped with another `'` to be interpreted literally).
- Concept searching is off by default for Metamorph queries following `LIKE`, but can be selectively invoked on a word using the tilde `'~'`.
- Syntax for complete use of Metamorph query language is covered in Chapter 7.
- Queries using `LIKE` can make use of any indexing which has been done. An alternate form of `LIKE` may also be used called `LIKE3`, which uses indexing exclusively with no post search. See Chapter 7 for a thorough explanation of all types of text searches possible with `LIKE` and `LIKE3`, and their relation to indexed information.

Example: Let’s start with a simple example. You wish to retrieve all reports where “ink” is part of the title, without knowing the full title.

If you enter the statement:

```

SELECT  TITLE
FROM    REPORT
WHERE   TITLE LIKE 'ink' ;

```

the result displayed will be:

| TITLE |
|---------------------------------------|
| Innovations in Disappearing Ink |
| Disappearing Ink Promotional Campaign |
| Ink Color Panorama |

In this query, you are retrieving the titles of all reports whose title is “like” the pattern “ink”.

In other cases you may not know the exact words you are looking for. A simple example where a wildcard ‘*’ is used follows.

```

SELECT  AUTHOR, DEPT
FROM    REPORT
WHERE   AUTHOR LIKE 'san*' ;

```

The result will be:

| AUTHOR | DEPT |
|--------------------|------|
| Sanchez, Carla | MKT |
| Sanders, George G. | FIN |
| Claus, Santa | MKT |

4.4.4 Relevance Ranking Using LIKER and LIKEP

In addition to the Metamorph searches listed above there is another type of search based on Metamorph. This will return rows in order of relevance, with the most relevant record first (unless other clauses alter this order, e.g. an ORDER BY). LIKER calculates a relevance based solely on the presence or absence of the terms in the document. LIKEP uses this same information, but also uses the proximity of the terms to calculate relevance.

There are several restrictions and points to note about LIKER and LIKEP. The conditions that must be met to obtain a relevancy search are that a Metamorph index exists on the field in question. LIKER can only work with an index; while LIKEP can work without such an index, it performs best with one. The other condition is that the query should consist of word terms only. None of the other pattern matchers are available with LIKER; they are available with LIKEP, but at a cost in performance (post-processing is required).

The query is a list of terms to be searched for. The words are weighted by their uniqueness in the document set being searched. This means that infrequent words are weighted more than common words.

The weight that was calculated for the record is available by selecting the generated field `$rank`, which will contain the rank value. The rank value for `LIKER` is unscaled. With `LIKEP` the number will range between 0 and 1000, where greater values indicate greater computed relevance to the query.

The default ordering of `LIKER` and `LIKEP` (rank-descending) may be changed by an `ORDER BY` clause. Historically (prior to Taxis version 8), an `ORDER BY` containing `$rank` (or potentially any expression containing `$rank`) would usually order descending as well – despite the typical default `ORDER BY` order being *ascending* – because rank-descending is considered more useful (and often low-rank results are eliminated prior to ordering anyway). However, this caused confusion when giving the `DESC` flag, as then `ORDER BY $rank DESC` would return *ascending* results.

Thus, in Taxis version 8 and later, `ORDER BY` clauses containing `$rank` will order consistently with other `ORDER BY` clauses – i.e. numerically ascending unless the `DESC` flag is given. This means that in version 8 and later, most `ORDER BY $rank` clauses should probably be `ORDER BY $rank DESC`, to get rank-descending behavior. This behavior may be altered by the `legacyversion7orderbyrank` SQL setting (p. 171).

For fuller details on using the logic operators in `LIKE` see Chapter 7

Command Discussion

Result ranking is a useful feature, although due to the variety of cases where you might want to use ranking, there are a number of variables that control the ranking algorithm.

The first major choice will be whether proximity is important. This will indicate if you want to use `LIKER` or `LIKEP`. `LIKER` uses the index to determine the frequencies of the terms, and the presence or absence of the terms in each document to determine the rank for each document. Each term is assigned a weight between 0 and 1000, and the rank value for the document is the sum of the weights for all the terms that occur.

`LIKER` has a threshold value, such that documents with a lower rank value than the threshold value will not be returned. This prevents a large number of irrelevant documents from being returned. Initially the threshold is set to the weight of the term with the highest weight. If there are more than five terms then the threshold is doubled, and if there are more than 10 terms the threshold is doubled again. This keeps queries containing a lot of terms from returning irrelevant hits. It is possible to force the threshold lower if desired to return more records. This can be performed either by specifying the maximum number of records a term should occur in, and still be returned by `LIKER`. This is the `likerrows` variable. For example, in a three term query, where the terms occur in 400, 900 and 1400 records respectively, setting `likerrows` to 1000 would allow records containing only the second search term to be returned.

In general `LIKEP` will perform the same initial step as `LIKER` to determine which documents to rank. `LIKEP` then looks at the `likeprows` highest ranked documents from `LIKER`, and recalculates the rank by actually looking inside the document to see where the matching terms occur. Because of this it will be slower than `LIKER`, although if you are using a Metamorph inverted index the ranks may still be determinable from the index alone, saving actual table accesses.

There are a number of variables that can be set with `LIKEP`, which affect both how documents are ranked, as well as how many documents are returned. See the “Rank knobs” (p. 173) and “Other ranking properties” (p. 173) discussions in the Server Properties section of the manual.

4.4.5 Query searching using `LIKEIN`

`LIKEIN` is used for doing profiling, where you have a lot of queries and you want to find which queries match the given text. This is typically used when the number of queries is large and relatively constant, and there is a stream of new texts to match. `LIKEIN` will find the queries that would match the text. To work efficiently you should have a `METAMORPH COUNTER` index created on the field containing the queries.

4.4.6 Search Condition Using `MATCHES`

The `MATCHES` keyword allows you to match fields against expressions. This is most useful when you have fields with a small amount of text and do not need the full power of Metamorph. Typical uses would be names, part numbers or addresses.

In the query an underscore will match any single character, and a percent sign will match any number of characters. For example

```
SELECT  AUTHOR, DEPT
FROM    REPORT
WHERE   AUTHOR MATCHES 'San%' ;
```

The result will be:

| AUTHOR | DEPT |
|--------------------|------|
| Sanchez, Carla | MKT |
| Sanders, George G. | FIN |

The special characters used with `MATCHES` can be changed using the `set matchmode` SQL statement. The default value of 0 produces the behavior documented above which is standard in SQL. Setting `MATCHMODE` to 1 will change the special characters such that asterisk will match any number of characters, and a question mark will match any single character, which is more familiar to many people.

Comparing the results to the earlier example using `LIKE` you will see that Claus, Santa does not match, as the match has to occur at the beginning of the field.

`MATCHES` can make use of a regular index on the field. It will not use a Metamorph index.

4.5 Sorting Your Results

The output from the above queries may not be in the desired order. For example, you may want the list of departments arranged alphabetically. Sorting is the process of rearranging data into some specific order. To sort the output into a desired sequence, a field or fields are specified that determine the order in which the results are arranged. These fields are called *sort keys*.

For example, if the department data is sorted into alphabetical order by department, the department name is the sort key. The budget field is the sort key if the department table is sorted by amount of budget. Note that the sort key can be numeric (budget) or character (department name).

Results can be sorted into ascending or descending sequence by sort key. Ascending means increasing order, and descending means decreasing order. For example, sorting the department table in ascending order by budget means the department data will be arranged so that the department with the lowest budget is first and the department with the highest budget is last. If we instead sorted in descending order, the department with the highest budget would appear first, the department with the lowest budget would appear last.

Sorting character data in ascending or descending order is based on a coding, or collating, sequence assigned to numbers and letters by the computer. For example, when department name is the sort key and you want the data arranged alphabetically, that indicates ascending order. If you want the data arranged in reverse alphabetical order, then specify descending order.

To sort your results using Taxis, add the `ORDER BY` clause to the `SELECT` statement. The form of this clause is:

```
ORDER BY column-name [DESC]
```

where `DESC` indicates the rows are to be arranged in descending order. If `DESC` is omitted, your output is sorted in ascending order.

This clause fits into the `SELECT` expression following the `WHERE` clause, as shown below:

```
SELECT      column-name1 [,column-name2] ...
FROM        table-name
[WHERE      search-condition]
[ORDER BY   column-name [DESC] ] ;
```

Example: Retrieve a list of departments arranged by division, and within that division, arranged by highest budget first.

If you enter the statement:

```
SELECT      DNAME, DIV, BUDGET
FROM        DEPARTMENT
ORDER BY    DIV, BUDGET DESC ;
```

Output will appear in ascending order automatically if `DESC` is omitted.

The result displayed will be:

| DNAME | DIV | BUDGET |
|-------------------------------------|------|--------|
| Corporate Legal Support | CORP | 28000 |
| Finance and Accounting | CORP | 26000 |
| Management and Administration | CORP | 22000 |
| Recruitment and Personnel | CORP | 15000 |
| Supplies and Procurement | CORP | 10500 |
| Strategic Planning and Intelligence | INFO | 28500 |
| Information Systems Management | INFO | 22500 |
| Corporate Library | INFO | 18500 |
| Manufacturing | PROD | 32000 |
| Research and Development | PROD | 27500 |
| Product Marketing and Sales | PROD | 25000 |
| Customer Support and Service | PROD | 11000 |

Notice that all departments in the same division are listed together, with the divisions listed in ascending order, as the default ordering for DIV. Within each division, the department with the highest budget is listed first, since descending order was specified for BUDGET.

It is possible to have as many as 50 sort keys. The order in which the sort keys are listed is the order in which the data will be arranged.

This chapter has introduced several ways to retrieve rows and columns from a table. In the next chapter, you will learn how to perform calculations on data stored in a table.

Chapter 5

Queries Involving Calculated Values

While Taxis focuses on manipulation of textual information, data can also be operated on numerically. Queries can be constructed which combine calculated values with text search.

To illustrate the material in this chapter, we'll use an employee table which the Personnel Department keeps to manage salaries and benefits. A sampling of the data stored in this table follows:

| EID | ENAME | DEPT | SALARY | BENEFITS |
|-----|-------------------|------|--------|----------|
| 101 | Aster, John A. | MKT | 32000 | FULL |
| 102 | Barrington, Kyle | MGT | 45000 | FULL |
| 103 | Chapman, Margaret | LIB | 22000 | PART |
| 104 | Jackson, Herbert | RND | 30000 | FULL |
| 105 | Price, Stella | FIN | 42000 | FULL |
| 106 | Sanchez, Carla | MKT | 35000 | FULL |
| 107 | Smith, Roberta | RND | 25000 | PART |

5.1 Arithmetic Calculations

This section covers the computational features of Taxis. They are adequate to allow the user to perform computations on the data and/or retrieve rows based on conditions involving computations. For example, you can adjust salaries for a 5 percent across-the-board increase, or you can compute weekly salaries (i.e., salary divided by 52).

Arithmetic calculations are performed on fields, or columns, in the database. An *arithmetic expression* is used to describe the desired computation. The expression consists of column names and numeric constants connected by parentheses and arithmetic operators. Table 5.1 shows the arithmetic operators used in Taxis.

Typically, the arithmetic expression is used in the `SELECT` clause to perform calculations on data stored in the table.

Example: Next year every employee will receive a 5 percent salary increase. List the names of each employee, his or her current salary, and next year's salary.

Table 5.1: Arithmetic Operators Supported in Taxis

| Arithmetic Operation | Taxis Operator | Example |
|----------------------|----------------|---------------|
| Addition | + | SALARY + 2000 |
| Subtraction | - | SALARY - 1000 |
| Multiplication | * | SALARY * 1.05 |
| Division | / | SALARY / 26 |

Enter this statement:

```
SELECT  ENAME, SALARY, SALARY * 1.05
FROM    EMPLOYEE ;
```

Where “SALARY * 1.05” is the arithmetic expression.

The results are:

| ENAME | SALARY | SALARY * 1.05 |
|-------------------|--------|---------------|
| Aster, John A. | 32000 | 33600 |
| Barrington, Kyle | 45000 | 47250 |
| Chapman, Margaret | 22000 | 23100 |
| Jackson, Herbert | 30000 | 31500 |
| Price, Stella | 42000 | 44100 |
| Sanchez, Carla | 35000 | 36750 |
| Smith, Roberta | 25000 | 26250 |

The expression “SALARY * 1.05” results in each value in the salary column being multiplied by 1.05. The results are then displayed in a new column that is labeled SALARY * 1.05.

If more than one arithmetic operator is used in an arithmetic expression, parentheses can be used to control the order in which the arithmetic calculations are performed. The operations enclosed in parentheses are computed before operations that are not enclosed in parentheses. For example, the expression:

```
12 * (SALARY + BONUS)
```

means bonus is added to salary, and then this result is multiplied by 12.

If parentheses are omitted or if several operations are included within the parentheses, the order in which calculations are performed is as follows:

1. First, all multiplication, division and modulo¹ operations are performed.
2. Then, all addition and subtraction operations are performed.

¹The modulo operator (%) was added in Taxis version 8; it is supported for integral types.

For example, in the expression:

```
SALARY + SALARY * .05
```

the value in the SALARY column is multiplied by .05, and then the salary value is added to this intermediate result.

When two or more computations in an expression are at the same level (e.g., multiplication and division), the operations are executed from left to right. For example, in the expression:

```
SALARY / 12 * 1.05
```

the salary value is first divided by 12, and then this result is multiplied by 1.05.

Arithmetic calculation can also be used in a WHERE clause to select rows based on a calculated condition. In addition, arithmetic expressions can be used in the HAVING and ORDER BY clauses, which will be discussed in later sections of this chapter.

Example: List the names of all employees earning a monthly salary above \$3000.

This query:

```
SELECT  ENAME
FROM    EMPLOYEE
WHERE   (SALARY/12) > 3000 ;
```

results in:

```
ENAME
Barrington, Kyle
Price, Stella
```

The rows in the EMPLOYEE table are retrieved if the condition “salary divided by 12” is greater than \$3000. This was true only for Barrington and for Price, whose annual salaries (respectively \$45,000 and \$42,000) are greater than \$3000 when divided by 12 months.

5.2 Manipulating Information By Date

In Taxis dates are stored as integers representing an absolute number of seconds from January 1, 1970, Greenwich Mean Time. This is done for efficiency, and to avoid confusions stemming from differences in relative times assigned to files from different time zones. The allowable range of years is 1970 through 2037. Years between 1902 and 1970 may be stored and compared for equality (=) but will not compare correctly using less than (<) and greater than (>).

Counters may also be treated as dates for comparison purposes. They may be compared to date fields or date strings. When compared with dates only the date portion of the counter is considered and the sequence number is ignored.

The comparison operators as given in Table 4.1 are used to compare date values, so that dates may be used as qualifying statements in the `WHERE` clause.

Example: The Strategic Planning and Intelligence Department is responsible for polling online news information on a daily basis, looking for information relevant to Acme's ongoing business. Articles of interest are stored in an archived `NEWS` table which retains the full text of the article along with its subject, byline, source, and date. The date column is named `NDATE`, for "News Date", as "date" is a special reserved SQL name and can't be used for column names.

A Date field may be compared to a number representing the number of seconds since 1/1/70 0:0:0 GMT (e.g.: 778876248). It may also be compared to a string representing a human readable date in the format 'YYYY-MM-DD [HH:MM[:SS] [AM|PM]]' (e.g.: '1994-03-05 06:30 pm' or '1994-07-04'). The date string may also be preceded by "begin of" or "end of" meaning the first or last second of a day, respectively.

Enter this query:

```
SELECT    NDATE, SUBJECT
FROM      NEWS
WHERE     NDATE BETWEEN 'begin of 1993-07-30'
          AND 'end of 1993-07-30' ;
```

Although the date column is stored with an absolute value, it is converted to the correct relative value when displayed. However, a date assigned to a file is to the second, and to match that time, you must match the same number of seconds. Stating the date as 1993-07-30 refers to a particular second of that day. An article which came in at 2 p.m. would not match in seconds. Thus you state the range of seconds that span the 24 hour period called "'1993-07-30'" by specifying a range between the first to last moment of the day.

In this example, all the articles which were saved from July 30, 1993 are displayed with their subject lines. The date as formatted by Taxis when displaying the date column is the format used inside the single quotes. It is put in quotes because it is a text string rather than an absolute value.

Dates are usually used to limit the amount of text retrieved based on some other search requirement, and would be so used along with other qualifying statements in the `WHERE` clause. The next query is identical to the last, but it adds another requirement.

```
SELECT    NDATE, SUBJECT
FROM      NEWS
WHERE     NDATE BETWEEN 'begin of 1993-07-30'
          AND 'end of 1993-07-30'
AND       BODY LIKE 'bill gates' ;
```

Now we can retrieve articles from July 30, 1993, but only a list of those articles whose text body mentions Bill Gates. A listing of Date and Subject of the article will be displayed, as dictated in `SELECT`. Now we know which articles are available and can pick any we would want to read in full.

This example uses a text query to find sentences in the body of the information with reference to "Bill

Gates”. Use of this type of query in the `LIKE` clause is explained in Chapter 7. The following articles are retrieved:

| NDATE | SUBJECT |
|---------------------|---|
| 1993-30-07 04:46:04 | High-Technology R&D Has Lost Its Cost-Effect... |
| 1993-30-07 13:10:08 | Heavy R&D Spending No Longer the Magic Route... |

Date fields can use any of the comparison operators as shown in Table 4.1 to manipulate information. We could broaden the date range of this search by increasing the `BETWEEN` range, or we could do it as follows:

```
SELECT  NDATE, SUBJECT
FROM    NEWS
WHERE   BODY LIKE 'bill gates'
AND     NDATE > 'begin of 1993-07-30'
AND     NDATE < 'end of 1993-08-01' ;
```

Remember that the actual value of the date is in a number of seconds. Therefore, greater than (`>`) translates to “a greater number of seconds than the stated value”, and therefore means “newer than”, while lesser than (`<`) translates to “a fewer number of seconds than the stated value”, and therefore means “older than”.

This would increase the output list to include dates in the specified range; that is, between July 30th and August 1st 1993.

| NDATE | SUBJECT |
|---------------------|--|
| 1993-07-30 04:46:04 | High-Technology R&D Has Lost Its Cost-Effect... |
| 1993-07-30 13:10:08 | Heavy R&D Spending No Longer the Magic Route... |
| 1993-07-31 07:56:44 | Microsoft-Novell battle out in the open |
| 1993-07-31 16:40:28 | Microsoft to Undergo Justice Department Scrutiny |
| 1993-08-01 09:50:24 | Justice Dept. Reportedly to Study Complaints ... |

Date strings have some additional operators, “today” and “now”. When used following `DATE` they are converted to today’s date and time in seconds for both “today” and “now”. A time period of seconds, minutes, hours, days, weeks, or months, can also be specified. A leading plus (+) or minus (−) may also be specified to indicate past or future. Using our example from the `NEWS` table, the form of the command would be:

```
SELECT  NDATE, SUBJECT
FROM    NEWS
WHERE   NDATE > '-7 days' ;
```

This query requests all articles less than seven days old and would produce a list of their subjects and date.

```
SELECT  NDATE, SUBJECT
FROM    NEWS
WHERE   NDATE < '-1 minute'
AND     NDATE > '-1 hour' ;
```

This query would produce a list of articles which came in over the last hour. The date must be older than 1 minute ago, but newer than 1 hour ago.

5.3 Summarizing Values: GROUP BY Clause and Aggregate Functions

So far, the examples presented have shown how to retrieve and manipulate values from individual rows in a table. In this section, we will illustrate how summary information can be obtained from groups of rows in a table.

Often we find it useful to group data by some characteristic of the group, such as department or division, or benefit level, so that summary statistics about the group (totals, averages, etc.) can be calculated. For example, to calculate average departmental salaries, the user could group the salaries of all employees by department. In Taxis, the GROUP BY clause is used to divide the rows of a table into groups that have matching values in one or more columns. The form of this clause is:

```
GROUP BY    column-name1 [,column-name2] ...
```

and it fits into the SELECT expression in the following manner.

```
SELECT      column-name1 [,column-name2] ...
FROM        table-name
[WHERE      search-condition]
[GROUP BY   column-name1 [,column-name2] ... ]
[ORDER BY   column-name1 [DESC] [,column-name2] [DESC] ] ... ;
```

The column(s) listed in the GROUP BY clause are used to form groups. The grouping is based on rows with the same value in the specified column or columns being placed in the same group. It is important to note that grouping is conceptual; the table is not physically rearranged.

As an extension Taxis also allows the GROUP BY clause to consist of expressions instead of just column names. This should be used with caution, and the same expression should be used in the SELECT as in the GROUP BY clause. This is especially true if the expression will fold multiple values together, such as dividing a number by 1000 to group quantities together if they are in the same 1000. If you select SALARY, and GROUP BY SALARY/1000 you will see one sample salary from the matching group.

The GROUP BY clause is normally used along with five built-in, or “aggregate” functions. These functions perform special operations on an entire table or on a set, or group, of rows rather than on each row and then return one row of values for each group.

Table 5.2 lists the aggregate functions available with Taxis.

Aggregate functions are used in place of column names in the SELECT statement. The form of the function is:

```
Function name ([DISTINCT] argument)
```

In all situations the argument represents the column name to which the function applies. For example, if the sum of all salaries is needed, then the function SUM is used and the argument is the column SALARY.

Table 5.2: Taxis Aggregate Function Names

| Function Name | Meaning | Example |
|------------------|---|--------------|
| SUM(column name) | Total of the values in a numeric column | SUM (SALARY) |
| AVG(column name) | Average of the values in a column | AVG (SALARY) |
| MAX(column name) | Largest value in a column | MAX (SALARY) |
| MIN(column name) | Smallest value in a column | MIN (SALARY) |
| COUNT(*) | Count of the number of rows selected | COUNT (*) |

When COUNT is used an asterisk (*) can be placed within the parentheses instead of a column name to count all the rows without regard to field.

If the DISTINCT keyword is used then only the unique values are processed. This is most useful with COUNT to find the number of unique values. If you use DISTINCT then you must supply a column name. DISTINCT will work with the other aggregate functions, although there is typically very little need for them. The DISTINCT feature was added in version 4.00.1002000000

Example: What is the average salary paid in each department?

Enter this statement:

```
SELECT      DEPT, AVG (SALARY)
FROM        EMPLOYEE
GROUP BY    DEPT ;
```

Syntax Notes:

- AVG is the aggregate function name.
- (SALARY) is the column on which the average is computed.
- DEPT is the column by which the rows will be grouped.

The above statement will produce the following results:

```
DEPT      AVG (SALARY)

MKT        33500
MGT        45000
LIB        22000
RND        27500
FIN        42000
```

In this query, all rows in the EMPLOYEE table that have the same department codes are grouped together. The aggregate function AVG is calculated for the salary column in each group. The department code and the average departmental salary are displayed for each department.

A `SELECT` clause that contains an aggregate function cannot contain any column name that does not apply to a group; for example:

The statement:

```
SELECT      ENAME, AVG (SALARY)
FROM        EMPLOYEE
GROUP BY    DEPT ;
```

results in the message

```
Error at Line 1: Not a GROUP BY Expression
```

It is not permissible to include column names in a `SELECT` clause that are not referenced in the `GROUP BY` clause. The only column names that can be displayed, along with aggregate functions, must be listed in the `GROUP BY` clause. Since `ENAME` is not included in the `GROUP BY` clause, an error message results.

Example: The chair of the Marketing Department plans to participate in a national salary survey for employees in Marketing Departments. Determine the average salary paid to the Marketing Department employees.

This statement:

```
SELECT      COUNT (*), AVG (SALARY)
FROM        EMPLOYEE
WHERE       DEPT = 'MKT'
```

Results in:

| COUNT (*) | AVG (SALARY) |
|-----------|--------------|
| 2 | 33500 |

In this example, the aggregate function `AVG` is used in a `SELECT` statement that has a `WHERE` clause. Taxis selects the rows that represent Marketing Department employees and then applies the aggregate function to these rows.

You can divide the rows of a table into groups based on values in more than one column. For example, you might want to compute total salary by department and then, within a department, want subtotals by benefits classification.

Example: What is the total salary paid by benefits classification in each department?

Enter this statement:

```
SELECT      DEPT, BENEFITS, SUM (SALARY)
FROM        EMPLOYEE
GROUP BY    DEPT, BENEFITS ;
```

In this example, we are grouping by department, and within department, by benefits classification.

We'll get the following results:

| DEPT | BENEFITS | SUM(SALARY) |
|------|----------|-------------|
| FIN | FULL | 42000 |
| LIB | PART | 22000 |
| MGT | FULL | 45000 |
| MKT | FULL | 67000 |
| RND | FULL | 30000 |
| RND | PART | 25000 |

In this query, the rows are grouped by department and, within each department, employees with the same benefits are grouped so that totals can be computed. Notice that the columns DEPT and BENEFITS can appear in the SELECT statement since both columns appear in the GROUP BY clause.

If the GROUP BY clause is omitted when an aggregate function is used, then the entire table is considered as one group, and the group function displays a single value for the entire table.

Example: What is the total salary paid to all employees?

The statement:

```
SELECT    SUM(SALARY)
FROM      EMPLOYEE ;
```

results in:

```
SUM(SALARY)

231000
```

5.4 Groups With Conditions: HAVING Clause

Sometimes you may want to specify a condition that applies to groups rather than to individual rows. For example, you might want a list of departments where the average departmental salary is above \$30,000. To express such a query, the HAVING clause is used. This clause specifies which groups should be selected and is used in combination with the GROUP BY clause. The form of this clause is as follows:

```
[GROUP BY  column-name1 [,column-name2] ...
[HAVING    search-condition ]
```

Conditions in the HAVING clause are applied after groups are formed. The search condition of the HAVING clause examines the grouped rows and produces a row for each group where the search condition in the

HAVING clause is true. The clause is similar to the WHERE clause, except the HAVING clause applies to groups.

Example: Which departments have an average salary above \$30,000? Order the results by average salary, with highest average salary appearing first.

The statement:

```
SELECT      DEPT, AVG (SALARY) AS AVG_SALARY
FROM        EMPLOYEE
GROUP BY    DEPT
HAVING      AVG_SALARY > 30000
ORDER BY    AVG_SALARY DESC ;
```

Syntax Notes:

- When HAVING is used, it always follows a GROUP BY clause.
- When referring to aggregate values in the HAVING and ORDER BY clauses of a GROUP BY you must assign an alternative name to the field, and use that in the HAVING and ORDER BY clauses.

The results are:

| DEPT | AVG_SALARY |
|------|------------|
| MGT | 45000 |
| FIN | 42000 |
| MKT | 33500 |

In this query, the average salary for all departments is computed, but only the names of those departments having an average salary above \$30,000 are displayed. Notice that Research and Development's average of \$27,500 is not displayed, nor is the Library's average of \$22,000.

The GROUP BY clause does not sort the results, thus the need for the ORDER BY clause. Finally, note that the ORDER BY clause must be placed after the GROUP BY and HAVING clauses.

This chapter has covered the computational capabilities of Taxis. In the next chapter, you will learn how to develop more complex queries by using the join operation and the nesting of queries.

5.5 Server functions

The Taxis server has a number of functions built into it which can operate on fields. This can occur anywhere an expression can occur in a SQL statement. It is possible that the server at your site has been extended with additional functions. Each of the arguments can be either a single field name, or another expression.

5.5.1 File functions

fromfile, fromfiletext

The `fromfile` and `fromfiletext` functions read a file. The syntax is

```
fromfile(filename[, offset[, length]])  
fromfiletext(filename[, offset[, length]])
```

These functions take one required, and two optional arguments. The first argument is the filename. The second argument is an offset into the file, and the third argument is the length of data to read. If the second argument is omitted then the file will be read from the beginning. If the third argument is omitted then the file will be read to the end. The result is the contents of the file. This can be used to load data into a table. For example if you have an indirect field and you wish to see the contents of the file you can issue SQL similar to the following.

The difference between the two functions is the type of data that is returned. `fromfile` will return varbyte data, and `fromfiletext` will return varchar data. If you are using the functions to insert data into a field you should make sure that you use the appropriate function for the type of field you are inserting into.

```
SELECT  FILENAME, fromfiletext(FILENAME)  
FROM    DOCUMENTS  
WHERE   DOCID = 'JT09113' ;
```

The results are:

| |
|--|
| <pre>FILENAME fromfiletext(FILENAME) /docs/JT09113.txt This is the text contained in the document that has an id of JT09113.</pre> |
|--|

totext

Converts data or file to text. The syntax is

```
totext(filename[, args])  
totext(data[, args])
```

This function will convert the contents of a file, if the argument given is an indirect, or else the result of the expression, and convert it to text. It does this by calling the program `anytotx`, which must be in the path. The `anytotx` program (obtained from Thunderstone) will handle PDF as well as many other file formats.

As of version 2.06.935767000 the `totext` command will take an optional second argument which contains arguments to the `anytotx` program. See the documentation for `anytotx` for details on its arguments.

```
SELECT  FILENAME, totext(FILENAME)
FROM    DOCUMENTS
WHERE   DOCID = 'JT09113' ;
```

The results are:

| FILENAME | totext(FILENAME) |
|-------------------|---|
| /docs/JT09113.pdf | This is the text contained in the document that has an id of JT09113. |

toind

Create a Taxis managed indirect file. The syntax is

```
toind(data)
```

This function takes the argument, stores it into a file, and returns the filename as an `indirect` type. This is most often used in combination with `fromfile` to create a Taxis managed file. For example:

```
INSERT INTO DOCUMENTS
VALUES('JT09114', toind(fromfile('srcfile')))
```

The database will now contain a pointer to a copy of `srcfile`, which will remain searchable even if the original is changed or removed. An important point to note is that any changes to `srcfile` will not be reflected in the database, unless the table row's `indirect` column is modified (even to the same value, this just tells Taxis to re-index it).

canonpath

Returns canonical version of a file path, i.e. fully-qualified and without symbolic links:

```
canonpath(path[, flags])
```

The optional `flags` is a set of bit flags: bit 0 set if error messages should be issued, bit 1 set if the return value should be empty instead of `path` on error. Added in version 5.01.1139446515 20060208.

pathcmp

File path comparison function; like C function `strcmp()` but for paths:

```
pathcmp(pathA, pathB)
```

Returns an integer indicating the sort order of `pathA` relative to `pathB`: 0 if `pathA` is the same as `pathB`, less than 0 if `pathA` is less than `pathB`, greater than 0 if `pathA` is greater than `pathB`. Paths are compared case-insensitively if and only if the OS is case-insensitive for paths, and OS-specific alternate directory separators are considered the same (e.g. “\” and “/” in Windows). Multiple consecutive directory separators are considered the same as one. A trailing directory separator (if not also a leading separator) is ignored. Directory separators sort lexically before any other character.

Note that the paths are only compared lexically: no attempt is made to resolve symbolic links, “.” path components, etc. Note also that no inference should be made about the magnitude of negative or positive return values: greater magnitude does not necessarily indicate greater lexical “separation”, nor should it be assumed that comparing the same two paths will always yield the same-magnitude value in future versions. Only the sign of the return value is significant. Added in version 5.01.1139446515 20060208.

basename

Returns the base filename of a given file path.

```
basename(path)
```

The `basename` is the contents of `path` after the last path separator. No filesystem checks are performed, as this is a text/parsing function; thus “.” and “.” are not significant. Added in version 7.00.1352510000 20121109.

dirname

Returns the directory part of a given file path.

```
dirname(path)
```

The directory is the contents of `path` before the last path separator (unless it is significant – e.g. for the root directory – in which case it is retained). Added in version 7.00.1352510000 20121109. No filesystem checks are performed, as this is a text/parsing function; thus “.” and “.” are not significant.

fileext

Returns the file extension of a given file path.

```
fileext(path)
```

The file extension starts with and includes a dot. The file extension is only considered present in the `basename` of the path, i.e. after the last path separator. Added in version 7.00.1352510000 20121109.

joinpath

Joins one or more file/directory path arguments into a merged path, inserting/removing a path separator between arguments as needed. Takes one to 5 path component arguments. E.g.:

```
joinpath('one', 'two/', '/three/four', 'five')
```

yields

```
one/two/three/four/five
```

Added in version 7.00.1352770000 20121112. Redundant path separators internal to an argument are not removed, nor are “.” and “.” path components removed. Prior to version 7.07.1550082000 20190213 redundant path separators between arguments were not removed.

joinpathabsolute

Like `joinpath`, except that a second or later argument that is an absolute path will overwrite the previously-merged path. E.g.:

```
joinpathabsolute('one', 'two', '/three/four', 'five')
```

yields

```
/three/four/five
```

Under Windows, partially absolute path arguments – e.g. “/dir” or “C:dir” where the drive or dir is still relative – are considered absolute for the sake of overwriting the merge.

Added in version 7.00.1352770000 20121112. Redundant path separators internal to an argument are not removed, nor are “.” and “.” path components removed. Prior to version 7.07.1550082000 20190213 partially absolute arguments were not considered absolute.

5.5.2 String Functions**abstract**

Generate an abstract of a given portion of text. The syntax is

```
abstract(text[, maxsize[, style[, query]]])
```


The abstract will be less than `maxsize` characters long, and will attempt to end at a word boundary. If `maxsize` is not specified (or is less than or equal to 0) then a default size of 230 characters is used.

The `style` argument is a string or integer, and allows a choice between several different ways of creating the abstract. Note that some of these styles require the `query` argument as well, which is a Metamorph query to look for:

- `dumb` (0)
Start the abstract at the top of the document.
- `smart` (1)
This style will look for the first meaningful chunk of text, skipping over any headers at the top of the text. This is the default if neither `style` nor `query` is given.
- `querysingle` (2)
Center the abstract contiguously on the best occurrence of `query` in the document.
- `querymultiple` (3)
Like `querysingle`, but also break up the abstract into multiple sections (separated with “. . .”) if needed to help ensure all terms are visible. Also take care with URLs to try to show the start and end.
- `querybest`
An alias for the best available query-based style; currently the same as `querymultiple`. Using `querybest` in a script ensures that if improved styles become available in future releases, the script will automatically “upgrade” to the best style.

If no `query` is given for the `query...` modes, they fall back to `dumb` mode. If a `query` is given with a *non-query...* mode (`dumb/smart`), the mode is promoted to `querybest`. The current locale and index expressions also have an effect on the abstract in the `query...` modes, so that it more closely reflects an index-obtained hit.

```
SELECT      abstract(STORY, 0, 1, 'power struggle')
FROM        ARTICLES
WHERE       ARTID = 'JT09115' ;
```

text2mm

Generate `LIKEP` query. The syntax is

```
text2mm(text[, maxwords])
```

This function will take a text expression, and produce a list of words that can be given to `LIKEP` or `LIKEP` to find similar documents. `text2mm` takes an optional second argument which specifies how many words should be returned. If this is not specified then 10 words are returned. Most commonly `text2mm` will be given the name of a field. If it is an `indirect` field you will need to call `fromfile` as shown below:

```

SELECT    text2mm(fromfile(FILENAME))
FROM      DOCUMENTS
WHERE     DOCID = 'JT09115' ;

```

You may also call it as `texttomm()` instead of `text2mm()` .

keywords

Generate list of keywords. The syntax is

```
keywords(text[, maxwords])
```

`keywords` is similar to `text2mm` but produces a list of phrases, with a linefeed separating them. The difference between `text2mm` and `keywords` is that `keywords` will maintain the phrases. `keywords` also takes an optional second argument which indicates how many words or phrases should be returned.

length

Returns the length in characters of a `char` or `varchar` expression, or number of strings/items in other types. The syntax is

```
length(value[, mode])
```

For example:

```

SELECT  NAME, length(NAME)
FROM    SYSTABLES

```

The results are:

| NAME | length(NAME) |
|--------------|--------------|
| SYSTABLES | 9 |
| SYSCOLUMNS | 10 |
| SYSINDEX | 8 |
| SYSUSERS | 8 |
| SYSPERMS | 8 |
| SYSTRIG | 7 |
| SYSMETAINDEX | 12 |

The optional `mode` argument is a `stringcomparemode`-style compare mode to use; see the Vortex manual on `<apicp stringcomparemode>` for details on syntax and the default. If `mode` is not given, the current `apicp stringcomparemode` is used. Currently the only pertinent mode flag is

“iso-8859-1”, which determines whether to interpret `value` as ISO-8859-1 or UTF-8. This can alter how many characters long the string appears to be, as UTF-8 characters are variable-byte-sized, whereas ISO-8859-1 characters are always mono-byte. The `mode` argument was added in version 6.

In version 5.01.1226622000 20081113 and later, if given a `strlst` type value, `length()` returns the number of string values in the list. For other types, it returns the number of values, e.g. for `varint` it returns the number of integer values.

lower

Returns the text expression with all letters in lower-case. The syntax is

```
lower(text[, mode])
```

For example:

```
SELECT  NAME, lower(NAME)
FROM    SYSTABLES
```

The results are:

| NAME | lower(NAME) |
|--------------|--------------|
| SYSTABLES | systables |
| SYSCOLUMNS | syscolumns |
| SYSINDEX | sysindex |
| SYSUSERS | sysusers |
| SYSPERMS | sysperms |
| SYSTRIG | systrig |
| SYSMETAINDEX | sysmetaindex |

Added in version 2.6.932060000.

The optional `mode` argument is a string-folding mode in the same format as `<apicp stringcomparemode>`; see the Vortex manual for details on the syntax and default. If `mode` is unspecified, the current `apicp stringcomparemode` setting – with “+lowercase” added – is used. The `mode` argument was added in version 6.

upper

Returns the text expression with all letters in upper-case. The syntax is

```
upper(text[, mode])
```

For example:

```
SELECT  NAME, upper (NAME)
FROM    SYSTABLES
```

The results are:

| NAME | upper (NAME) |
|--------------|--------------|
| SYSTABLES | SYSTABLES |
| SYSCOLUMNS | SYSCOLUMNS |
| SYSINDEX | SYSINDEX |
| SYSUSERS | SYSUSERS |
| SYSPERMS | SYSPERMS |
| SYSTRIG | SYSTRIG |
| SYSMETAINDEX | SYSMETAINDEX |

Added in version 2.6.932060000.

The optional mode argument is a string-folding mode in the same format as `<apicp stringcomparemode>`; see the Vortex manual for details on the syntax and default. If mode is unspecified, the current `apicp stringcomparemode` setting – with “+uppercase” added – is used. The mode argument was added in version 6.

initcap

Capitalizes text. The syntax is

```
initcap(text[, mode])
```

Returns the text expression with the first letter of each word in title case (i.e. upper case), and all other letters in lower-case. For example:

```
SELECT  NAME, initcap (NAME)
FROM    SYSTABLES
```

The results are:

| NAME | initcap (NAME) |
|--------------|----------------|
| SYSTABLES | Systables |
| SYSCOLUMNS | Syscolumns |
| SYSINDEX | Sysindex |
| SYSUSERS | Sysusers |
| SYSPERMS | Sysperms |
| SYSTRIG | Systrig |
| SYSMETAINDEX | Sysmetaindex |

Added in version 2.6.932060000.

The optional `mode` argument is a string-folding mode in the same format as `<apicp stringcomparemode>`; see the Vortex manual for details on the syntax and default. If `mode` is unspecified, the current `apicp stringcomparemode` setting – with “+titlecase” added – is used. The `mode` argument was added in version 6.

sandr

Search and replace text.

```
sandr(search, replace, text)
```

Returns the text expression with the search REX expression replaced with the replace expression. See the REX documentation and the Vortex `sandr` function documentation for complete syntax of the search and replace expressions.

```
SELECT  NAME, sandr('>>=SYS=', 'SYSTEM TABLE ', NAME) DESC
FROM    SYSTABLES
```

The results are:

| NAME | DESC |
|--------------|------------------------|
| SYSTABLES | SYSTEM TABLE TABLES |
| SYSCOLUMNS | SYSTEM TABLE COLUMNS |
| SYSINDEX | SYSTEM TABLE INDEX |
| SYSUSERS | SYSTEM TABLE USERS |
| SYSPERMS | SYSTEM TABLE PERMS |
| SYSTRIG | SYSTEM TABLE TRIG |
| SYSMETAINDEX | SYSTEM TABLE METAINDEX |

Added in version 3.0

separator

Returns the separator character from its `strlst` argument, as a `vchar` string:

```
separator(strlstValue)
```

This can be used in situations where the `strlstValue` argument may have a nul character as the separator, in which case simply converting `strlstValue` to `vchar` and looking at the last character would be incorrect. Added in version 5.01.1226030000 20081106.

stringcompare

Compares its string (`varchar`) arguments `a` and `b`, returning -1 if `a` is less than `b`, 0 if they are equal, or 1 if `a` is greater than `b`:

```
stringcompare(a, b[, mode])
```

The strings are compared using the optional `mode` argument, which is a string-folding mode in the same format as `<apicp stringcomparemode>`; see the Vortex manual for details on the syntax and default. If `mode` is unspecified, the current `apicp stringcomparemode` setting is used. Function added in version 6.00.1304108000 20110429.

stringformat

Returns its arguments formatted into a string (`varchar`), like the equivalent Vortex function `<strfmt>` (based on the C function `sprintf()`):

```
stringformat(format[, arg[, arg[, arg[, arg]]]])
```

The `format` argument is a `varchar` string that describes how to print the following argument(s), if any. See the Vortex manual for `<strfmt>` for details. Added in version 6.00.1300386000 20110317.

5.5.3 Math functions

The following basic math functions are available in Taxis: `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `fmod`, `log`, `log10`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.

All of the above functions call the ANSI C math library function of the same name, and return a result of type `double`. `pow`, `atan2` and `fmod` take two `double` arguments, the remainder take one `double` argument. Added in version: 2.6.931790000

In addition, the following math-related functions are available:

- `isNaN(x)`
Returns 1 if `x` is a float or double NaN (Not a Number) value, 0 if not. This function should be used to test for NaN, rather than using the equality operator (e.g. `x = 'NaN'`), because the IEEE standard defines `NaN == NaN` to be false, not true as might be expected. Added in version 5.01.1193955406 20071101.

5.5.4 Date functions

The following date functions are available in Taxis: `dayname`, `month`, `monthname`, `dayofmonth`, `dayofweek`, `dayofyear`, `quarter`, `week`, `year`, `hour`, `minute`, `second`.

All the functions take a date as an argument. `dayname` and `monthname` will return a string with the full day or month name based on the current locale, and the others return a number.

The `dayofweek` function returns 1 for Sunday. The quarter is based on months, so April 1st is the first day of quarter 2. Week 1 begins with the first Sunday of the year.

Added in version: 3.0.948300000

The following functions were added in version 3.01.990400000: `monthseq`, `weekseq` and `dayseq` which will return the number of months, weeks and days since an arbitrary past date. These can be used when comparing dates to see how many months, weeks or days separate them.

5.5.5 Bit manipulation functions

These functions are used to manipulate integers as bit fields. This can be useful for efficient set operations (e.g. set membership, intersection, etc.). For example, categories could be mapped to sequential bit numbers, and a row's category membership stored compactly as bits of an `int` or `varint`, instead of using a string list. Category membership can then be quickly determined with `bitand` on the integer.

In the following functions, bit field arguments `a` and `b` are `int` or `varint` (32 bits per integer, all platforms). Argument `n` is any integer type. Bits are numbered starting with 0 as the least-significant bit of the first integer. 31 is the most-significant bit of the first integer, 32 is the least-significant bit of the second integer (if a multi-value `varint`), etc. These functions were added in version 5.01.1099455599 of Nov 2 2004.

- `bitand(a, b)`
Returns the bit-wise AND of `a` and `b`. If one argument is shorter than the other, it will be expanded with 0-value integers.
- `bitor(a, b)`
Returns the bit-wise OR of `a` and `b`. If one argument is shorter than the other, it will be expanded with 0-value integers.
- `bitxor(a, b)`
Returns the bit-wise XOR (exclusive OR) of `a` and `b`. If one argument is shorter than the other, it will be expanded with 0-value integers.
- `bitnot(a)`
Returns the bit-wise NOT of `a`.
- `bitsize(a)`
Returns the total number of bits in `a`, i.e. the highest bit number plus 1.
- `bitcount(a)`
Returns the number of bits in `a` that are set to 1.
- `bitmin(a)`
Returns the lowest bit number in `a` that is set to 1. If none are set to 1, returns -1.

- `bitmax(a)`
Returns the highest bit number in `a` that is set to 1. If none are set to 1, returns -1.
- `bitlist(a)`
Returns the list of bit numbers of `a`, in ascending order, that are set to 1, as a `varint`. Returns a single -1 if no bits are set to 1.
- `bitshiftleft(a, n)`
Returns `a` shifted `n` bits to the left, with 0s padded for bits on the right. If `n` is negative, shifts right instead.
- `bitshiftright(a, n)`
Returns `a` shifted `n` bits to the right, with 0s padded for bits on the left (i.e. an unsigned shift). If `n` is negative, shifts left instead.
- `bitrotateleft(a, n)`
Returns `a` rotated `n` bits to the left, with left (most-significant) bits wrapping around to the right. If `n` is negative, rotates right instead.
- `bitrotateright(a, n)`
Returns `a` rotated `n` bits to the right, with right (least-significant) bits wrapping around to the left. If `n` is negative, rotates left instead.
- `bitset(a, n)`
Returns `a` with bit number `n` set to 1. `a` will be padded with 0-value integers if needed to reach `n` (e.g. `bitset(5, 40)` will return a `varint(2)`).
- `bitclear(a, n)`
Returns `a` with bit number `n` set to 0. `a` will be padded with 0-value integers if needed to reach `n` (e.g. `bitclear(5, 40)` will return a `varint(2)`).
- `bitisset(a, n)`
Returns 1 if bit number `n` is set to 1 in `a`, 0 if not.

5.5.6 Internet/IP address functions

The following functions manipulate IP network and/or host addresses; most take `inet` style argument(s). This is an IPv4 address string, optionally followed by a netmask.

For IPv4, the format is dotted-decimal, i.e. $N[.N[.[N.N]]]$ where N is a decimal, octal or hexadecimal integer from 0 to 255. If $x < 4$ values of N are given, the last N is taken as the last $5 - x$ bytes instead of 1 byte, with missing bytes padded to the right. E.g. `192.258` is valid and equivalent to `192.1.2.0`: the last N is 2 bytes in size, and covers $5 - 2 = 3$ needed bytes, including 1 zero pad to the right. Conversely, `192.168.4.1027` is not valid: the last N is too large.

An IPv4 address may optionally be followed by a netmask, either of the form `/B` or `:IPv4`, where B is a decimal, octal or hexadecimal netmask integer from 0 to 32, and `IPv4` is a dotted-decimal IPv4 address of the same format described above. If an `:IPv4` netmask is given, only the largest contiguous set of most-significant 1 bits are used (because netmasks are contiguous). If no netmask is given, it will be

calculated from standard IPv4 class A/B/C/D/E rules, but will be large enough to include all given bytes of the IP. E.g. 1.2.3.4 is Class A which has a netmask of 8, but the netmask will be extended to 32 to include all 4 given bytes.

In version 7.07.1554395000 20190404 and later, error messages are reported.

The `inet` functions were added in version 5.01.1113268256 of Apr 11 2005 and include the following. See also the Vortex `<urlutil>` equivalents:

- `inetabbrev(inet)`
Returns a possibly shorter-than-canonical representation of `$inet`, where trailing zero byte(s) of an IPv4 address may be omitted. All bytes of the network, and leading non-zero bytes of the host, will be included. E.g. `<urlutil inetabbrev "192.100.0.0/24">` returns `192.100.0/24`. The `/B` netmask is included, except if (in version 7.07.1554840000 20190409 and later) the network is host-only (i.e. netmask is the full size of the IP address). Empty string is returned on error.
- `inetcanon(inet)`
Returns canonical representation of `$inet`. For IPv4, this is dotted-decimal with all 4 bytes. The `/B` netmask is included, except if (in version 7.07.1554840000 20190409 and later) the network is host-only (i.e. netmask is the full size of the IP address). Empty string is returned on error.
- `inetnetwork(inet)`
Returns string IP address with the network bits of `inet`, and the host bits set to 0. Empty string is returned on error.
- `inethost(inet)`
Returns string IP address with the host bits of `inet`, and the network bits set to 0. Empty string is returned on error.
- `inetbroadcast(inet)`
Returns string IP broadcast address for `inet`, i.e. with the network bits, and host bits set to 1. Empty string is returned on error.
- `inetnetmask(inet)`
Returns string IP netmask for `inet`, i.e. with the network bits set to 1, and host bits set to 0. Empty string is returned on error.
- `inetnetmasklen(inet)`
Returns integer netmask length of `inet`. -1 is returned on error.
- `inetcontains(inetA, inetB)`
Returns 1 if `inetA` contains `inetB`, i.e. every address in `inetB` occurs within the `inetA` network. 0 is returned if not, or -1 on error.
- `inetclass(inet)`
Returns class of `inet`, e.g. A, B, C, D, E or `classless` if a different netmask is used (or the address is IPv6). Empty string is returned on error.
- `inet2int(inet)`

Returns integer representation of IP network/host bits of `$inet` (i.e. without netmask); useful for compact storage of address as integer(s) instead of string. Returns -1 is returned on error (note that -1 may also be returned for an all-ones IP address, e.g. 255.255.255.255).

- `int2inet(i)`
Returns `inet` string for 1- or 4-value `varint $i` taken as an IP address. Since no netmask can be stored in the integer form of an IP address, the returned IP string will not have a netmask. Empty string is returned on error.

urlcanonicalize

Canonicalize a URL. Usage:

```
urlcanonicalize(url[, flags])
```

Returns a copy of `url`, canonicalized according to case-insensitive comma-separated `flags`, which are zero or more of:

- `lowerProtocol`
Lower-cases the protocol.
- `lowerHost`
Lower-cases the hostname.
- `removeTrailingDot`
Removes trailing dot(s) in hostname.
- `reverseHost`
Reverse the host/domains in the hostname. E.g. `http://host.example.com/` becomes `http://com.example.host/`. This can be used to put the most-significant part of the hostname leftmost.
- `removeStandardPort`
Remove the port number if it is the standard port for the protocol.
- `decodeSafeBytes`
URL-decode safe bytes, where semantics are unlikely to change. E.g. “%41” becomes “A”, but “%2F” remains encoded, because it would decode to “/”.
- `upperEncoded`
Upper-case the hex characters of encoded bytes.
- `lowerPath`
Lower-case the (non-encoded) characters in the path. May be used for URLs known to point to case-insensitive filesystems, e.g. Windows.
- `addTrailingSlash`
Adds a trailing slash to the path, if no path is present.

Default flags are all but `reverseHost`, `lowerPath`. A flag may be prefixed with the operator `+` to append the flag to existing flags; `-` to remove the flag from existing flags; or `=` (default) to clear existing flags first and then set the flag. Operators remain in effect for subsequent flags until the next operator (if any) is used. Function added in Taxis version 7.05.

5.5.7 Geographical coordinate functions

The geographical coordinate functions allow for efficient processing of latitude / longitude operations. They allow for the conversion of a latitude/longitude pair into a single “geocode”, which is a single `long` value that contains both values. This can be used to easily compare it to other geocodes (for distance calculations) or for finding other geocodes that are within a certain distance.

azimuth2compass

```
azimuth2compass(double azimuth [, int resolution [, int verbosity]])
```

The `azimuth2compass` function converts a numerical azimuth value (degrees of rotation from 0 degrees north) and converts it into a compass heading, such as N or Southeast. The exact text returned is controlled by two optional parameters, `resolution` and `verbosity`.

`Resolution` determines how fine-grained the values returned are. There are 4 possible values:

- 1 - Only the four cardinal directions are used (N, E, S, W)
- 2 (default) - Inter-cardinal directions (N, NE, E, etc.)
- 3 - In-between inter-cardinal directions (N, NNE, NE, ENE, E, etc.)
- 4 - “by” values (N, NbE, NNE, NEbN, NE, NEbE, ENE, EbN, E, etc.)

`Verbosity` affects how verbose the resulting text is. There are two possible values:

- 1 (default) - Use initials for direction values (N, NbE, NNE, etc.)
- 2 - Use full text for direction values (North, North by east, North-northeast, etc.)

For an azimuth value of 105, here are some example results of `azimuth2compass`:

```
azimuth2compass(105) : E
azimuth2compass(105, 3) : ESE
azimuth2compass(105, 4) : EbS
azimuth2compass(105, 1, 2) : East
azimuth2compass(105, 3, 2) : East-southeast
azimuth2compass(105, 4, 2) : East by south
```

azimuthgeocode

```
azimuthgeocode(geocode1, geocode2 [, method])
```

The `azimuthgeocode` function calculates the directional heading going from one geocode to another. It returns a number between 0-360 where 0 is north, 90 east, etc., up to 360 being north again.

The third, optional `method` parameter can be used to specify which mathematical method is used to calculate the direction. There are two possible values:

- `greatcircle` (*default*) - The “Great Circle” method is a highly accurate tool for calculating distances and directions on a sphere. It is used by default.
- `pythagorean` - Calculations based on the pythagorean method can also be used. They’re faster, but less accurate as the core formulas don’t take the curvature of the earth into consideration. Some internal adjustments are made, but the values are less accurate than the `greatcircle` method, especially over long distances and with paths that approach the poles.

EXAMPLE

For examples of using `azimuthgeocode`, see the `geocode` script in the `taxis/samples` directory.

azimuthlatlon

```
azimuthlatlon(lat1, lon1, lat2, lon2, [, method])
```

The `azimuthlatlon` function calculates the directional heading going from one latitude-longitude point to another. It operates identically to `azimuthgeocode`, except `azimuthlatlon` takes its parameters in a pair of latitude-longitude points instead of geocode values.

The third, optional `method` parameter can be used to specify which mathematical method is used to calculate the direction. There are two possible values:

- `greatcircle` (*default*) - The “Great Circle” method is a highly accurate tool for calculating distances and directions on a sphere. It is used by default.
- `pythagorean` - Calculations based on the pythagorean method can also be used. They’re faster, but less accurate as the core formulas don’t take the curvature of the earth into consideration. Some internal adjustments are made, but the values are less accurate than the `greatcircle` method, especially over long distances and with paths that approach the poles.

dms2dec, dec2dms

```
dms2dec(dms)
dec2dms(dec)
```

The `dms2dec` and `dec2dms` functions are for changing back and forth between the deprecated Taxis/Vortex “degrees minutes seconds” (DMS) format (west-positive) and “decimal degree” format for latitude and longitude coordinates. All SQL geographical functions expect decimal degree parameters (the Vortex `<code2geo>` and `<geo2code>` Vortex functions expect Taxis/Vortex DMS).

Taxis/Vortex DMS values are of the format *DDMMSS*. For example, 35°15’ would be represented as 351500.

In decimal degrees, a degree is a whole digit, and minutes & seconds are represented as fractions of a degree. Therefore, 35°15’ would be 35.25 in decimal degrees.

Note that the Taxis/Vortex DMS format has *west*-positive longitudes (unlike ISO 6709 DMS format), and decimal degrees have *east*-positive longitudes. It is up to the caller to flip the sign of longitudes where needed.

distgeocode

```
distgeocode(geocode1, geocode2 [, method] )
```

The `distgeocode` function calculates the distance, in miles, between two given geocodes. It uses the “Great Circle” method for calculation by default, which is very accurate. A faster, but less accurate, calculation can be done with the Pythagorean theorem. It is not designed for distances on a sphere, however, and becomes somewhat inaccurate at larger distances and on paths that approach the poles. To use the Pythagorean theorem, pass a third string parameter, “`pythagorean`”, to force that method. “`greatcircle`” can also be specified as a method.

For example:

- New York (JFK) to Cleveland (CLE), the Pythagorean method is off by .8 miles (.1%)
- New York (JFK) to Los Angeles (LAX), the Pythagorean method is off by 22.2 miles (.8%)
- New York (JFK) to South Africa (PLZ), the Pythagorean method is off by 430 miles (5.2%)

EXAMPLE

For examples of using `distgeocode`, see the `geocode` script in the `taxis/samples` directory.

SEE ALSO

`distlatlon`

For a very fast method that leverages geocodes for selecting cities within a certain radius, see the `<code2geo>` and `<geo2code>` functions in the Vortex manual.

distlatlon

```
distlatlon(lat1, lon1, lat2, lon2 [, method] )
```

The `distlatlon` function calculates the distance, in miles, between two points, represented in latitude/longitude pairs in decimal degree format.

Like `distgeocode`, it uses the “Great Circle” method by default, but can be overridden to use the faster, less accurate Pythagorean method if “pythagorean” is passed as the optional `method` parameter.

For example:

- New York (JFK) to Cleveland (CLE), the Pythagorean method is off by .8 miles (.1%)
- New York (JFK) to Los Angeles (LAX), the Pythagorean method is off by 22.2 miles (.8%)
- New York (JFK) to South Africa (PLZ), the Pythagorean method is off by 430 miles (5.2%)

SEE ALSO

`distgeocode`

latlon2geocode, latlon2geocodearea

```
latlon2geocode(lat[, lon])
latlon2geocodearea(lat[, lon], radius)
```

The `latlon2geocode` function encodes a given latitude/longitude coordinate into one `long` return value. This encoded value – a “geocode” value – can be indexed and used with a special variant of Taxis’ `BETWEEN` operator for bounded-area searches of a geographical region.

The `latlon2geocodearea` function generates a bounding area centered on the coordinate. It encodes a given latitude/longitude coordinate into a *two-* value `varlong`. The returned geocode value pair represents the southwest and northeast corners of a square box centered on the latitude/longitude coordinate, with sides of length two times `radius` (in decimal degrees). This bounding area can be used with the Taxis `BETWEEN` operator for fast geographical searches. `latlon2geocodearea` was added in version 6.00.1299627000 20110308; it replaces the deprecated Vortex `<geo2code>` function.

The `lat` and `lon` parameters are doubles in the decimal degrees format. (To pass *DDMMSS* “degrees minutes seconds” (DMS) format values, convert them first with `dms2dec` or `parselatitude()/parselongitude()`). Negative numbers represent south latitudes and west longitudes, i.e. these functions are east-positive, and decimal format (unlike Vortex `<geo2code>` which is west-positive, and DMS-format).

Valid values for latitude are -90 to 90 inclusive. Valid values for longitude are -360 to 360 inclusive. A longitude value less than -180 will have 360 added to it, and a longitude value greater than 180 will have

360 subtracted from it. This allows longitude values to continue to increase or decrease when crossing the International Dateline, and thus avoid a non-linear “step function”. Passing invalid `lat` or `lon` values to `latlon2geocode` will return -1. These changes were added in version 5.01.1193956000 20071101.

In version 5.01.1194667000 20071109 and later, the `lon` parameter is optional: both latitude and longitude (in that order) may be given in a single space- or comma-separated text (`varchar`) value for `lat`. Also, a N/S suffix (for latitude) or E/W suffix (for longitude) may be given; S or W will negate the value.

In version 6.00.1300154000 20110314 and later, the latitude and/or longitude may have just about any of the formats supported by `parselatitude()`/`parselongitude()` (p. 77), provided they are disambiguated (e.g. separate parameters; or if one parameter, separated by a comma and/or fully specified with degrees/minutes/seconds).

EXAMPLE

```
-- Populate a table with latitude/longitude information:
create table geotest(city varchar(64), lat double, lon double,
                    geocode long);
insert into geotest values('Cleveland, OH, USA', 41.4, -81.5, -1);
insert into geotest values('Seattle, WA, USA', 47.6, -122.3, -1);
insert into geotest values('Dayton, OH, USA', 39.75, -84.19, -1);
insert into geotest values('Columbus, OH, USA', 39.96, -83.0, -1);
-- Prepare for geographic searches:
update geotest set geocode = latlon2geocode(lat, lon);
create index xgeotest_geocode on geotest(geocode);
-- Search for cities within a 3-degree-radius "circle" (box)
-- of Cleveland, nearest first:
select city, lat, lon, distlatlon(41.4, -81.5, lat, lon) MilesAway
from geotest
where geocode between (select latlon2geocodearea(41.4, -81.5, 3.0))
order by 4 asc;
```

For more examples of using `latlon2geocode`, see the `geocode` script in the `taxis/samples` directory.

CAVEATS

The geocode values returned by `latlon2geocode` and `latlon2geocodearea` are platform-dependent in format and accuracy, and should not be copied across platforms. On platforms with 32-bit longs a geocode value is accurate to about 32 seconds (around half a mile, depending on latitude). -1 is returned for invalid input values (in version 5.01.1193955804 20071101 and later).

NOTES

The geocodes produced by these functions are compatible with the codes used by the deprecated Vortex functions `<code2geo>` and `<geo2code>`. However, the `<code2geo>` and `<geo2code>` functions take Taxis/Vortex DMS format (*DDMMSS* “degrees minutes seconds”, as described in the `dec2dms` and `dms2dec` SQL functions).

SEE ALSO

`geocode2lat`, `geocode2lon`

geocode2lat, geocode2lon

```
geocode2lat (geocode)
geocode2lon (geocode)
```

The `geocode2lat` and `geocode2lon` functions decode a geocode into a latitude or longitude coordinate, respectively. The returned coordinate is in the decimal degrees format. In version 5.01.1193955804 20071101 and later, an invalid geocode value (e.g. -1) will return NaN (Not a Number).

If you want *DDMMSS* “degrees minutes seconds” (DMS) format, you can use `dec2dms` to convert it.

EXAMPLE

```
select city, geocode2lat(geocode), geocode2lon(geocode) from geotest;
```

CAVEATS

As with `latlon2geocode`, the geocode value is platform-dependent in accuracy and format, so it should not be copied across platforms, and the returned coordinates from `geocode2lat` and `geocode2lon` may differ up to about half a minute from the original coordinates (due to the finite resolution of a `long`). In version 5.01.1193955804 20071101 and later, an invalid geocode value (e.g. -1) will return NaN (Not a Number).

SEE ALSO

`latlon2geocode`

parselatitude, parselongitude

```

parselatitude(latitudeText)
parselongitude(longitudeText)

```

The `parselatitude` and `parselongitude` functions parse a text (varchar) latitude or longitude coordinate, respectively, and return its value in decimal degrees as a double. The coordinate should be in one of the following forms (optional parts in square brackets):

```

[H] nnn [U] [:] [H] [nnn [U] [:] [nnn [U]]] [H]
DDMM[.MMM...]
DDMMSS[.SSS...]

```

where the terms are:

- *nnn*
A number (integer or decimal) with optional plus/minus sign. Only the first number may be negative, in which case it is a south latitude or west longitude. Note that this is true even for *DDMMSS* (DMS) longitudes – i.e. the ISO 6709 east-positive standard is followed, not the deprecated Taxis/Vortex west-positive standard.
- *U*
A unit (case-insensitive):
 - d
 - deg
 - deg.
 - degrees
 - ' (single quote) for minutes
 - m
 - min
 - min.
 - minutes
 - " (double quote) for seconds
 - s (iff d/m also used for degrees/minutes)
 - sec
 - sec.
 - seconds
 - Unicode degree-sign (U+00B0), in ISO-8559-1 or UTF-8

If no unit is given, the first number is assumed to be degrees, the second minutes, the third seconds. Note that “s” may only be used for seconds if “d” and/or “m” was also used for an earlier degrees/minutes value; this is to help disambiguate “seconds” vs. “southern hemisphere”.

- *H*

A hemisphere (case-insensitive):

- N
- north
- S
- south
- E
- east
- W
- west

A longitude hemisphere may not be given for a latitude, and vice-versa.

- *DD*

A two- or three-digit degree value, with optional sign. Note that longitudes are east-positive ala ISO 6709, not west-positive like the deprecated Taxis standard.

- *MM*

A two-digit minutes value, with leading zero if needed to make two digits.

- *.MMM...*

A zero or more digit fractional minute value.

- *SS*

A two-digit seconds value, with leading zero if needed to make two digits.

- *.SSS...*

A zero or more digit fractional seconds value.

Whitespace is generally not required between terms in the first format. A hemisphere token may only occur once. Degrees/minutes/seconds numbers need not be in that order, if units are given after each number. If a 5-integer-digit *DDMM[.MMM...]* format is given and the degree value is out of range (e.g. more than 90 degrees latitude), it is interpreted as a *DMMSS[.SSS...]* value instead. To force *DDMMSS[.SSS...]* for small numbers, pad with leading zeros to 6 or 7 digits.

EXAMPLE

```
insert into geotest(lat, lon)
  values(parselatitude('54d 40m 10"'),
         parselongitude('W90 10.2'));
```

CAVEATS

An invalid or unparseable latitude or longitude value will return NaN (Not a Number). Extra unparsed/unparsable text may be allowed (and ignored) after the coordinate in most instances. Out-of-range values (e.g. latitudes greater than 90 degrees) are accepted; it is up to the caller to bounds-check the result. The `parselatitude` and `parselongitude` SQL functions were added in version 6.00.1300132000 20110314.

5.5.8 JSON functions

The JSON functions allow for the manipulation of `varchar` fields and literals as JSON objects.

JSON Path Syntax

The JSON Path syntax is standard Javascript object access, using `$` to represent the entire document. If the document is an object the path must start `$.`, and if an array `$[`.

JSON Field Syntax

In addition to using the JSON functions it is possible to access elements in a `varchar` field that holds JSON as if it was a field itself. This allows for creation of indexes, searching and sorting efficiently. Arrays can also be fetched as `strlst` to make use of those features, e.g.

```
SELECT Json.$.name FROM tablename WHERE 'SQL' IN Json.$.skills[*];
```

isjson

```
isjson(JsonDocument)
```

The `isjson` function returns 1 if the document is valid JSON, 0 otherwise.

```
isjson('{ "type" : 1 }'): 1
isjson('{}'): 1
isjson('json this is not'): 0
```

json_format

```
json_format(JsonDocument, FormatOptions)
```

The `json_format` formats the `JsonDocument` according to `FormatOptions`. Multiple options can be provided either space or comma separated.

Valid `FormatOptions` are:

- `COMPACT` - remove all unnecessary whitespace
- `INDENT(N)` - print the JSON with each object or array member on a new line, indented by N spaces to show structure
- `SORT-KEYS` - sort the keys in the object. By default the order is preserved
- `EMBED` - omit the enclosing `{ }` or `[]` is using the snippet in another object
- `ENSURE_ASCII` - encode all Unicode characters outside the ASCII range
- `ENCODE_ANY` - if not a valid JSON document then encode into a JSON literal, e.g. to encode a string.
- `ESCAPE_SLASH` - escape forward slash `/` as `\/`

json_type

```
json_type(JsonDocument)
```

The `json_type` function returns the type of the JSON object or element. Valid responses are:

- `OBJECT`
- `ARRAY`
- `STRING`
- `INTEGER`
- `DOUBLE`
- `NULL`
- `BOOLEAN`

Assuming a field `Json` containing: `"items": ["Num": 1, "Text": "The Name", "First": true , "Num": 2.0, "Text": "The second one", "First": false , null]`

```
json_type(Json): OBJECT
json_type(Json.$.items[0]): OBJECT
json_type(Json.$.items): ARRAY
json_type(Json.$.items[0].Num): INTEGER
json_type(Json.$.items[1].Num): DOUBLE
json_type(Json.$.items[0].Text): STRING
json_type(Json.$.items[0].First): BOOLEAN
json_type(Json.$.items[2]): NULL
```

json_value

```
json_value(JsonDocument, Path)
```

The `json_value` extracts the value identified by `Path` from `JsonDocument`. `Path` is a `varchar` in the JSON Path Syntax. This will return a scalar value. If `Path` refers to an array, object, or invalid path no value is returned.

Assuming the same `Json` field from the previous examples:

```
json_value(Json, '$'):
json_value(Json, '$.items[0]'):
json_value(Json, '$.items'):
json_value(Json, '$.items[0].Num'): 1
json_value(Json, '$.items[1].Num'): 2.0
json_value(Json, '$.items[0].Text'): The Name
json_value(Json, '$.items[0].First'): true
json_value(Json, '$.items[2]'):
```

json_query

```
json_query(JsonDocument, Path)
```

The `json_query` extracts the object or array identified by `Path` from `JsonDocument`. `Path` is a `varchar` in the JSON Path Syntax. This will return either an object or an array value. If `Path` refers to a scalar no value is returned.

Assuming the same `Json` field from the previous examples:

```
json_query(Json, '$')
-----
{"items":[{"Num":1,"Text":"The Name","First":true},~
  ↳{"Num":2.0,"Text":"The second one","First":false},null]}

json_query(Json, '$.items[0]')
-----
{"Num":1,"Text":"The Name","First":true}

json_query(Json, '$.items')
-----
[{"Num":1,"Text":"The Name","First":true},~
  ↳{"Num":2.0,"Text":"The second one","First":false},null]
```

The following will return an empty string as they refer to scalars or non-existent keys.

```
json_query(Json, '$.items[0].Num')
json_query(Json, '$.items[1].Num')
```

```
json_query(Json, '$.items[0].Text')
json_query(Json, '$.items[0].First')
json_query(Json, '$.items[2]')
```

json_modify

```
json_modify(JsonDocument, Path, NewValue)
```

The `json_modify` function returns a modified version of `JsonDocument` with the key at `Path` replaced by `NewValue`.

If `Path` starts with `append`, then the `NewValue` is appended to the array referenced by `Path`. It is an error if `Path` refers to anything other than an array.

```
json_modify('{}', '$.foo', 'Some "quote"')
-----
{"foo":"Some \"quote\""}

json_modify('{ "foo" : { "bar": [40, 42] } }', 'append $.foo.bar', 99)
-----
{"foo":{"bar":[40,42,99]}}

json_modify('{ "foo" : { "bar": [40, 42] } }', '$.foo.bar', 99)
-----
{"foo":{"bar":99}}
```

json_merge_patch

```
json_merge_patch(JsonDocument, Patch)
```

The `json_merge_patch` function provides a way to patch a target JSON document with another JSON document. The patch function conforms to ([href=https://tools.ietf.org/html/rfc7386](https://tools.ietf.org/html/rfc7386)) RFC 7386

Keys in `JsonDocument` are replaced if found in `Patch`. If the value in `Patch` is `null` then the key will be removed in the target document.

```
json_merge_patch('{ "a": "b" }', '{ "a": "c" }'
-----
{"a": "c"}

json_merge_patch('{ "a": [{"b": "c"}] }', '{ "a": [1] }'
-----
{"a": [1]}
```

```

json_merge_patch('[1,2]',          '{"a":"b", "c":null}'
-----
{"a":"b"}

% - - - - -
\subsubsection{json\_merge\_preserve}

\begin{verbatim}
  json_merge_preserve(JsonDocument, Patch)

```

The `json_merge_preserve` function provides a way to patch a target JSON document with another JSON document while preserving the content that exists in the target document.

Keys in `JsonDocument` are merged if found in `Patch`. If the same key exists in both the target and patch file the result will be an array with the values from both target and patch.

If the value in `Patch` is `null` then the key will be removed in the target document.

```

json_merge_preserve('{"a":"b"}',          '{"a":"c"}'
-----
{"a":["b","c"]}

json_merge_preserve('{"a": [{"b":"c"}]}', '{"a": [1]}'
-----
{"a":[{"b":"c"},1]}

json_merge_preserve('[1,2]',          '{"a":"b", "c":null}'
-----
[1,2,{"a":"b","c":null}]

```

5.5.9 Other Functions

exec

Execute an external command. The syntax is

```
exec(commandline[, INPUT[, INPUT[, INPUT[, INPUT]]]]);
```

Allows execution of an external command. The first argument is the command to execute. Any subsequent arguments are written to the standard input of the process. The standard output of the command is read as the return from the function.

This function allows unlimited extensibility of Taxis, although if a particular function is being used often then it should be linked into the Taxis server to avoid the overhead of invoking another process.

For example this could be used to OCR text. If you have a program which will take an image filename on the command line, and return the text on standard out you could issue SQL as follows:

```

UPDATE      DOCUMENTS
SET          TEXT = exec('ocr '+IMGFILE)
WHERE        TEXT = '';

```

Another example would be if you wanted to print envelopes from names and addresses in a table you might use the following SQL:

```

SELECT exec('envelope ', FIRST_NAME+' '+LAST_NAME+'
', STREET + '
', CITY + ', ' + STATE + ' ' + ZIP)
FROM ADDRESSES;

```

Notice in this example the addition of spaces and line-breaks between the fields. Taxis does not add any delimiters between fields or arguments itself.

mminfo

This function lets you obtain Metamorph info. You have the choice of either just getting the portions of the document which were the hits, or you can also get messages which describe each hit and subhits.

The SQL to use is as follows:

```

SELECT mminfo(query,data[,nhits,[0,msgs]]) from TABLE
[where CONDITION];

```

query Query should be a string containing a metamorph query.

data The text to search. May be literal data or a field from the table.

nhits The maximum number of hits to return. If it is 0, which is the default, you will get all the hits.

msgs An integer; controls what information is returned. A bit-wise OR of any combination of the following values:

- 1 to get matches and offset/length information
- 2 to suppress text from data which matches; printed by default
- 4 to get a count of hits (up to nhits)
- 8 to get the hit count in a numeric parseable format
- 16 to get the offset/length in the original query of each search set

Set offset/length information (value 16) is of the form:

```
Set N offset/len in query: setoff setlen
```


Where N is the set number (starting with 1), `setoff` is the byte offset from the start of the query where set N is, and `setlen` is the length of the set. This information is available in version 5.01.1220640000 20080905 and later.

Hit offset/length information is of the form:

```
300 <Data from Taxis> offset length suboff sublen [suboff sublen]..
301 End of Metamorph hit
```

Where:

- `offset` is the offset within the data of the overall hit context (sentence, paragraph, etc...)
- `length` is the length of the overall hit context
- `suboff` is the offset within the hit of a matching term
- `sublen` is the length of the matching term
- `suboff` and `sublen` will be repeated for as many terms as are required to satisfy the query.

Example:

```
select mminfo('power struggle @0 w/.',Body,0,0,1) inf from html
      where Title\Meta\Body like 'power struggle';
```

Would give something of the form:

```
300 <Data from Taxis> 62 5 0 5
power
301 End of Metamorph hit
300 <Data from Taxis> 2042 5 0 5
power
301 End of Metamorph hit
300 <Data from Taxis> 2331 5 0 5
POWER
301 End of Metamorph hit
300 <Data from Taxis> 2892 8 0 8
STRUGGLE
301 End of Metamorph hit
```

convert

The `convert` function allows you to change the type of an expression. The syntax is

```
CONVERT(expression, 'type-name' [, 'mode'])
```

The type name should in general be in lower case.

This can be useful in a number of situations. Some cases where you might want to use `convert` are

- The display format for a different format is more useful. For example you might want to convert a field of type COUNTER to a DATE field, so you can see when the record was inserted, for example:

```
SELECT convert(id, 'date')
FROM LOG;
```

```
CONVERT(id, 'date')
1995-01-27 22:43:48
```

- If you have an application which is expecting data in a particular type you can use convert to make sure you will receive the correct type.

Caveat: Note that in Taxis version 7 and later, `convert()` ing data from/to `varbyte/varchar` no longer converts the data to/from hexadecimal by default (as was done in earlier versions) in programs other than `tsql`; it is now preserved as-is (though truncated at nul for `varchar`). See the `binto hex()` and `hextobin()` functions (p. 88) for hexadecimal conversion, and the `hexifybytes` SQL property (p. 186) for controlling automatic hex conversion.

Also in Taxis version 7 and later, an optional third argument may be given to `convert()`, which is a `varchar to strlst sep` mode value (p. 184). This third argument may only be supplied when converting to type `strlst` or `varstrlst`. It allows the separator character or mode to be conveniently specified locally to the conversion, instead of having to alter the global `varchar to strlst sep` mode.

seq

Returns a sequence number. The number can be initialized to any value, and the increment can be defined for each call. The syntax is:

```
seq(increment [, init])
```

If `init` is given then the sequence number is initialized to that value, which will be the value returned. It is then incremented by `increment`. If `init` is not specified then the current value will be retained. The initial value will be zero if `init` has not been specified.

Examples of typical use:

```
SELECT NAME, seq(1)
FROM SYSTABLES
```

The results are:

| NAME | seq(1) |
|--------------|--------|
| SYSTABLES | 0 |
| SYSCOLUMNS | 1 |
| SYSINDEX | 2 |
| SYSUSERS | 3 |
| SYSPERMS | 4 |
| SYSTRIG | 5 |
| SYSMETAINDEX | 6 |

```
SELECT  seq(0, 100)
FROM    SYSDUMMY;
```

```
SELECT  NAME, seq(1)
FROM    SYSTABLES
```

The results are:

| | |
|--------------|--------|
| seq(0, 100) | |
| 100 | |
| NAME | seq(1) |
| SYSTABLES | 100 |
| SYSCOLUMNS | 101 |
| SYSINDEX | 102 |
| SYSUSERS | 103 |
| SYSPERMS | 104 |
| SYSTRIG | 105 |
| SYSMETAINDEX | 106 |

random

Returns a random `int`. The syntax is:

```
random(max [, seed])
```

If `seed` is given then the random number generator is seeded to that value. The random number generator will only be seeded once in each session, and will be randomly seeded on the first call if no seed is supplied. The `seed` parameter is ignored in the second and later calls to `random` in a process.

The returned number is always non-negative, and never larger than the limit of the C lib's random number generator (typically either 32767 or 2147483647). If `max` is non-zero, then the returned number will also be less than `max`.

This function is typically used to either generate a random number for later use, or to generate a random ordering of result records by adding `random` to the `ORDER BY` clause.

Examples of typical use:

```
SELECT  NAME, random(100)
FROM    SYSTABLES
```

The results might be:

| NAME | random(100) |
|--------------|-------------|
| SYSTABLES | 90 |
| SYSCOLUMNS | 16 |
| SYSINDEX | 94 |
| SYSUSERS | 96 |
| SYSPERMS | 1 |
| SYSTRIG | 84 |
| SYSMETAINDEX | 96 |

```
SELECT  ENAME
FROM    EMPLOYEE
ORDER BY random(0);
```

The results would be a list of employees in a random order.

binto hex

Converts a binary (`varbyte`) value into a hexadecimal string.

```
binto hex(varbyteData[, 'stream|pretty'])
```

A string (`varchar`) hexadecimal representation of the `varbyteData` parameter is returned. This can be useful to visually examine binary data that may contain non-printable or nul bytes. The optional second argument is a comma-separated string of any of the following flags:

- `stream`: Use the default output mode: a continuous stream of hexadecimal bytes, i.e. the same format that `convert(varbyteData, 'varchar')` would have returned in Taxis version 6 and earlier.
- `pretty`: Return a “pretty” version of the data: print 16 byte per line, space-separate the hexadecimal bytes, and print an ASCII dump on the right side.

The `binto hex()` function was added in Taxis version 7. Caveat: Note that in version 7 and later, `convert()` ing data from/to `varbyte/varchar` no longer converts the data to/from hexadecimal by default (as was done in earlier versions) in programs other than `tsql`; it is now preserved as-is (though truncated at nul for `varchar`). See the `hexifybytes` SQL property (p. 186) to change this.

hextobin

Converts a hexadecimal stream to its binary representation.

```
hextobin(hexString[, 'stream|pretty'])
```

The hexadecimal `varchar` string `hexString` is converted to its binary representation, and the `varbyte` result returned. The optional second argument is a comma-separated string of any of the following flags:

- `stream`: Only accept the `stream` format of `bintoHex()`, i.e. a stream of hexadecimal bytes, the same format that `convert(varbyteData, 'varchar')` would have returned in Taxis version 6 and earlier. Whitespace is acceptable, but only between (not within) hexadecimal bytes. Case-insensitive. Non-conforming data will result in an error message and the function failing.
- `pretty`: Accept either `stream` or `pretty` format data; if the latter, only the hexadecimal bytes are parsed (e.g. ASCII column is ignored). Parsing is more liberal, but may be confused if the data deviates significantly from either format.

The `hextobin()` function was added in Taxis version 7. Caveat: Note that in version 7 and later, `convert()` ing data from/to `varbyte/varchar` no longer converts the data to/from hexadecimal by default (as was done in earlier versions) in programs other than `tsql`; it is now preserved as-is (though truncated at nul for `varchar`). See the `hexifybytes` SQL property (p. 186) to change this.

identifylanguage

Tries to identify the predominant language of a given string. By returning a probability in addition to the identified language, this function can also serve as a test of whether the given string is really natural-language text, or perhaps binary/encoded data instead. Syntax:

```
identifylanguage(text[, language[, samplesize]])
```

The return value is a two-element `strlst`: a probability and a language code. The probability is a value from 0.000 to 1.000 that the `text` argument is composed in the language named by the returned language code. The language code is a two-letter ISO-639-1 code.

If an ISO-639-1 code is given for the optional `language` argument, the probability for that particular language is returned, instead of for the highest-probability language of the known/built-in languages (currently `de`, `es`, `fr`, `ja`, `pl`, `tr`, `da`, `en`, `eu`, `it`, `ko`, `ru`).

The optional third argument `samplesize` is the initial integer size in bytes of the `text` to sample when determining language; it defaults to 16384. The `samplesize` parameter was added in version 7.01.1382113000 20131018.

Note that since a `strlst` value is returned, the probability is returned as a `strlst` element, not a `double` value, and thus should be cast to `double` during comparisons. In Vortex with `arrayconvert` on (the default), the return value will be automatically split into a two-element Vortex `varchar` array.

The `identifylanguage()` function is experimental, and its behavior, syntax, name and/or existence are subject to change without notice. Added in version 7.01.1381362000 20131009.

lookup

By combining the `lookup()` function with a `GROUP BY`, a column may be grouped into bins or ranges – e.g. for price-range grouping – instead of distinct individual values. Syntax:

```
lookup(keys, ranges[, names])
```

The `keys` argument is one (or more, e.g. `strlst`) values to look up; each is searched for in the `ranges` argument, which is one (or more, e.g. `strlst`) ranges. All range(s) that the given key(s) match will be returned. If the `names` argument is given, the corresponding `names` value(s) are returned instead; this allows ranges to be renamed into human-readable values. If `names` is given, the number of its values must equal the number of ranges.

Each range is a pair of values (lower and upper bounds) separated by “.” (two periods). The range is optionally surrounded by square (bound included) or curly (bound excluded) brackets. E.g.:

```
[10..20}
```

denotes the range 10 to 20: including 10 (“[”) but not including (“}”) 20. Both an upper and lower bracket must be given if either is present (though they need not be the same type). The default if no brackets are given is to include the lower bound but exclude the upper bound; this makes consecutive ranges non-overlapping, if they have the same upper and lower bound and no brackets (e.g. “0..10,10..20”). Either bound may be omitted, in which case that bound is unlimited. Each range’s lower bound must not be greater than its upper bound, nor equal if either bound is exclusive.

If a `ranges` value is not `varchar/char`, or does not contain “.”, its entire value is taken as a single inclusive lower bound, and the exclusive upper bound will be the next `ranges` value’s lower bound (or unlimited if no next value). E.g. the `varint` lower-bound list:

```
0,10,20,30
```

is equivalent to the `strlst` range list:

```
[0..10}, [10..20}, [20..30}, [30..]
```

By using the `lookup()` function in a `GROUP BY`, a column may be grouped into ranges. For example, given a table `Products` with the following `SKUs` and `float` prices:

```
SKU      Price
-----
```

| | |
|------|--------|
| 1234 | 12.95 |
| 1235 | 5.99 |
| 1236 | 69.88 |
| 1237 | 39.99 |
| 1238 | 29.99 |
| 1239 | 25.00 |
| 1240 | 50.00 |
| 1241 | -2.00 |
| 1242 | 499.95 |
| 1243 | 19.95 |
| 1244 | 9.99 |
| 1245 | 125.00 |

they may be grouped into price ranges (with most-products first) with this SQL:

```
SELECT  lookup(Price, convert('0..25,25..50,50...', 'strlst', 'lastchar'),
        convert('Under $25,$25-49.99,$50 and up', 'strlst', 'lastchar'))
        PriceRange, count(SKU) NumberOfProducts
FROM Products
GROUP BY lookup(Price, convert('0..25,25..50,50...', 'strlst', 'lastchar'),
        convert('Under $25,$25-49.99,$50 and up', 'strlst', 'lastchar'))
ORDER BY 2 DESC;
```

or this Vortex:

```
<$binValues =    "0..25"        "25..50"        "50..">
<$binDisplays = "Under $$25" "$$25-49.99" "$$50 and up">
<sql row "select lookup(Price, $binValues, $binDisplays) PriceRange,
        count(SKU) NumberOfProducts
        from Products
        group by lookup(Price, $binValues, $binDisplays)
        order by 2 desc">
    <fmt "%10s: %d\n" $PriceRange $NumberOfProducts>
</sql>
```

which would give these results:

| PriceRange | NumberOfProducts |
|--------------|------------------|
| Under \$25, | 4 |
| \$50 and up, | 4 |
| \$25-49.99, | 3 |
| | 1 |

The trailing commas in `PriceRange` values are due to them being `strlst` values, for possible multiple ranges. Note the empty `PriceRange` for the fourth row: the -2 `Price` matched no ranges, and hence an empty `PriceRange` was returned for it.

CAVEATS

The `lookup()` function as described above was added in Taxis version 7.06.1528745000 20180611.

A different version of the `lookup()` function was first added in version 7.01.1386980000 20131213: it only took the second range syntax variant (single lower bound); range values had to be in ascending order (by `keys` type); only the first matching range was returned; and if a key did not match any range the first range was returned.

SEE ALSO

`lookupCanonicalizeRanges`, `lookupParseRange`

lookupCanonicalizeRanges

The `lookupCanonicalizeRanges()` function returns the canonical version(s) of its `ranges` argument, which is zero or more ranges of the syntaxes acceptable to `lookup()` (p. 90):

```
lookupCanonicalizeRanges(ranges, keyType)
```

The canonical version always includes both a lower and upper inclusive/exclusive bracket/brace, both lower and upper bounds (unless unlimited), the “.” range operator, and is independent of other ranges that may be in the sequence.

The `keyType` parameter is a `varchar` string denoting the SQL type of the key field that would be looked up in the given range(s). This ensures that comparisons are done correctly. E.g. for a `strlst` range list of “0, 500, 1000”, `keyType` should be “integer”, so that “500” is not compared alphabetically with “1000” and considered invalid (greater than).

This function can be used to verify the syntax of a range, or to transform it into a standard form for `lookupParseRange()` (p. 93).

CAVEATS

For an implicit-upper-bound range, the upper bound is determined by the *next* range’s lower bound. Thus the full list of ranges (if multiple) should be given to `lookupCanonicalizeRanges()` – even if only one range needs to be canonicalized – so that each range gets its proper bounds.

The `lookupCanonicalizeRanges()` function was added in version 7.06.1528837000 20180612. The `keyType` parameter was added in version 7.06.1535500000 20180828.

SEE ALSO

lookup, lookupParseRange

lookupParseRange

The `lookupParseRange()` function parses a single `lookup()`-style range into its constituent parts, returning them as strings in one `strlst` value. This can be used by Vortex scripts to edit a range. Syntax:

```
lookupParseRange(range, parts)
```

The `parts` argument is zero or more of the following part tokens as strings:

- `lowerInclusivity`: Returns the inclusive/exclusive operator for the lower bound, e.g. “{” or “[”
- `lowerBound`: Returns the lower bound
- `rangeOperator`: Returns the range operator, e.g. “. .”
- `upperBound`: Returns the upper bound
- `upperInclusivity`: Returns the inclusive/exclusive operator for the upper bound, e.g. “}” or “]”

If a requested part is not present, an empty string is returned for that part. The concatenation of the above listed parts, in the above order, should equal the given range. Non-string range arguments are not supported.

The `lookupParseRange()` function was added in version 7.06.1528837000 20180612.

EXAMPLE

```
lookupParseRange('10..20', 'lowerInclusivity')
```

would return a single empty-string `strlst`, as there is no lower-bound inclusive/exclusive operator in the range “10..20”.

```
lookupParseRange('10..20', 'lowerBound')
```

would return a `strlst` with the single value “10”.

CAVEATS

For an implicit-upper-bound range, the upper bound is determined by the *next* range's lower bound. Since `lookupParseRange()` only takes one range, passing such a range to it may result in an incorrect (unlimited) upper bound. Thus the full list of ranges (if multiple) should always be given to `lookupCanonicalizeRanges()` first, and only then the desired canonicalized range passed to `lookupParseRange()`.

SEE ALSO

`lookup`, `lookupCanonicalizeRanges`

hasFeature

Returns 1 if given feature is supported, 0 if not (or unknown). The syntax is:

```
hasFeature(feature)
```

where `feature` is one of the following `varchar` tokens:

- RE2 For RE2 regular expression support in REX

This function is typically used in Vortex scripts to test if a feature is supported with the current version of Taxis, and if not, to work around that fact if possible. For example:

```
<if hasFeature( "RE2" ) = 1>
  ... proceed with RE2 expressions ...
<else>
  ... use REX instead ...
</if>
```

Note that in a Vortex script that does not support `hasFeature()` itself, such an `<if>` statement will still compile and run, but will be false (with an error message).

Added in version 7.06.1481662000 20161213. Some feature tokens were added in later versions.

ifNull

Substitute another value for NULL values. Syntax:

```
ifNull(testVal, replaceVal)
```

If `testVal` is a SQL NULL value, then `replaceVal` (cast to the type of `testVal`) is returned; otherwise `testVal` is returned. This function can be used to ensure that NULL value(s) in a column are replaced with a non-NULL value, if a non-NULL value is required:

```
SELECT ifNull(myColumn, 'Unknown') FROM myTable;
```

Added in version 7.02.1405382000 20140714. Note that SQL NULL is not yet fully supported in Taxis (including in tables). See also `isNull`.

isNull

Tests a value, and returns a long value of 1 if NULL, 0 if not. Syntax:

```
isNull(testVal)
```

```
SELECT isNull(myColumn) FROM myTable;
```

Added in version 7.02.1405382000 20140714. Note that SQL NULL is not yet fully supported in Taxis (including in tables). Also note that Taxis `isNull` behavior differs from some other SQL implementations; see also `ifNull`.

xmlTreeQuickXPath

Extracts information from an XML document.

```
xmlTreeQuickXPath(string xmlRaw, string xpathQuery  
    [, string[] xmlns])
```

Parameters:

- `xmlRaw` - the plain text of the xml document you want to extract information from
- `xpathQuery` - the XPath expression that identifies the nodes you want to extract the data from
- `xmlns` (*optional*) - an array of `prefix=URI` namespaces to use in the XPath query

Returns:

- String values of the node from the XML document `xmlRaw` that match `xpathQuery`

`xmlTreeQuickXPath` allows you to easily extract information from an XML document in a one-shot function. It is intended to be used in SQL statements to extract specific information from a field that contains XML data.

It is essentially a one statement version of the following:

```
<$doc = (xmlTreeNewDocFromString($xmlRaw))>
<$xpath = (xmlTreeNewXPath($doc))>
<$nodes = (xmlTreeXPathExecute($xpathQuery))>
<loop $nodes>
    <$ret = (xmlTreeGetAllContent($nodes))>
    <$content = $content $ret>
</loop>
```

EXAMPLE

if the `xmlData` field of a table has content like this:

```
<extraInfo>
    <price>8.99</price>
    <author>John Doe</author>
    <isbn>978-0-06-051280-4</isbn>
</extraInfo>
```

Then the following SQL statement will match that row:

```
SELECT * from myTable where xmlTreeQuickXPath(data,
'/extraInfo/author') = 'John Doe'
```

Chapter 6

Advanced Queries

This chapter is divided into three sections. The first one focuses on using the join operation to retrieve data from multiple tables. The second section covers nesting of queries, also known as subqueries. The final section introduces several advanced query techniques, including self-joins, correlated subqueries, subqueries using the EXISTS operator.

6.1 Retrieving Data From Multiple Tables

All the queries looked at so far have been answered by accessing data from one table. Sometimes, however, answers to a query may require data from two or more tables.

For example, for the Corporate Librarian to display a list of contributing authors with their long form department name requires data from the REPORT table (author) and data from the DEPARTMENT table (department name). Obtaining the data you need requires the ability to combine two or more tables. This process is commonly referred to as “*joining the tables*”.

Two or more tables can be combined to form a single table by using the *join operation*. The join operation is based on the premise that there is a logical association between two tables based on a common attribute that links the tables. Therefore, there must be a common column in each table for a join operation to be executed. For example, both the REPORT table and the DEPARTMENT table have the department identification code in common. Thus, they can be joined.

Joining two tables in Taxis is accomplished by using a SELECT statement. The general form of the SELECT statement when a join operation is involved is:

```
SELECT    column-name1 [,column-name2] ...
FROM      table-name1, table-name2
WHERE     table-name1.column-name = table-name2.column-name ;
```

The combination of table name with column name as stated in the WHERE clause describes the Join condition.

Command Discussion

1. A join operation pulls data from two or more tables listed in the `FROM` clause. These tables represent the source of the data to be joined.
2. The `WHERE` clause specifies the relationship between the tables to be joined. This relationship represents the *join condition*. Typically, the join condition expresses a relationship between rows from each table that match on a common attribute.
3. When the tables to be joined have the same column name, the column name is prefixed with a table name in order for Taxis to know from which table the column comes. Taxis uses the notation:

`table-name.column-name`

The table name in front of the column name is referred to as a *qualifier*.

4. The common attributes in the join condition need not have the same column name, but they should represent the same kind of information. For example, where the attribute representing names of people submitting resumes was named `RNAME` in table 1, and the attribute for names of employees was named `ENAME` in table 2, you could still join the tables on the common character field by specifying:

`WHERE table-name1.RNAME = table-name2.ENAME`

While the above is true, it is still a good rule of thumb in database design to give the same name to all columns referring to data of the same type and meaning. Columns which are designed to be a key, and intended as the basis for joining tables would normally be given the same name.

5. If a row from one of the tables never satisfies the join condition, that row will not appear in the joined table.
6. The tables are joined together, and then Taxis extracts the data, or columns, listed in the `SELECT` clause.
7. Although tables can be combined if you omit the `WHERE` clause, this would result in a table of all possible combinations of rows from the tables in the `FROM` clause. This output is usually not intended, nor meaningful, and can waste much computer processing time. Therefore, be careful in forming queries that involve multiple tables.

Example: The corporate librarian wants to distribute a list of authors who have contributed reports to the corporate library, along with the name of that author's department. To fulfill this request, data from both the `REPORT` table (author) and the `DEPARTMENT` table (department name) are needed.

You would enter this statement:

```
SELECT  AUTHOR, DNAME
FROM    REPORT, DEPARTMENT
WHERE   REPORT.DEPT = DEPARTMENT.DEPT ;
```

Syntax Notes:

- REPORT and DEPARTMENT indicate the tables to be joined.
- The WHERE clause statement defines the condition for the join.
- The notation “REPORT.” in “REPORT.DEPT”, and “DEPARTMENT.” in “DEPARTMENT.DEPT” are the qualifiers which indicate from which table to find the column.

This statement will result in the following joined table:

| AUTHOR | DNAME |
|------------------|-------------------------------|
| Jackson, Herbert | Research and Development |
| Sanchez, Carla | Product Marketing and Sales |
| Price, Stella | Finance and Accounting |
| Smith, Roberta | Research and Development |
| Aster, John A. | Product Marketing and Sales |
| Jackson, Herbert | Research and Development |
| Barrington, Kyle | Management and Administration |

In this query, we are joining data from the REPORT and the DEPARTMENT tables. The common attribute in these two tables is the department code. The conditional expression:

```
REPORT.DEPT = DEPARTMENT.DEPT
```

is used to describe how the rows in the two tables are to be matched. Each row of the joined table is the result of combining a row from the REPORT table and a row from the DEPARTMENT table for each comparison with matching codes.

To further illustrate how the join works, look at the rows in the REPORT table below where DEPT is “MKT”:

| TITLE | AUTHOR | DEPT | FILENAME |
|----------------------------|------------------|------|---------------------|
| Disappearing Ink | Jackson, Herbert | RND | /data/rnd/ink.txt |
| > INK PROMOTIONAL CAMPAIGN | SANCHEZ, CARLA | MKT | /data/MKT/PROMO.RPT |
| Budget for 4Q 92 | Price, Stella | FIN | /data/ad/4q.rpt |
| Round Widgets | Smith, Roberta | RND | /data/rnd/widge.txt |
| > PAPERCLIPS | ASTER, JOHN A. | MKT | /data/MKT/CLIP.RPT |
| Color Panorama | Jackson, Herbert | RND | /data/rnd/color.txt |
| Meeting Schedule | Barrington, Kyle | MGT | /data/mgt/when.rpt |

Now look at the rows in the DEPARTMENT table below where DEPT is “MKT”. These are matching rows since the department code (“MKT”) is the same.

| DEPT | DNAME | DHEAD | DIV | BUDGET |
|------|-------|-------|-----|--------|
|------|-------|-------|-----|--------|

| | | | | |
|-------|-------------------------------------|------------|------|-------|
| MGT | Management and Administration | Barrington | CORP | 22000 |
| FIN | Finance and Accounting | Price | CORP | 26000 |
| LEG | Corporate Legal Support | Thomas | CORP | 28000 |
| SUP | Supplies and Procurement | Sweet | CORP | 10500 |
| REC | Recruitment and Personnel | Harris | CORP | 15000 |
| RND | Research and Development | Jones | PROD | 27500 |
| MFG | Manufacturing | Washington | PROD | 32000 |
| CSS | Customer Support and Service | Ferrer | PROD | 11000 |
| > MKT | PRODUCT MARKETING AND SALES | BROWN | PROD | 25000 |
| ISM | Information Systems Management | Dedrich | INFO | 22500 |
| LIB | Corporate Library | Krinski | INFO | 18500 |
| SPI | Strategic Planning and Intelligence | Peters | INFO | 28500 |

The matching rows can be conceptualized as combining a row from the REPORT table with a matching row from the DEPARTMENT table. Below is a sample of rows from both tables, matched on the department code “MKT”:

| DEPT | DNAME | DHEAD | DIV | BUDGET | TITLE | AUTHOR | FILENAME |
|------|-----------|-------|------|--------|------------|---------|---------------------|
| MKT | Marketing | Brown | PROD | 25000 | Ink | Sanchez | /data/mkt/promo.rpt |
| MKT | Marketing | Brown | PROD | 25000 | Paperclips | Aster | /data/mkt/clip.rpt |

This operation is carried out for all matching rows; i.e., each row in the REPORT table is combined, or matched, with a row having the same department code in the DEPARTMENT table:

| DEPT | DNAME | DHEAD | DIV | BUDGET | TITLE | AUTHOR | FILENAME |
|------|------------|-------|------|--------|------------|---------|---------------------|
| RND | Research | Jones | PROD | 27500 | Ink | Jackson | /data/rnd/ink.txt |
| MKT | Marketing | Brown | PROD | 25000 | Ink Promo | Sanchez | /data/mkt/promo.rpt |
| FIN | Finance | Price | CORP | 26000 | Budget | Price | /data/ad/4q.rpt |
| RND | Research | Jones | PROD | 27500 | Widgets | Smith | /data/rnd/widge.txt |
| MKT | Marketing | Brown | PROD | 25000 | Paperclips | Aster | /data/mkt/clip.rpt |
| RND | Research | Jones | PROD | 27500 | Panorama | Jackson | /data/rnd/color.txt |
| MGT | Management | Barri | CORP | 22000 | Schedule | Barring | /data/mgt/when.rpt |

The columns requested in the SELECT statement determine the final output for the joined table:

| | |
|------------------|-------------------------------|
| AUTHOR | DNAME |
| Jackson, Herbert | Research and Development |
| Sanchez, Carla | Product Marketing and Sales |
| Price, Stella | Finance and Accounting |
| Smith, Roberta | Research and Development |
| Aster, John A. | Product Marketing and Sales |
| Jackson, Herbert | Research and Development |
| Barrington, Kyle | Management and Administration |

Observe that the joined table does not include any data on several departments from the `DEPARTMENT` table, where that department did not produce any contributing authors as listed in the `REPORT` table. The joined table includes only rows where a match has occurred between rows in both tables. If a row in either table does not match any row in the other table, the row is not included in the joined table.

In addition, notice that the `DEPT` column is not included in the final joined table. Only two columns are included in the joined table since just two columns are listed in the `SELECT` clause, and `DEPT` is not one of them.

The next example illustrates that conditions other than the join condition can be used in the `WHERE` clause. It also shows that even though the results come from a single table, the solution may require that data from two or more tables be joined in the `WHERE` clause.

Example: Assume that you cannot remember the department code for Research and Development, but you want to know the titles of all reports submitted from that department.

Enter this statement:

```
SELECT    TITLE
FROM      DEPARTMENT, REPORT
WHERE     DNAME = 'RESEARCH AND DEVELOPMENT'
AND       REPORT.DEPT = DEPARTMENT.DEPT ;
```

Syntax Notes:

- The tables to be joined are listed after `FROM`.
- The condition for the join operation is specified after `AND` (as part of `WHERE`).

The results follow:

```
TITLE
Innovations in Disappearing Ink
Improvements in Round Widgets
Ink Color Panorama
```

Since you don't know Research and Development's department code, you use the department name found in the `DEPARTMENT` table in order to find the row that stores Research and Development's code, which is 'RND'. Conceptually, visualize the join operation to occur as follows:

1. The conditional expression `DNAME = 'RESEARCH AND DEVELOPMENT'` references one row from the `DEPARTMENT` table; i.e., the 'RND' row.
2. Now that the RND code is known, this row in the `DEPARTMENT` table is joined with the rows in the `REPORT` table that have `DEPT = RND`. The joined table represents the titles of the reports submitted by authors from the Research and Development department.

As the next example illustrates, more than two tables can be joined together.

Example: Provide a list of salaries paid to those people in the Product Division who contributed reports to the Corporate Library. The report should include the author's name, department name, and annual salary.

You would enter this statement:

```
SELECT  AUTHOR, DNAME, SALARY
FROM    REPORT, DEPARTMENT, EMPLOYEE
WHERE   DEPARTMENT.DIV = 'PROD'
        AND REPORT.DEPT = DEPARTMENT.DEPT
        AND REPORT.DEPT = EMPLOYEE.DEPT ;
```

Syntax Notes:

- The order of the joins in the WHERE clause is not important.
- The three tables to be joined are listed after FROM.
- The first AND statement (in WHERE clause) is the condition for joining the REPORT and DEPARTMENT tables.
- The second AND statement (in WHERE clause) is the condition for joining the REPORT and EMPLOYEE tables.
- While department code happens to be a column which all three tables have in common, it would be possible to join two tables with a common column, and the other two tables with a different common column, such as ENAME in the EMPLOYEE table and AUTHOR in the REPORT table. (The latter would not be as efficient, nor as reliable, so department name was chosen instead.)

The results would be:

| AUTHOR | DNAME | SALARY |
|------------------|-----------------------------|--------|
| Jackson, Herbert | Research and Development | 30000 |
| Sanchez, Carla | Product Marketing and Sales | 35000 |
| Smith, Roberta | Research and Development | 25000 |
| Aster, John A. | Product Marketing and Sales | 32000 |

In this example, data from three tables (REPORT, DEPARTMENT, EMPLOYEE) are joined together.

Conceptually, the DEPARTMENT table references the rows that contain PROD; this gives us the departments in the Product Division. The departments in the Product Division (RND, MFG, CSS, MKT) are matched against the departments in the DEPT column of the REPORT table. The tables are joined for the Research and Development (RND) and Product Marketing and Sales (MKT) departments. This yields an intermediate table containing all the columns from both the DEPARTMENT and REPORT tables for RND and MKT rows.

This intermediate table is joined with the EMPLOYEE table, based on the second join condition REPORT.DEPT = EMPLOYEE.DEPT to form a combination of columns from all 3 tables, for the matching rows.

Finally, the `SELECT` clause indicates which columns in the intermediate joined table that you want displayed. Thus the author, department name, and annual salary are shown as in the above example.

As a final point, the order in which you place the conditions in the `WHERE` clause does not affect the way Taxis accesses the data. Taxis contains an “*optimizer*” in its underlying software, which chooses the best access path to the data based on factors such as index availability, size of tables involved, number of unique values in an indexed column, and other statistical information. Thus, the results would not be affected by writing the same query in the following order:

```
SELECT  AUTHOR, DNAME, SALARY
FROM    REPORT, DEPARTMENT, EMPLOYEE
WHERE   REPORT.DEPT = EMPLOYEE.DEPT
        AND REPORT.DEPT = DEPARTMENT.DEPT
        AND DEPARTMENT.DIV = 'PROD' ;
```

6.2 Nesting Queries

At times you may wish to retrieve rows in one table based on conditions in a related table. For example, suppose Personnel needed to call in any employees in the Information Division receiving only partial benefits, to discuss options for upgrading to the full benefit program. To answer this query, you have to retrieve the names of all departments in the Information Division, found in the `DEPARTMENT` table, and then the employees with partial benefits in the Information Division departments, found in the `EMPLOYEE` table.

In other situations, you may want to formulate a query from one table that required you to make two passes through the table in order to obtain the desired results. For example, you may want to retrieve a list of staff members earning a salary higher than Jackson, but you don’t know Jackson’s salary. To answer this query, you first find Jackson’s salary; then you compare the salary of each staff member to his.

One approach is to develop a *subquery*, which involves embedding a query (`SELECT-verbFROM`“-WHERE block) within the `WHERE` clause of another query. This is sometimes referred to as a “*nested query*”.

The format of a nested query is:

```
SELECT  column-name1 [,column-name2]
FROM    table-name
WHERE   column-name IN
        (SELECT  column-name
         FROM    table-name
         WHERE   search-condition) ;
```

Syntax Notes:

- The first `SELECT-verbFROM`“-WHERE block is the outer query.
- The second `SELECT-verbFROM`“-WHERE block in parentheses is the subquery.

- The `IN` operator is normally used if the inner query returns many rows and one column.

Command Discussion

Here are some points concerning the use of nested queries:

1. The above statement contains two `SELECT-FROM-WHERE` blocks. The portion in parentheses is called the subquery. The subquery is evaluated first; then the outer query is evaluated based on the result of the subquery. In effect, the nested query can be looked at as being equivalent to:

```
SELECT    column-name1 [,column-name2] ...
FROM      table-name
WHERE     column-name IN (set of values from the subquery) ;
```

where the set of values is determined from the inner `SELECT-FROM-WHERE` block.

2. The `IN` operator is used to link the outer query to the subquery when the subquery returns a set of values (one or more). Other comparison operators, such as `<`, `>`, `=`, etc., can be used to link an outer query to a subquery when the subquery returns a single value.
3. The subquery must have only a single column or expression in the `SELECT` clause, so that the resulting set of values can be passed back to the next outer query for evaluation.
4. You are not limited to one subquery. Though it isn't advised, there could be as many as 16 levels of subqueries, with no fixed limitation except limits of memory and disk-space on the machine in use. Any of the operators (`IN`, `=`, `<`, `>`, etc.) can be used to link the subquery to the next higher level.

Example: List the names of all personnel in the Information Division by entering this statement:

```
SELECT    ENAME
FROM      EMPLOYEE
WHERE     DEPT IN
        (SELECT    DEPT
         FROM      DEPARTMENT
         WHERE     DIV = 'INFO' ) ;
```

Parentheses are placed around the subquery, as shown below the outer `WHERE` clause.

The results are:

```
ENAME
Chapman, Margaret
Dedrich, Franz
Krinski, Wanda
Peters, Robert
```

To understand how this expression retrieves its results, work from the bottom up in evaluating the `SELECT` statement. In other words, the subquery is evaluated first. This results in a set of values that can be used as the basis for the outer query. The innermost `SELECT` block retrieves the following set of department codes, as departments in the Information ('INFO') Division: ISM, LIB, SPI.

In the outermost `SELECT` block, the `IN` operator tests whether any department code in the `EMPLOYEE` table is contained in the set of department codes values retrieved from the inner `SELECT` block; i.e., ISM, LIB, or SPI.

In effect, the outer `SELECT` block is equivalent to:

```
SELECT  ENAME
FROM    EMPLOYEE
WHERE   DEPT IN ( 'ISM', 'LIB', 'SPI' ) ;
```

where the values in parentheses are values from the subquery.

Thus, the employee names Chapman, Dedrich, Krinski and Peters are retrieved.

Subqueries can be nested several levels deep within a query, as the next example illustrates.

Example: Acme Industrial's ink sales are up, and management wishes to reward everyone in the division(s) most responsible. List the names of all employees in any division whose personnel have contributed reports on ink to the corporate library, along with their department and benefit level.

Use this statement:

```
SELECT  ENAME, DEPT, BENEFITS
FROM    EMPLOYEE
WHERE   DEPT IN
      (SELECT  DEPT
       FROM    DEPARTMENT
       WHERE   DIV IN
            (SELECT  DIV
             FROM    DEPARTMENT
             WHERE   DEPT IN
                  (SELECT  DEPT
                   FROM    REPORT
                   WHERE   TITLE LIKE 'ink') ) ) ;
```

`IN` is used for each subquery since in each case it is possible to retrieve several values. You could use `'='` instead where you knew only one value would be retrieved; e.g. where you wanted only the division with the greatest number of reports rather than all divisions contributing reports.

Results of the above nested query are:

| ENAME | DEPT | BENEFITS |
|------------------|------|----------|
| Aster, John A. | MKT | FULL |
| Jackson, Herbert | RND | FULL |

| | | |
|-----------------|-----|------|
| Sanchez, Carla | MKT | FULL |
| Smith, Roberta | MKT | PART |
| Jones, David | RND | FULL |
| Washington, G. | MFG | FULL |
| Ferrer, Miguel | CSS | FULL |
| Brown, Penelope | MKT | FULL |

Again, remember that a nested query is evaluated from the bottom up; i.e., from the innermost query to the outermost query. First, a text search is done (`TITLE LIKE 'INK'`) of report titles from the `REPORT` table. Two such titles are located: “Disappearing Ink” by Herbert Jackson from Research and Development (RND), and “Ink Promotional Campaign” by Carla Sanchez from Product Marketing and Sales (MKT). Thus the results of the innermost query produces a list of two department codes: RND and MKT.

Once the departments are known, a search is done of the `DEPARTMENT` table, to locate the division or divisions to which these departments belong. Both departments belong to the Product Division (PROD); thus the results of the next subquery produces one item: PROD.

A second pass is made through the same table, `DEPARTMENT`, to find all departments which belong to the Product Division. This search produces a list of four Product Division departments: MKT, RND, MFG, and CSS, adding Manufacturing as well as Customer Support and Service to the list.

This list is passed to the outermost query so that the `EMPLOYEE` table may be searched for all employees in those departments. The final listing is retrieved, as above.

Here is another example specifically designed to illustrate the use of a subquery making two passes through the same table to find the desired results.

Example: List the names of employees who have salaries greater than that of Herbert Jackson. Assume you do not know Jackson’s salary.

Enter this statement:

```
SELECT  ENAME, SALARY
FROM    EMPLOYEE
WHERE   SALARY >
        (SELECT  SALARY
         FROM    EMPLOYEE
         WHERE   ENAME = 'Jackson, Herbert') ;
```

The compare operator `>` can be used (as could `=` and other compare operators) where a single value only will be returned from the subquery.

Using the sample information in our `EMPLOYEE` table, the results are as follows:

| ENAME | SALARY |
|------------------|--------|
| Aster, John A. | 32000 |
| Barrington, Kyle | 45000 |
| Price Stella | 42000 |
| Sanchez, Carla | 35000 |

The subquery searches the `EMPLOYEE` table and returns the value 30000, the salary listed for Herbert Jackson. Then the outer `SELECT` block searches the `EMPLOYEE` table again to retrieve all employees with `SALARY > 30000`. Thus the above employees with higher salaries are retrieved.

6.3 Forming Complex Queries

The situations covered in this section are more technical than most end users have need to conceptualize. However, a system administrator may require such complex query structures to efficiently obtain the desired results.

6.3.1 Joining a Table to Itself

In some situations, you may find it necessary to join a table to itself, as though you were joining two separate tables. This is referred to as a *self join*. In the self join, the combined result consists of two rows from the same table.

For example, suppose that within the `EMPLOYEE` table, personnel are assigned a `RANK` of “STAFF”, “DHEAD”, and so on. To obtain a list of employees that includes employee name and the name of his or her department head requires the use of a self join.

To join a table to itself, the table name appears twice in the `FROM` clause. To distinguish between the appearance of the same table name, a temporary name, called an *alias* or a *correlation name*, is assigned to each mention of the table name in the `FROM` clause. The form of the `FROM` clause with an alias is:

```
FROM    table-name [alias1] [,table-name [alias2] ] ...
```

To help clarify the meaning of the query, the alias can be used as a qualifier, in the same way that the table name serves as a qualifier, in `SELECT` and `WHERE` clauses.

Example: As part of an analysis of Acme’s salary structure, you want to identify the names of any regular staff who are earning more than a department head.

Enter this query:

```
SELECT    STAFF.ENAME, STAFF.SALARY
FROM      EMPLOYEE DHEAD, EMPLOYEE STAFF
WHERE     DHEAD.RANK = 'DHEAD' AND STAFF.RANK = 'STAFF'
AND       STAFF.SALARY > DHEAD.SALARY ;
```

Using a sampling of information from the `EMPLOYEE` table, we would get these results:

| ENAME | SALARY |
|----------------|--------|
| Sanchez, Carla | 35000 |

In this query, the `EMPLOYEE` table, using the alias feature, is treated as two separate tables named `DHEAD` and `STAFF`, as shown here (in shortened form):

| DHEAD Table | | | | | | STAFF Table | | | | | |
|-------------|---------|------|-------|------|--------|-------------|---------|------|-------|------|--------|
| EID | ENAME | DEPT | RANK | BEN | SALARY | EID | ENAME | DEPT | RANK | BEN | SALARY |
| 101 | Aster | MKT | STAFF | FULL | 32000 | 101 | Aster | MKT | STAFF | FULL | 32000 |
| 109 | Brown | MKT | DHEAD | FULL | 37500 | 109 | Brown | MKT | DHEAD | FULL | 37500 |
| 103 | Chapman | LIB | STAFF | PART | 22000 | 103 | Chapman | LIB | STAFF | PART | 22000 |
| 110 | Krinski | LIB | DHEAD | FULL | 32500 | 110 | Krinski | LIB | DHEAD | FULL | 32500 |
| 106 | Sanchez | MKT | STAFF | FULL | 35000 | 106 | Sanchez | MKT | STAFF | FULL | 35000 |

Now the join operation can be made use of, as if there were two separate tables, evaluated as follows.

First, using the following compound condition:

```
DHEAD.RANK = 'DHEAD' AND STAFF.RANK = 'STAFF'
```

each department head record (Brown, Krinski) in the `DHEAD` table is joined with each staff record (Aster, Chapman, Sanchez) from the `STAFF` table to form the following intermediate result:

| DHEAD Table | | | | | | STAFF Table | | | | | |
|-------------|---------|------|-------|------|--------|-------------|---------|------|-------|------|--------|
| EID | ENAME | DEPT | RANK | BEN | SALARY | EID | ENAME | DEPT | RANK | BEN | SALARY |
| 109 | Brown | MKT | DHEAD | FULL | 37500 | 101 | Aster | MKT | STAFF | FULL | 32000 |
| 109 | Brown | MKT | DHEAD | FULL | 37500 | 103 | Chapman | LIB | STAFF | PART | 22000 |
| 109 | Brown | MKT | DHEAD | FULL | 37500 | 106 | Sanchez | MKT | STAFF | FULL | 35000 |
| 110 | Krinski | LIB | DHEAD | FULL | 32500 | 101 | Aster | MKT | STAFF | FULL | 32000 |
| 110 | Krinski | LIB | DHEAD | FULL | 32500 | 103 | Chapman | LIB | STAFF | PART | 22000 |
| 110 | Krinski | LIB | DHEAD | FULL | 32500 | 106 | Sanchez | MKT | STAFF | FULL | 35000 |

Notice that every department head row is combined with each staff record.

Next, using the condition:

```
STAFF.SALARY > DHEAD.SALARY
```

for each row of the joined table, the salary value from the `STAFF` portion is compared with the corresponding salary value from the `DHEAD` portion. If `STAFF.SALARY` is greater than `DHEAD.SALARY`, then `STAFF.ENAME` and `STAFF.SALARY` are retrieved in the final table.

The only row in the joined table satisfying this condition of staff salary being greater than department head salary is the last one, where Carla Sanchez from Marketing, at a salary of \$35,000, is earning more than Wanda Krinski, as department head for the Corporate Library, at a salary of \$32,500.

6.3.2 Correlated Subqueries

All the previous examples of subqueries evaluated the innermost query completely before moving to the next level of the query. Some queries, however, cannot be completely evaluated before the outer, or main,

query is evaluated. Instead, the search condition of a subquery depends on a value in each row of the table named in the outer query. Therefore, the subquery is evaluated repeatedly, once for each row selected from the outer table. This type of subquery is referred to as a *correlated subquery*.

Example: Retrieve the name, department, and salary, of any employee whose salary is above average for his or her department.

Enter this query:

```
SELECT    POSSIBLE.ENAME, POSSIBLE.DEPT, POSSIBLE.SALARY
FROM      EMPLOYEE POSSIBLE
WHERE     SALARY >
        (SELECT    AVG (SALARY)
         FROM      EMPLOYEE AVERAGE
         WHERE     POSSIBLE.DEPT = AVERAGE.DEPT) ;
```

Syntax Notes:

- The outer SELECT-FROM-WHERE block is the main query.
- The inner SELECT-FROM-WHERE block in parentheses is the subquery.
- POSSIBLE (following EMPLOYEE in the outer query) and AVERAGE (following EMPLOYEE in the subquery) are alias table names for the EMPLOYEE table, so that the information may be evaluated as though it comes from two different tables.

It results in:

| ENAME | DEPT | SALARY |
|-----------------|------|--------|
| Krinski, Wanda | LIB | 32500 |
| Brown, Penelope | MKT | 37500 |
| Sanchez, Carla | MKT | 35000 |
| Jones, David | RND | 37500 |

The column AVERAGE.DEPT correlates with POSSIBLE.DEPT in the main, or outer, query. In other words, the average salary for a department is calculated in the subquery using the department of each employee from the table in the main query (POSSIBLE). The subquery computes the average salary for this department and then compares it with a row in the POSSIBLE table. If the salary in the POSSIBLE table is greater than the average salary for the department, then that employee's name, department, and salary are displayed.

The process of the correlated subquery works in the following manner. The department of the first row in POSSIBLE is used in the subquery to compute an average salary. Let's take Krinski's row, whose department is the corporate library (LIB). In effect, the subquery is:

```
SELECT    AVG (SALARY)
FROM      EMPLOYEE AVERAGE
WHERE     'LIB' = AVERAGE.DEPT ;
```

LIB is the value from the first row in POSSIBLE, as alias for EMPLOYEE.

This pass through the subquery results in a value of \$27,250, the average salary for the LIB dept. In the outer query, Krinski's salary of \$32,500 is compared with the average salary for LIB; since it is greater, Krinski's name is displayed.

This process continues; next, Aster's row in POSSIBLE is evaluated, where MKT is the department. This time the subquery is evaluated as follows:

```
SELECT    AVG (SALARY)
FROM      EMPLOYEE AVERAGE
WHERE     'MKT' = AVERAGE.DEPT ;
```

The results of this pass through the subquery is an average salary of \$34,833 for MKT, the Product Marketing and Sales Department. Since Aster has a salary of \$32,000, a figure lower than the average, this record is not displayed.

Every department in POSSIBLE is examined in a similar manner before this subquery is completed.

6.3.3 Subquery Using EXISTS

There may be situations in which you are interested in retrieving records where there exists at least one row that satisfies a particular condition. For example, the resume records stored in the RESUME table may include some individuals who are already employed at Acme Industrial and so are entered in the EMPLOYEE table. If you wanted to know which employees were seeking new jobs at the present time, an existence test using the keyword EXISTS can be used to answer such a query.

This type of query is developed with a subquery. The WHERE clause of the outer query is used to test the existence of rows that result from a subquery. The form of the WHERE clause that is linked to the subquery is:

```
WHERE [NOT] EXISTS (subquery)
```

This clause is satisfied if there is at least one row that would be returned by the subquery. If so, the subquery does not return any values; it just sets an indicator value to true. On the other hand, if no elements satisfy the condition, or the set is empty, the indicator value is false.

The subquery should return a single column only.

Example: Retrieve a list of Acme employees who have submitted resumes to personnel for a different job placement.

Enter this query:

```
SELECT    EID, ENAME
FROM      EMPLOYEE
WHERE     EXISTS
          (SELECT RNAME
```

```

FROM    RESUME
WHERE   EMPLOYEE.ENAME = RESUME.RNAME) ;

```

The results are:

```

EID  ENAME
107  Smith, Roberta
113  Ferrer, Miguel

```

In this query, the subquery cannot be evaluated completely before the outer query is evaluated. Instead, we have a correlated subquery. For each row in `EMPLOYEE`, a join of `EMPLOYEE` and `RESUME` tables is performed (even though `RESUME` is the only table that appears in the subquery's `FROM` clause) to determine if there is a resume name in `RESUME` that matches a name in `EMPLOYEE`.

For example, for the first row in the `EMPLOYEE` table (`ENAME = 'Smith, Roberta'`) the subquery evaluates as “true” if at least one row in the `RESUME` table has `RNAME = 'Smith, Roberta'`; otherwise, the expression evaluates as “false”. Since there is a row in `RESUME` with `RNAME = 'Smith, Roberta'`, the expression is true and Roberta Smith's row is displayed. Each row in `EMPLOYEE` is evaluated in a similar manner.

The following is an example of the interim join (in shortened form) between the `EMPLOYEE` and `RESUME` Tables, for the above names which satisfied the search requirement by appearing in both tables:

| EMPLOYEE Table | | | RESUME Table | | | |
|----------------|----------------|------|--------------|----------------|----------|----------------------|
| EID | ENAME | DEPT | RES_ID | RNAME | JOB | EXISTS (subquery) |
| 107 | Smith, Roberta | RND | R406 | Smith, Roberta | Engineer | TRUE |
| 113 | Ferrer, Miguel | CSS | R425 | Ferrer, Miguel | Analyst | TRUE |

Note in this example that there is no key ID field connecting the two tables; therefore the character field for name is being used to join the two tables, which might have been entered differently and therefore is not an altogether reliable join. This indicates that such a search is an unusual rather than a usual action.

Such a search would be a good opportunity to use a Metamorph `LIKE` qualifier rather than a straight join on a column as above, where `ENAME` must match exactly `RNAME`. A slightly more thorough way of searching for names appearing in both tables which were not necessarily intended to be matched exactly would use Metamorph's approximate pattern matcher, indicated by a percent sign `%` preceding the name. For example:

```

SELECT  EID, ENAME
FROM    EMPLOYEE
WHERE   EXISTS
        (SELECT *
         FROM  RESUME
         WHERE EMPLOYEE.ENAME LIKE '%' + RESUME.RNAME) ;

```

In this example a name approximately like each `RNAME` in the `RESUME` table would be compared to each `ENAME` in the `EMPLOYEE` table, increasing the likelihood of a match. (String concatenation is used to

append the name found in the resume table to the percent sign (%) which signals the approximate pattern matcher XPM.)

Often, a query is formed to test if no rows are returned in a subquery. In this case, the following form of the existence test is used:

```
WHERE NOT EXISTS (subquery)
```

Example: List any authors of reports submitted to the online corporate library who are not current employees of Acme Industrial. To find this out we would need to know which authors listed in the REPORT table are not entered as employees in the EMPLOYEE table.

Use this query:

```
SELECT  AUTHOR
FROM    REPORT
WHERE   NOT EXISTS
        (SELECT *
         FROM  EMPLOYEE
         WHERE EMPLOYEE.ENAME = REPORT.AUTHOR) ;
```

which would likely result in a list of former employees such as:

```
AUTHOR
Acme, John Jacob Snr.
Barrington, Cedrick II.
Rockefeller, George G.
```

Again, we have an example of a correlated subquery. Below is illustrated (in shortened form) how each row which satisfied the search requirement above in REPORT is evaluated with the records in EMPLOYEE to determine which authors are not (or are no longer) Acme employees.

| REPORT Table | EMPLOYEE Table | EXISTS |
|-------------------|-------------------------|--------|
| TITLE | AUTHOR | |
| Company Origin | Acme, John Jacob Snr. | FALSE |
| Management Art | Barrington, Cedrick II. | FALSE |
| Financial Control | Rockefeller, George G. | FALSE |

In this example each of the above authors from the REPORT Table are tested for existence in the EMPLOYEE Table. When they are not found to exist there it returns a value of FALSE. Since the query condition in the WHERE clause is that it NOT EXISTS, this changes the false value to true, and these rows are displayed.

For each of the queries shown in this section, there are probably several ways to obtain the same kind of result. Some correlated subqueries can also be expressed as joins. These examples are given not so much as the only definitive way to state these search requests, but more so as to give a model for what kinds of things are possible.

6.4 Virtual Fields

To improve the capabilities of Taxis, especially with regard to Metamorph searching multiple fields we implemented the concept of virtual fields. This allows you to treat the concatenation of any number of text fields as a single field. As a single field you can create an index on the fields, search the fields, and perform any other operation allowable on a field. Concatenation is represented by the `\` operator. For example:

```
SELECT TITLE
FROM PAPERS
WHERE ABSTRACT\BODY LIKE 'ink coloration';
```

would display the title of all papers whose abstract or body matched the query `ink coloration`. By itself this is helpful, but the real change is that you could create an index on this virtual field as follows:

```
CREATE METAMORPH INDEX IXMMABSOD ON PAPERS (ABSTRACT\BODY);
```

which could greatly improve the performance of this query. You can create any type of index on a virtual field, although it is important to remember that for non Metamorph indices the sum of the fields should not exceed 2048 bytes. If your keys are text fields this method allows you to create a unique index across several fields.

6.5 Column Aliasing

Similar to the ability to alias the name of a table in the from clause it is also possible to alias column names. An alias can have up to 35 characters (case is significant).

This has several possible uses. One is simply to produce a more informative report, for example:

```
SELECT COUNT(*) EMPLOYEES
FROM EMPLOYEES;
```

might produce the following output

| |
|-----------------|
| EMPLOYEES 42 |
|-----------------|

Another important use is when using the create table as select statement. This allows you to rename a field, or to name a calculated field.

```
CREATE TABLE INVENTORY AS
SELECT PROD_ID, SALES * 3 MAX_LEVEL, SALES MIN_LEVEL
FROM SALES;
```

Would create a new table with three fields, PROD_ID, MAX_LEVEL, and MIN_LEVEL.

This chapter has illustrated various complex query constructions possible with Taxis, and has touched on the use of Metamorph in conjunction with standard SQL queries. The next chapter will explain Metamorph query language in depth and give examples of its use in locating relevant narrative text.

Chapter 7

Intelligent Text Search Queries

This manual has concentrated so far on the manipulation of fields within a relational database. As a text information server Taxis provides all the capabilities one would expect from a traditional RDBMS.

Taxis was also designed to incorporate large quantities of narrative full text stored within the database. This is evidenced by its data types as presented in Chapter 3, *Table Definition*. When textual content becomes the focus of the search, the emphasis shifts from one of document management to one of research. Taxis has embodied within it special programs geared to accomplish this.

Metamorph was originally a stand-alone program designed to meet intelligent full text retrieval needs on full text files. Since 1986 it has been used in a variety of environments where concept oriented text searching is desired. Now within the `LIKE` clause, all of what is possible on full text information with Metamorph is possible with Taxis, within the framework of a relational database.

Metamorph is covered in a complete sense in its own section in this manual and can be studied of itself. Please refer to the *Metamorph Intelligent Text Query Language* sections for a full understanding of all Metamorph's theory and application.

This chapter deals with the use of Metamorph within the `LIKE` portion of the `WHERE` clause. Taxis can accomplish any Metamorph search through the construction of a `SELECT-FROM-WHERE` block, where the Metamorph query is enclosed in single quotes 'query' following `LIKE`.

7.1 Types of Text Query

There are two primary types of text query that can be performed by Taxis. The first form is used to find those records which match the query, and is used when you only care if the record does or does not match. The second form is used when you want to rank the results and produce the best answers first.

The first type of query is done with `LIKE`, or `LIKE3` to avoid post processing if the index can not fully resolve the query. The ranking queries are done with `LIKEP` or `LIKER`. `LIKER` is a faster, and less precise ranking figure than the one returned by `LIKEP`. The ranking takes into account how many of the query terms are present, as well as their weight, and for `LIKEP`, how close together the words are, and where in the document they occur. Most queries will use `LIKE` or `LIKEP`, with `LIKE3` and `LIKER` used in special

circumstances when you want to avoid the post-processing that would fully resolve the query.

There are also two forms of Metamorph index, the Metamorph inverted index and the Metamorph index. The inverted form contains additional information which allows phrases to be resolved and `LIKEP` rankings to be calculated entirely using the index. This improved functionality comes at a cost in terms of space.

If your queries are single word `LIKE` queries then the Metamorph index has all the information needed, so the additional overhead of the inverted index is not needed.

7.2 Definition of Metamorph Terms

Query: A Metamorph Query is the question or statement of search items to be matched in the text within specified beginning and ending delimiters. A Query is comprised of one or more search items which can be of different types, and a “within delimiter” specification which is either understood or expressed. In Taxis the Metamorph Query refers to what is contained within single quotes ‘query’ following `LIKE` in the `WHERE` clause.

Hit: A Hit is the text Metamorph retrieves in response to a query, whose meaning matches the Query to the degree specified.

Search Item: A Search Item is an English word or a special expression inside a Metamorph Query. A word is automatically processed using certain linguistic rules. Special searches are signaled with a special character leading the item, and are governed respectively by the rules of the pattern matcher invoked.

Set: A Set is the group of possible strings a pattern matcher will look for, as specified by the Search Item. A Set can be a list of words and word forms, a range of characters or quantities, or some other class of possible matches based on which pattern matcher Metamorph uses to process that item.

Intersection: A portion of text where at least one member of two Sets each is matched.

Delimiters: Delimiters are repeating patterns in the text which define the bounds within which search items are found in proximity to each other. These patterns are specified as regular expressions. A within operator is used to specify delimiters in a Metamorph Query.

Intersection Quantity: The number of unions of sets existing within the specified Delimiters. The maximum number of Intersections possible for any given Query is the maximum number of designated Sets minus one.

Hits can have varying degrees of relevance based on the number of set intersections occurring within the delimited block of text, definition of proximity bounds, and weighting of search items for inclusion or exclusion.

Intersection quantity, Delimiter bounds, and Logic weighting can be adjusted by the user as part of Metamorph Query specification.

7.3 Adjusting Linguistic Controls

Concept sets can be edited to include special vocabulary, acronyms, and slang. There is sufficient vocabulary intelligence off the shelf so that editing is not required to make good use of the program immediately upon installation. However, such customization is encouraged to keep online research in rapport with users' needs, especially as search routines and vocabulary evolve.

A word need not be "known" by Metamorph for it to be processed. The fact of a word having associations stored in the Thesaurus makes abstraction of concept possible, but is not required to match word forms. Such word stemming knowledge is inherent. And, any string of characters can be matched exactly as entered.

You can edit the special word lists Metamorph uses to process English if you wish. As it may not be immediately apparent to what degree these word lists may affect general searching, it is cautioned that such editing be used sparingly and with the wisdom of experience. Even so, what Metamorph deems to be Noise, Prefixes, and Suffixes is all under user control.

See the Metamorph portion of this manual for a complete explanation of all these terms and other background information.

7.4 Constructing a Metamorph Query

The following types of searches are all possible within a Metamorph Query, as contained in single quotes following the `LIKE` portion of the `WHERE` clause, in any `SELECT-FROM-WHERE` block.

7.4.1 Keyword Search

Your search can be as simple as a single word or string. If you want references to do with dogs, type in the word "dog".

Example: Let's say Corporate Legal Support maintains a table called `CODES` which includes full text of the local ordinances of the town in which Acme Industrial has its headquarters. The full text field of each ordinance is stored in a column called `BODY`.

To find ordinances containing references to dogs, the `WHERE` clause takes this form:

```
WHERE column-name LIKE 'metamorph-query'
```

You can put any Metamorph query in the quotes ('metamorph-query') although you would need to escape a literal ' with another ' by typing '' , if you want the character ' to be part of the query.

Using Metamorph's defaults, a sentence in the body of the ordinance text will be sought which contains a match to the query. Whatever is dictated in the `SELECT` portion of the statement is what will be displayed. All outer logic applies, so that multiple queries can be sought through use of `AND`, `OR`, `NOT`, and so on. See Chapter 4, especially the section, *Additional Comparison Operators*, for a complete understanding of how the `LIKE` clause fits into a `SELECT-FROM-WHERE` statement.

In this example, the WHERE clause would look like this:

```
WHERE BODY LIKE 'dog'
```

When Taxis executes the search, ordinances whose bodies contain matching sentences would be retrieved. An example of a qualifying sentence would be:

```
DOG:  any member of the canine family.
```

And this sentence:

```
It shall be unlawful and a nuisance for any DOG owner to  
permit a dog to be free of restraint in the city.
```

An English word entered in a Metamorph Query retrieves occurrences of forms of that word in both lower and upper case, regardless of how it was entered; i.e., the default keyword search is case insensitive.

Each matched sentence is called a *HIT*. Metamorph locates all such *hits* containing “dog” and any other “dog” word forms adhering to the linguistic settings in place. There would normally be quite a few hits for a common keyword query like this.

7.4.2 Refining a Query

To refine a query, thereby further qualifying what is judged a hit, add any other keywords or concepts which should appear within the same concept grouping.

Example:

```
WHERE BODY LIKE 'dog fine'
```

Fewer hits will be retrieved than when only one search item is entered (i.e., “dog”), as you are requiring both “dog” and “fine” to occur in the same sentence. This sentence would qualify:

```
The owner of any DOG who permits such a dog to be free of  
restraint in violation of Section 4.2 of this article shall  
pay a FINE of not less than twenty-five dollars.
```

You may enter as many query items as you wish, to qualify the hits to be found.

Example:

```
WHERE BODY LIKE 'dog owner vaccination city'
```

Such a query locates this sentence:

```
Every veterinarian who VACCINATES any cat or DOG within the
CITY limits shall issue a certification of vaccination to
such OWNER.
```

You needn't sift through references which seem too broad or too numerous. Refine your query so it produces only what you judge to be relevant to the goal of your search.

7.4.3 Adjusting Proximity Range by Specifying Delimiters

By default Taxis considers the entire field to be a hit when the full text is retrieved.

If you want your search items to occur within a more tightly constrained proximity range this can be adjusted. If you are using Vortex you will need to allow within operators which are disabled by default due to the extra processing required.

Add a “within” operator to your query syntax; “w/line” indicates a line; “w/para” indicates a paragraph; “w/sent” indicates a sentence; “w/all” indicates the entire field; “w/#” indicates # characters. The default proximity is “w/all”.

Example: Using the legal ordinance text, we are searching the full text bodies of those ordinances for controls issued about dogs. The following query uses sentence proximity to qualify its hits.

```
WHERE BODY LIKE 'dog control w/sent'
```

This sentence qualifies as a hit because “control” and “dogs” are in the same sentence.

```
Ordinances provide that the animal CONTROL officer takes
possession of DOGS which are free of restraint.
```

Add a within operator to the Metamorph query to indicate both stated search items must occur within a single line of text, rather than within a sentence.

```
WHERE BODY LIKE 'dog control w/line'
```

The retrieved concept group has changed from a sentence to a line, so “dog” and “control” must occur in closer proximity to each other. Now the line, rather than the sentence, is the hit.

```
CONTROL officer takes possession of DOGS
```

Expanding the proximity range to a paragraph broadens the allowed distance between located search words.

```
WHERE BODY LIKE 'dog control w/para'
```

The same query with a different “within” operator now locates this whole paragraph as the hit:

The mayor, subject to the approval of the city council, shall appoint an animal CONTROL officer who is qualified to perform the duties of an animal control officer under the laws of this state and the ordinances of the city. This officer shall take possession of any DOG which is free of restraint in the city.

The words “control” and “dog” span different lines and different sentences, but are within the same paragraph.

These “within” operators for designating proximity are also referred to as delimiters. Any delimiter can be designed by creating a regular expression using REX syntax which follows the “w/”. Anything following “w/” that is not one of the previously defined special delimiters is assumed to be a REX expression. For example:

```
WHERE BODY LIKE 'dog control w/\RSECTION'
```

What follows the ‘w/’ now is a user designed REX expression for sections. This would work on text which contained capitalized headers leading with “SECTION” at the beginning of each such section of text.

Delimiters can also be expressed as a number of characters forward and backwards from the located search items. For example:

```
WHERE BODY LIKE 'dog control w/500'
```

In this example “dog” and “control” must occur within a window of 500 characters forwards and backwards from the first item located.

More often than not the beginning and ending delimiters are the same. Therefore if you do not specify an ending delimiter (as in the above example), it will be assumed that the one specified is to be used for both. If two expressions are specified, the first will be beginning, the second will be ending. Specifying both would be required most frequently where special types of messages or sections are used which follow a prescribed format.

Another factor to consider is whether you want the expression defining the text unit to be included inside that text unit or not. For example, the ending delimiter for a sentence obviously belongs with the hit. However, the beginning delimiter is really the end of the last sentence, and therefore should be excluded.

Inclusion or exclusion of beginning and ending delimiters with the hit has been thought out for the defaults provided with the program. However, if you are designing your own beginning and ending expressions, you may wish to specify this.

Delimiter Syntax Summary

```
w/{abbreviation}  
or
```

```

w/{number}
or
w/{expression}
or
W/{expression}
or
w/{expression} W/{expression}
or
W/{expression} w/{expression}
or
w/{expression} w/{expression}
or
W/{expression} W/{expression}

```

Rules of Delimiter Syntax

- The above can be anywhere in a Metamorph query, and is interpreted as “within {the following delimiters}”.
- Accepted built-in abbreviations following the slash ‘/’ are:

Table 7.1: Metamorph delimiter abbreviations

| Abbreviation | Meaning |
|--------------|--------------------------|
| line | within a line |
| sent | within a sentence |
| para | within a paragraph |
| page | within a page |
| all | within a field |
| NUMBER | within NUMBER characters |

| REX Expression | Meaning |
|--------------------------------|---|
| \$ | 1 new line |
| [^\digit\upper][.?!][\space'"] | not a digit or upper case letter, then a period, question, or exclamation point, then any space character, single or double quote |
| \x0a=\space+ | a new line + some space |
| \x0c | form feed for printer output |

- A number following a slash ‘/’ means the number of characters before and after the first search item found. Therefore “w/250” means “within a proximity of 250 characters”. When the first occurrence of a valid search item is found, a window of 250 characters in either direction will be used to

determine if it is a valid hit. The implied REX expression is: “. { , 250 }” meaning “250 of any character”.

- If what follows the slash ‘/’ is not recognized as a built-in, it is assumed that what follows is a REX expression.
- If one expression only is present, it will be used for both beginning and ending delimiter. If two expressions are present, the first is the beginning delimiter, the second the ending delimiter. The exception is within-*N* (e.g. “w/250”), which always specifies both start and end delimiters, overriding any preceding “w/”.
- The use of a small ‘w’ means to exclude the delimiters from the hit.
- The use of a capital ‘W’ means to include the delimiters in the hit.
- Designate small ‘w’ and capital ‘W’ to exclude beginning delimiter, and include ending delimiter, or vice versa. Note that for within-*N* queries (e.g. “w/250”), the “delimiter” is effectively always included in the hit, regardless of the case of the w.
- If the same expression is to be used, the expression need not be repeated. Example: “w/[. ? !] W/” means to use an ending punctuation character as both beginning and end delimiter, but to exclude the beginning delimiter from the hit, and include the end delimiter in the hit.

7.4.4 Using Set Logic to Weight Search Items

Set Logic and Intersections Defined

Any search item entered in a query can be weighted for determination as to what qualifies as a hit.

All search items indicate to the program a set of possibilities to be found. A keyword is a set of valid derivations of that word’s root (morpheme). A concept set includes a list of equivalent meaning words. A special expression includes a range of strings that could be matched.

Therefore, whatever weighting applies to a search item applies to the whole set, and is referred to as “set logic”.

The most usual logic in use is “AND” logic. Where no other weighting is given, it is understood that all entered search items have equal weight, and you want each one to occur in the targeted hit.

Here is an example of a typical query, where no special weighting has been assigned:

```
WHERE BODY LIKE 'mayor powers duties city'
```

The query equally weights each item, and searches for a sentence containing “mayor” and “powers” and “duties” and “city” anywhere within it, finding this sentence:

```
In the case of absence from the CITY or the failure,
inability or refusal of both the MAYOR and mayor pro tempore
```

to perform the DUTIES of mayor, the city council may elect an acting mayor pro tempore, who shall serve as mayor with all the POWERS, privileges, and duties.

Only those words required to qualify the sentence as a hit are located by the program, for maximum search efficiency.

In this example, there are several occurrences of the search items “mayor”, “duties”, and “city”. It was only necessary to locate each item once to confirm validity of the hit. Such words may be found by the search program in any order.

The existence of more than one matched search item in a hit is called an intersection. Specifying two keywords in a query indicates you want both keywords to occur, or intersect, in the sentence.

A 2 item search is common, and can be thought of as 1 intersection of 2 sets.

Example:

```
WHERE BODY LIKE '~alcohol ~consumption'
```

In the above example, the tilde (~) preceding “alcohol” and preceding “consumption” enables concept expansion on both words, thereby including the set of associations listed for each word in the Thesaurus.

Where something from the concept set “alcohol” and something from the concept set “consumption” meet within a sentence, there is a hit. This default set logic finds a 1 intersection sentence:

It shall be unlawful to USE the city swimming pool or enter the enclosure in which it is located when a person is INTOXICATED or under the influence of illegal drugs.

“Use” is in the “consumption” concept set; “intoxicated” is in the “alcohol” concept set.

These two sets have herein intersected, forcing the context of the set members to be relevant to the entered query.

Maximum Intersections Possible (“AND”)

Adding a search item dictates stricter relevance requirements. Here, a sentence has to contain 2 intersections of 3 search items to be deemed a valid hit.

Example:

```
WHERE BODY LIKE '~alcohol ~sweets ~consumption'
```

Such a 2 intersection search finds this hit:

any public sleeping or EATING place, or any place or vehicle

where food or DRINK is manufactured, prepared, stored, or any manufacturer or vendor of CANDIES or manufactured sweets.

Default intersection logic is to find the maximum number of set intersections possible in the stated query; that is, an “and” search where an intersection of all search items is required.

Specifying Fewer Intersections

The casual user can use the “AND” logic default. Even so, here is another way to write the above 2 intersection query, where the number of desired intersections (2) is preceded by the at sign (@):

Example:

```
WHERE BODY LIKE '~alcohol ~sweets ~consumption @2'
```

The “@2” designation is redundant as it is understood by the program to be the default maximum number of intersections possible, but it would yield the same results.

It is possible to find different permutations of which items must occur inside a hit. Even where the maximum number of intersections possible is being sought, this is still seen as a permutation, and is referred to as *permuted* logic.

The meaning of “*permuted*” takes on more significance when fewer intersections of items are desired.

If you wanted only one intersection of these three items, it would create an interesting range of possibilities. You might find an intersection of any of the following combinations:

```
alcohol (AND) sweets
alcohol (AND) consumption
sweets (AND) consumption
```

Specify one intersection only (@1), while listing the 3 possible items.

Example:

```
WHERE BODY LIKE '~alcohol ~sweets ~consumption @1'
```

This 1 intersection search finds the following, where any 2 occurrences from the 3 specified sets occur within the hit. Hits for a higher intersection number (@2) as shown above also appear.

```
~consumption (and) ~alcohol @1
It shall be unlawful to USE the city swimming pool or enter
the enclosure in which it is located when a person is
INTOXICATED or under the influence of illegal drugs.
```



```
~consumption (and) ~sweets @1
  any public sleeping or EATING place, or any place or vehicle
  where food or drink is manufactured, prepared, stored, or
  any manufacturer or vendor of CANDIES or manufactured
  sweets.
```

```
~sweets (and) ~alcohol @1
  subject to inspection are: Bakery or CONFECTIONERY shop
  (retail), Beverage sale ALCOHOLIC ...
```

```
~consumption (and) ~alcohol @1
  A new EATING and DRINKING establishment shall be one which
  is newly erected or constructed at a given location.
```

```
~alcohol (and) ~consumption @1
  involving the sale of spirituous, vinous, or malt LIQUORS,
  including beer in unbroken packages for off-premises
  CONSUMPTION.
```

The “@#” intersection quantity designation is not position dependent; it can be entered anywhere in the Metamorph query.

Any number of intersections may be specified, provided that number does not exceed the number of intersections possible for the entered number of search items.

Specifying No Intersections (“OR”)

Using this intersection quantity model, what is commonly understood to be an “OR” search is any search which requires no (zero) intersections at all. In an “or” search, any occurrence of any item listed qualifies as a hit; the item need not intersect with any other item.

Designate an “or” search using the same intersection quantity syntax, where zero (0) indicates no intersections are required (@0):

Example:

```
WHERE BODY LIKE '~alcohol ~sweets ~consumption @0'
```

In addition to the hits listed above for a higher number of intersections, the following 0 intersection hits would be found, due to the presence of only one item (a or b or c) required:

```
~alcohol @0
  Every person licensed to sell LIQUOR, wine or beer or mixed
  beverages in the city under the Alcoholic Beverage Code
  shall ...
```

`~sweets @0`

An establishment preparing and selling at retail on the premises, cakes, pastry, CANDIES, breads and similar food items.

`~consumption @0`

To regulate the disposal and prohibit the BURNING of garbage and trash; ...

All such items are considered *permuted*, at intersection number zero (0).

Weighting Items for Precedence (+)

Intersection logic treats all search items as equal to each other, regardless of the number of understood or specified intersections. You can indicate a precedence for a particular search item which falls outside the intersection quantity setting.

A common example is where you are interested chiefly in one subject, but you want to see occurrences of that subject in proximity to one or more of several specified choices. This would be an “or” search in conjunction with one item marked for precedence. You definitely want A, along with either B, or C, or D.

Use the plus sign (+) to mark search items for mandatory inclusion. Use @0 to signify no intersections are required of the unmarked permuted items. The number of intersections required as specified by ‘@#’ will apply to those permuted items remaining.

Example:

`WHERE BODY LIKE '+license plumbing alcohol taxes @0'`

This search requires (+) the occurrence of “license”, which must be found in the same sentence with either “plumbing”, “alcohol”, or “taxes”.

The 0 intersection designation applies only to the unmarked permuted sets. Since “license” is weighted with a plus (+), the “@0” designation applies to the other search items only.

This query finds the following hits:

`+license (and) @0 alcohol`

Every person licensed to sell liquor, wine or beer or mixed beverages in the city under the ALCOHOLIC Beverage Code shall pay to the city a LICENSE fee equal to the maximum allowed as provided for in the Alcoholic Beverage Code.

`+license (and) @0 plumbing`

Before any person, firm or corporation shall engage in the PLUMBING business within the city, he shall be qualified as set forth herein, and a LICENSE shall be obtained from the

State Board of Plumbing Examiners as required.

+license (and) @0 taxes

The city may assess, levy and collect any and all character of TAXES for general and special purposes on all subjects or objects, including occupation taxes, LICENSE taxes and excise taxes.

More than one search item may be marked with a plus (+) for inclusion, and any valid intersection quantity (@#) may be used to refer to the other unmarked items. Any search item, including phrases and special expressions, may be weighted for precedence in this fashion.

Marking Items for Exclusion (“NOT”) (-)

You can exclude a hit due to the presence of one or more search items. Such mandatory exclusion logic for a particular search item falls outside the intersection quantity setting, as does inclusion, and applies to the whole set in the same manner. This is sometimes thought of as “NOT” logic, designated with a minus sign (-).

A common example is where one item is very frequently used in the text, so you wish to rule out any hits where it occurs. You want an intersection of A and B, but not if C is present.

Use the minus sign (-) to mark search items for exclusion. Default or specified intersection quantities apply to items not marked with a plus (+) or minus (-). The number of intersections required will apply to the remaining permuted items.

Example:

```
WHERE BODY LIKE 'license ~alcohol -drink'
```

This search has the goal of finding licensing issues surrounding alcohol. However, the presence of the word “drink” might incorrectly produce references about restaurants that do not serve alcohol.

Excluding the hit if it contains “drink” retrieves these hits:

```
license (and) ~alcohol -drink
```

```
Every person licensed to sell LIQUOR, wine or beer or mixed
beverages in the city shall pay to the city a LICENSE fee
equal to the maximum allowed ...
```

```
license (and) ~alcohol -drink
```

```
State law reference(s)--Authority to levy and collect
LICENSE fee, V.T.C.A., ALCOHOLIC Beverage Code 11.38, 61.36.
```

But excludes this hit:

```
license (and) ~alcohol -drink {Excluded Hit}
  The city council shall have the power to regulate, LICENSE
  and inspect persons, firms, or associations operating,
  managing, or conducting any place where food or DRINK is
  manufactured, prepared, or otherwise handled within city
  limits.
```

More than one search item may be marked with a minus (-) for exclusion, along with items marked with plus (+) for inclusion, and any valid intersection quantity (@#) specification. Any search item, including phrases and special expressions, may be marked for exclusion in this fashion.

Combinatorial Logic

Weighting search items for inclusion (+) or exclusion (-) along with an intersection specification which is less than the maximum quantity possible (the default “AND” search) can be used in any combination.

Default logic is adequate for the casual user to get very satisfactory search results with no special markings, so this extra syntax need not be learned. However, such syntax is available to the more exacting user if desired.

A rather complicated but precise query might make use of weighting for inclusion, exclusion, and also a specified intersection quantity, as follows.

Example:

```
WHERE BODY LIKE '+officer @1 law power pay duties -mayor'
```

The above query makes these requirements:

- “Officer” must be present, plus ...
- 1 intersection of any 2 of the unmarked search items “law”, “power”, “pay”, “duties”, but ...
- Not if “mayor” is present (i.e., exclude it).

This query retrieves the following hits, while excluding hits containing “mayor”.

```
power, duties +officer (but not) -mayor
  The city council shall have POWER from time to time to
  require other further DUTIES of all OFFICERS whose duties
  are herein prescribed.

law, duties +officer (but not) -mayor
  Proof shall be made before some OFFICER authorized by the
  LAW to administer oaths, and filed with the person
  performing the DUTIES of city secretary.
```

law, power +officer (but not) -mayor

In case of any irreconcilable conflict between the provisions of this Charter and any superior LAW, the POWERS of the city and its OFFICERS shall be defined in such superior laws.

duties, law +officer (but not) -mayor

The plan must be designed to enable the records management OFFICER to carry out his or her DUTIES prescribed by state LAW and this article effectively.

pay, duties +officer (but not) -mayor

PAYMENT of firefighting OFFICIAL performing DUTIES outside of territorial limits of city.

power, duties +officer (but not) -mayor

POWERS and DUTIES of OFFICIAL assigned to assist in the city.

Any search item, including keywords, wildcards, concept searches, phrases, and special expressions, can be weighted for inclusion, exclusion, and combinatorial set logic.

Combinatorial Logic and LIKER

When using LIKER the default weighting of the terms has two factors. The first is based on the words location in the query, with the most important term first. The second is based on word frequency in the document set, where common words have a lesser importance attached to them. The logic operators (+ and -) remove the second factor. Additionally the not operator (-) negates the first factor.

Metamorph Logic Rules Summary

Logic operators apply to any entered search item.

1. Precedence (mandatory inclusion) is expressed by a plus sign (+) preceding the concept or expression. A plus (+) item is "Required".
2. Exclusion ("not" logic) is expressed by a minus sign (-) preceding the concept or expression. A minus (-) item is "Excluded".
3. Equivalence ("and" logic) may be expressed by an equal sign (=) preceding the concept or expression. An equal sign is assumed where no other logic operator is assigned. An equal (=) item is "Permuted".
4. Search items not marked with (+) or (-) are considered to be equally weighted. Intersection quantity logic (@#) applies to these unmarked (=) permuted sets only.

5. Where search items are not otherwise marked, the default set logic in use is “And” logic. The maximum number of intersections possible is sought.
6. Designate “Or” logic with zero intersections (@0), applying to any unmarked permuted search items.
7. Logic operators and intersection quantity settings can be used in combination with each other, referred to as combinatorial logic.

Metamorph’s use of logic should not be confused with Boolean operators. Metamorph deals with these logic operators as sets rather than single strings, a different methodology.

7.5 Other Metamorph Features

Metamorph contains many special kinds of searches. Again, any Metamorph search can be constructed as part of the `LIKE` clause. See the Metamorph section of this manual for a complete treatment of this subject.

Chapter 8

Indexing for Increased Performance

8.1 When and How to Index

There are two basic mechanisms for accessing Taxis tables: a table space scan which is sequential, and an index based scan which is direct. Index based retrieval is usually more efficient than table space scan with some exceptions.

The use of indexes is one of the major ways in which the performance of queries (i.e. the speed with which results are retrieved) can be improved in relational databases. Indexes allow the DBMS to retrieve rows from a table without scanning the entire table, much as a library user can use a card catalog to find books without scanning the entire library.

The creation of indexes improves the performance associated with processing large tables. However, an excessive number of indexes can result in an increase in processing time during update operations because of the additional effort needed to maintain the indexes. Thus, for tables undergoing frequent change, there could be a “cost” associated with an excessive number of indexes.

In addition, as the number of indexes increases, the storage requirements needed to hold the indexes becomes significant. Other cautions include not indexing small amounts of data as doing so may slow down searches due to the overhead of looking in an index. The point at which it makes sense to use an index will depend upon the system in use. And it would be important to use the correct kind of index for the job.

Since there are so many factors involved in the decision as to whether and what kind of index will most optimize the search, Taxis largely takes over the management of these decisions. The user can suggest those tables on which an index ought to be created. Beyond this, the user would not know the status of the index, which is always in flux, nor whether it has been updated at the time of the search.

Unlike other systems, Taxis ensures that all information which has been added to any table can be searched immediately, regardless of whether it has been indexed, and regardless of whether it has been suggested that an index be maintained on that table or not. Sequential table space scans and index based scans are efficiently managed by Taxis so that the database can always be searched in the most optimized manner, with the most current information available to the user.

To this end, there are two types of indexes supported by Taxis:

1. The canonical sorted order alphabetical index found in traditional SQL systems, and
2. The Metamorph index, optimized for systems containing a large number of rows, or a lot of text, or both. The Metamorph index is used when it is expected that `LIKE` or its variants will be common on the field being indexed.

When an index is created, neither an end user nor an application programmer need (nor can) reference the index in a query. Indexes are used automatically by Taxis when needed to choose the best path to the data.

8.2 Creating An Index

The `CREATE INDEX` command is used to establish an index. The form of this command is:

```
CREATE INDEX index-name
ON table-name (column-name [DESC] [, column-name [DESC]] ...)
[WITH option-name [value] [option-name [value] ...]] ;
```

Command Discussion

- Each index is assigned a name and is related to a particular table based on the `ON table-name` clause.
- The entries in an index are ordered on a specified column within the specified table, and arranged in either ascending or descending order.
- The term *index key* refers to the column (or set of columns) in a table that is used to determine the order of entries in an index. Where an index does consist of multiple columns, the most important column is placed first.
- Further options controlling how and what type of index is created may be set in the `WITH` clause; see p. 138.
- A table can have many different indexes associated with its columns. Each index is created by issuing a separate `CREATE INDEX` command.

Example: Many queries to the `EMPLOYEE` table reference the employee's department; therefore, the database designer decides to create an index using the `DEPT` column to improve performance related to these queries.

This command:

```
CREATE INDEX DEPTINDEX
ON EMPLOYEE (DEPT) ;
```


would direct the creation of an index called `DEPTINDEX` (the Index name) on the table `EMPLOYEE`, using the `DEPT` column as indicated in parentheses as the Index key.

The index can be used by the system to provide quick access to employee data that are subject to conditions related to the employee's department.

8.3 Creating a Unique Index

The *primary key* is an important concept in data processing. The primary key is a field or combination of fields in a record that allows you to uniquely identify a record from all the other records in a table. For example, companies assign IDs to employees as unique identifiers. Thus, an employee ID can serve as a primary key for personnel records.

When a table is created in Taxis, duplicate records can be stored in the table. The uniqueness characteristic is not enforced automatically. To prevent duplicate records from being stored in a table, some steps must be taken.

First, a separate file called an “index” must be created. In this case the index is created so that the DBMS can ensure that all values in a special column or columns of a table are unique. For example, the `EMPLOYEE` table can be indexed on `EID` (employee ID) so that each row of the `EMPLOYEE` table contains a different employee ID value (i.e., no duplicate `EIDs` can be entered.)

A variation of the `CREATE INDEX` command, `CREATE UNIQUE INDEX`, is used to establish an index that assures no duplicate primary key values are stored in a table. The form of this command is:

```
CREATE [UNIQUE] INDEX index-name
ON table-name (column-name [DESC] [,column-name [DESC]] ... ) ;
```

Command Discussion

- The keyword `UNIQUE` in the clause `CREATE UNIQUE INDEX` specifies that in the creation and maintenance of the index no two records in the index table can have the same value for the index column (or column combination). Thus, any `INSERT` or `UPDATE` command that attempts to add a duplicate row in the index would be rejected.
- Each index is assigned a name and is related to a particular table based on the `ON table-name` clause.
- An index is based on the specified column within the specified table, and will be arranged in ascending order.

Example: Create an index for the `EMPLOYEE` table that prevents records with the same employee ID from being stored in the table with this command:

```
CREATE UNIQUE INDEX EMPINDEX
ON EMPLOYEE (EID) ;
```

This command directs the creation of a unique index on the `EMPLOYEE` table, where the index name is `EMPINDEX`. The `EID` column as indicated in parentheses is the Index key.

In other words, an index called `EMPINDEX` has been created on the `EMPLOYEE` table for the employee ID number.

The index is stored separately from the `EMPLOYEE` table. The example below shows the relationship between `EMPLOYEE` and `EMPINDEX` after ten employees have been added to the `EMPLOYEE` table. Each row of the index, `EMPINDEX`, consists of a column value for the index column and a pointer, or physical address, to the location of a row in the `EMPLOYEE` table. As employees are added or deleted from the `EMPLOYEE` table, Taxis automatically updates the index in the most efficient and timely manner.

To conceptualize how the index works, assume you didn't realize Chapman's record was already stored in the `EMPLOYEE` table and you attempt to add her record again. You enter the command:

```
INSERT INTO EMPLOYEE
VALUES ('103','Chapman, Margaret','LIB','STAFF','PART',22000) ;
```

and Taxis responds with an error message, such as:

```
ERROR: Duplicate Value in Index
```

This message occurs because the value 103, the employee ID (`EID`), is already stored in `EMPINDEX` and attempting to add another 103 value results in a duplicate value, which is not permitted in a unique index.

When we add a new employee named Krinski with an `EID` equal to 110 by entering this command:

```
INSERT INTO EMPLOYEE
VALUES ('110','Krinski','LIB','DHEAD','FULL',32500) ;
```

the record is successfully created. The value 110 did not previously exist in the unique index `EMPINDEX`, and so it was allowed to be entered as a new row in the `EMPLOYEE` table. As the `EMPINDEX` is in sorted order, it is much faster to ascertain that information than it would be by searching the entire `EMPLOYEE` table.

The relationship between `EMPINDEX` the index, and `EMPLOYEE` the table, appear below as they would containing 10 employee records. The dashed lines indicate pointers from the index to rows in the table. However, this is conceptual rather than actual, and not all pointers are shown.

| EMPINDEX | | EMPLOYEE | | | | | |
|----------|---------|----------|-------------------|------|-------|----------|--------|
| Index | | Table | | | | | |
| | | EID | ENAME | DEPT | RANK | BENEFITS | SALARY |
| 101 | -----> | 101 | Aster, John A. | MKT | STAFF | FULL | 32000 |
| 102 | --+ | 109 | Brown, Penelope | MKT | DHEAD | FULL | 37500 |
| 103 | | 104 | Jackson, Herbert | RND | STAFF | FULL | 30000 |
| 104 | +--> | 108 | Jones, David | RND | DHEAD | FULL | 37500 |
| 105 | +-- --> | 102 | Barrington, Kyle | MGT | DHEAD | FULL | 45000 |
| 106 | | 106 | Sanchez, Carla | MKT | STAFF | FULL | 35000 |
| 108 | -----+ | 105 | Price, Stella | FIN | DHEAD | FULL | 42000 |
| 107 | | 103 | Chapman, Margaret | LIB | STAFF | PART | 22000 |
| 109 | | 107 | Smith, Roberta | RND | STAFF | PART | 25000 |
| 110 | -----> | 110 | Krinski, Wanda | LIB | DHEAD | FULL | 32500 |

8.4 Creating a Metamorph Index

A sorted order index is optimized for columns containing values of limited length, which can easily be canonically managed. In some cases, especially when a column contains a large amount of text, there is a need for an index which goes beyond the methods used in these previous examples.

For example, let us take the case of the News database being archived on a daily basis by the Strategic Planning and Intelligence Department. The entire body of the news article is stored in a table, whether the data type in use is `VARCHAR`, indicating a variable length number of characters, or `INDIRECT`, indicating it points elsewhere to the actual location of the files. While subjects, dates, and bylines are important, the most often queried part is the body of the article, or the text field itself. The column we want to index is a text column rather than something much more concise like an employee ID number.

To accurately find text in the files, where search items are to be found in proximity to other search items within some defined delimiters, all the words of all the text in question must be indexed in an efficient manner which still allows everything relevant to be found based on its content, even after it has been archived away. A Metamorph index combines indexing technology with a linear free text scan of selected portions of the database where appropriate in order to accomplish this. This linear scan following the index lookup is referred to as a *post-search* or *post-processing*.

Metamorph query language as used following `LIKE` and its variants is described in detail in Chapter 7, *Intelligent Text Search Queries*. Where you anticipate such `LIKE` queries will be common on that field, it would be appropriate to create a Metamorph index.

The form of the command is:

```
CREATE METAMORPH [INVERTED|COUNTER] INDEX index-name
ON table-name (column-name [, column-name...]) ;
```

Syntax is the same as in the previous `CREATE INDEX` examples, except that you are specifying the type of index you want created (i.e. a Metamorph index).

Example: The news database that is being accumulated from selected news articles is getting too large to

search from beginning to end for content based searches which make heavy use of the `LIKE` clause. A Metamorph index should be created for the Strategic Planning and Intelligence Department to enhance their research capability. The column containing the text of the articles is called `BODY`.

An index called `BODYINDEX` will be created and maintained on the `BODY` column of the `NEWS` table, which contains the full text of all collected news articles. Now content searches can stay fast as well as accurate, regardless of how large this database becomes.

Additional columns can be specified in addition to the text field to be indexed. These should be fixed length fields, such as dates, counters or numbers. The extra data in the index can be used to improve searches which combine a `LIKE` statement with restrictions on the other fields, or which `ORDER BY` some or all of the other fields.

8.4.1 Metamorph Index Types: Inverted vs. Compact vs. Counter

There are three types of Metamorph index: inverted, compact and counter. All are used to aid in resolving `LIKE`/`LIKEP`/`LIKE3`/`LIKER`/`LIKEIN` queries, and are created with some variant of the syntax `CREATE METAMORPH INDEX`.

Inverted

An inverted Metamorph index is the most commonly used type of Metamorph index, and is created with `CREATE METAMORPH INVERTED INDEX`. In Taxis version 7 (and `compatibilityversion 7`) and later, this is the default Metamorph index type created when no other flags are given, e.g. `CREATE METAMORPH INDEX`; in version 6 (or `compatibilityversion 6`), a compact index is created. The version 7 index option `WORDPOSITIONS 'on'` (p. 138) also explicitly creates this type of Metamorph index (same effect as the `INVERTED` flag after `METAMORPH`).

An inverted Metamorph index maintains knowledge not only of what rows words occur in, but also what position in each row the words occur in (the `WORDPOSITIONS`). With such an index Taxis can often avoid a post-search altogether, because the index contains all the information needed for phrase resolution and rank computation. This can speed up searches more than a compact Metamorph index, especially for ranking queries using `LIKEP`, or phrase searches. Because of the greater range of queries resolvable with an inverted Metamorph index (vs. compact), in Taxis version 7 and later it is the default Metamorph type created. However, an inverted Metamorph index consumes more disk space, typically 20-30% of the text size versus about 7% for a compact Metamorph index. Index updating is also slower because of this.

Compact

A compact Metamorph index maintains knowledge of what rows words occur in, but does not store word position information. In Taxis version 7 and later, it is created by adding the index option `WORDPOSITIONS 'off'` (p. 138). In Taxis version 6 and earlier, this was the default Metamorph index type, and was created with `CREATE METAMORPH INDEX` (no flags/options).

Because of the lack of word position information, a compact Metamorph index only consumes about 7% of the text size in disk space (vs. about 20-30% for a Metamorph inverted index); this compact size can also

speed up its usage. However, a post-process search is needed after index usage if the query needs word-position information (e.g. to resolve phrases, within “w/N” operators, LIKEP ranking), which can greatly slow such queries. Thus a compact Metamorph index is best suited to queries that do not need word position information, such as single non-phrased words with no special pattern matchers, and no ranking (e.g. LIKE). A LIKER or LIKE3 search (below), which never does a post-search, can also use a compact Metamorph index without loss of performance.

Counter

A Metamorph counter index contains the same information that a compact Metamorph index has, but also includes additional information which improves the performance of LIKEIN queries. If you are doing LIKEIN queries then you should create this type of index, otherwise you should use either the normal or inverted forms of the Metamorph index. A Metamorph counter index is created with CREATE METAMORPH COUNTER INDEX; in Taxis version 7 and later the COUNTS 'on' index option (p. 138) can be given instead of the COUNTER flag to accomplish the same action.

8.4.2 Metamorph Index Capabilities and Limitations

As with any tool the best use can be obtained by knowing the capabilities and limitations of the tool. The Metamorph index allows for rapid location of records containing one or more keywords. The Metamorph index also takes care of some of the set logic.

The following should be noted when generating queries. The most important point is the choice of keywords. If a keyword is chosen that occurs in many files, then the index will have to do more work to keep track of all the files possibly containing that word. A good general rule of thumb is “The longer the word, the faster the search”.

Also, neither type of Metamorph index is useful for special pattern matchers (REX, XPM, NPM) as these terms cannot be indexed. If other indexable terms are present in the query, the index will be used with them to narrow the result list, but a post-search or possibly even a complete linear scan of the table may be needed to resolve special pattern matchers.

8.5 Using LIKE3 for Index Only Search (No Post-Search)

In certain special cases, such as static information which does not change at all except under very controlled circumstances easily managed by the system administrator, there may be instances where an index based search with no qualifying linear post-search may be done without losing meaningful responses to entered queries.

This kind of search is completely optimized based on certain defaults which would be known to be acceptable, including:

- All the search items can be keywords (i.e., no special pattern matchers will be used).

- All searches can be done effectively within the length of the field (i.e., the delimiters used to define proximity of search items is the length of the whole text field).

As more often than not maintaining all the above rules is impractical, dispensing with the post-search would not be done very frequently. However, in some circumstances where these rules fit, the search requirements are narrow, and speed is of the essence, the post-search can be eliminated for optimization purposes.

Texis will examine the query given to `LIKE`, and if it can guarantee the same results without the post-search it will not perform the post-search, and `LIKE` will then be equivalent to `LIKE3`. With these caveats in mind, `LIKE3` may be substituted for `LIKE` in any of the queries illustrated in the previous chapters.

8.6 Creating an Inverted Index

The inverted index (not to be confused with a Metamorph inverted index—a different type of index) is a highly specialized index, which is designed to speed up one class of query only, and as such has some limitations on its use. The primary limitation is that it can currently only be used on a field that is of the type `UNSIGNED INT` or `DATE`.

Inverted indexes can be used to speed up the `ORDER BY` operation in the case that you are ordering by the field that was indexed only. For maximum effect you should also have the indexed ordered in the same manner as the `ORDER BY`. In other words if you want a descending order, you should have a descending index.

An inverted index can be created using this command:

```
CREATE INVERTED INDEX SALINDEX
ON EMPLOYEE (SALARY) ;
```

8.7 Index Options

A series of index options may be specified using a `WITH` clause at the end of any `CREATE [index-type] INDEX` statement:

```
CREATE [index-type] INDEX index-name
ON table-name (column-name [DESC] [, column-name [DESC]] ...)
[WITH option-name [value] [option-name [value] ...]] ;
```

Index options control how the index is made, or what sub-type of index is created. Many options are identical to global or server properties set with the `SET` statement (p. 165), but as options are set directly in the `CREATE INDEX` statement, they override those server properties, yet only apply to the statement they are set in. Thus, using index options allows modularization of `CREATE INDEX` statements, making it clearer what settings apply to what index by directly specifying them in the `CREATE` statement, and avoiding side-effects on later statements.

Note that the `WITH` clause is only supported in Taxis version 7 and later. Previous releases can only set server-wide properties, via the `SET` statement.

Available Options

The available index options are as follows. Note that some options are only applicable to certain index types, as noted; using an option that does not apply to the given index type will result in an error:

- `counts 'on' | 'off'`
For Metamorph or Metamorph counter index types only. If set to “on”, creates a Metamorph counter type index (useful for `LIKEIN` searches); if “off” (the default), a regular Metamorph or Metamorph inverted index is created.
- `indexmaxsingle N`
For Metamorph, Metamorph inverted, and Metamorph counter index types only. Same as the `indexmaxsingle` server property (p. 181).
- `indexmem N`
- `indexmeter N|type`
- `indexspace N`
- `indexvalues type`
These options have the same effect as the same-named server properties set with `SET`.
- `indexversion N`
- `keepnoise 'on' | 'off'`
- `noiselist ('word', 'word', ...)`
- `textsearchmode mode`
- `wordexpressions ('expr', 'expr', ...)`
For Metamorph, Metamorph inverted, and Metamorph counter index types only. Same effect as the same-named server properties.
- `wordpositions 'on' | 'off'`
For Metamorph and Metamorph inverted index types only. If “on” (the default in version or `compatibilityversion 7` and later), creates a full-inversion (Metamorph inverted) index; if “off”, creates a compact (Metamorph) index.
- `max_index_text N`
- `stringcomparemode mode`
For regular index types only. Same effect as the same-named server properties.

8.8 Dropping an Index

Any index – unique or non-unique, sorted order or Metamorph – can be eliminated if it is no longer needed. The `DROP INDEX` command is used to remove an index. The format of this command is similar to the `DROP TABLE` command illustrated in Chapter 3.

```
DROP INDEX index-name ;
```

Example: Let's say the `DEPTINDEX` is no longer needed. Delete it with this statement:

```
DROP INDEX DEPTINDEX ;
```

The table on which the index was created would not be touched. However, the index created for it has been removed.

Chapter 9

Keeping the Database Current

To keep the data in your database current, three types of transactions must be performed on the data. These transactions are:

1. Adding new records.
2. Changing existing records.
3. Deleting records no longer needed.

9.1 Adding New Records

Once a table has been defined and before any data can be retrieved, data must be entered into the table. Initially, data can be entered into the table in several ways:

- Batch mode: Data is loaded into the table from a file.
- Interactive mode: Data for each record is added by interactive prompting of each column in a record.
- Line Input: A row of data is keyed for insertion into a table using a line editor and then is submitted to the database.

Generally a Load Program would be used to load data into the tables at the outset. It would be unusual to use line input especially to get started, but it is used in the following examples so that the correct syntax can be clearly seen.

9.1.1 Inserting One Row at a Time

In addition to initially loading data into tables, records can be added at any time to keep a table current. For example, if a new employee is hired, a new record, or row, would be added to the EMPLOYEE table. The `INSERT` command is used to enter a row into a table. The command has two formats:

1. Entering one row at a time.
2. Entering multiple rows at a time.

In the first format, the user enters values one row at a time, using the following version of the `INSERT` command:

```
INSERT INTO table-name [(column-name1 [,column-name2] ... )]  
VALUES (value1, value2 ... ) ;
```

Command Discussion

- The `INSERT INTO` clause indicates that you intend to add a row to a table.
- Following the `INSERT INTO` clause, the user specifies the name of the table into which the data is to be inserted.
- When data values are being entered in the same order the columns were created in there is no need to list the column names following the `INSERT INTO` clause. However, sometimes when a row is added, the correct ordering of column values is not known. In those cases, the columns being added must be listed following the table name in the order that the values will be supplied.
- Following the keyword `VALUES` are the values to be added to one row of a table. The entire row of values is placed within parentheses. Each data value is separated from the next by a comma. The first value corresponds to the first column in the table; the second value corresponds to the second column in the table, and so on.

Example: When the `EMPLOYEE` table is first created, it has no employee data stored in it. Add the first record to the table, where the data values we have are as follows:

```
EID = 101  
ENAME = Aster, John A.  
DEPT = MKT  
RANK = STAFF  
BENEFITS = FULL  
SALARY = 32000
```

You can create a record containing these values by entering this command:

```
INSERT INTO EMPLOYEE  
VALUES (101,'Aster, John A.', 'MKT', 'STAFF', 'FULL', 32000) ;
```

Quotes are placed around character values so Taxis can distinguish data values from column names.

A new employee record gets added to the `EMPLOYEE` table, so that the table now looks like this, if this were the only row entered:

| EID | ENAME | DEPT | RANK | BENEFITS | SALARY |
|-----|----------------|------|-------|----------|--------|
| 101 | Aster, John A. | MKT | STAFF | FULL | 32000 |

9.1.2 Inserting Text

There are a few different ways to manage large quantities of text in a database. The previous examples given for the REPORT table concentrated on the VARCHAR (variable length character) column which held a filename as a character string; e.g., ' /data/rnd/ink.txt ' as stored in the FILENAME column. This column manages the filename only, not the text contained in that file.

In the examples used in Chapter 3, *Table Definition*, a RESUME table is created which uses a VARCHAR field of around 2000 characters to hold the text of the resumes. In this case, the job experience text of each resume is stored in the column EXP. A Load Program would be used to insert text of this length into the column of a table.

Another way Taxis has of managing text is to allow the files to remain outside the confines of the table. Where the INDIRECT data type is used, a filename can be entered as a value which points to an actual file, rather than treated as a character string. The INDIRECT type looks at the contents of the file when doing LIKE, and these contents can be retrieved using the API (Application Program Interface).

The form of the INSERT INTO command is the same as above. Where a data type is defined as INDIRECT, a filename may be entered as the value of one or more columns.

Example: Let's say we have the following information available for a resume to be entered into the RESUME table, and that the job experience column EXP has been defined as INDIRECT.

```
RES_ID = R421
RNAME = Smith, James
JOB = Jr Analyst
EDUC = B.A. 1982 Radford University
EXP = contained in the resume file "/usr/local/resume/smith.res"
```

Use this INSERT INTO statement to add a row containing this information to the RESUME table:

```
INSERT INTO RESUME
VALUES ('R421','Smith, James','Jr Analyst',
      'B.A. 1982 Radford University',
      '/usr/local/resume/smith.res') ;
```

The EXP column acts as a pointer to the full text files containing the resumes. As such, the text in those files responds to all SELECT-FROM-WHERE statements. Thus Metamorph queries used after LIKE can be done on the text content manipulated by Taxis in this table.

9.1.3 Inserting Multiple Rows at a Time

In addition to adding values to a table one row at a time, you can also use a variation of the `INSERT` command to load some or all data from one table into another table. The second form of the `INSERT` command is used when you want to create a new table based on the results of a query against an existing table. The form of this `INSERT` command is:

```
INSERT INTO table-name
  SELECT expression1 [,expression2] ...
  FROM   table-name
  [WHERE search-condition] ;
```

Command Discussion

- The `INSERT INTO` clause indicates that you intend to add a row or rows to a table.
- Following the `INSERT INTO` clause, the user specifies the name of the table to be updated.
- The query is evaluated, and a copy of the results from the query is stored in the table specified after the `INSERT INTO` clause. If rows already exist in the table being copied to, then the new rows are added to the end of the table.
- Block inserts of text columns using `INDIRECT` respond just as any other column.

Example: Finance wants to do an analysis by department of the consequences of a company wide 10% raise in salaries, as it would affect overall departmental budgets. We want to manipulate the relational information stored in the database without affecting the actual table in use.

Step 1: Create a new table named `EMP_RAISE`, where the projected results can be studied without affecting the live stored information. Use this `CREATE TABLE` statement, which defines data types as in the original table, `EMPLOYEE`, creating an empty table.

```
CREATE TABLE EMP_RAISE
  (EID      INTEGER
   ENAME    CHAR(15)
   DEPT     CHAR(3)
   RANK     CHAR(5)
   BENEFITS CHAR(4)
   SALARY   INTEGER) ;
```

Step 2: Copy the data in the `EMPLOYEE` table to the `EMP_RAISE` table. We will later change salaries to the projected new salaries using the `UPDATE` command. For now, the new table must be loaded as follows:

```
INSERT INTO EMP_RAISE
  SELECT *
  FROM   EMPLOYEE ;
```

The number of records which exist in the EMPLOYEE table at the time this INSERT INTO command is done is the number of records which will be created in the new EMP_RAISE table. Now that the new table has data values, it can be queried and updated, without affecting the data in the EMPLOYEE table.

An easier way to create a copy of the table is to use the following syntax:

```
CREATE TABLE EMP_RAISE AS
  SELECT *
  FROM EMPLOYEE ;
```

which creates the table, and copies it in one statement. Any indexes on the original table will not be created on the new one.

9.2 Updating Records

Very often data currently stored in a table needs to be corrected or changed. For example, a name may be misspelled or a salary figure increased. To modify the values of one or more columns in one or more records of a table, the user specifies the UPDATE command. The general form of this statement is:

```
UPDATE table-name
SET column-name1 = expression1
  [,column-name2 = expression2] ...
[WHERE search-condition] ;
```

Command Discussion

- The UPDATE clause indicates which table is to be modified.
- The SET clause is followed by the column or columns to be modified. The expression represents the new value to be assigned to the column. The expression can contain constants, column names, or arithmetic expressions.
- The record or records being modified are found by using a search condition. All rows that satisfy the search condition are updated. If no search condition is supplied, all rows in the table are updated.

Example: Change the benefits for the librarian Margaret Chapman from partial to full with this statement:

```
UPDATE EMPLOYEE
SET BENEFITS = 'FULL'
WHERE EID = 103 ;
```

The value 'FULL' is the change being made. It will replace the current value 'PART' listed in the BENEFITS column for Margaret Chapman, whose employee ID number is 103. A change is made for all records that satisfy the search condition; in this example, only one row is updated.

Example: The finance analysis needs to include the effects of a 10% pay raise to all staff; i.e., to all employees whose RANK is STAFF.

Use this statement to update all staff salaries with the intended raise:

```
UPDATE  EMP_RAISE
SET     SALARY = SALARY * 1.1
WHERE   RANK = 'STAFF' ;
```

If a portion of the EMP_RAISE table looked like this before the update:

| EID | ENAME | DEPT | RANK | BENEFITS | SALARY |
|-----|-------------------|------|-------|----------|--------|
| 101 | Aster, John A. | MKT | STAFF | FULL | 32000 |
| 102 | Barrington, Kyle | MGT | DHEAD | FULL | 45000 |
| 103 | Chapman, Margaret | LIB | STAFF | PART | 22000 |
| 104 | Jackson, Herbert | RND | STAFF | FULL | 30000 |
| 105 | Price, Stella | FIN | DHEAD | FULL | 42000 |
| 106 | Sanchez, Carla | MKT | STAFF | FULL | 35000 |
| 107 | Smith, Roberta | RND | STAFF | PART | 25000 |

It would look like this after the update operation:

| EID | ENAME | DEPT | RANK | BENEFITS | SALARY |
|-----|-------------------|------|-------|----------|--------|
| 101 | Aster, John A. | MKT | STAFF | FULL | 35200 |
| 102 | Barrington, Kyle | MGT | DHEAD | FULL | 45000 |
| 103 | Chapman, Margaret | LIB | STAFF | PART | 24200 |
| 104 | Jackson, Herbert | RND | STAFF | FULL | 33000 |
| 105 | Price, Stella | FIN | DHEAD | FULL | 42000 |
| 106 | Sanchez, Carla | MKT | STAFF | FULL | 38500 |
| 107 | Smith, Roberta | RND | STAFF | PART | 27500 |

Notice that only the STAFF rows are changed to reflect the increase. DHEAD row salaries remain as they were. As a word of caution, it's easy to “accidentally” modify all rows in a table. Check your statement carefully before executing it.

9.3 Making a Taxis Owned File

When a file is inserted into an INDIRECT column, the ownership and location of the file remains as it was when loaded. If the resume file called “/usr/local/resume/smith.res” was owned by the Library, it will remain so when pointed to by the INDIRECT column unless you take steps to make it otherwise. For example, if Personnel owns the RESUME table but not the files themselves, an attempt to update the resume files would not be successful. The management and handling of the resume files is still in the domain of the Library.

The system of INDIRECT data types is a system of pointers to files. The file pointed to can either exist on the system already and remain where it is, or you can instruct Taxis to create a copy of the file under its own ownership and control. In either case, the file still exists outside of Taxis.

Where you want Taxis to own a copy of the data, a Taxis owned file can be made with the TOIND function. You can then do whatever you want with one version without affecting the other, including removing the original if that is appropriate. The permissions on such Taxis owned files will be the same as the ownership and permissions assigned to the Taxis table which owns it.

The file is copied into the table using an UPDATE statement. The form of UPDATE is the same, but with special use of the expression for the column name following SET. The form of this portion of the UPDATE statement would be:

```
UPDATE  table-name
SET      column-name = toind (fromfile ('local-file') ) ;
```

The change you are making is to the named column. With SET, you are taking text from the file (“fromfile”) as it currently exists on the system (“local-file”), and copying it to an INDIRECT text column (“toind”) pointed to by the Taxis table named by UPDATE. The name of the local file is in quotes, as it is a character string, and is in parentheses as the argument of the function “fromfile”. The whole “fromfile” function is in parentheses as the argument of the function “toind”.

Example: To make a Taxis owned copy of the Smith resume file for the RESUME table, use this UPDATE statement:

```
UPDATE  RESUME
SET      EXP = toind (fromfile ('/usr/local/resume/smith.res') ) ;
```

The “smith.res” file now exists as part of the Taxis table RESUME, while still remaining outside it. Once you have made Taxis owned copies of any such files, you can operate on the text in the table without affecting the originals. And you can decide whether it is prudent to retain the original copies of the files or whether that would now be an unnecessary use of space.

9.4 Deleting Records

Records are removed from the database when they are no longer relevant to the application. For example, if an employee leaves the company, data concerning that person can be removed. Or if we wish to remove the data from certain departments which are not of interest to the pay raise analysis, we can delete those records from the temporary analysis table.

Deleting a record removes all data values in a row from a table. One or more rows from a table can be deleted with the use of the DELETE command. This command has the following form:

```
DELETE FROM  table-name
[WHERE  search-condition] ;
```

Command Discussion

- The `DELETE FROM` clause indicates you want to remove a row from a table. Following this clause, the user specifies the name of the table from which data is to be deleted.
- To find the record or records being deleted, use a search condition similar to that used in the `SELECT` statement.
- Where `INDIRECT` text columns are concerned, such rows will be deleted just as any other when `DELETE FROM` is used. However, the files pointed to by `INDIRECT` will only be removed where managed by Taxis, as defined in the previous section on Taxis owned files.

An employee whose ID number is 117 has quit his job. Use this statement to delete his record from the `EMPLOYEE` table.

```
DELETE FROM EMPLOYEE
WHERE  EID = 117 ;
```

All records which satisfy the search condition are deleted. In this case, one record is deleted from the table. Note that the entire record:

```
117  Peters, Robert      SPI      DHEAD  FULL      34000
```

is deleted, not just the column specified in the `WHERE` clause.

When you delete records, aim for consistency. For example, if you intend to delete Peters' record in the `EMPLOYEE` table, you must also delete the reference to Peters as department head in the `DEPARTMENT` table and so on. This would involve two separate operations.

Example: Let's say we want to delete all the department heads from the `EMP_RAISE` table as they are not really part of the analysis. Use this statement:

```
DELETE FROM EMP_RAISE
WHERE  RANK = 'DHEAD' ;
```

The block of all records of employees who are department heads are removed from the `EMP_RAISE` table, leaving the table with just these entries:

| EID | ENAME | DEPT | RANK | BENEFITS | SALARY |
|-----|-------------------|------|-------|----------|--------|
| 101 | Aster, John A. | MKT | STAFF | FULL | 32000 |
| 103 | Chapman, Margaret | LIB | STAFF | PART | 22000 |
| 104 | Jackson, Herbert | RND | STAFF | FULL | 30000 |
| 106 | Sanchez, Carla | MKT | STAFF | FULL | 35000 |
| 107 | Smith, Roberta | RND | STAFF | PART | 25000 |

If the finance analyst wanted to empty the table of existing entries and perhaps load in new ones from a different part of the organization, this could be done with this statement:

```
DELETE FROM EMP_RAISE ;
```

All rows of EMP_RAISE would be deleted, leaving an empty table. However, the definition of the table has not been deleted; it still exists even though it has no data values, so rows can be added to the table at any time.

It is important to note the difference between the DELETE command and the DROP TABLE command. In the former, you eliminate one or more rows from the indicated table. However, the structure of the table is still defined, and rows can be added to the table at any time. In the case of the DROP TABLE command, the table definition is removed from the system catalog. You have removed not only access to the data in the table, but also access to the table itself. Thus, to add data to a “dropped” table, you must first create the table again.

Chapter 10

Security

Many people have access to a database: managers, analysts, data-entry clerks, programmers, temporary workers, and so on. Each individual or group needs different access to the data in the database. For example, the Finance Director needs access to salary data, while the receptionist needs access only to names and departments. R&D needs access to the library's research reports, while Legal needs access to depositions in pertinent court cases.

Taxis maintains permissions which work in conjunction with the operating system security. Taxis will not change the operating system permissions on a table, but it will change the permissions on the indices to match those on the table.

This scheme allows the operating system to give a broad class of security, while Taxis maintains finer detail. The reason for the combination is that Taxis can not control what the user does with the operating system, and the operating system does not have the detailed permissions required for a database.

When a table is created it initially has full permissions for the creator, and read/write operating system permissions for the creator only.

When using Taxis permissions the operating system can still deny access which Taxis believes is proper. To prevent this from happening Taxis should always be run as one user id, which owns the database. The easiest way of doing this on Unix is to set the `suid` bit on all the programs that form the Taxis package, as well as any user programs written with the direct library, and change the user to a common user, for example `taxis`. Alternative methods may exist for other operating systems.

10.1 Creating Users and Logging In

When a database is created there are two users created by default. The default users are `PUBLIC` and `_SYSTEM`. `PUBLIC` has the minimal permissions possible in Taxis, and `_SYSTEM` has the maximum permissions in Taxis. When these are created they are created without any password. You should change the password on `_SYSTEM` to prevent security issues. The password on `PUBLIC` can be left blank to allow anonymous access to the database, or it can be set to restrict access.

When logging into Taxis the username will default to `PUBLIC` if none is specified. This means that tables

will be created owned by `PUBLIC`, and all users will have permissions to use the tables. It is possible to use only the `PUBLIC` user while developing a database, although if multiple people are working on the database you should see the previous comments about operating system permissions.

To create new users in the database you must may either use the program `tsql -a a`, and you will be prompted for the user's information, or you can use the `CREATE USER SQL` statement. You must log in as `_SYSTEM` to add or delete users.

The syntax for the user administration command in `SQL` are as follows:

```
CREATE USER username IDENTIFIED BY password ;
```

```
ALTER USER username IDENTIFIED BY password ;
```

```
DROP USER username ;
```

You may issue the `ALTER USER` command if you are logged in as the same user that is given in the statement, or if you are `_SYSTEM`. The password should be given as a string in single quotes. The `ALTER USER` statement is used to change a user's password; the new password being specified in the command. Dropping a user will not remove any tables owned by that user.

10.2 Granting Privileges

In Taxis, the creator of the database would be the automatic administrator of security. This individual can grant to other users different powers, such as the ability to read only, to modify, or to delete data in the database. Through the authorization subsystem, user names and password control which users can see what data. Each user signs onto the computer system with his or her own user name and password (i.e., user identification) and cannot access without permission tables created by some other user with a different user name.

The person who creates a table is considered the "owner" of the table. Initially, that person is the only one who can access, update, and destroy the table. The owner, however, can grant to other users the right, or privilege, to do the following:

- Access the tables created by the owner.
- Add, change, or delete values in a table.
- Grant rights the user receives from the owner to other users.

The owner of the table can grant to other users privileges that include the following:

`SELECT`: Retrieve rows without changing values in a table.

`INSERT`: Add new rows to a table.

UPDATE: Change values in a table.

DELETE: Remove rows from a table.

The authorization subsystem of Taxis is based on privileges that are controlled by the statements GRANT and REVOKE. The GRANT command allows the “owner” of a table to specify the operations, or privileges, that other users may perform on a table. The format of the command is:

```
GRANT    [ALL]
         privilege1 [,privilege2] ...
ON       table-name1
TO       PUBLIC
         userid1 [,userid2] ...
[WITH GRANT OPTION] ;
```

Command Discussion

- GRANT is a required keyword that indicates you are granting access to tables to other users.
- Privilege refers to the type of privilege or privileges you are granting. One or more of the following privileges can be granted: SELECT, INSERT, UPDATE, DELETE, and ALTER. Alternatively, ALL can be specified if all of the above actions are to be granted to the user.
- ON indicates the table(s) to which these privileges are being assigned.
- PUBLIC is used if the privileges are to be granted to all users. If you want only certain users to have privileges assigned to this table, you must list the user identifications (“userid’s”) of all those who will be allowed to share the table.
- If the clause WITH GRANT OPTION is specified, the recipient of the privileges specified can grant these privileges to other users.

Example: The Systems Administrator in the Information Systems Management Department created the EMPLOYEE table, and is therefore its owner. As owner of the EMPLOYEE table, he grants the SELECT privilege to the firm’s CPA in Accounting. As owner of the table he issues the following command:

```
GRANT    SELECT
ON       EMPLOYEE
TO       CPA ;
```

Syntax Notes:

- When the SELECT privilege is granted, it is done so with read-only access. Therefore the person granted the SELECT privilege can read the data in the table, but cannot write to it, or in other words, cannot change it with UPDATE or other such privileges.

- ON refers to the table these privileges are being granted on; in this case, the EMPLOYEE table.
- What follows TO is the user ID (userid) of the person to whom the privilege is granted. In this case the SELECT privilege is granted to the person in accounting whose user ID is “CPA”.

Example: The owner of the EMPLOYEE table allows the clerks in Personnel to add and modify employee data with this command:

```
GRANT    UPDATE, INSERT
ON       EMPLOYEE
TO       CLERK1, CLERK2 ;
```

In this case there are two clerks with two separate user ID’s, “CLERK1” and “CLERK2”. Both are granted privileges to UPDATE and INSERT new information into the EMPLOYEE table.

Example: The owner of the EMPLOYEE table, the System Administrator, gives the Director of Personnel complete access (SELECT, INSERT, UPDATE, DELETE, ALTER) to the EMPLOYEE table, along with permission to assign these privileges to others. This statement is used:

```
GRANT    ALL
ON       EMPLOYEE
TO       PERS
WITH GRANT OPTION ;
```

ALL following GRANT includes all 5 of the privileges. PERS is the user ID of the Director of Personnel. WITH GRANT OPTION allows the Director of Personnel to grant these privileges to other users.

Example: A systems analyst in the Strategic Planning and Intelligence Department has created and is owner of the NEWS table in which they are daily archiving online news articles of interest. It is decided to give all employees read-only access to this database. Owner of the table can do so with this command:

```
GRANT    SELECT
ON       NEWS
TO       PUBLIC ;
```

Anyone with access to the server on which the news table is stored will have permission to read the articles in the NEWS table, since the SELECT privilege has been granted to PUBLIC.

10.3 Removing Privileges

Privileges assigned to other users can be taken away by the person who granted them. In Taxis, the REVOKE statement would be used to remove privileges granted by the GRANT command. The general form of this statement is:

```
REVOKE  [ALL]
        privilege1 [,privilege2] ...
ON      table-name1
TO      PUBLIC
        userid1 [,userid2] ... ;
```

Command Discussion

- REVOKE is a required keyword that indicates you are removing access to tables .
- Privilege refers to the type of privilege or privileges you are revoking. One or more of the following privileges can be revoked: SELECT, INSERT, UPDATE, DELETE, and ALTER. Alternatively, ALL can be specified if all of the above actions are to be taken away from the user.
- The ON clause indicates the table(s) from which these privileges are being removed.
- PUBLIC is used if the privileges are taken away from all users of the indicated table(s). Otherwise, you list the user names of only those who are no longer allowed to share the table.

Example: The Personnel clerks no longer need to access the EMPLOYEE table. Revoke their privileges as follows:

```
REVOKE  UPDATE, INSERT
ON      EMPLOYEE
FROM    CLERK1, CLERK2 ;
```

This completes the discussion of security features in Taxis. In the next chapter, you will be introduced to certain other administrative features that can be implemented in Taxis.

Chapter 11

Administration of the Database

This chapter covers topics related to the administration of the database. The topics include the following:

- Accessing information about the database by using Taxis's system catalog.
- Taxis reserved words to avoid in naming tables and columns.

11.1 System Catalog

In Taxis, information about the database, such as the names of tables, columns, and indexes, is maintained within a set of tables referred to as the *system catalog*. Taxis automatically maintains these tables in the system catalog in response to commands issued by users. For example, the catalog tables are updated automatically when a new table is defined using the `CREATE TABLE` command.

Database administrators and end users can access data in the system catalog just as they access data in other Taxis tables by using the `SELECT` statement. This enables a user to inquire about data in the database and serves as a useful reference tool when developing queries.

Table 11.1 lists the tables that make up the system catalog for Taxis.

Table 11.1: Overview of System Catalog Tables in Taxis

| Table Name | Description |
|--------------|---|
| SYSTABLES | Contains one row per table in the database. |
| SYS_COLUMNS | Contains one row per column for each database table. |
| SYSINDEX | Contains one row per canonical index in the database. |
| SYSPERMS | Holds the permissions information. |
| SYSUSERS | Contains information about users known to the system. |
| SYSSTRIG | Contains one row per trigger defined to the system. |
| SYSMETAINDEX | Contains one row per Metamorph index in the database. |

One commonly referenced table, SYSTABLES, contains a row for each table that has been defined. For each table, the name of the table, authorized ID of the user who created the table, type of table, and so on is maintained. When users access SYSTABLES, they see data pertaining to tables that they can access.

Texis's system catalog table, "SYSTABLES" has these columns, defined with the following data types:

| | | |
|---------|---|------------|
| NAME | – | CHAR (20) |
| TYPE | – | CHAR |
| WHAT | – | CHAR (255) |
| FC | – | BYTE |
| CREATOR | – | CHAR (20) |
| REMARK | – | CHAR (80) |

Each field is fixed length rather than variable length, so the designated size limits do apply.

NAME is the name of the table. Each of the tables comprising the system catalog are entered here, as well as each of the other database relations existing as "normal" tables.

TYPE indicates the type of table.

S indicates a System table, and is Texis owned. 'S' is assigned to all tables where the user who created the table is "texis".

T indicates a normal Table.

V indicates a normal View.

B indicates a Btree table. A Btree is a special type of table that can be created through the API only, that contains all the data in the index. It is of limited special purpose use. It is somewhat quicker and more space efficient if you have a few, small fields, and if you will never need to index on the fields in a different order. Use of the API is covered in Part V, Chapter 1.

t indicates a temporary table. These are not directly accessible, and exist only briefly. They are used when a temporary table is needed by the system – for example when compacting a table – and may have the same name as another, normal table. They are automatically removed when no longer needed.

D indicates a Deleted table. On some operating systems (such as Windows), when a table is DROPPed, it cannot be removed immediately and must continue to exist – as a deleted table – for a short time. Deleted tables are not directly accessible, and are automatically removed as soon as possible.

WHAT is the filename designating where the table actually exists on the system.

FC stands for Field Count. It shows how many columns have been defined for each table entered.

CREATOR is a User ID and shows who created the table.

REMARK is reserved for any explanatory comments regarding the table.

Example: Provide a list of all tables in the database with this statement:

```
SELECT  NAME, TYPE
FROM    SYSTABLES ;
```

The result will be a listing of the available tables, as follows:

| NAME | TYPE |
|--------------|------|
| SYSCOLUMNS | S |
| SYSINDEX | S |
| SYSMETAINDEX | S |
| SYSTABLES | S |
| CODES | T |
| DEPARTMENT | T |
| EMPLOYEE | T |
| NEWS | T |
| REPORT | T |
| RESUME | T |

In the above example, the first four tables: SYSCOLUMNS, SYSINDEX, SYSMETAINDEX, and SYSTABLES, comprise the system catalog and are marked as type S, for “*system*”.

The next six in the list are the tables which have been used for examples throughout this manual: CODES, DEPARTMENT, EMPLOYEE, NEWS, REPORT, and RESUME. These are marked as type T, for “*table*”.

The table SYSCOLUMNS contains a row for every column of every table in the database. For each column, its name, name of the table to which it belongs, data type, length, position in the table, and whether NULL is permitted in the columns is maintained information. Users querying SYSCOLUMNS can retrieve data on columns in tables to which they have access.

Texis’s system catalog table “SYSCOLUMNS” has these columns, defined with the following data types:

```
NAME      -  CHAR (20)
TBNAME    -  CHAR (20)
TYPE      -  CHAR (15)
INDEX     -  CHAR (20)
NONNULL   -  BYTE
REMARK    -  CHAR (80)
```

NAME is the column name itself.

TBNAME is the table the column is in.

TYPE is the data type assigned to the column, defined as a string. TYPE might contain “char”, “varchar”, “integer”, “indirect”, and so on.

INDEX is the name of an index created on this column. (This field is reserved for use in future versions of Texis. As it is not currently being used, one should not be surprised if the INDEX field is empty.)

NONNULL indicates whether NULL fields should be disallowed. (This field is reserved for use in future versions of Taxis. As it is not currently being used, one should not be surprised if the INDEX field is empty.)

REMARK is reserved for any user comment about the column.

Example: A user wants to obtain data about employees in the R&D Department, but doesn't know any of the column names in the EMPLOYEE table. Assume that the user does know there is a table named EMPLOYEE.

This statement:

```
SELECT  NAME
FROM    SYSCOLUMNS
WHERE   TBNAME = 'EMPLOYEE' ;
```

would result in the following:

```
NAME
EID
ENAME
DEPT
RANK
BENEFITS
SALARY
```

In this way one can find out what kind of data is stored, so as to better formulate queries which will reveal what you actually want to know.

Taxis has two other system catalog tables called "SYSINDEX" and "SYSMETAINDEX". Taxis's system catalog table "SYSINDEX" has these columns, defined with the following data types:

```
NAME      - CHAR(20)
TBNAME    - CHAR(20)
FNAME     - CHAR(20)
ORDER     - CHAR
TYPE      - BYTE
UNIQUE    - BYTE
FIELDS    - CHAR(20)
```

NAME is the name of the index.

TBNAME is the table the index is on.

FNAME is the file name of the index.

ORDER indicates sort order. ‘A’ indicates *ascending*; ‘D’ indicates *descending*. This field is not currently used, but is planned for future releases.

TYPE indicates the type of index, either Btree or Metamorph.

UNIQUE indicates whether the values entered should be unique. This field is not currently used, but is planned for future releases.

FIELDS indicates which field is indexed.

“SYSMETAINDEX” controls a demon that checks Metamorph indexes, those indexes used on text oriented columns. The demon waits a certain number of seconds between checks, and has a threshold in bytes at which size the update process is required to run.

Texis’s system catalog table “SYSMETAINDEX” has these columns, defined with the following data types:

| | | |
|--------|---|-----------|
| NAME | – | CHAR (20) |
| WAIT | – | INTEGER |
| THRESH | – | INTEGER |

NAME is the name of the Metamorph index.

WAIT indicates how long to wait in seconds between index checks.

THRESH is a number of bytes which have changed. This is the threshold required to re-index.

The system catalog tables are a good place to start when initially becoming familiar with what a database has to offer.

11.2 Optimization

11.2.1 Table Compaction

After a table has been extensively modified, its disk file(s) may accumulate a certain amount of unused free space, especially if a large number of rows have been deleted. This free space will be re-used as much as possible whenever new rows are inserted or updated, to try to avoid expanding the table’s disk footprint. However, if the table is no longer to be modified in the future – e.g. it is now a search-only archive – this free space will never be reclaimed. It is now wasted disk space, as well as a potential performance impairment, as larger seeks may be needed by the operating system to access actual payload data.

Free space in a table may be reclaimed by compacting the table (retaining all payload data), with the following SQL:

```
ALTER TABLE name COMPACT
```

This will compact the table *name* to eliminate its free space. The process may take some time for a large table, or where there are many indexes on it. Also, while the end result will generally be less disk usage for

the table, *during* the compaction disk usage will temporarily increase, as copies of the table and most of its index files are created. Therefore, before starting, ensure that there is free disk space (in the database's partition) at least equal to the combined size of the table and its indexes.

Because extensive modifications are needed, the table will not be modifiable during compaction: attempts to insert, delete or update rows will block until compaction is finished. The table is readable during compaction, however, so `SELECT`s are possible. Progress meters may be printed during compaction by setting the SQL property `meter` to `'compact'`. The `ALTER TABLE name COMPACT` syntax was added in version 6.00.1291080000 20101129. **NOTE: Versions prior to version 6.00.1291080000 20101129 should not attempt to access the table during compaction, or corruption may result.**

Note that compacting a table is generally only useful when the table will no longer be modified, or has undergone a large amount of deletions that will not be replaced by inserts. Conversely, a “steady-state” continuously-modified table rarely benefits from compaction, because it will merely accumulate free space again: the short-term gains of compaction are outweighed by the significant cost and delay of repeatedly running the compaction.

11.2.2 Index Maintenance

B-tree (regular) and inverted indexes never require explicit optimization by the database administrator, as they are automatically kept up-to-date (optimized) at every table modification (`INSERT`, `DELETE` or `UPDATE`).

However, this is not possible for Metamorph indexes due to their fundamentally different nature. Instead, table changes are logged for incorporation into the index at the next optimization (index update), and Taxis must linearly search the changed data until then. Thus, the more a table has been modified since a Metamorph index's last optimization, the more its search performance potentially degrades. When the index is re-optimized, those changes are indexed and merged into the Metamorph index, restoring its performance. A Metamorph index may be optimized in one of several ways, as follows.

Manual Index Optimization via `CREATE METAMORPH INDEX`

A Metamorph index may be optimized manually simply by re-issuing the same `CREATE METAMORPH [INVERTED|COUNTER] INDEX ...` statement that was used to create it. Instead of producing an error noting that the index already exists – as would happen with regular or inverted indexes – the Metamorph index is re-optimized. (If the index is already fully optimized, the statement returns success immediately.)

Note that the `ALTER INDEX` statement (p. 162) is an easier method of optimizing indexes.

Manual Index Optimization via `ALTER INDEX`

Since the full syntax of the original `CREATE` statement may not be known, or may be cumbersome to remember and re-enter, a Metamorph index may also be optimized with an `ALTER INDEX` statement:

```
ALTER INDEX indexName [ALL [ON tableName]]
      OPTIMIZE|REBUILD
```

This will optimize the index named *indexName*, or all indexes in the database if `ALL` is given. Adding the optional `ON tableName` clause will limit the index(es) optimized to only those on the table named *tableName*. If a non-Metamorph index is specified, it will be silently ignored, as non-Metamorph indexes are always in an optimized state.

If the keyword `REBUILD` is given instead of `OPTIMIZE`, the index is rebuilt from scratch instead. This usually takes more time, as it is the same action as the initial creation of the index, and thus the whole table must be indexed, not just changes since last optimization. Any index type may be rebuilt, not just Metamorph indexes. During rebuilding, the original index is still available for search use; however inserts, deletes and updates may be postponed until the rebuild completes. Rebuilding is not generally needed, but may be useful if the index is suspected to be corrupt. The `ALTER INDEX` syntax was added in Taxis version 7.

Automatic Index Optimization via `chkind`

Another method of Metamorph index optimization is automatically, via the `chkind` daemon, and is enabled by default. This is a process that runs automatically in the background (as part of the database monitor), and periodically checks how out-of-date Metamorph indexes are. When an index reaches a certain (configurable) threshold of “staleness”, it is re-optimized. See p. ?? for more details on `chkind` and its configuration.

Choosing Manual vs. Automatic Index Optimization

Whether to optimize Metamorph indexes manually (via a SQL statement) or automatically (via `chkind`) depends on the nature of table changes and searches.

Deployments where table changes occur in batches, and/or search load predictably ebbs and flows, are good candidates for manual optimization. The optimizations can be scheduled for just after the batch table updates, and if possible when search load is low. This will keep the index(es) up-to-date (and thus performing best) for the longest amount of time, while also avoiding the performance penalty of updating both the table and the index simultaneously. Optimizing at off-peak search times also improves peak-load search performance by freeing up resources during the peak. Contrast this with automatic optimization, which cannot know about upcoming table updates or search load, and thus might trigger an index update that coincides with either, negatively impacting performance.

Applications where tables are changed at a more constant rate (e.g. a steady stream of changes) may be better candidates for automatic updating. There may not be any predictable “best time” to run the optimization, nor may it be known how much the indexes are out-of-date. Thus the decision on when to optimize can be left to `chkind`’s automatic out-of-date scan, which attempts to minimize both staleness of the index and frequency of index optimizations.

Some situation may call for a combination, e.g. `chkind` to handle miscellaneous table updates, and an occasional manual optimization after batch updates, or just before peak search load.

11.3 Reserved Words

The following words are reserved words in SQL. Taxis makes use of many of them, and future versions may make use of others. These words should not be used as ordinary identifiers in forming names.

Allowances will be made in future versions of Taxis so that the words may be used as delimited identifiers if deemed vital, by enclosing them between double quotation marks.

| | | | |
|----------------|---------------|------------|------------|
| ADA | DELETE | INTO | REFERENCES |
| ADD | DESC | IS | REVOKE |
| ALL | DESCRIPTOR | KEY | ROLLBACK |
| ALTER | DISTINCT | LANGUAGE | SCHEMA |
| AND | DOUBLE | LIKE | SECQTY |
| ANY | DROP | LIKE3 | SELECT |
| AS | EDITPROC | LOCKSIZE | SET |
| ASC | END-EXEC | MATCHES | SMALLINT |
| AUTHORIZATION | ERASE | MAX | SOME |
| AVG | ESCAPE | METAMORPH | SQLCODE |
| BETWEEN | EXECUTE | MIN | STOGROUP |
| BLOB | EXISTS | MODULE | SUM |
| BUFFERPOOL | FETCH | NOT | SYNONYM |
| BY | FIELDPROC | NULL | TABLE |
| C | FLOAT | NUMERIC | TABLESPACE |
| CHAR (ACTER) ? | FOR | NUMPARTS | TO |
| CHECK | FOREIGN | OF | UNION |
| CLOSE | FORTTRAN | ON | UNIQUE |
| CLUSTER | FOUND | OPEN | UPDATE |
| COBOL | FROM | OPTION | USER |
| COLUMN | GO | OR | USING |
| COMMIT | GO [\t] *TO | ORDER | VALIDPROC |
| COMPACT | GOTO | PART | VALUES |
| CONTINUE | GRANT | PASCAL | VARCHAR |
| COUNT | GROUP | PLAN | VCAT |
| CREATE | HAVING | PLI | VIEW |
| CTIME | IMMEDIATE | PRECISION | VOLUMES |
| CURRENT | IN | PRIMARY | WHENEVER |
| CURSOR | INDEX | PRIQTY | WHERE |
| DATABASE | INDICATOR | PRIVILEGES | WITH |
| DATE | INDIRECT | PROCEDURE | WORK |
| DECIMAL | INSERT | PUBLIC | |
| DECLARE | INT (EGER) ? | REAL | |
| DEFAULT | | | |

11.4 Server Properties

There are a number of properties that are settable in the SQL Engine. They do not need to be changed unless the behavior of the system must be modified. The properties are set using the following SQL syntax:

```
SET property = value;
```

The `value` can be one of three types depending on the property: numeric, boolean or string. A boolean value is either an integer—0 is false, anything else is true—or one of the following strings: “on”, “off”, “true”, “false”, “yes” or “no”.

The settings are grouped as follows:

11.4.1 Search and optimization parameters

These settings affect the way that Taxis will process the search. They include settings which change the meaning of the search, as well as how the search is performed.

defaultlike Defines which sort of search should occur when a `like` or `contains` operator is in the query. The default setting of “like” behaves in the normal manner. Other settings that can be set are “like3”, “likep”, “liker” and “matches”. In each case the `like` operator will act as if the specified operator had been used instead.

matchmode Changes the behavior of the `matches` clause. The default behavior is to use underscore and percent as the single and multi-character character wildcards. Setting `matchmode` to 1 will change the wildcards to question-mark and asterisk.

predopttype The Taxis engine can reorder the `where` clause in an attempt to make it faster to evaluate. There are a number of ways this can be done; the `predopttype` property controls the way it reorders. The values are 0 to not reorder, 1 to evaluate `and` first, 2 to evaluate `or` first. The default is 0.

ignorecase **Note:** Deprecated; see `stringcomparemode` setting which supercedes this. Setting `ignorecase` to true will cause string comparisons (equals, sorting, etc.) in the SQL engine to ignore case, e.g. “A” will compare identical to “a”. (This is distinct from *text* comparisons, e.g. the `LIKE` operator, which ignore case by default and are unaffected by `ignorecase`.) **Note:** This setting will also affect any indices that are built; the value set at index creation will be saved with the index and used whenever that index is used. **Note:** In versions prior to version 5.01.1208300000 20080415, the value of `ignorecase` *must* be explicitly set the same when an index is created, when it or its table is updated and when it is used in a search, or incorrect results and/or corrupt indexes may occur. In later versions, this is not necessary; the saved-at-index-creation value will automatically be used. In version 6 and later, this setting toggles the `ignorecase` flag of the `stringcomparemode` setting, which supercedes it.

textsearchmode Sets the APICP `textsearchmode` property; see Vortex manual for details and important caveats. Added in version 6.

stringcomparemode Sets the APICP `stringcomparemode` property; see Vortex manual for details and important caveats. Added in version 6.

tracemetamorph Sets the `tracemetamorph` debug property; see Vortex manual for details. Added in version 7.00.1375225000 20130730.

tracerowfields Sets the `tracerowfields` debug property; see Vortex manual for details. Added in version 7.02.1406754000 20140730.

tracekdbf Sets the `tracekdbf` debug property; see Vortex manual for details.

tracekdbffile Sets the `tracekdbffile` debug property; see Vortex manual for details.

kdbfiostats Sets the `kdbfiostats` debug property; see Vortex manual for details.

btrecachesize Index pages are cached in memory while the index is used. The size of the memory cache can be adjusted to improve performance. The default is 20, which means that 20 index pages can be cached. This can be increased to allow more pages to be cached in memory. This will only help performance if the pages will be accessed in random order, more than 20 will be accessed, and the same page is likely to be accessed at different times. This is most likely to occur in a join, when a large number of keys are looked up in the index. Increasing the size of the cache when not needed is likely to hurt performance, due to the extra overhead of managing a larger cache. The cache size should not be decreased below the default of 20, to allow room for all pages which might need to be accessed at the same time.

ramrows When ordering large result sets, the data is initially ordered in memory, but if more than `ramrows` records are being ordered the disk will be used to conserve memory. This does slow down performance however. The default is 10000 rows. Setting `ramrows` to 0 will keep the data in memory.

ramlimit `ramlimit` is an alternative to `ramrows`. Instead of limiting the number of records, the number of bytes of data in memory is capped. By default it is 0, which is unlimited. If both `ramlimit` and `ramrows` are set then the first limit to be met will trigger the use of disk.

bubble Normally Taxis will bubble results up from the index to the user. That is a matching record will be found in the index, returned to the user, then the next record found in the index, and so forth till the end of the query. This normally generates the first results as quickly as possible. By setting `bubble` to 0 the entire set of matching record handles will be read from the index first, and then each record processed from this list.

optimize,nooptimize Enable or disable optimizations. The argument should be a comma separated list of optimizations that you want to enable or disable. The available optimizations are:

join Optimize join table order. The default is enabled. When enabled Taxis will arrange the order of the tables in the `FROM` clause to improve the performance of the join. This can be disabled if you believe that Taxis is optimizing incorrectly. If it is disabled then Taxis will process the tables in the left to right order, with the first table specified being the driving table. Added in version 02.06.927235551.

compoundindex Allow the use of compound indexes to resolve searches. For example if you create an index on table (field1, field2), and then search where field1 = value and field2 = value, it will use the index to resolve both portions of this. When disabled it would only look for field1 in the index. Added in version 02.06.929026214.

countstar Use any regular index to determine the number of records in the table. If disabled Taxis will read each record in the table to count them. Added in version 02.06.929026214.

minimallocking Controls whether the table will be locked when doing reads of records pointed to by the index used for the query. This is enabled by default, which means that read locks will not be used. This is the optimal setting for databases which are mostly read, with few writes and small records. Added in version 03.00

groupby This setting is enabled by default and will cause the data to be read only once to perform a group by operation. The query should produce identical results whether this is enabled or disabled, with the performance being the only difference. Added in version 03.00

faststats When enabled, which is the default, and when the appropriate indexes exist Taxis will try and resolve aggregate functions directly from the index that was used to perform the WHERE clause. Added in version 03.00

readlock When enabled, which is the default, Taxis will use readlocks more efficiently if there are records that are scanned, but don't match the query. Taxis will hold the read lock until a matching record is found, rather than getting and releasing a read lock for every record read. If you are suffering from lock contention problems, with writes waiting, then this can be disabled, which will allow more opportunity for the write locks to be granted. This is not normally suggested, as the work required to grant and release the locks would typically negate the benefit. Added in version 03.00

analyze When enabled, which is the default, Taxis will analyze the query for which fields are needed. This can allow for more efficient query processing in most cases. If you are executing a lot of different SQL statements that are not helped by the analysis you can disable this. Added in version 03.00

skipahead When enabled, which is the default, Taxis will skipahead as efficiently as possible, typically used with the SKIP parameter in Vortex. If disabled Taxis will perform full processing on each skipped record, and discard the record. Added in version 03.00

likewithnots When enabled (default), LIKE/LIKEP-type searches with NOT sets (negated terms) are optimized for speed. Added in version 4.02.1041535107 Jan 2 2003.

shortcuts When enabled (default), a fully-indexed LIKE/LIKEIN clause ORed with another fully-indexed LIKE/LIKEIN should not cause an unnecessary post-process for the LIKES (and entire query). Added in version 4.03.1061229000 20030818 as optimization18; in version 7.06.1475000000 20160927, alias shortcuts added.

likehandled When enabled (default), a fully-indexed LIKE/LIKEIN clause ORed with another fully-indexed non-LIKE/LIKEIN clause should not cause an unnecessary post-process for the LIKE (and entire query).

Also, linear and post-process LIKE/LIKEIN operations caused not by the Metamorph query itself, but by the presence of another ORed/ANDED clause, do not check alllinear nor alpostproc when this optimization is disabled (i.e. they will perform the linear or post-process regardless of settings, silently). E.g. fully-indexed LIKE ORed with linear clause, or two fully-indexed LIKES ANDED (where the first's results are under maxlinearrows),

could cause linear search or post-processing, respectively, of an otherwise fully-indexable Metamorph query.

Added in version 7.06.1475014000 20160927.

indexbatchbuild When enabled, indexes are built as a batch, i.e. the table is read-locked continuously. When disabled (the default), the table is read-locked intermittently if possible (e.g. Metamorph index), allowing table modifications to proceed even during index creation. A continuous read lock allows greater read buffering of the table, possibly increasing index build speed (especially on platforms with slow large-file `lseek` behavior), at the expense of delaying table updates until after the index is nearly built, which may be quite some time. Note that non-Metamorph indexes are *always* built with a continuous read lock – regardless of this setting – due to the nature of the index. Added in version 5.01.1177455498 20070424.

indexdataonlycheckpredicates When enabled (the default), allows the index-data-only optimization¹ to proceed even if the `SELECT` columns are renamed or altered in expressions. Previously, the columns had to be selected as-is with no renaming or expressions. Added in version 7.00.1369437000 20130524.

indexvirtualfields When enabled (the default), attempts to reduce memory usage when indexing virtual fields (especially with large rows) by freeing certain buffers when no longer needed. Currently only applies to Metamorph and Metamorph inverted indexes. Added in version 6.00.1322890000 20111203.

Example: `set nooptimize='minimallocking'`

options,nooptions Enable or disable certain options. The argument should be a comma separated list of options to enable or disable. All options are off by default. The available options are:

triggers When on, *disable* the creation of triggers.

indexcache Cache certain Metamorph index search results, so that an immediately following Metamorph query with the same `WHERE` clause might be able to re-use the index results without re-searching the index. E.g. may speed up a `SELECT field1, field2, ... Metamorph` query that follows a `SELECT count(*)` query with the same `WHERE` clause.

ignoremissingfields Ignore missing fields during an `INSERT` or `UPDATE`, i.e. do not issue a message and fail the query if attempting to insert a non-existent field. This may be useful if a `SQL INSERT` statement is to be used against a table where some fields are optional and may not exist.

Example: `set options='indexcache'`

ignorenewlist When processing a Metamorph query you can instruct Taxis to ignore the unoptimized portion of a Metamorph index by issuing the `SQL set ignorenewlist = 1;`. If you have a continually changing dataset, and the index is frequently updated then the default of processing the unoptimized portion is probably correct. If the data tends to change in large batches, followed by a reoptimization of the index then the large batch can cause significant processing overhead. In that case it may be wise to enable the `ignorenewlist` option. If the option is enable then records that

¹The index-data-only optimization allows Taxis to not only use the index to resolve the `WHERE` clause, but also the `SELECT` clause in certain circumstances, potentially avoiding a read of the table altogether and speeding up results. One of the prerequisites for this optimization is that the `SELECT` clause only refer to columns available in the index.

have been updated in the batch will not be found with Metamorph queries until the index has been optimized. Added in version 02.06.934400000.

indexwithin How to use the Metamorph index when processing “within *N*” (*w/N*) LIKE-type queries. It is an integer combination of bit flags:

0x01 : Use index for *w/N* searches when `withinmode` is “char [span]”

0x02 : Use index for *w/N* searches when `withinmode` is “word [span]”

0x04 : Optimize within-chars window down

0x08 : Do not scale up intervening (non-query) words part of window to account for words matching multiple index expressions, which rarely occur; this reduces false (too wide) hits from the index. Also do not require post-processing if multiple index expressions. In rare cases valid hits may be missed if an intervening word does index-match multiply; the *N* value can simply be increased in the query to return these.

The default is 0xf in version 7.06.1525203000 20180501 and later, when support for 0x8 was also added. In version 5.01.1153865548 20060725 up to then, the default was 0x7. The setting was added in version 4.04.1075255999 20040127 with a default of 0.

wildoneword Whether wildcard expressions in Metamorph queries span a single word only, i.e. for multi-substring wildcards. If 0 (false), the query “`st*ion`” matches “stallion” as well as “stuff an onion”. If 1 (true), then “`st*ion`” only matches “stallion”, and linear-dictionary index searches are possible (if enabled), because there are no multi-word matches to (erroneously) miss. **Note:** prior to version 5.01.1208472000 20080417, this setting did not apply to linear searches; linear or post-process searches may have experienced different behavior. The default is 1 in version 6 and later, 0 in version 5 and earlier. Added in version 4.03.1058230349 20030714.

wildsufmatch Whether wildcard expressions in Metamorph queries suffix-match their trailing substrings to the end of words. If 0 (false), the query “`*so`” matches “also” as well as “absolute”. If 1 (true), then “`*so`” only matches “also”. Affects what terms are matched during linear-dictionary index searches. **Note:** prior to version 5.01.1208472000 20080417, this setting did not apply to linear searches; linear or post-process searches may have experienced different behavior. The default is 1 in version 6 and later, 0 in version 5 and earlier. Added in version 4.03.1058230349 20030714.

wildsingle An alias for setting `wildoneword` and `wildsufmatch` together, which is usually desired. Added in version 4.03.1058230349 20030714.

alllineardict Whether to allow linear-dictionary Metamorph index searches. Normally a Metamorph query term is either binary-index searchable (fastest), or else must be linear-table searched (slowest). However, certain terms, while not binary-index searchable, can be linear-dictionary searched in the index, which is slower than binary-index, yet faster than linear-table search. Examples include leading-prefix wildcards such as “`*tion`”. The default is 0 (false), since query protection is enabled by default. Note that `wildsingle` should typically be set true so that wildcard syntax is more likely to be linear-dictionary searchable. Added in version 4.03.1058230349 20030714.

indexminsublen The minimum number of characters that a Metamorph index word expression must match in a query term, in order for the term to utilize the index. A term with fewer than `indexminsublen` indexable characters is assumed to potentially match too many words in the index for an index search to be more worthwhile/faster than a linear-table search.

For binary-index searchable terms, `indexminsublen` is tested against the minimum prefix length; e.g. for query `test.#@` the length tested is 4 (assuming default index word expression of `\alnum{2,99}`). For linear-dictionary index searches, the length tested is the total of all non-wildcard characters; e.g. for query `ab*cd*ef` the length tested is 6.

The default for `indexminsublen` is 2. Added in version 4.03.1058230349 20030714. Note that the query – regardless of index or linear search – must also pass the `qminprelen` setting.

dropwordmode How to remove words from a query set when too many are present (`qmaxsetwords` or `qmaxwords` exceeded) in an index search, e.g. for a wildcard term. The possible values are 0 to retain suffixes and most common words up to the word limit, or 1 to drop the entire term. The default is 0. Added in version 3.00.947633136 20000111.

metamorphstrlstmode How to convert a `strlst` Metamorph query (perhaps generated by Vortex `arrayconvert`) to a regular string Metamorph query. For example, for the `strlst` query composed of the 3 strings `“one”`, `“two”`, and `“bear arms”`, the various modes would convert as follows:

- `allwords`
Space-separate each string, e.g. `“one two bear arms”`.
- `anywords`
Space-separate each string and append `“@0' ”`, e.g. `‘one two bear arms @0’`.
- `allphrases`
Space-separate and double-quote each string, e.g. `“"one" "two" "bear arms"”`.
- `anywords`
Space-separate and double-quote each string, and append `“@0' ”`, e.g. `“"one" "two" "bear arms" @0’`.
- `equivlist`
Make the string list into a parenthetical comma-separated list, e.g. `“(one,two,bear arms)”`.

The default is `equivlist`. Added in version 5.01.1225240000 20081028. See also the `varchartostrlstsep` setting (p. 184), which affects conversion of `varchar` to `strlst` in other contexts.

compatibilityversion Sets the Taxis compatibility version – the version to attempt to behave as – to the given string, which is a Taxis version of the form `“major[.minor[.release]]”`, where *major* is a major version integer, *minor* is a minor version integer, and *release* is a release integer. Added in version 7. See the `<vxcp compatibilityversion>` setting in Vortex for details. See also the `[Taxis] Compatibility Version setting (p. ??)` in `taxis.ini`, which the `compatibilityversion` setting defaults to.

failifincompatible Whenever set nonzero/true, and the most recent `compatibilityversion` setting attempt failed, then all future SQL statements will fail with an error message. Since there is no conditional (“if”) statement in SQL, this allows a SQL script to essentially abort if it tries to set a Taxis compatibility version that is unsupported, rather than continue with possibly undesired side effects. Added in version 7. See also `<vxcp compatibilityversion>` in Vortex, which obviates the need for this setting, as it has a checkable error return.

groupbymem When set nonzero/true (the default), try to minimize memory usage during GROUP BY/DISTINCT operations (e.g. when using an index and sorting is not needed). Added in version 7.00.1370039228 20130531.

legacyversion7orderbyrank If on, an ORDER BY \$rank (or \$rank-containing expression) uses legacy version 7 behavior, i.e. typically orders in numerically descending order, but may change to ascending (and have other idiosyncrasies) depending on index, expression and DESC flag use. If disabled, such ORDER BYs are consistent with others: numerically ascending unless DESC flag given (which would typically be given, to maintain descending-numerical-rank order).

The default is the value of the [Taxis] Legacy Version 7 Order By Rank setting (p. ??) in `conf/taxis.ini`, which is off by default with `compatibilityversion 8` and later, on in earlier versions (`compatibilityversion` defaults to Taxis Version). Added in version 7.06.1508871000 20171024.

Note that this setting may be removed in a future release, as its enabled behavior is deprecated. Its existence is only to ease transition of old code when upgrading to Taxis version 8, and thus should only be used temporarily. Old code should be updated to reflect version 8 default behavior – and this setting removed – soon after upgrading.

11.4.2 Metamorph parameters

These settings affect the way that text searches are performed. They are equivalent to changing the corresponding parameter in the profile, or by calling the Metamorph API function to set them (if there is an equivalent). They are:

minwordlen The smallest a word can get due to suffix and prefix removal. Removal of trailing vowel or double consonant can make it a letter shorter than this. Default 255.

keepnoise Whether noise words should be stripped from the query and index. Default off.

suffixproc Whether suffixes should be stripped from the words to find a match. Default on.

prefixproc Whether prefixes should be stripped from the words to find a match. Turning this on is not suggested when using a Metamorph index. Default off.

rebuild Make sure that the word found can be built from the root and appropriate suffixes and prefixes. This increases the accuracy of the search. Default on.

useequiv Perform thesaurus lookup. If this is on then the word and all equivalences will be searched for. If it is off then only the query word is searched for. Default off. Aka **keepeqvs** in version 5.01.1171414736 20070213 and later.

inc_sdexp Include the start delimiter as part of the hit. This is not generally useful in Taxis unless hit offset information is being retrieved. Default off.

inc_edexp Include the end delimiter as part of the hit. This is not generally useful in Taxis unless hit offset information is being retrieved. Default on.

sdexp Start delimiter to use: a regular expression to match the start of a hit. The default is no delimiter.

edexp End delimiter to use: a regular expression to match the start of a hit. The default is no delimiter.

intersects Default number of intersections in Metamorph queries; overridden by the @ operator. Added in version 7.06.1530212000 20180628.

hyphenphrase Controls whether a hyphen between words searches for the phrase of the two words next to each other, or searches for the hyphen literally. The default value of 1 will search for the two words as a phrase. Setting it to 0 will search for a single term including the hyphen. If you anticipate setting hyphenphrase to 0 then you should modify the index word expression to include hyphens.

wordc For language or wildcard query terms during linear (non-index) searches, this defines which characters in the document constitute a word. When a match is found for language/wildcard terms, the hit is expanded to include all surrounding word characters, as defined by this setting. The resulting expansion must then match the query term for the hit to be valid. (This prevents the query “pond” from inadvertently matching the text “correspondence”, for example.) The value is specified as a REX character set. The default setting is `[\alpha\']` which corresponds to all letters and apostrophe. For example, to exclude apostrophe and include digits use: `set wordc='[\alnum]'` Added in version 3.00.942260000. Note that this setting is for linear searches: what constitutes a word for Metamorph *index* searches is controlled by the index expressions (**addexp** property, p. 175). Also note that non-language, non-wildcard query terms (e.g. 123 with default settings) are not word-expanded.

langc Defines which characters make a query term a language term. A language term will have prefix/suffix processing applied (if enabled), as well as force the use of **wordc** to qualify the hit (during linear searches). Normally **langc** should be set the same as **wordc** with the addition of the phrase characters space and hyphen. The default is `[\alpha\ ' \-]` Added in version 3.00.942260000.

withinmode A space- or comma-separated unit and optional type for the “within-*N*” operator (e.g. `w/5`). The unit is one of:

- `char` for within-*N* characters
- `word` for within-*N* words

The optional type determines what distance the operator measures. It is one of the following:

- `radius` (the default if no type is specified when set) indicates all sets must be within a radius *N* of an “anchor” set, i.e. there is a set in the match such that all other sets are within *N* units right of its right edge or *N* units left of its left edge.
- `span` indicates all sets must be within an *N*-unit span

Added in version 4.04.1077930936 20040227. The optional type was added in version 5.01.1258712000 20091120; previously the only type was implicitly `radius`. In version 5 and earlier the default setting was `char` (i.e. `char radius`); in version 6 and later the default is `word span`.

phrasewordproc Which words of a phrase to do suffix/wildcard processing on. The possible values are `mono` to treat the phrase as a monolithic word (i.e. only last word processed, but entire phrase counts towards **minwordlen**); `none` for no suffix/wildcard processing on phrases; or `last` to process just the last word. Note that a phrase is multi-word, i.e. a single word in double-quotes is not considered a

phrase, and thus **phrasewordproc** does not apply. Added in version 4.03.1082000000 20040414.
Mode `none` supported in version 5.01.1127760000 20050926.

mdparmodifyterms If nonzero, allows the Metamorph query parser to modify search terms by compression of whitespace and quoting/unquoting. This is for back-compatibility with earlier versions; enabling it will break the information from bit 4 of `mminfo()` (query offset/lengths of sets). Added in version 5.01.1220640000 20080905.

11.4.3 Rank knobs

The following properties affect the document ranks from `likep` and `like` queries, and hence the order of returned documents for `likep`. Each property controls a factor used in the rank. The property's value is the relative importance of that factor in computing the rank. The properties are settable from 0 (factor has no effect at all) to 1000 (factor has maximum relative importance).

It is important to note that these property weights are relative to the sum of all weights. For example, if `likepleadbias` is set to 1000 and the remaining properties to 0, then a hit's rank will be based solely on lead bias. If `likeproximity` is then set to 1000 as well, then lead bias and proximity each determine 50% of the rank.

likeproximity Controls how important proximity of terms is. The closer the hit's terms are grouped together, the better the rank. The default weight is 500.

likepleadbias Controls how important closeness to document start is. Hits closer to the top of the document are considered better. The default weight is 500.

likeporder Controls how important word order is: hits with terms in the same order as the query are considered better. For example, if searching for "bear arms", then the hit "arm bears", while matching both terms, is probably not as good as an in-order match. The default weight is 500.

likepdofreq Controls how important frequency in document is. The more occurrences of a term in a document, the better its rank, up to a point. The default weight is 500.

likeptblfreq Controls how important frequency in the table is. The more a term occurs in the table being searched, the *worse* its rank. Terms that occur in many documents are usually less relevant than rare terms. For example, in a web-walk database the word "HTML" is likely to occur in most documents: it thus has little use in finding a specific document. The default weight is 500.

11.4.4 Other ranking properties

These properties affect how `LIKEP` and some `LIKE` queries are processed.

likeprows Only the top `likeprows` relevant documents are returned by a `LIKEP` query (default 100). This is an arbitrary cut-off beyond which most results would be increasingly useless. It also speeds up the query process, because fewer rows need to be sorted during ranking. By altering `likeprows` this threshold can be changed, e.g. to return more results to the user (at the potential cost of more search time). Setting this to 0 will return all relevant documents (no limit).

Note that in some circumstances, a `LIKEP` query might return more than `likeprows` results, if for example later processing requires examination of all `LIKEP`-matching rows (e.g. certain `AND` queries). Thus a SQL statement containing `LIKEP` may or may not be limited to `likeprows` results, depending on other clauses, indexes, etc.

likepmode Sets the mode for `LIKEP` queries. This can be either 0, for early, or 1 for late. The default is 1, which is the correct setting for almost all cases. Does not apply to most Metamorph index searches.

likepallmatch Setting this to 1 forces `LIKEP` to only consider those documents containing *all* (non-negated) query terms as matches (i.e. just as `LIKE` does). By default, since `LIKEP` is a ranking operator it returns the best results even if only some of the set-logic terms (non-+ or - prefix) can be found. (Note that required terms - prefixed with a + - are always required in a hit regardless of this setting. Also note that if `likepobeyintersects` is true, an @ operator value in the query will override this setting.)

likepobeyintersects Setting this to 1 forces `LIKEP` to obey the intersects operator (@) in queries (even when `likepallmatch` is true). By default `LIKEP` does not use it, because it is a ranking operator. Setting both `likepallmatch` and `likepobeyintersects` to 1 will make `LIKEP` respect queries the same as `LIKE`. (Note: `apicp alintersects` may have to be enabled in Vortex as well.)

likepinfthresh This controls the “infinity” threshold in `LIKE` and `LIKEP` queries: if the estimated number of matching rows for a set is greater than this, the set is considered infinitely-occurring. If all the search terms found in a given document are such infinite sets, the document is given an estimated rank. This saves time ranking irrelevant but often-occurring matches, at the possible expense of rank position. The default is 0, which means infinite (no infinite sets; rank all documents).

likepindexthresh Controls the maximum number of matching documents to examine (default infinite) for `LIKEP` and `LIKE`. After this many matches have been found, stop and return the results obtained so far, even if more hits exist. Typically this would be set to a high threshold (e.g. 100000): a query that returns more than that many hits is probably not specific enough to produce useful results, so save time and don’t process the remaining hits. (It’s also a good bet that something useful was already found in the initial results.) This helps keep such noisy queries from loading a server, by stopping processing on them early. A more specific query that returns fewer hits will fall under this threshold, so all matches will be considered for ranking.

Note that setting `likepindexthresh` is a tradeoff between speed and accuracy: the lower the setting, the faster queries can be processed, but the more queries may be dropping potentially high-ranking hits.

11.4.5 Indexing properties

indexspace A directory in which to store the index files. The default is the empty string, which means use the database directory. This can be used to put the indexes onto another disk to balance load or for space reasons. If `indexspace` is set to a non-default value when a Metamorph index is being updated, the new index will be stored in the new location.

indexblock When a Metamorph index is created on an indirect field, the indirect files are read in blocks. This property allows the size of the block used to be redefined.

indexmem When indexes are created Taxis will use memory to speed up the process. This setting allows the amount of memory used to be adjusted. The default is to use 40% of physical memory, if it can be determined, and to use 16MB if not. If the value set is less than 100 then it is treated as a percentage of physical memory. If the number is greater than 100 then it is treated as the number of bytes of memory to use. Setting this value too high can cause excessive swapping, while setting it too low causes unneeded extra merges to disk.

indexmeter Whether to print a progress meter during index creation/update. The default is 0 or 'none', which suppresses the meter. A value of 1 or 'simple' prints a simple hash-mark meter (with no tty control codes; suitable for redirection to a file and reading by other processes). A value of 2 or 'percent' or 'pct' prints a hash-mark meter with a more detailed percentage value (suitable for large indexes). Added in version 4.00.998688241 Aug 24 2001.

meter A semicolon-separated list of processes to print a progress meter for. Syntax:

```
{process [=type]} | type [; ...]
```

A *process* is one of `index`, `compact`, or the catch-all alias `all`. A *type* is a progress meter type, one of `none`, `simple`, `percent`, `on` (same as `simple`) or `off` (same as `none`). The default *type* if not given is `on`. E.g. to show a progress meter for all meterable processes, simply set `meter` to `on`. Added in version 6.00.1290500000 20101123.

addexp An additional REX expression to match words to be indexed in a Metamorph index. This is useful if there are non-English words to be searched for, such as part numbers. When an index is first created, the expressions used are stored with it so they will be updated properly. The default expression is `\alnum{2,99}`. **Note:** Only the expressions set when the index is initially created (i.e. the first `CREATE METAMORPH ...` statement – later statements are index updates) are saved. Expressions set during an update (issuance of “`create metamorph [inverted] index`” on an existent index) will *not* be added.

delexp This removes an index word expression from the list. Expressions can be removed either by number (starting with 0) or by expression.

lstexp Lists the current index word expressions. The value specified is ignored (but required syntactically).

addindextmp Add a directory to the list of directories to use for temporary files while creating the index. If temporary files are needed while creating a Metamorph index they will be created in one of these directories, the one with the most space at the time of creation. If no `addindextmp` dirs are specified, the default list is the index's destination dir (e.g. `database` or `indexspace`), and the environment variables `TMP` and `TMPDIR`.

delindextmp Remove a directory from the list of directories to use for temporary files while creating a Metamorph index.

lstindextmp List the directories used for temporary files while creating Metamorph indices. Aka `listindextmp`.

indexvalues Controls how a regular (B-tree) index stores table values. If set to `splitstrlst` (the default), then `strlst`-type fields are split, i.e. a separate (item,recid) tuple is stored for *each* (varchar) item in the `strlst`, rather than just one for the whole (strlst,recid) tuple. This allows

the index to be used for some set-like operators that look at individual items in a `strlst`, such as most `IN`, `SUBSET` (p. 37) and `INTERSECT` (p. 38) queries.

If `indexvalues` is set to `all` – or the index is not on a `strlst` field, or is on multiple fields – such splitting does not occur, and the index can generally not be used for set-like queries (with some exceptions; see p. 38 for details).

Note that if index values are split (i.e. `splitstrlst` set and index is one field which is `strlst`), table rows with an empty (zero-items) `strlst` value will not be stored in the index. This means that queries that require searching for or listing empty-`strlst` table values cannot use such an index. For example, a subset query with a non-empty parameter on the right side and a `strlst` table column on the left side will not be able to return empty-`strlst` rows when using an index, even though they match. Also, subset queries with an empty-`strlst` or empty-`varchar` parameter (left or right side) must use an `indexvalues=all` index instead. Thus if empty-`strlst` subset query parameters are a possibility, both types of index (`splitstrlst` and `all`) should be created.

As with `stringcomparemode`, only the creation-time `indexvalues` value is ever used by an index, not the current value, and the optimizer will attempt to choose the best index at search time. The `indexvalues` setting was added in Taxis version 7; previous versions effectively had `indexvalues` set to `splitstrlst`. **Caveat:** A version 6 Taxis will issue an error when encountering an `indexvalues=all` index (as it is unimplemented in version 6), and will refuse to modify the index or the table it is on. **A version 5 or earlier Taxis, however, may silently corrupt an `indexvalues=all` index during table modifications.**

btreethreshold This sets a limit as to how much of an index should be used. If a particular portion of the query matches more than the given percent of the rows the index will not be used. It is often more efficient to try and find another index rather than use an index for a very frequent term. The default is set to 50, so if more than half the records match, the index will not be used. This only applies to ordinary indices.

btreelog Whether to log operations on a particular B-tree, for debugging. Generally enabled only at the request of tech support. The value syntax is:

```
[on=|off=][dir/]file[.btr]
```

Prefixing `on=` or `off=` turns logging on or off, respectively; the default (if no prefix) is `on`. Logging applies to the named B-tree file; if a relative path is given, logging applies to the named B-tree in any database accessed.

The logging status is also saved in the B-tree file itself, if the index is opened for writing (e.g. at create or update). This means that once logging is enabled and saved, *every* process that accesses the B-tree will log operations, not just ones that have `btreelog` explicitly set. This is critical for debugging, as every operation must be logged. Thus, `btreelog` can just be set once (e.g. at index create), without having to modify (and track down) every script that might use the B-tree. Logging can be disabled later, by setting “`off=file`” and accessing the index for an update.

Operations are logged to a text file with the same name as the B-tree, but ending in “`.log`” instead of “`.btr`”. The columns in the log file are as follows; most are for tech support analysis, and note that they may change in a future Taxis release:

- **Date** Date

- **Time** Time (including microseconds)
- **Script and line** Vortex script and line number, if known
- **PID** Process ID
- **DBTBL handle** DBTBL handle
- **Read locks** Number of read locks (DBTBL.nireadl)
- **Write locks** Number of write locks (DBTBL.niwrite)
- **B-tree handle** BTREE handle
- **Action** What action was taken:
 - open B-tree open: **Recid** is root page offset
 - create B-tree create
 - close B-tree close
 - RDroot Read root page
 - dump B-tree dump
 - WRhdr Write B-tree header: **Recid** is root page offset
 - WRdd Write data dictionary: **Recid** is DD offset. (Read DD at open is not logged.)
 - delete Delete key: **Recid** is for the key
 - append Append key
 - insert Insert key
 - search Search for key
 - RDpage Read page: **Recid** is for the page
 - WRpage Write page
 - CRpage Create page
 - FRpage Free page
 - FRdbf Free DBF block
- **Result** Result of action:
 - ok Success
 - fail Failure
 - dup Duplicate (e.g. duplicate insert into unique B-tree)
 - hit Search found the key
 - miss Search did not find the key
- **Search mode** Search mode:
 - B Find before
 - F Find
 - A Find after
- **Index guarantee** DBTBL.indguar flag (1 if no post-process needed)
- **Index type** Index type:
 - N DBIDX_NATIVE (bubble-up)
 - M DBIDX_MEMORY (RAM B-tree)
 - C DBIDX_CACHE (RAM cache)

- **Recid** Record id; see notes for **Action** column
- **Key size** Key size (in bytes)
- **Key flags** Flags for each key value, separated by commas:
 - D OF_DESCENDING
 - I OF_IGNORE_CASE
 - X OF_DONT_CARE
 - E OF_PREFER_END
 - S OF_PREFER_START
- **Key** Key, i.e. value being inserted, deleted etc.; multiple values separated with commas

Unavailable or not-applicable fields are logged with a dash. Note that enabling logging can produce a large log file quickly; free disk space should be monitored. The `btreetlog` setting was added in version 5.01.1134028000 20051208.

btreedump Dump B-tree indexes, for debugging. Generally enabled only at the request of tech support. The value is an integer whose bits are defined as follows:

Bits 0-15 define what to dump. Files are created that are named after the B-tree, with a different extension:

- Bit 0: Issue a `putmsg` about where dump file(s) are
- Bit 1: `.btree` file: Copy of in-mem BTREE struct
- Bit 2: `.btrcopy` file: Copy of `.btr` file
- Bit 3: `.cache` file: Page cache from BCACHE, BPAGE
- Bit 4: `.his` file: History from BTRL
- Bit 5: `.core` file: `fork()` and dump core

Bits 16+ define when to dump:

- Bit 16: At “Cannot insert value” messages
- Bit 17: At “Cannot delete value” messages
- Bit 18: At “Trying to insert duplicate value” messages

The files are for tech support analysis. Formats and bits subject to change in future Taxis releases. The `btreedump` setting was added in version 5.01.1131587000 20051109.

maxlinearrows This set the maximum number of records that should be searched linearly. If using the indices to date yield a result set larger than `maxlinearrows` then the program will try to find more indices to use. Once the result set is smaller than `maxlinearrows`, or all possible indices are exhausted, the records will be processed. The default is 1000.

likerrows How many rows a single term can appear in, and still be returned by `liker`. When searching for multiple terms with `liker` and `likep` one does not always want documents only containing a very frequent term to be displayed. This sets the limit of what is considered frequent. The default is 1000.

indexaccess If this option is turned on then data from an index can be selected as if it were a table. When selecting from an ordinary (B-tree) index, the fields that the index was created on will be listed. When selecting from a Metamorph index a list of words (`Word` column), count of rows containing each word (`RowCount`), and – for Metamorph inverted indexes – count of all hits in all rows (`OccurrenceCount`) for each word will be returned.

indexchunk In versions of Taxis after October 1998, the `indexchunk` setting is deprecated and unused. In prior releases, when creating a Metamorph index temporary files are used which in the worst case can grow to twice the size of the data being indexed. This process can be broken into stages, such that after indexing a certain amount of data the temporary files are processed, to generate a partial index, and then the process repeats for the rest of the data. By default the amount of free disk space is checked on startup, and used to calculate when it will need to perform the processing step. If the system does not report free disk space accurately, or to free more disk space, this value can be changed. The default is 0, which automatically calculates a value. Otherwise it is set to the number of bytes of data to index before processing the temporary files. Lower values conserve disk space, at the expense of more time to process intermediate files.

cleanupwait *Windows/NT specific* After updating a Metamorph index the database will wait this long before trying to remove the old copy of the index. This is to allow any other process currently using the index time to stop using the index, so it can be removed. The default is twenty seconds. If a whole batch of Metamorph indices are being updated right after another, it may be useful to set this to 0 for all but the last index, as an attempt will be made to remove all old indices after every index update.

dbcleanupverbose Integer whose bit flags control some tracing messages about database cleanup housekeeping (e.g. removal of unneeded temporary or deleted indexes and tables). A bit-wise OR of the following values:

- 0x01: Report successful removal of temporary/deleted indexes/tables.
- 0x02: Report failed removal of such indexes/tables.
- 0x04: Report on in-use checks of temporary indexes/tables.

The default is 0 (i.e. no messages). Note that these cleanup actions may also be handled by the Database Monitor; see also the `[Monitor] DB Cleanup Verbose` setting in `conf/taxis.ini`. Added in version 6.00.1339712000 20120614.

indextrace For debugging: trace index usage, especially during searches, issuing informational `putmsgs`. Greater values produce more messages. Note that the meaning of values, as well as the messages printed, are subject to change without notice. Aka `traceindex`, `traceidx`. Added in version 3.00.942186316 19991109.

tracerecid For debugging: trace index usage for this particular `recid`. Added in version 3.01.945660772 19991219.

indexdump For debugging: dump index `recids` during search/usage. Value is a bitwise OR of the following flags:

- Bit 0** for new list
- Bit 1** for delete list

Bit 2 for token file

Bit 3 for overall counts too

The default is 0.

indexmmap Whether to use memory-mapping to access Metamorph index files, instead of `read()`. The value is a bitwise OR of the following flags:

Bit 0 for token file

Bit 1 for `.dat` file

The default is 1 (i.e. for token file only). Note that memory-mapping may not be supported on all platforms.

indexreadbufsz Read buffer size, when reading (not memory-mapping) Metamorph index `.tok` and `.dat` files. The default is 64KB; suffixes like “KB” are respected. During search, actual read block size could be less (if predicted) or more (if blocks merged). Also used during index create/update. Decreasing this size when creating large indexes can save memory (due to the large number of intermediate files), at the potential expense of time. Aka `indexreadbufsize`. Added in version 4.00.1006398833 20011121.

indexwritebufsz Write buffer size for creating Metamorph indexes. The default is 128KB; suffixes like “KB” are respected. Aka `indexwritebufsize`. Added in version 4.00.1007509154 20011204.

indexmmapbufsz Memory-map buffer size for Metamorph indexes. During search, it is used for the `.dat` file, if it is memory-mapped (see `indexmmap`); it is ignored for the `.tok` file since the latter is heavily used and thus fully mapped (if `indexmmap` permits it). During index update, `indexmmapbufsz` is used for the `.dat` file, if it is memory-mapped; the `.tok` file will be entirely memory-mapped if it is smaller than this size, else it is read. Aka `indexmmapbufsize`. The default is 0, which uses 25% of RAM. Added in version 3.01.959984092 20000602. In version 4.00.1007509154 20011204 and later, “KB” etc. suffixes are allowed.

indexslurp Whether to enable index “slurp” optimization during Metamorph index create/update, where possible. Optimization is always possible for index create; during index update, it is possible if the new insert/update recids all occur after the original recids (e.g. the table is insert-only, or all updates created a new block). Optimization saves about 20% of index create/update time by merging piles an entire word at a time, instead of word/token at a time. The default is 1 (enabled); set to 0 to disable. Added in version 4.00.1004391616 20011029.

indexappend Whether to enable index “append” optimization during Metamorph index update, where possible. Optimization is possible if the new insert recids all occur after the original recids, and there were no deletes/updates (e.g. the table is insert-only); it is irrelevant during index create. Optimization saves index build time by avoiding original token translation if not needed. The default is 1 (enabled); set to 0 to disable. Added in version 4.00.1006312820 20011120.

indexwritesplit Whether to enable index “write-split” optimization during Metamorph index create/update. Optimization saves memory by splitting the writes for (potentially large) `.dat` blocks into multiple calls, thus needing less buffer space. The default is 1 (enabled); set to 0 to disable. Added in version 4.00.1015532186 20020307.

indexbtreeexclusive Whether to optimize access to certain index B-trees during exclusive access. The optimization may reduce seeks and reads, which may lead to increased index creation speed on platforms with slow large-file `lseek` behavior. The default is 1 (enabled); set to 0 to disable. Added in version 5.01.1177548533 20070425.

mergeflush Whether to enable index “merge-flush” optimization during Metamorph index create/update. Optimization saves time by flushing in-memory index piles to disk just before final merge; generally saves time where `indexslurp` is not possible. The default is 1 (enabled); set to 0 to disable. Added in version 4.00.1011143988 20020115.

indexversion

`indexversion` Which version of Metamorph index to produce or update, when creating or updating Metamorph indexes. The supported values are 0 through 3; the default is 2. Setting version 0 sets the default index version for that Taxis release. Note that old versions of Taxis may not support version 3 indexes. Version 3 indexes may use less disk space than version 2, but are considered experimental. Added in version 3.00.954374722 20000329.

indexmaxsingle For Metamorph indexes; the maximum number of locations that a single-recid dictionary word may have and still be stored solely in the `.btr` B-tree file (without needing a `.dat` entry). Single-recid-occurrence words usually have their data stored solely in the B-tree to save a `.dat` access at search time. However, if the word occurs many times in that single recid, the data (for a Metamorph inverted index) may be large enough to bloat the B-tree and thus negate the savings, so if the single-recid word occurs more than `indexmaxsingle` times, it is stored in the `.dat`. The default is 8.

uniquewlist Whether/how to unique the new list during Metamorph index searches. Works around a potential bug in old versions of Taxis; not generally set. The possible values are:

- 0** : do not unique at all
- 1** : unique auxillary/compound index new list only
- 2** : unique all new lists
- 3** : unique all new lists and report first few duplicates

The default is 0.

tablereadbufsz Size of read buffer for tables, used when it is possible to buffer table reads (e.g. during some index creations). The default is 16KB. When setting, suffixes such as “KB” etc. are supported. Set to 0 to disable read buffering. Added in version 5.01.1177700467 20070427. Aka `tablereadbufsize`.

11.4.6 Locking properties

These properties affect the way that locking occurs in the database engine. Setting these properties without understanding the consequences can lead to inaccurate results, and even corrupt tables.

singleuser This will turn off locking completely. *This should be used with extreme caution.* The times when it is safe to use this option are if the database is read-only, or if there is only one connection to the database. Default off. This replaces the prior setting of `nolocking`.

lockmode This can be set to either manual or automatic. In manual mode the person writing the program is responsible for getting and releasing locks. In automatic mode Taxis will do this itself. Manual mode can reduce the number of locks required, or implement specific application logic. In manual mode care must be taken that reads and writes can not occur at the same time. The two modes can co-exist, in that one process can have manual mode, and the other automatic. Default automatic.

locksleppmethod Determines whether to use a portable or OS specific method of sleeping while waiting for a lock. By default the OS specific method is used. This should not need to be changed.

locksleeptime How long to wait between attempts to check the lock. If this value is too small locks will be checked too often, wasting CPU time. If it is too high then the process might be sleeping when there is no lock, delaying database access. Generally the busier the system the higher this setting should be. It is measured in thousandths of a second. The default is 20.

locksleppmaxtime The lock sleep time automatically increments the if unable to get a lock to allow other processes an opportunity to get the CPU. This sets a limit on how long to sleep. It is measured in thousandths of a second. The default is 100. Added in version 4.00.1016570000.

fairlock Whether to be fair or not. A process which is running in fair mode will not obtain a lock if the lock which has been waiting longest would conflict. A process which is not in fair mode will obtain the lock as soon as it can. This can cause a process to wait forever for a lock. This typically happens if there are lots of processes reading the table, and one trying to write. Setting `fairlock` to true will guarantee that the writer can obtain the lock as long as the readers are getting and releasing locks. Without `fairlock` there is no such guarantee, however the readers will see better performance as they will rarely if ever wait for the writer. This flag only affects the process which sets the flag. It is not possible to force another process to be fair. The default is that it operates in fair mode.

lockverbose How verbose the lock code should be. The default minimum level of 0 will report all serious problems in the lock manager, as they are detected and corrected. A verbosity level of 1 will also display messages about less serious problems, such as processes that have exited without closing the lock structure. Level 2 will also show when a lock can not be immediately obtained. Level 3 will show every lock as it is released. In version 5.01.1160010000 20061004 and later, the level can be bitwise OR'd with 0x10 and/or 0x20 to report system calls before and after (respectively). Levels 1 and above should generally only be used for debugging. In version 7.07.1565800000 20190814 and later, 0x40 and 0x80 may be set to report before and after semaphore locking/unlocking.

debugbreak Stop in debugger when set. Internal/debug use available in some versions. Added in version 4.02.1045505248 Feb 17 2003.

debugmalloc Integer; controls debug malloc library. Internal/debug use in some versions. Added in version 4.03.1050682062 Apr 18 2003.

11.4.7 Miscellaneous Properties

These properties do not fit nicely into a group, and are presented here.

tablespace Similar to `indexspace` above. Sets a directory into which tables created will be placed. This property does not stay set across invocations. Default is empty string, which means the database directory.

datefmt This is a `strftime` format used to format dates for conversion to character format. This will affect `tsql`, as well as attempts to retrieve dates in ASCII format. Although the features supported by different operating systems will vary, some of the more common format codes are:

- `%%` Output `%`
- `%a` abbreviated weekday name
- `%A` full weekday name
- `%b` abbreviated month name
- `%B` full month name
- `%c` local date and time representation
- `%d` day of month (01 - 31)
- `%D` date as `%m/%d/%y`
- `%e` day of month (1 - 31)
- `%H` Hour (00 - 23)
- `%I` Hour (01 - 12)
- `%j` day of year (001 - 366)
- `%m` month (01 - 12)
- `%M` Minute (00 - 59)
- `%P` AM/PM
- `%S` Seconds (00 - 59)
- `%U` Week number (beginning Sunday) (00-53)
- `%w` Week day (0-6) (0 is Sunday)
- `%W` Week number (beginning Monday) (00-53)
- `%x` local date representation
- `%X` local time representation
- `%y` two digit year (00 - 99)
- `%Y` Year with century
- `%Z` Time zone name

Default `%Y-%m-%d %H:%M:%S`, which can be restored by setting `datefmt` to an empty string. Note that in version 6.00.1300386000 20110317 and later, the `stringformat()` SQL function can be used to format dates (and other values) without needing to set a global property.

timezone Change the default timezone that Taxis will use. This should be formatted as for the TZ environment variable. For example for US Eastern time you should set `timezone` to `EST5EDT`. Some systems may allow alternate representations, such as `US/Eastern`, and if your operating system accepts them, so will Taxis.

locale Can be used to change the locale that Taxis uses. This will impact the display of dates if using names, as well as the meaning of the character classes in REX expressions, so `\alpha` will be correct. Also with the correct locale set (and OS support), Metamorph will work case insensitively correctly (with mono-byte character sets and Taxis version 5 or earlier; see `textsearchmode` for UTF-8/Unicode and version 6 or later support).

indirectcompat Setting this to 1 sets compatibility with early versions of Taxis as far as display of indirects go. If set to 1 a trailing @ is added to the end of the filename. Default 0.

indirectspace Controls where indirects are created. The default location is a directory called indirects in the database directory. Taxis will automatically create a directory structure under that directory to allow for efficient indirect access. At the top level there will be 16 directories, 0 through 9 and a through f. When you create the directory for indirects you can precreate these directories, or use them as mount points. You should make sure that the Taxis user has permissions to the directories. Added in version 03.00.940520000

triggermode This setting changes the way that the command is treated when creating a trigger. The default behavior is that the command will be executed with an extra arg, which is the filename of the table containing the records. If `triggermode` is set to 1 then the strings `$db` and `$table` are replaced by the database and table in that database containing the records. This allows any program which can access the database to retrieve the values in the table without custom coding.

paramchk Enables or disables the checking of parameters in the SQL statement. By default it is enabled, which will cause any unset parameters to cause an error. If `paramchk` is set to 0 then unset parameters will not cause an error, and will be ignored. This lets a single complex query be given, yet parameter values need only be supplied for those clauses that should take effect on the query.

message,nomessage Enable or disable messages from the SQL engine. The argument should be a comma separated list of messages that you want to enable or disable. The known messages are:

duplicate Message Trying to insert duplicate value () in index when an attempt is made to insert a record which has a duplicate value and a unique index exists. The default is enabled.

vchartostrlstsep The separator character or mode to use when converting a `varchar` string into a `strlst` list of strings in Taxis. The default is set by the `conf/taxis.ini` setting `[Taxis] Vchar To Strlst Sep (p. ??)`; if that is not set, the “factory” built-in default is `create` in version 7 (or `compatibilityversion 7`) and later, or `lastchar` in version 6 (or `compatibilityversion 6`) and earlier.

A value of `create` indicates that the separator is to be created: the entire string is taken intact as the sole item for the resulting `strlst`,² and a separator is created that is not present in the string (to aid re-conversion to `varchar`). This can be used in conjunction with Vortex’s `<sqlcp arrayconvert>` setting to ensure that single-value as well as multi-value Vortex variables are converted consistently when inserted into a `strlst` column: single-value vars by `vchartostrlstsep`, multi-value by `arrayconvert`.

The value `lastchar` indicates that the last character in the source string should be the separator; e.g. “a,b,c,” would be split on the comma and result in a `strlst` of 3 values: “a”, “b” and “c”.

`vchartostrlstsep` may also be a single byte character, in which case that character is used as the separator. This is useful for converting CSV-type strings e.g. “a,b,c” without having to modify the string and append the separator character first (i.e. for `lastchar` mode).

²In version 7 (or `compatibilityversion 7`) and later, note that in `create` mode, an empty source string will result in an empty (zero-items) `strlst`: this helps maintain consistency of empty-string meaning empty-set for `strlst`, as is true in other contexts. In version 6 and earlier an empty source string produced a one-empty-string-item `strlst` in `create` mode.

`varcharostrlstsep` may also be set to default to restore the default (`conf/texis.ini`) setting. It may also be set to `builtindefault` to restore the “factory” built-in default (which changes under `compatibilityversion`, see above); these values were added in version 5.01.1231553000 20090109. If no `conf/texis.ini` value is set, default is the same as `builtindefault`.

`varcharostrlstsep` was added in version 5.01.1226978000 20081117. See also the `metamorphstrlstmode` setting (p. 170), which affects conversion of `strlst` values into Metamorph queries; and the `convert SQL` function (p. 85), which in Taxis version 7 and later can take a `varcharostrlstsep` mode argument. The `compatibilityversion` property (p. 170), when set, affects `varcharostrlstsep` as well.

multivaluetomultirow Whether to split multi-value fields (e.g. `strlst`) into multiple rows (e.g. of `varchar`) when appropriate, i.e. during `GROUP BY` or `DISTINCT` on such a field. If `nonzero/true`, a `GROUP BY` or `DISTINCT` on a `strlst` field will split the field into its `varchar` members for processing. For example, consider the following table:

```
create table test(Colors strlst);
insert into test(Colors)
  values(convert('red,green,blue,', 'strlst', 'lastchar'));
insert into test(Colors)
  values(convert('blue,orange,green,', 'strlst', 'lastchar'));
```

With `multivaluetomultirow` set true, the statement:

```
select count(Colors) Count, Colors from test group by Colors;
```

generates the following output:

| Count | Colors |
|-------|--------|
| 2 | blue |
| 2 | green |
| 1 | orange |
| 1 | red |

Note that the `strlst` values have been split, allowing the two blue and green values to be counted individually. This also results in the returned `Colors` type being `varchar` instead of its declared `strlst`, and the sum of `Count` values being greater than the number of rows in the table. Note also that merely `SELECTING` a `strlst` will not cause it to be split: it must be specified in the `GROUP BY` or `DISTINCT` clause.

The `multivaluetomultirow` was added in version 5.01.1243980000 20090602. It currently only applies to `strlst` values and only to single-column `GROUP BY` or `DISTINCT` clauses. A system-wide default for this SQL setting can be set in `conf/texis.ini` with the `[Taxis]`

Multi Value To Multi Row setting. If unset, it defaults to true through version 6 (or `compatibilityversion` 6), and false in version 7 and later (because in general `GROUP BY/DISTINCT` are expected to return true table rows for results). The `compatibilityversion` property (p. 170), when set, affects this property as well.

inmode How the `IN` operator should behave. If set to `subset`, `IN` behaves like the `SUBSET` operator (p. 37). If set to `intersect`, `IN` behaves like the `INTERSECT` operator (p. 38). Added in version 7, where the default is `subset`. Note that in version 6 (or `compatibilityversion` 6) and earlier, `IN` always behaved in an `INTERSECT`-like manner. The `compatibilityversion` property (p. 170), when set, affects this property as well.

hexifybytes Whether conversion of `byte` to `char` (or vice-versa) should encode to (or decode from) hexadecimal. In Taxis version 6 (or `compatibilityversion` 6) and earlier, this always occurred. In Taxis version 7 (or `compatibilityversion` 7) and later, it is controllable with the `hexifybytes` SQL property: 0 for off/as-is, 1 for hexadecimal conversion. This property is on by default in `tsql` (i.e. hex conversion ala version 6 and earlier), so that `SELECT`ing from certain system tables that contain `byte` columns will still be readable from the command line. However, the property is off by default in version 7 and later non-`tsql` programs (such as Vortex), to avoid the hassle of hex conversion when raw binary data is needed (e.g. images), and because Vortex etc. have more tools for dealing with binary data, obviating the need for hex conversion. (The `hextobin()` and `bintohehex()` SQL functions may also be useful, p. 88.) The `hexifybytes` property was added in version 7. It is also settable in the `conf/taxis.ini` config file (p. ??). The `compatibilityversion` property (p. 170), when set, affects this property as well.

unalignedbufferwarning Whether to issue “Unaligned buffer” warning messages when unaligned buffers are encountered in certain situations. Messages are issued if this setting is `true/nonzero` (the default). Added in version 7.00.1366400000 20130419.

unneededrexescapewarning Whether to issue “REX: Unneeded escape sequence ...” warnings when a REX expression uses certain unneeded escapes. An unneeded escape is when a character is escaped that has no special meaning in the current context in REX, either alone or escaped. Such escapes are interpreted as just the literal character alone (respect-case); e.g. “`\w`” has no special meaning in REX, and is taken as “`w`”.

While such escapes have no meaning currently, some may take on a specific new meaning in a future Taxis release, if REX syntax is expanded. Thus using them in an expression now may unexpectedly (and silently) result in their behavior changing after a Taxis update; hence the warning message. Expressions using such escapes should thus have them changed to the unescaped literal character.

If updating the code is not feasible, the warning may be silenced by setting `unneededrexescapewarning` to 0 – at the risk of silent behavior change at an upgrade. Added in version 7.06.1465574000 20160610. Overrides `[Taxis] Unneeded REX Escape Warning` setting (p. ??) in `conf/taxis.ini`.

nulloutputstring The string value to output for SQL `NULL` values. The default is “`NULL`”. Note that this is different than the output string for zero-integer date values, which are also shown as “`NULL`”. Added in version 7.02.1405382000 20140714.

validatebtrees Bit flags for additional consistency checks on B-trees. Added in version 7.04.1449078000 20151202. Overrides `[Taxis] Validate Btrees` setting (p. ??) in `conf/taxis.ini`.

Part II

Metamorph Intelligent Query Language

Chapter 1

Metamorph: The Program Inside the Program

1.1 Background

Deep inside of Taxis stands Metamorph, the original incarnation of Thunderstone's text retrieval methodology. While Metamorph has grown into Taxis, its intelligent text query language has remained intact. It carries with it a whole philosophy of concept based text retrieval which sets a layer of assumptions about how text, particularly English, should be processed to retain its meaning.

Metamorph is made reference to in many places throughout Thunderstone's program documentation. While originally it was a stand-alone program, it lives on as functions which are used at all levels of the search, chiefly thought about only in relation to the formulation of queries themselves, and attendant processing of the vocabulary base of the language itself.

Most of what you need to know about Metamorph is included at the appropriate places in our documentation. This section is devoted to some of the aspects of Metamorph which were documented early on in the Research & Development of our product line, but may still be of interest for context rich applications or just to the curious user.

1.2 Original Design Mandate

Based on their past experience, many people have become accustomed to certain fixed ways of viewing the information dilemma; i.e., how to deal with too much available stored information. However, most of these accepted solutions contain inherent flaws which can be readily seen and solved once one realizes that there are choices now which we did not have not too long ago.

Thunderstone took a fresh approach to text analysis and data search problems from the very core; that is, in the methods of pattern matching itself. It has always been our premise that if we could develop pattern matching algorithms fast and precise enough, we could take a different approach to data research which would provide more accurate and reliable results, while being less labor intensive in the processing of the raw data.

Different algorithms are most efficient for different tasks; therefore we have learned how to make use of the best tool for the best job by integrating a number of smaller programs through a senior program. The Metamorph Search Engine calls upon the right pattern matcher for the right job, encompassing several different pattern matchers: PPM (Parallel Pattern Matcher), SPM (Single Pattern Matcher), Wildcard '*' matcher, REX (Regular EXpression pattern matcher), XPM (ApproXimate Pattern Matcher), and NPM (Numeric Pattern Matcher). Each pattern matcher handles a certain class of search problems and is optimized for a particular type of task, making the overall search environment as fast and efficient as possible.

In light of the rapid advances being made in hardware configurations, we have maintained that the best approach to the creation of all data manipulation tools must be totally software solutions. In this way we can as required modify our software to take advantage of new hardware breakthroughs, rather than be tied to outdated hardware systems. We believe this represents a more cost effective solution to our clients so that only their hardware budget concerns need be weighty, knowing our software will be flexible enough to be portable to the hardware configuration of their choice.

By way of example, we are in fact operational on more than 25 different Unix platforms, evidencing more portability than any other software company around. Using our Application Program Interfaces (API's), we can extend portability of the Search Engine to almost any platform. And now that Metamorph is inside Taxis, a complete and multi-faceted Relational DataBase Management System, its use can be extended truly to any application that might be desired, now in this exciting global village of totally connected Internet and World Wide Web applications.

Some hardware pattern matching solutions have surfaced along the way, some quite impressive. However, there are certain inherent problems with these techniques no matter how fast they appear to be. What can be located is limited, and there is limited portability. If the host configuration is greatly changed, an entirely new piece of hardware will in most cases be required. Metamorph on the other hand, being entirely software, always has retained the potential of being moved to a new hardware environment. Our technology has shown flexibility over time in DOS, Unix, MVS, OS/2, MACH, Macintosh, Windows, and NT environments, spanning micro, mini and mainframe applications.

1.3 Tokenization and Inverted Files

Two leading techniques in text retrieval that have heretofore been used have been file inversion and tokenization. The dominant problem with both of these techniques is that they require modification of the original data to be searched in order for it to be accessible to the data retrieval tool. The second problem, which has deeper ramifications, is that in order to perform file inversion or tokenization such programs make certain predisposed determinations about the lexical items that will be later identified.

How good such programs are will depend in great part upon their ability to identify and then locate specified lexical items. In most cases the set of lexical items identified by the inversion or tokenization routines is simply insufficient to guarantee the retrieval of all things that one might want to search for. But even where the set of identifiable lexical items is reasonably good, one always has a certain basic limitation to contend with: one will never be able to locate a superset of the lexical item listed in the look up table.

It is for these reasons that when we make use of indexing techniques, we supplement them with a final linear text read where required. Many content oriented definitive searches must contain a linear read of

context to make accurate determinations involving relevance.

In systems where a lookup table exists containing either file pointers or tokens, context is missing. You cannot search for something next to something else, as no adequate record of related locations of items is contained in the lookup table. You may yet find what you are looking for, but you may have to convert the file to its original form before you can do so.

It will be hard to find a program which stores any, let alone all, possible combinations of words making up phrases, idiomatic expressions, and acronyms in the lookup table. While you can look for “Ming”, or you can look for “Ling”, you cannot directly look for “Ming-Ling”. Another tricky category is that of regular expressions involving combinations of lexical items. If you are searching the Bible by verse, you want to find a pattern of “digit, digit, colon, digit”. This cannot be done when occurrences of digits are stored separately from the occurrences of colons.

Our own database tool Taxis has a modifiable lexical analyzer, which goes further than other indexing programs to attend to this problem. However, a linear free text scan still gives the maximum flexibility for looking for any type of pattern in relation to another pattern, and therefore has been included in Taxis as part of the search technique used to qualify hits.

Metamorph is the search engine inside of Taxis which contains maximum capability for identification of a diverse variety of lexical items. No other program has such an extended capability to recognize these items. We can look for special expressions by themselves or in proximity to sets of concepts. Logical set operators ‘and’, ‘or’, and ‘not’ are applied to whole sets of lexical items, rather than just single, specified lexical items. Because Metamorph is so fast, benchmarked even in its very early years of development as searching up to 4.5 megabytes per second in some Unix environments, we can read text live where required and get extremely impressive results.

Where stream text is involved, such as for a message analysis system where large amounts of information are coming in at a steady rate, or a news alert profiling system, you could not practically speaking tokenize all the data as it was coming in before reading it to find out if there was anything worth attending to in that data. Using a tokenized system to search incoming messages at a first pass would be very unwieldy, as well as inefficient and lacking in discretion. Indexes are more appropriately useful when searching large amounts of archived data. Taxis makes use of Metamorph search engines internally where required to read the stream text in all of its rich context but without losing speed of efficiency.

Where Taxis sends out a Metamorph query, it is fast and thorough in its search and its retrieval. A parser picks out phrases without having to mark them as such, along with any known associations. Search questions are automatically expanded to sets of concepts without you having to tell it what those sets are, or having to knowledge engineer connections in the files. Regular expressions can be located which other systems might miss. Numeric quantities entered as text and misspellings and typos can be found. Intersections of all the permutations of the defined sets can be located, within a defined unit of text defined by the user. No other search program is capable of this.

Even were you to find a comparable Database Management System with which to manage your text (which we challenge you to do!), at the bottom line, you could not find all the specific things that the Metamorph search engine would let you find. In Taxis we now have a completely robust body; inside, it yet retains the heart and soul of Metamorph.

1.4 Metamorph Query Language Highlights

Metamorph allows you to search for intersections of sets of lexical items, while also performing prefix and suffix morpheme processing. Once your target is found the question arises: what rules govern proximity of the items you wish to find? In traditional searching tools, this has been done only on a line by line basis, or by using some quantitative proximity range. Metamorph can search by an intelligent textual unit, a sentence. Whether searching by paragraph, page, chart entry, or memo, in all respects it is intended that the user may define real qualitative units of communication inside of which the concepts he is interested in connecting are located.

The user can specify right within his or her query the delimiters of choice: i.e., he can look within a sentence, paragraph, a proximity of 500 characters, or a specially defined textual unit such as a memo. To the degree that lexical items can be defined and located as beginning and end delimiters, your intersections will be located within those parameters.

REX, Metamorph's Regular EXpression pattern matcher, can be used outside Taxis as a special text processing tool. REX can locate uniquely repeated patterns in files, such as headers, footers, captions, diagram references, and so on. If the existing patterns aren't adequate to your needs, you can put them into your files rather easily. For example, using REX's incrementing counter and its search and replace facility, one could locate paragraph starts and number them. Such pattern identification can be made use of by other applications.

Metamorph allows for editing word sets, by hand or using the Backref program. This means that you may select which associations you would like in connection to any search; you can create your own concept sets permanently for future use. You can fine tune the search to use associations of only a certain part of speech. You can enter all known spelling variations of any particular search word in the same way. You can generally customize the program to include your own nomenclature and vocabulary, making it increasingly intelligent the longer it is in use.

You can call up the ApproXimate Pattern Matcher (XPM) and tell it to look for a certain percentage of proximity to an entered string, finding misspelled names and typos. You can also look for numeric quantities entered as text with the Numeric Pattern Matcher (NPM), finding "four score and seven years ago" in the Gettysburg address when searching for events 80 to 100 years ago.

The Metamorph Query Language was designed so that the text searcher can get rudimentary satisfaction of result right away without needing to know much of anything. At the same time, a more complex query can be written with just a little self-training time on the advanced search syntax possibilities. We like to say that there's *nothing* that can't be found with a Metamorph query. This flexibility enhancing Taxis, means the system designer setting up the search environment and wanting to customize it to certain applications can accomplish all his goals.

Taxis, with Metamorph inside it, is intended to be a modular set of tools to attack the formidable problem of how to get at and deal with large quantities of information, when you don't really know what you want to know or where to find it; and in the most dynamic, efficient, and pragmatic way possible. It is intended for discrete analysis where the human supplies the final cognition.

1.5 Your Basic Metamorph Search

Metamorph has often been classified as a form of Artificial Intelligence since its functions fall into the categories of knowledge acquisition, natural language processing, and intelligent text retrieval.

The software attempts in its own way to understand your question, represent its understanding to the data in the files, and come up with relevant responses as retrieved portions of full text information which best correspond to your questions.

Metamorph's vocabulary is around 250,000+ word connections, constructed in a dense web of associations and equivalences. Search parameters can be adjusted to dynamically dictate surface and deep inference. The program's responses can be controlled so that they are direct or abstract in relation to your questions. Proximity of concept can be fine tuned so as to qualify degree of relevance, providing matches which are sometimes concrete, sometimes abstract.

Think of your text as a field of information which was put together by a human being for a stated purpose; the Equivalence File acts as an intelligent language filter through which relevant associations occurring as common denominators can be located and retrieved out of the information in those files.

Metamorph retrieves data as a match to queries for response from any text. Metamorph can search files which are not flat ASCII, but it is the ASCII characters which will be recognized.

To help you get started, some demo text files are supplied with the Taxis package. These are files in the "c:\morph3\text" directory if you installed on Drive C in DOS or Windows; or in the "/usr/local/morph3/text" directory if you are on Unix:

| Filename | Description |
|----------|---|
| alien | science fiction excerpts |
| constn | the US Constitution |
| declare | the US Declaration of Independence |
| garden | descriptive prose |
| kids | children's adventure stories |
| liberty | Patrick Henry's "liberty or death" speech |
| events | downloaded news information from mid 1990 |
| qadhafi | magazine interview with Mohamar Qadhafi |
| socrates | summary from Plato's Republic about Socrates. |

Let's say you have set things up one way or another to be searching these demo text files, and have Taxis set up to type in a query. The easiest thing to do is think of a few concepts or keywords you'd like to see matched near each other in a sentence. For example, "power" and "struggle". This can be entered on the query line as "power struggle", where your default proximity is set to search by sentence. Or if it has not, you can enter your query as "power struggle w/sent". If you have enabled Metamorph hit markup, you might retrieve a passage of text which could be set up to look something like this:

```
File: c:\morph3\text\events
PageNo: 11
Query: power struggle
```

```
million whites and 28 million blacks. []Charges range from using
excessive FORCE on antigovernment protests and torture of detainees
to openly backing the ANC's rival Zulu movement Inkatha in bloody
CLASHES in Natal province.[]
```

```
"It's a Frankenstein which has been created and inherited by a
racist set-up," Slovo told a news conference, noting the ANC
"reserves the right" to resume the armed struggle "should the
government fail to carry out its undertakings."
```

In this example, this section of text was selected because the sentence marked by [] blocks at beginning and end matched the search request; i.e.:

Charges range from using excessive **force** on anti-government protests and torture of detainees to openly backing the ANC's rival Zulu movement Inkatha in bloody **clashes** in Natal province.

This sentence was selected as a hit because it contained a concept match to both concepts entered on the query line: i.e., "force" matched "power", and "clashes" matched "struggle".

1.6 More Complex Query Syntax

As you learn more about how the Metamorph Query Language works, your queries can become more complex, if so desired. Two key factors which are part of any query, along with a statement of the search items you are looking for, are intersection quantity and delimited text unit.

Search items can be weighted, by marking them for inclusion with a plus sign '+', or for exclusion with a minus sign '-'. All other search items are considered equally weighted.

It is understood that the maximum number of unmarked search items will always be looked for, unless a designation follows those sets of '@#'; i.e., the at sign '@' followed by the desired number of intersections; i.e., 0-9+. Designating "@0" would mean "zero intersections required".

For example, one might enter this query:

```
+Near East @1 military political economic involvement
```

This query would locate any sentences which definitely contained a reference to one of the countries in the Near East set, and also contained at least one intersection of 2 of the 4 specified search sets: "military", "political", "economic", and "involvement". Thus it would retrieve the following sentence, where the words in curly braces indicate the set to which the preceding word member belongs:

Troops in **Turkey** {Near East} became **engaged** {involvement} in a heated **battle** {military} when a training exercise was misinterpreted as a hostile initiative.

You can further qualify the type of search results you are after by changing the delimiters which dictate the proximity of entered concepts. This can be done by adding to the query line “w/delim” where “delim” is either “line”, “sent”, “para”, or “page”. (NOTE: “page” only works where page formatting characters exist in the text.) Or you can just choose an arbitrary number of characters to search within; like: “w/250” to indicate a window of 500 characters (250 forward, 250 back) around the first search item found.

Therefore you could change the nature of the above query:

```
+Near East @1 military political economic involvement w/para
```

By adding a delimiter specification “w/para”, you are instructing Metamorph to search by paragraph rather than by sentence. The required proximity of concept is now much broader. More paragraphs will be found to fit all the search requirements than sentences. But the sentences are likely to be closer matches to the query, as the correlation of concept had to be a closer match.

In addition to entering English questions and keywords, Metamorph has several special pattern matchers which allow the user to search for practically any type of expression. Any such expression is a valid search item, which can be assigned a logic operator, and searched for in proximity to other keywords, concepts, or expressions, as outlined in the previous section.

All the above things are covered elsewhere in detail. Their strength can be drawn upon where very specific types of results are desired.

Metamorph lets you create new and varied viewpoints as to what the originator of the files might have intended, without hours of preprocessing or knowledge engineering. These new views and impressions can be informative, educational, and useful, enhancing content analysis and data correlation.

Chapter 2

Hits, Intersections and Sets

2.1 Definition and Background

A “*hit*” is the term used to refer to a portion of text Metamorph retrieves in response to a specified query or search request. We use the term “hit” to mean the program has found or “hit” on something it was looking for. Hits can have varying degrees of relevance or validity, based on the number of intersections of concept matched within that delimited block of text.

An “*intersection*” is defined as an occurrence of at least one member of each of two sets of strings located within a section of text as demarcated by a specified beginning and ending delimiter.

A “*set*” is the group of possible strings a pattern matcher will look for, as specified by the search item. A set can be a list of words and word forms, a range of characters or quantities, or some other class of possible matches based on which pattern matcher Metamorph uses to process that item.

Intersection quantity refers to the number of unions of sets existing within the specified delimiters. The maximum number of intersections possible for any given query will be the maximum number of designated sets minus one.

The number of intersections present in any given hit directly determines how relevant that response will be to the stated query. Therefore, the designated intersection quantity can modulate the amount of abstraction or inference you wish to allow away from the original concept as stated in your query.

Because intersection quantity can be adjusted by the user to determine how tightly correlated the retrieved responses must be, the term “correlation threshold level” has in the past been used interchangeably with “intersection quantity” in earlier writings and versions of Metamorph. However, intersection quantity is a more precise lexical concept, and is therefore the preferred and more accurate term.

Intersections, whether calculated automatically by Metamorph or set by the user, signify the number of intersections of specified sets being searched for, to be matched anywhere within the delimited block of text; this could be within a sentence, paragraph or page or some other designated amount of text.

Metamorph provides a template of variables which go into a search, all of which can be stored so that one can analyze exactly what produced the search results and adjust them as desired. The intersection quantity variable is the most important in terms of designating how closely you wish to have the retrieved responses

(hits) correlated to your search query.

2.2 Basic Operation

A default Metamorph search will calculate the maximum number of intersections possible in a hit based on your query.

Metamorph picks out the important elements of your question and creates a series of sets of words and/or patterns; counting the number of sets, it does a search for any hit which contains all those elements. If you ask: “Was there a power struggle?” the program will look for hits containing 1 intersection of the 2 sets “power” and “struggle”. In this case the program is executing the search at intersection quantity 1, as it is looking for text containing 1 union of 2 sets.

When you first enter the query, Metamorph determines how many sets of concepts it will be searching for. Upon execution, the program will attempt to find hits containing the maximum number of intersections of something from each of those sets, within the defined delimiters of a sentence.

A “sentence” is specified as the block of text from one sentence type delimiter [. ? !] to the next sentence type delimiter [. ? !]. Any sentence found which contains that maximum number of intersections will be considered a valid hit, and can be made available for viewing, in the context of the full text of the file in which it is located.

2.3 Intersections and Search Logic

A common and simple search would be to enter a few search items on the query line, in hopes of locating a sentence which matches the idea of what you have entered. You want a sentence containing some correlation to the ideas you have entered. For example, enter on the query line:

```
property evaluation
```

This should find a relevant response containing a connection to both entered keywords. The user would be happy to locate the sentence:

The broker came to **assess** the current market value of our **house**.

Here “assess” is linked to “evaluation”, and “house” is linked to “property”, within the bounds of one sentence. Therefore it is a hit, and can be brought up for viewing.

Requiring an occurrence of 2 search items is referred to as 1 intersection. The occurrence of 3 search items is 2 intersections. The default search logic is to look for the maximum number of intersections possible of all entered search items. If you were searching large quantities of text, you could add several qualifiers, as follows:

```
property tax market assessment
```

The maximum number of intersections would be looked for; i.e., an occurrence of “property”, “tax”, “market”, and “assessment”, anywhere inside the text unit. This can be thought of as “and” logic.

To override the default “and” logic, you would enter the desired number of intersections with ‘@#’: as “@0” (“at zero intersections”), “@1” (“at one intersection”), “@2” (“at two intersections”), and so on.

To get “or” logic you would enter “@0” on the query line with your search items. In this example:

```
@0 property tax
```

“@0” (read as: “at zero intersections”) means that zero intersections are required; therefore you are specifying that you want an occurrence of either “property” or “tax” anywhere inside of the delimited text, or sentence.

To obtain different permutations of logic you can specify a number of intersections greater than zero but less than an intersection of all specified items. For example:

```
@1 property tax value
```

requires one intersection of any two of the three specified items. Therefore, the text unit will be retrieved if it contains an occurrence of “property” and “tax”, “property” and “value”, or “tax” and “value”.

Any search item not marked with a ‘+’ or a minus ‘-’ is assumed to be equally weighted. Each unmarked item could be preceded with ‘=’ but it is not required on the query line as it is understood. Intersection quantities “@#” apply to these equally weighted sets not otherwise marked with ‘+’ or ‘-’.

Mandatory inclusion ‘+’ and mandatory exclusion ‘-’ logic can be assigned, and can be used with combinatorial logic. For example, you might enter this query:

```
+tax -federal @1 market assessment value property assets w/page
```

This query means that within any page (“w/page”), you must include an occurrence of “tax” (designated with the plus sign ‘+’), but must exclude the hit if it contains any reference to “federal” (designated with the minus sign ‘-’). The “@1” means that you also want 1 intersection of any 2 of the other 5 equally weighted (unmarked with ‘+’ or ‘-’) query items: “market”, “assessment”, “value”, “property”, and “assets”.

2.4 Intersection Quantity Specification Further Explained

If you want all possibilities you can find, you will find very adequate answers to your search requests usually by setting the quantity to 1 or 2. If you don’t get answers at that quantity, drop it lower.

With intersections at 0, the program will retrieve hits containing 0 set intersections; i.e., it will locate any occurrence of any set element. The denotation of 0 indicates no intersection of specified sets is required. In other words, you want Metamorph to locate any hit in which there is any equivalence to any of the set elements in your question. This is comparable to an “or” search.

With intersections at 1, the program will retrieve hits containing an intersection of any 2 specified set elements; for example, an intersection of “life” and “death”. With intersections at 2, the program will retrieve hits containing any 2 intersections of any 3 specified set elements; e.g., “life” and “death” plus “death” and “infinity”. With the intersection variable at 3, the program might locate an occurrence of “life” and “death”, “death” and “infinity”, plus “infinity” and “money”. The higher the number of intersections being sought, the more precise the retrieved response must be.

The lower you set the intersection quantity, the more hits are likely to be found in relation to the question asked; but many of them are likely to be quite abstract. The higher you set the intersection quantity, the fewer hits are likely to be found in relation to the question asked; but those hits are likely to be more intelligently and precisely related to the search query, as they require a higher number of matching sets.

A smaller intersection quantity will also retrieve any larger intersection quantity responses as a matter of course. However, the program only searches for the number of intersections of elements it is directed to search for, so it will only highlight the requested number of sets.

Conversely, the program cannot locate hits containing more intersections than are possible. If you have set intersections to 6, but you enter a question containing only 3 set elements, realize that one pass through the data will find no answers, as you have designated more intersections than are possible to find.

2.5 How ‘+’ and ‘-’ Logic Affects Intersection Settings

The assignment of intersection quantity refers to the number of intersections of sets being sought in the search. This presumes that all sets are equal to each other in value. A search for:

```
life love death
=life =love =death (same as above)
```

presumes that each set has equal value; if something from each set is found, it would reflect an intersection quantity of 2, as it contains 2 intersections of “equal” value. In Metamorph terms, this is referred to as “set logic”.

When you assign a ‘+’ (“must include”) logic operator to a set, this means that you must include something from that designated set in any hit that is found. When you assign a ‘-’ (“not”) logic operator to a set, this means that you must exclude any hit which contains any element from that designated set.

Examples of these might be:

```
+spirit =life =love =death
      or
-alien =life =love =death
```

It doesn’t matter whether the equal sign ‘=’ is entered or not. It also doesn’t matter what sequence these operand sets are entered in, or how many of each. The only rule is you cannot enter only “not” sets.

In the above two examples, the maximum intersection quantity is still 2. The “must include” (+) set (“and logic”) and the “must exclude” (-) set (“not logic”) are outside the calculation of maximum intersections possible, as both are special cases. In a search where you designate something like this:

```
war peace -Reagan +diplomacy
```

the maximum intersection quantity is only 1, referring to an intersection of the “war” and the “peace” sets. Therefore, if you wish to definitely find something related to “diplomacy”, but to exclude any references related to “Reagan”, and you want to intersect what is found with either war or peace, set intersections to 0:

```
@0 war peace -Reagan +diplomacy
```

The 0 intersection setting applies to either the “war” set or the “peace” set (the unmarked equally weighted sets), as both “Reagan” and “diplomacy” are special cases and not included in the intersection quantity calculation.

2.6 Modulating Set Size

Metamorph can draw upon a vast built-in intelligence comprised of equivalence sets, as contained in a 2MB+ collection of 250,000+ word associations. The user has the option of how much of this “mind” he wants to draw upon at any given time, by regulating to what degree he wishes to make use of the built-in connective structure in this Equivalence File.

When enabled via `keepeqvs/Synonyms` (for the entire query) or the `~` operator (for a particular query term), each query term is expanded by pulling from the Equivalence File the set of known associations listed with each entry. This lets you draw upon the full power of all the associations in all of its equivalence sets. Still, you have the option of quantitatively limiting or expanding this set expansion, or selectively editing their content.

2.6.1 Limiting Sets to Roots Only

There may be times when you want to limit set size to root words only from your search query. This is the default in `tsql`, Vortex and the Search Appliance. If using the MM3 API, or if equivalences have been enabled with `keepeqvs` (Vortex/`tsql`) or `Synonyms` (Search Appliance), then equivalences can be excluded for a particular word – while still retaining morpheme processing – by preceding the word with a tilde (`~`). This should be done where you wish to look for intersections of concepts while holding abstraction to a minimum, and you do not want any automatic expansion of those words into word lists. If equivalences are not currently enabled via `keepeqvs/Synonyms`, then the `~` operator has the opposite effect: it *enables* equivalences for that term only. Thus, it always toggles the pre-set behavior for its word.

Regardless of settings, you can also give explicit equivalences directly in the query, instead of the thesaurus-provided ones, using parentheses; see p. 231.

Where you wish to use morpheme processing on root words only, restricting concept expansion completely, turn off Equivalence File access altogether by setting the global APICP flag “`keepeqvs`” off. Where “`keepeqvs`” is set to off, the tilde (`~`) is not required (indeed, it would re-enable concept expansion).

Restricting set expansion is useful for proper nouns which you do not want expanded into abstract concepts (e.g., `President Bush`), technical or legal terminology, or simply any precise discrete search. Selectively cut off the set expansion by designating a tilde (~) preceding the word you are looking for.

Using REX syntax by preceding the word with a forward slash (/), can further delimit the pattern you are looking for, though in a different manner. Note however, that using non-SPM/PPM pattern matchers such as REX may slow the query, as Metamorph indexes cannot be used for such terms.

To check this out for yourself, in an application where Metamorph hit markup has been set up, compare the results of the following queries (assuming Vortex or `tsql` defaults):

```
Query 1:  President Bush
Query 2:  ~President ~Bush
Query 3:  "President Bush"
Query 4:  /President /Bush
Query 5:  /\RPresident /\RBush
```

Query 1 `President Bush`: In the first example you would get any hit containing an occurrence of the word “President” and the word “Bush”, including other related word forms (suffixes etc.). So you would get a hit like “**President Bush** came to tea.”; as well as “**Bush** attended a conference of corporation **presidents**.” There are no equivalences added to the “President” set, or the “Bush” set.

Query 2 `~President ~Bush`: In the second example you have elected to keep the full set size, so you would obtain references to “President” and “Bush” while also allowing for other abstractions. Since the word “chief” is associated with “president”, and the word “jungle” is associated with “bush”, you would retrieve a sentence such as, “We met the **chief** at his home deep in the Amazon **jungle**.”

Query 3 `"President Bush"`: The third example calls for “President” and “Bush” as a two word phrase by putting it in quotes, so that it will be treated as one set rather than as two. It has no equivalences, because the phrase “President Bush” has no equivalences known by the Equivalence File; you could add equivalences to that phrase if you wished by editing the User Equiv File. While you would retrieve the hit “**President Bush** came to tea.”, you would exclude the hit “**Bush** attended a conference of corporation **presidents**.” You would get a hit like “We elected a new **president Bush**.”

Query 4 `/President /Bush`: In the fourth example you have limited the root word set in a different way. Signalling REX with the forward slash ‘/’ means that you will use REX to accomplish a string search on whatever comes after the ‘/’. Therefore, you can find “Our **president’s** name is **Bush**.” and “We planted those **bushes** near the **President’s** house.” This search gets similar yet different results than Example 1. Look at exactly what is highlighted by the Metamorph hit markup to see the difference in what was located.

Query 5 `/\RPresident /\RBush`: In the fifth example there is better reason to use REX syntax, so that you can limit the set even further by specifying proper nouns only. The designation ‘\R’ means to “respect case”, and would retrieve the sentence “**President Bush** came to tea.”, but would rule out the sentence “**Bush** attended a conference of corporation **presidents**.” It would also rule out the hit “We elected a new **president Bush**.”, and “We planted those **bushes** near the **President’s** house.”

NOTE: In the previous example, the “respect case” designation (`\R`) must follow a forward slash (`/`) which indicates that REX syntax follows. Remember that words in a query are not case sensitive unless you so designate, using REX. (See Chapter on REX syntax.)

2.6.2 Expanding Sets by Use of See References

You can exponentially increase the denseness of connectivity in the equivalence file by turning on the option called “See References”. This greatly increases set size overall.

The concept of a “See” reference is the same as in a dictionary. When you are looking up the word “cat”, you’ll get a description and a few definitions which try to outline exactly what the concept of “cat” is all about. Then it may say at the end, “See also pet”.

With See References set to on, the equivalence list associated with the word “cat” would be automatically expanded to include all equivalences associated with the word “pet”. Not all equivalences have “See also” notations, but with See References on, all equivalences associated with any “See also” root word will be included as part of the original. With See References off, only the root word and its equivalences will be included in that word set.

In the Equivalence File, See References are denoted with the at sign (`@`). In this way, additional second level references can be added or deleted from a root word entry.

For some of the more common words like “take” and “life” and such, allowing See References can make a set surprisingly huge, so you should think twice before operating in this mode. It is really intended to expand the range of opportunity to the user, so that there is more to pick and choose from when editing the content of the word lists.

If you are using the default equivalence sets it is generally better to run with the “See” option “off”. You would not normally want to be doing searches where the word lists are too large; we suggest limiting the set size to well under 100 in any case, whether you are running with See References either on or off.

In a theoretical sense, given no limitations, using endless levels of See References could eventually create a circle which covered the whole spectrum of life, and led right back to the word one had started with. Therefore Metamorph has intentionally built-in limitations on set size to prevent this connectivity of concept from becoming redundant or out of hand. Firstly, See References are restricted to only one level of reference. Secondly, if a word set approaches around 256, Metamorph will automatically truncate it in the search process, cutting off any words outside the numerical limit.

To make use of the See References option, set the corresponding APICP flag “see”.

2.7 Modulating Set Content

The ability to dictate precise content of all word sets passed to the Metamorph Search Engine is left open to the user on an optional basis. We have tried to build into the program enough intelligence through its Equivalence File so that you would never be required to pay any attention to the lists of equivalences if you didn’t want to.

Even if Metamorph has no known associations with a word it will still process it with no difficulty, using its

built-in morpheme processing capability. You'll find that more often than not you are not required to teach Metamorph anything about your search vocabulary to get satisfactory results, even when using technical terminology.

When you want to control exactly what associations are made with any or all the words or expressions in your searches, you can do so by editing the equivalence set associated with any word already known by the Equivalence File, or by creating associations for a new or created word not yet known. This is covered in detail under the Chapter on *Thesaurus Customization*.

2.8 Modulating Hit Size by Adjusting Delimiters

2.8.1 Discussion

Delimiters can be defined as the beginning and ending patterns in the text which define the text unit inside of which your query items will be located. Concept proximity is adjusted through delimiters.

If you look for the words “bear” and “woods” within a sentence, the result will be a tight match to your query. Looking for “bear” and “woods” inside a paragraph is less restrictive. Requiring only that “bear” and “woods” occur inside the same section might or might not have much to do with what you are looking for. Knowing that, you would probably add more qualifying search items when searching by section; e.g., “bear” “woods” “winchester” “rifle”. In short, the relevance of your search results will differ greatly based on the type of text unit by which you are searching.

Beginning and ending delimiters are defined by repeating patterns in the text, technically known as regular expressions. A sentence can be identified by its ending punctuation (period, question mark, or exclamation point); a paragraph is often identified by a few new lines in a row; a page is often identified by the presence of a page formatting character.

In a Metamorph search, the entered query concepts will be searched for within the bounds of two such expressions, called delimiters. These regular expressions are defined with REX syntax for Regular Expressions. If you know how to write a regular expression using REX you can enter delimiters of your own design. In lieu of that you can rely upon the defaults that have been provided.

Since the most common types of search are by line, sentence, paragraph, page, or whole document, these expressions have been written into the Metamorph Query Language so that you can signal their use dynamically from within any query, using English. The expression “w/delim” means “within a ...” where 4 characters following “w/” are an abbreviation for a commonly delimited unit of text. If you want to search by paragraph, you would add “w/para” as an item on the query line.

For example:

```
power struggle w/para
```

Such a search will look within any paragraph for an occurrence of the concept “power” and the concept “struggle”.

You can designate a quantitative proximity, using the same syntax, by stating how many characters you want before and after the located search items. For example:


```
power struggle w/150
```

Such a search will look within a 300 character range (150 before, 150 after the first item located) for an occurrence of the concept `power` and the concept `struggle`. This is useful for text which doesn't follow any particular text pattern, such as source code.

You can also write your own expressions, useful for section heads and/or tails. To enter delimiters of your own design, create a REX expression first which works, and enter it following the "w/". For example:

```
power struggle w/\n\RSECTION
```

This search uses the pattern "SECTION" where it begins on a new line and is in Caps only, as both beginning and ending delimiters. Thus an occurrence of the set "power" and the set "struggle" need only occur within the same section as so demarcated, which might be several pages long.

You can figure out such useful section delimiter expressions when setting up an application, and use the corresponding APICP flag `sdexp` (start delimiter expression) and `edexp` (end delimiter expression) in a Vortex script to make use of them.

The following examples of queries would dictate the proximity of the search items "power" and "struggle", by specifying the desired delimiters. Noting the highlighted search items, you might try these searches on the demo text files to see the difference in what is retrieved.

| | |
|--|--|
| <code>power struggle w/line</code> | within a line (1 new line) |
| <code>power struggle w/sent</code> | within a sentence (ending punctuation) |
| <code>power struggle w/para</code> | within a paragraph (a new line + some space) |
| <code>power struggle w/page</code> | within a page (where format character exists) |
| <code>power struggle w/all</code> | within whole document (all of the record) |
| <code>power struggle w/500</code> | within a window of 500 characters forward and backwards |
| <code>power struggle w/\$\$\$</code> | within user designed expression for a section; where what follows the slash '/' is assumed to be a REX expression. (In this case, the expression means 3 new lines in a row.) |

More often than not the beginning and ending delimiters are the same. Therefore if you do not specify an ending delimiter (as in the above example), it will be assumed that the one specified is to be used for both. If two expressions are specified, the first will be beginning, the second will be ending. Specifying both would be required most frequently where special types of messages or sections are used which follow a prescribed format.

Another factor to consider is whether you want the expression defining the text unit to be included inside that text unit or not. For example, the ending delimiter for a sentence (ending punctuation from the located sentence) obviously belongs with the hit. However, the beginning delimiter (ending punctuation from the previous sentence) is really the end of the last sentence, and therefore should be excluded.

Inclusion or exclusion of beginning and ending delimiters with the hit has been thought out for the defaults provided. However, if you are designing your own beginning and ending expressions, you may wish to so specify.

2.8.2 Delimiter Syntax Summary

```

w/{abbreviation}
or
w/{number}
or
w/{expression}
or
W/{expression}
or
w/{expression} W/{expression}
or
W/{expression} w/{expression}
or
w/{expression} w/{expression}
or
W/{expression} W/{expression}

```

2.8.3 Rules of Delimiter Syntax

- The above can be anywhere on a query line, and is interpreted as “within {the following delimiters}”.
- Accepted built-in abbreviations following the slash ‘/’ are:

| ABBREV | MEANING |
|--------|-----------------------|
| line | within a line |
| sent | within a sentence |
| para | within a paragraph |
| page | within a page |
| all | within a whole record |

| REX EXPRESSION | MEANING |
|--------------------------------|---|
| \$ | 1 new line |
| [^\digit\upper][.?!][\space'"] | not a digit or upper case letter, then a period, question, or exclamation point, then any space character, single or double quote |
| \x0a=\space+ | a new line + some space |
| \x0c | form feed for printer output |
| " " | both start and end delims are empty |

- A number following a slash ‘/’ means the number of characters before and after the first search item found. Therefore “w/250” means “within a proximity of 250 characters”. When the first occurrence

of a valid search item is found, a window of 250 characters in either direction will be used to determine if it is a valid hit. The implied REX expression is: “. { 250 }” meaning “250 of any character”.

- If what follows the slash ‘/’ is not recognized as a built-in, it is assumed that what follows is a REX expression.
- If one expression only is present, it will be used for both beginning and ending delimiter. If two expressions are present, the first is the beginning delimiter, the second the ending delimiter. The exception is within-*N* (e.g. “w/250”), which always specifies both start and end delimiters, overriding any preceding “w/”.
- The use of a small ‘w’ means to exclude the delimiters from the hit.
- The use of a capital ‘W’ means to include the delimiters in the hit.
- Designate small ‘w’ and capital ‘W’ to exclude beginning delimiter, and include ending delimiter, or vice versa Note that for within-*N* queries (e.g. “w/250”), the “delimiter” is effectively always included in the hit, regardless of the case of the w.
- If the same expression is to be used, the expression need not be repeated. Example: “w/[. ? !] W/” means to use an ending punctuation character as both beginning and end delimiter, but to exclude the beginning delimiter from the hit, and include the end delimiter in the hit.

Chapter 3

Thesaurus Customization

3.1 Specializing Knowledge

You could say that Metamorph's general knowledge approximates that of the man on the street without the benefit of specialized knowledge. This knowledge is stored in a large Thesaurus containing some 250,000 word associations supplied with the program.

It is not necessary for a query item to be stored in the Thesaurus for Metamorph to search for it. In fact, for many technical terms you would want only forms of that exact word rather than any abstract associations.

If you wanted someone to intelligently correlate information from a medical, legal, or other specialized or technical data base, you would expect him to have a working knowledge of the nomenclature used in discussing those subjects. Therefore, we have created a facility for teaching Metamorph specialized vocabulary and domain specific inference sets to draw upon for its concept set expansion.

You are customizing Metamorph for your own use by adding semantic knowledge to the web of equivalences. In this way, the program can be made "smarter" and able to take on the "viewpoints" of the user where required. Metamorph becomes an increasingly powerful retrieval tool through this process of the user teaching it new language and associations.

By passing this ability onto the user, and allowing for thesaurus customization to be an integral part of a search environment, the job of evolving vocabulary becomes entirely the user's responsibility and under his control. We provide a basic foundation of over 250,000 associations which is adequate to process any query with a reasonable amount of discretion and intelligence. The job of refining that vocabulary and keeping it up to date as language evolves is up to you.

3.2 Equivalence File Explanation

Metamorph draws upon a large interconnected web of root words and their equivalence sets called the Equivalence File. For every English word entered as a search item, a lookup is done in this file, and whatever list of associations exists therein is included with the specified root word.

The entered keyword is called the "root". The associations pulled from the Equivalence File are called the

equivalences, or “equivs”.

Each list of words created from such Equivalence File lookup is passed to PPM (the Parallel Pattern Matcher) for text searching. Morpheme processing is done on all equivalences of all root words. Therefore, all valid word forms of all words in a concept set are searched for.

There are over 250,000 of these word associations contained in the Equivalence File, making a Metamorph search fairly intelligent without the need for customization before trying it.

While the Equivalence File is similar to a thesaurus, it is more accurately comprised of associations of different types, signalling paths of equivalent weightings through a volume of vocabulary.

Where special nomenclature, slang, acronyms, or technical terminology is in abundance, one can customize the Equivalence File by making a User Equivalence File, through which tailored priorities can be ordered and new entries added.

3.3 Editing Concept Sets With Backref

It is intended that permanent Equivalence editing be done with the whole group’s needs in mind, rather than to please just one individual user. Vocabulary additions, specialized nomenclature, slang and acronyms that would be common to a particular group, subgroup, purpose or project, would be entered and saved in a User Equivalence File, as named with the corresponding APICP flag `ueqprefix`, and pointed to in the managing program; e.g., a Vortex script.

While it is possible to create several User Equivalence Files for different groups with differing acronyms and vocabulary, this is not always necessary. More often than not there is only one User Equivalence File per system, and multiples are created only to resolve conflicts in terminology.

Running with See References on produces anywhere from 75 to 200 equivalences associated with a particular word; more than 256 words are truncated. This is usually much more than is prudent, so it is intended that with See References on, you would edit such a list down to correspond more closely to what you really had in mind for the search question at hand.

To check existing equivalence set for a word, use the BACKREF program, where the syntax is:

```
backref -e input_file output_file
```

where `input_file` is your ASCII equiv source filename, and `output_file` is the backreferenced and indexed binary file which the Metamorph engine will use. The default User Equiv source file is `eqvsusr.lst` and its binary counterpart is `eqvsusr`.

When you enter a query like “power struggle” in a query input box, where concept search has been enabled, your search already knows 57 equivalences for `power`, and 23 equivalences for `struggle`. Using the above command for BACKREF, you can see what those sets are composed of, and you can edit them. When asked to enter a root term, enter `power`, and the list below would be revealed:

power;n 57 equivalences

| | | | |
|-----------------|-------------------|---------------|---------------|
| ability;n | intensity;n | primacy;n | might;u |
| acquisition;n | jurisdiction;n | regency;n | reign;u |
| ascendency;n | justice;n | restraint;n | rule;u |
| authority;n | kingship;n | scepter;n | sovereignty;u |
| carte blanche;n | leadership;n | skill;n | sway;u |
| clutches;n | majesty;n | strength;n | electrify;v |
| command;n | mastership;n | suction;n | |
| control;n | mastery;n | superiority;n | |
| domination;n | militarism;n | supremacy;n | |
| dominion;n | monarchy;n | vigor;n | |
| efficiency;n | nuclear fission;n | weight;n | |
| electricity;n | omnipotence;n | ability;u | |
| energy;n | persuasiveness;n | capability;u | |
| force;n | potency;n | control;u | |
| hegemony;n | predominance;n | energy;u | |
| imperialism;n | preponderance;n | faculty;u | |
| influence;n | pressure;n | function;u | |

struggle;n 23 equivalences

| | |
|---------------|------------|
| battle;n | agonize;v |
| combat;n | compete;v |
| competition;n | contest;v |
| conflict;n | fight;v |
| effort;n | flounder;v |
| exertion;n | strive;v |
| experience;n | |
| fight;n | |
| scuffle;n | |
| strife;n | |
| attempt;u | |
| clash;u | |
| conflict;u | |
| endeavor;u | |
| fight;u | |
| flight;u | |
| oppose;u | |

The root entry appears at the top, with its equivalences (each with an assigned class, or part of speech) listed underneath. The ‘n’ following (or preceding) “power” stands for “noun”, and is the class to which “power” has been assigned. (‘v’ means “verb”; ‘u’ means “unclassified”.)

Once editing a root word’s set of equivalences, you’ll have these choices, offered below:

Delete

Deletes equivalence (named by number).

Add

Opens word entry line below list to add new equivalence.

Zap

Deletes entire equivalence list.

| | |
|-----------------|--|
| Change-class | Prompts on line below for new word class assignment. |
| By-class-delete | Deletes all words in the entered class. |
| Save Changes | Saves all changes made to list to User Equiv File. |
| Undo Changes | Restores previous root word entry screen. |
| Redisplay | Refreshes the list with any changes made (or as it was). |

If you choose “Save Changes”, any changes made to the list will be saved to the named User Equivalence File when you quit the program with qq.

When a new word is added, you are prompted to enter its class. When the new word is added to the list, it will be sorted in alphabetically at its appropriate place in the list, under the class to which it belongs. Existing thesaurus entries have been classed according to the standard parts of speech as described below. However, you may create and assign any class you like.

In the example above under “struggle”, you might want to delete-by-class all those entries listed as verbs. Doing so would eliminate with one keystroke all equivalences classed as ‘v’: “agonize”, “compete”, “contest”, “fight”, “flounder”, and “strive”. The classes in use are as follows:

| | |
|---|--------------|
| P | Pronoun |
| c | conjunction |
| i | interjection |
| m | modifier |
| n | noun |
| p | preposition |
| u | unclassified |
| v | verb |

You can Undo Changes anytime while editing, to escape from the action you are in. This restores the entry screen, which lets you choose another root word to edit, add to, or delete. When you are finished editing a word, either Save or Undo Changes, then qq to Quit. At that point the source file will be indexed into its binary form usable by the search, if you have saved any changes.

3.4 Toggling Equiv Expansion On or Off

The APICP flag `keepeqvs` lets you set the default for concept expansion. If set on, unless otherwise marked, the search will use any equivalences found in the Equivalence File associated with any word entered in a query. This global condition can be selectively reversed by preceding a word in a query with a ~ tilde. If the global setting is off, the ~ will selectively enable concept expansion (i.e., equivalence look-up) on that word; if the global setting is on, the ~ will turn it off for that word. (Note that equivalences can also be explicitly specified in parentheses; see p. 231.)

While use of the Equivalence File is an integral part of the intelligence of a Metamorph search, in certain kinds of particularly specialized or technical data such abstraction may not be desired. Turning the `keepeqvs` flag off prevents any automatic lookup in the Equivalence File, changing the nature of the

search so that the emphasis is rather on intersections of valid word forms of specified English words, mixed in with special expressions. For example:

What RESEARCH has been done about HEALTH DRINKS?

Morpheme processing will be retained on the important (non-noise) words, but equivalences will not be included. An example of a sentence this question would retrieve, would be:

The company had been **researching** ingredients which would taste good in a **drink** while still promoting good **health**.

Where the global setting is on and you wish to selectively restrict Equivalence Lookup on some but not all words, you use tilde ‘~’ in front of those words where no equivalences are desired. This would be the preferred method of search for some types of technical material, such as medical case data.

Although at first glance it would seem that an effective Metamorph search could not be done on technical data until much specialized nomenclature was taught to the Equivalence File, this is not always the case. Often a technical term means only that, and the power is in intersecting some valid English form of that word with some other concept set.

An example of a very discrete query that requires no knowledge engineering beyond what comes “out of the box” with Metamorph might be as in this query (assuming `keepeqvs` turned on):

stomach ~cancer operation

The tilde ‘~’ is used to restrict equivalence lookup on “cancer” (toggle the `keepeqvs` setting, to off), as references to such things as “illness” rather than “cancer” would be too abstract. However, what you do want is concept expansion on the related words. Therefore, such a search would retrieve the sentence:

Suffering severe pains in his **abdomen**, it was first thought to be appendicitis; however this led to exploratory **surgery** which revealed **cancerous** tissue.

In this example, “cancerous” is a valid word form of “cancer” included through the morpheme process; “abdomen” was found because it was in the “stomach” equivalence list; “surgery” was found because it was in the “operation” equivalence list.

Again, with the `keepeqvs` flag set on, use of the tilde (~) reverses its meaning. Therefore, the above example would seek only valid word forms of the root words “stomach” and “operation”, but the tilde preceding “cancer” would selectively enable Equivalence Lookup on that word alone. So, with `keepeqvs` off, we might retrieve instead the following:

In the midst of the **operation** to remove her appendix, an abnormal **growth** was found in the **stomach** area.

This last response is matched because “operation” and “stomach” are forms of those root words; “growth” is in the equivalence list for “cancer” and was included in the set of possibilities due to the tilde (~) preceding it.

3.5 Creating a User Equivalence File By Hand

The User Equivalence File is read by Metamorph as an overlay to the Main Equivalence File. The search looks for matching root entries first in the User Equivalence File, and then in the Main Equivalence File. The information found in both places is combined, following certain rules.

When editing equivalences using the BACKREF program as shown, the changes you make are written to the named User Equivalence File. It is also possible to hand edit a User Equivalence File if you understand the syntax which is used when writing directly to it.

In order to precisely deal with issues such as precedence, substitution, removal, assignment, back referencing, and see references, a strict format must be adhered to. Any erroneous characters included in the User Equivalence File could be misinterpreted, causing unseen difficulties. Therefore, one must take care to ensure the User Equivalence file is flat ASCII.

If you want to create a User Equivalence file independent of the `backref -e` feature, follow these steps:

1. Using an ASCII only editor, open a file called “`eqvsusr.lst`”. If you already have a list of words and equivalences you want to add in a flat ASCII file, you can edit the entries into the prescribed format. Otherwise, simply begin entering root words with their equivalences as outlined in the following sections.
2. After creating the above User Equivalence File, it must still be indexed by the “`backref`” program supplied with your Taxis package. `Backref` takes an ASCII filename as the first argument, and creates a file of the name given in the second argument. For example, use this command on your ASCII User Equivalence File:

```
backref eqvsusr.lst eqvsusr
```

Where “`eqvsusr.lst`” is the ASCII file containing your User Equivalence entries, and “`eqvsusr`” is the file ready for use by the Metamorph search engine.

3. If you have not otherwise named a special User Equiv File in some managing program such as a vortex script, the indexed User Equivalence File must be called “`eqvsusr`”, and must be located in the “`morph3`” directory, in order for it to be used by a Metamorph search.

3.6 User Equivalence File Format

A User Equivalence File is an ASCII file created by the user, which corresponds to information in the larger 2+ megabyte Main Equivalence File which comes with the Taxis package.

Each root word is listed as a separate entry, on its own line with a list of known associations or equivalences (equivs).

The root words go down the lefthand side of the file, each one a new entry; the equivalences go out left to right as separated by commas. Word class (part of speech) and other optional weighting information may be stored with each entry.

Here is an example of a User Equivalence File. It contains no special information beyond root words and their equivalences. Its chief purpose would be the addition of domain specific vocabulary. Phrases are acceptable as roots or as equivalences.

```
chicken,bird,seed,egg,aviary,rooster
seed,food,feed,sprout
ranch,farm,pen,hen house,chicken house,pig sty
Farmer's Almanac,research,weather forecast,book
rose,flower,thorn,red flower
water,moisture,dew,dewdrop,Atlantic Ocean
bee pollen,mating,flower,pollination,Vitamin B
grow,mature,blossom,ripen
```

Root word entries should be kept to a reasonable line length; around 80 characters for standard screen display is prudent. In no case should a root word entry exceed 1K per line. Where more equivalences exist for a root word than can be fit onto one line, enter multiple entries where the root word is repeated as the first item. For example:

```
abort, cancel, cease, destroy, end, fail, kill
abort, miscarry, nullify, terminate, zap
```

It is important to remember that these are not just synonyms. They can be anything you wish to associate with a particular word: i.e., identities, generalities, or specifics of the word entry, plus associated phrases, acronyms, or spelling variations. Even antonyms could be listed if you wished, although that wouldn't generally be advisable.

The word "equivalence" is used in a programming sense, to indicate that each equivalence will be treated in weight exactly the same as every other equivalence in that set grouping when a search is executed.

3.7 Back Referencing

Both the Main Equivalence File and the User Equivalence File include a provision for "back referencing". That is, where a word is stored as the equivalence to another root entry, there is an inherent connection "backwards" to the root, when that equivalence is entered as a keyword (root) on the Query Line.

For example, imagine the following Equivalence File entry for the root acronym "A&R":

```
A&R, automation, robotics, automation and robotics
```

Metamorph, and so therefore Taxis, knows when you enter "robotics" as a root word in a query, that it should back search and associate it with "A&R". Therefore, when "robotics" is used as a keyword, "A&R" will be automatically associated as one of its equivalences.

Back referencing means that the following association is implicitly understood, and need not be separately entered:

robotics,A&R

Such automatic back searching capability exponentially increases the density of association and connection within the Equivalence File and User Equivalence File.

3.8 See Referencing

You can exponentially increase the denseness of connectivity in the Equivalence File by invoking “See References”, set with the APICP flag *see*.

The concept of a “See” reference is the same as in a dictionary. When looking up the word “cat”, you’ll get a description and a few definitions for cat. Then it may say at the end “see also pet”. With See Referencing on, the equivalence list associated with the word “cat” is expanded to also include all the equivalences associated with the word “pet”.

See Referencing greatly increases the general size of word sets in use in any search, increasing the chance for abstraction of concept. One would not normally invoke See Referencing, and would do so only where such abstraction was desired.

Not all equivalences have “See also” notations; but with See Referencing on, all equivalences associated with any “See also” root word will be included as part of the original. With See References off, only the root word and its equivalences will be included in that word set, regardless of whether a see reference exists or not.

See Referencing is restricted to only one level of reference, to prevent inadvertent “endless” looping or overlap of concepts. In any case, a word set will be truncated at the point it approaches 256 equivalences.

See References are denoted with the at sign (@). Enter the word preceded by ‘@’. For example:

```
cowboy, horse, cows, @rancher
rancher, plains, landowner
```

In normal usage, the search item “cowboy” expands to the set “cowboy”, “horse”, “cows” and “rancher”. With see referencing invoked, “cowboy” expands to “cowboy”, “horse”, and “cows” as well as “rancher”, “plains”, and “landowner”.

The see reference ‘@’ marking in a User Equivalence file will only connect entries which are in the User Equivalence file. In the above example, the entry for “rancher” must exist in the User Equivalence to be so linked.

3.9 Word Classes and Parts of Speech

Part of speech or other word class information can be stored with equivalence entries. Use the following abbreviations:

| | |
|-----------------|-----------------|
| P: Pronoun | n: noun |
| c: conjunction | p: preposition |
| i: interjection | u: unclassified |
| m: modifier | v: verb |

A part of speech abbreviation follows a semicolon (;), where the part of speech designation applies to that word and to all equivalences which follow it up to the next part of speech abbreviation on the line. For example:

```
wish;n,pie in the sky,dream;v,yearn,long,pine
```

The above is an entry for the root word “wish”. The equivalences “pie in the sky”, and “dream” are classed as nouns. The equivalences “yearn”, “long”, and “pine” are classed as verbs.

3.10 Rules of Precedence and Syntax

A root word and its equivalences are separated by commas. The comma (,) signifies addition of an item to a set.

Where an entry exists in a User Equivalence File and also in the Main Equivalence File, equivalences found for all entries of that root word are combined into one set. Example:

```
constellation,celestial heavens
```

Phrases are acceptable as roots or as equivalences, and locate matches as separated by a hyphen or any kind of white space, provided the separation is only one character long. Use spaces rather than hyphens to enter normally hyphenated words.

When “constellation” is entered as a search item on the Query Line, “celestial heavens” from the User Equivalence File will be added to the existing set, making the complete concept set:

```
constellation
celestial heavens
configuration of stars
galaxy
group of stars
nebula
star
zodiac
```

Equivs can be removed from a larger set by preceding them with a tilde (~) in the User Equivalence File. For example:

```
constellation~nebula~zodiac,big dipper
```

This entry for constellation reads “remove ‘nebula’, remove ‘zodiac’, and add ‘big dipper’”; making the complete concept set:

```
constellation
big dipper
configuration of stars
galaxy
group of stars
star
```

A whole equivalence set can be substituted for what is in the Main Equivalence File with a User Equivalence File entry which uses the equal sign (=) preceding the favored list of equivalences. For example:

```
constellation=constellation, galaxy, nebula, star
```

This entry for constellation replaces any entries in the Main Equivalence File, making the complete concept set:

```
constellation
galaxy
nebula
star
```

Don’t forget to include the root word following the equal sign (=), as the substitution is literal for the whole set, and the root word must be repeated to be included.

To permanently swap one word for another, you could make one entry only, having the effect of assignment. For example:

```
constellation=andromeda
```

Subsequent searches for “constellation” where concept searching is invoked will swap “constellation” for “andromeda”.

To permanently disable concept expansion for an item, use the equal sign (=) to replace a keyword with itself only. For example:

```
constellation=constellation
```

Any equivalences from the Main Equivalence File would be ignored, as the set is replaced by this entry.

The above rules for substitution apply where what immediately follows the equal sign (=) is alphanumeric. In the special case where the 1st character following the equal operator (=) is not alphanumeric, the entirety of what follows on the line is grabbed as a unit, rather than as a list of equivalences. Example:

```
lots=#>100
```

The root word “lots” will be replaced on the Query Line by the NPM expression which follows the equal sign “#>100”, therefore finding numeric quantities greater than 100, rather than finding English occurrences of the word “lots”.

All root and equivalence entries are case insensitive. If you need case sensitivity you must so specify with REX syntax on the Query Line. REX, NPM, XPM, and * (Wildcard) expressions cannot be entered as equivalences, as equivalences are sent directly to PPM which processes lists of English words.

The only way an English word may be linked in this way to a special expression is through the use of substitution. In this case the expression which follows an equal sign (=) will be substituted for the root word. Example:

```
bush=/\RBush
```

The root word “bush” will be replaced on the Query Line by the REX expression which follows the equal sign “/\RBush”; therefore finding only the proper name “Bush”, rather than the common noun “bush” along with any of its equivalences (“jungle”, “shrub”, “hedge”) as listed in the Main Equivalence File.

3.11 Specialized User Equivalence Files

Where you are creating your own specialized lexicon of terms, such as for medical, legal, technical, or acronym laden fields, you may be in a position to obtain your own digitized compilation of such which could be of some size.

Rather than starting from scratch, take the digitized file, flatten it to ASCII, then follow the User Equivalence rules as outlined in the previous sections to edit it into the correct format.

If you have questions on how you might speed up this activity, call Thunderstone technical support for discussion of details.

Chapter 4

Tailoring Metamorph's Linguistics

4.1 Editing Special Word Lists

There are special word lists which are called upon at various points in the Metamorph search routine and influence the way your search requests are processed, as well as which hits are deemed valid and presented for viewing. These lists can be tailored to user specification if desired.

While it is encouraged that you edit equivalences as often as you wish to reflect what you have in mind to search for, it should be clearly understood that editing equivalences that go with search words is a very different activity from editing the special word lists which govern Metamorph's approach to linguistically processing the English language.

The default content of the special lists is the result of much research on the part of the program developers, and reflects what has been locked inside Metamorph for a long time for effective use in most situations. Still, these lists are open to editing to conform to program purpose: that just about everything that is related to how Metamorph processes language is subject to modification.

The special word lists are what the program classifies as: Noise (which also includes Pronouns, Is-associations, and Question words), Prefixes, Suffixes, and Equiv-Suffixes.

For example, here is the default Pronoun list:

| | | |
|------------|-----------|------|
| anybody | ourselves | you |
| anyone | she | your |
| anything | somebody | |
| each | someone | |
| everyone | something | |
| everything | their | |
| he | them | |
| her | they | |
| him | this | |
| his | us | |
| i | we | |
| me | whatever | |

| | |
|--------|---------|
| mine | who |
| my | whoever |
| myself | whom |
| our | whose |

The content of this list is stored in ASCII format in a Metamorph profile, and can be edited there, if you have an application set up which can read a special profile. Otherwise trust the internal defaults of the program, and override them with an inline APICP directive if you need to change them, as shown under *Question Parsing*.

4.2 Question Parsing

A Metamorph query will by default be parsed for noise before the main words are sent out to the executed search. The intent is that a user can phrase his search request in a manner which reflects how he is thinking about it, without requiring him to categorize his thoughts into a syntactical procedure.

You do have the option to keep the noise words as significant words in the query. This choice can be invoked by setting the APICP flag `keepnoise` to `on`. If off, as is the Metamorph default used by Taxis, the question parser is invoked and allowed to exercise discretion in determining what is most important about your query. This selection process determines what should be treated as a search word, and therefore included in the set count and intersection quantity assigned to each significant search item.

A need to edit these lists would be most likely to come up if one were searching language from a different era or with a vastly different language style; perhaps for example the Bible. Where “thee’s” and “thou’s” replace “you” and “yours”, certain things should be filtered out from the query that might not be entered in the list. These categories can be defined as follows, where all of these categories are grouped for APICP programming purposes under “noise”.

Noise Small, common, relational words which appear frequently in a particular language and refine and fine tune specific communications, but do not majorly affect the larger concepts under discussion; e.g., about, in, on, whether.

Pronouns Words indicating person, number, and gender which are used in place of other nouns; e.g., her, his, we, them, its.

Is Words indicating state of being or existence, or used as auxiliary verbs as assistive in defining tense and use; e.g., is, being, was.

Questions Words indicating a question is being asked and requires an answer or response; e.g., what, where, when.

Syntax for setting a custom list inside a Vortex script is as follows, where what you write in would override the default internal noise list:

```
<$noiselist = "am" "are" "was" "were" "thee" "thou" "has" "hast"
    "from" "hence">
<apicp noise $noiselist>
```

4.3 Morpheme Processing

Metamorph's whole search process is built around its ability to deal with root morphemes in words, and in fact the program was originally christened "Metamorph" based on that ability. An inherent part of the search process is to take search words, strip them of suffixes and prefixes down to a recognizable morpheme, search for the morpheme, find a possible match, then go through the process again with the possible match to see if it is indeed what you were looking for.

The elements which affect morpheme processing are very subtle as they affect all language and therefore all of any text you are searching. Therefore, any changes entered into the defaults which dictate how or whether morpheme processing is done, will affect broadly the nature of your search results.

The options which affect morpheme processing are the following:

- Content of the Equiv-Suffixes List;
- Content of the Prefix List;
- Content of the Suffix List;
- Prefix processing on/off toggle;
- Suffix processing on/off toggle;
- Morpheme rebuild on/off toggle;
- Minimum word length setting.

The content of the `Equiv-Suffix` List determines which suffixes will be stripped before doing lookup for a matching entry in the Equivalence File.

The content of the `Prefix` and `Suffix` Lists determines precisely what will be stripped from a search word to obtain the morpheme which is passed to the Search Engine. Specifically this relates to all words in all equivalence sets.

To understand why the program is doing what it is doing you need to know more than just what is contained in the lists; you must also understand the sequence of the Morpheme Stripping process. If you understand these rules, you will have better judgment on how to edit the prefix and suffix lists, what happens when you turn off prefix and/or suffix processing altogether, what happens if you turn off the morpheme rebuild step, and where the minimum word length setting comes in.

4.3.1 Morpheme Stripping Routine

This routine is done by the Engine as a preliminary step before actually executing any search, using the content of the Prefix and Suffix Lists. This routine is used to get words from the Equivalence File, and only does the suffix stripping part, using the content of the Equiv-Suffix List.

1. When it is time to execute a search, the suffix and prefix lists are each sorted by descending size and ascending alphabetical order. The reason for and importance of descending order is so that suffixes

and prefixes can be stripped largest to smallest. There is no particular reason for alphabetical order except to provide a predictable ordering sequence.

2. Get the word to be checked from entered query; e.g., “antidisestablishmentarianism”.
3. Check the word’s length to see if it is greater than or equal to the length set in minimum word length. The default in Taxis is 255. A setting of 5 normally produces the best results with the default suffix list. If so, carry on. If not, there is no need to morpheme strip the word; it would just get searched for as is; e.g., “red”.
4. Check the word found against the list of suffixes to see if there is a match, starting from largest suffix on the list.
5. If a match is found strip it from the word. Note: This is why ordering by size is so important: because you want to remove suffixes (or prefixes) by the largest first, so as not to miss multiple suffixes, where one suffix may be a subset of another.
6. Continue checking against the list for the next match. Follow steps 4-5 until no more matches found. In the case of our example above, based on the default suffix list, we would be left with the following morpheme before prefix processing: “antidisestablishmentarian”. Note the following things:
 - The suffix “ism” was on the list and was stripped.
 - Neither “an”, “ian”, nor “arian” was on the suffix list, so it was not stripped.
 - The suffix “ment” is on the suffix list, but it was not left at the end of the word at any point, and therefore was not removed.
 - If “arian” and “ian” were both entered on the suffix list, “arian” would be removed first, so as not to remove “ian” and be left with “ar” at the end of the word which would not be strippable.
7. If suffix checking (only), remove any trailing vowels, or 1 of any double trailing consonants. This handles things like “strive”, which would be correctly stripped down to “striv” so that it won’t miss matches for “striving”, etc. (trailing vowel). And things like “travelling” would be stripped to “travell”; you have to strip the second ‘l’ so that you wouldn’t miss the word “travel” (trailing double consonant). Note: this is only done for suffix checking, not prefix checking.
8. Now repeat Steps 4-6 for prefix stripping against the prefix list. In our example, “antidisestablishmentarian” would get stripped down to “establishmentarian”. This is what you have left and is what goes to the pattern matcher.
9. When something is found, the pattern matcher builds it back up again to make sure it is truly a match to what you were looking for. This prevents things like taking “pressure” when you were really looking for “president”, “restive” when you were really looking for “restaurant”, and other such oddities.

4.3.2 Suffix List Editing Notes

It should be noted that the Suffix List used for Equivalence Lookup is different than the Suffix List used for matching words in the text. One should make changes in the Suffix List with great care, as it is so basic to what can be found in the text.

While caution should also be applied to editing the Equiv-Suffix List, doing so has a different effect upon the search results. The default Equiv-Suffix List is quite short by comparison, containing only 3 suffixes. This is because different forms of words tend to have different sets of concepts, as contained in the Equivalence File.

If you feel the suffix list is not broad enough, you might wish to carefully expand it. An example of a valid suffix to add might be “al”. Doing so would mean that typing in a query using the word “environmental” would pull up the equivalences listed for “environment”.

If an exact entry for your word exists, only its equivalences will be retrieved from the Equivalence File. You want “monumental” to match “colossal”, but you would probably be unhappy if it matched “tombstone” from the set associated with “monument”. If no such exact entry existed, you would have to be prepared for “monumental” to dutifully retrieve the word “obelisk” if it were linked to “monument”.

Keeping all these English nuances in mind, you do in fact have complete control over exactly what you can make the program retrieve.

4.3.3 Toggling Prefix/Suffix Processing On/Off

You can toggle prefix and suffix processing individually on and off. If you don’t like the way a particular word is being suffix or prefix processed, the easiest thing to do is simply turn off suffix or prefix processing temporarily and search for the word exactly as you want it. This is done by setting the corresponding APICP flag for `suffixproc` or `prefixproc`.

When one or both (suffix or prefix) morpheme processes are off, none of that respective routine (or routines) will be done at all. If both are off, you get a search for exactly the word you were looking for with no removal at all, and no check once found; in other words, the word you enter is passed directly from the search line to the PPM (Parallel Pattern Matcher) search algorithm.

Note that prefix processing is not supported by the Metamorph index, and thus enabling it will require a linear search (slower). This is why prefix processing is off by default in Vortex, `tsql` and the Search Appliance.

4.3.4 Toggling Morpheme Rebuild Step On/Off

You can also turn off the Morpheme Rebuild step if you want to, by setting it with the corresponding APICP flag `rebuild`. The Rebuild step, referenced in Step 9 above, is intended as a check to rebuild morphemes back up into words once found, to verify hits. If not otherwise set, the default is “on”; i.e., that this step is done.

It may happen however, that you either miss a hit you feel you should have gotten, or are getting a hit that

does not make sense to you. If so, you can try turning off the rebuild step, do your search, go into context and see exactly what was retrieved (as it will be highlighted) and why. This will at least serve to explain to you exactly what is going on in the search process.

Remember that the English language is based on more irregularities than regularities. Metamorph is designed to process the English language as best as it can, in a regular fashion. We leave it to specific applications to adjust these language toggles to one's own satisfaction, and expect that everything will not be set in exactly the same way every time.

4.3.5 Setting Minimum Word Length

Minimum word length is the approximate number of significant characters the program will deal with at a morpheme level. You increase it to obtain more exactness to the search pattern entered, and decrease it for less exactness to that pattern. The smaller the minimum word length, the slower the search will be, although the difference may be imperceptible.

Vortex scripts, `tsql` and the Search Appliance use a default minimum word length of 255, which essentially turns off morpheme stripping and allows for exact searching in locating documents. To use morpheme processing on a content oriented, English-smart application, set the APICP flag for `minwordlen` to 5. From years of experience we have established this as the best place to start, and really do not advise changing this setting arbitrarily.

As applies to the Morpheme Stripping Routine, note the following: in general (about 90% of the time) these rules are followed exactly, and the word would never be stripped smaller than the set length. But in certain cases to take into account certain overlapping rules and/or idiosyncrasies as it sees fit, it will sometimes strip down further than minimum word length; but never more than 1 character.

4.4 Warning to Linguistic Fiddlers

The program defaults and guidelines as described herein pass on to the user years of experience in what works best in terms of all these linguistic elements to accomplish in the main generally satisfactory search results. If you are not getting good results for some reason, or have allowed various situation specific choices at some points of your program, restore the defaults before continuing. This can be done by setting the APICP flag `defaults`, which restores the Vortex defaults to the next search.

4.5 APICP Metamorph Control Parameters Summary

You can modify these APICP settings by adjusting these flags to modify the Metamorph control parameters below. These are covered elsewhere in relation to their use in Taxis, Vortex, or the Metamorph API; see the appropriate manual. The settings are:

`suffixproc` : Whether to do suffix processing or not.

`prefixproc` : Whether to do suffix processing or not.

rebuild : Whether to do word rebuilding.
 incsd : Include start *w/* delimiters in hits (always on for *w/N*).
 inced : Include end *w/* delimiters in hits (always on for *w/N*).
 withinproc : Whether to respect the within (*w/*) operator.
 minwordlen : Minimum word length for morpheme processing.
 intersects : Default number of intersections (if no @).
 see : Whether to look up “see also” references.
 keepqvs : Whether to keep equivalences for words/phrases.
 keepnoise : Whether to keep noise words in the query.
 sdexp : Default start delimiter REX expression.
 edexp : Default end delimiter REX expression.
 eqprefix : Main equivalence file name.
 ueqprefix : User equivalence file name.
 suffix : Suffix list for suffix processing during search.
 suffixeq : Suffix list for suffix processing during equivalence lookup.
 prefix : Prefix list for prefix processing during search.
 noise : Noise word list.
 defaults : Restore defaults for all APICP settings.

Note that the default values for these (and other) settings may vary between Vortex, the Search Appliance, *tsql* and the Metamorph API. See the section on “Differences Between Vortex, *tsql* and Metamorph API” in the Vortex manual for a list of differences.

The easiest way to make use of these settings is in a Vortex script, e.g.:

```
<apicp suffixproc 1>
```

In *tsql*, they can be altered with a *SQL set* statement:

```
tsql "set suffixproc=1; select ..."
```

The various C APIs have their own calls, e.g. `n_setsuffixproc()`.

Some of the settings can also be changed inline with the query, for example: `horse @minwordlen=3 cat`. Such inline settings will apply to all terms after the setting.

You can also create thesaurus entries that apply specific settings, for example the thesaurus entry:

```
battery={@suffixproc=0,battery,batteries}
```

could be used to give exact word forms for just a specific word in a query. I.e. only the words `battery` and `batteries` will match for `battery`: no suffix processing for other suffixes will be done for that term, though such processing may be done for other terms. The { and } curly braces are required to enable inline settings in an equiv list. Also, the settings should be given first in the list, so that they apply to all the words in the list.

To use inline settings in a query-specified equiv list, you will need to escape the = which is special, e.g.:

```
"cell phone" ({@suffixproc\=0,battery,batteries})
```

The following settings may be specified inline:

`rebuild`

`defsufm`

`defsuffm`

`suffixproc`

`minwordlen`

`prefixproc`

Chapter 5

Types of Searches in Metamorph

5.1 Multiple Search Algorithms

The Metamorph Query Language allows for several methods of searching. You can enter a natural language question. You can specify which words, phrases, or regular expressions you wish to look for, and your search request will be processed accordingly. To accomplish all this, several different search algorithms are used which go about pattern matching in different ways. These are called up internally by Metamorph. The chief pattern matchers in use are:

SPM Metamorph's Single Pattern Matcher (includes wildcarding, the '*' operator)

PPM Metamorph's Parallel Pattern Matcher

REX Metamorph's Regular EXpression Pattern Matcher

XPM Metamorph's ApproXimate Pattern Matcher

NPM Metamorph's Numeric Pattern Matcher

When you enter the most common kind of Metamorph search, a normal English word, Metamorph calls SPM or PPM. SPM handles the morpheme processing for root words which have no equivalences; PPM handles the root words with their lists of equivalences expanded into sets of words. Where there is only a single word in a list (i.e., a root word which has no equivalences) SPM is used instead so as to optimize search speed. PPM searches for every occurrence of every valid word form for each item in a list in parallel, and will handle the multiple lists of words created from a routine query.

PPM and SPM make it possible to routinely execute such searches at tremendous speed, locating hits containing all combinations of all items from each of these lists.

Entering words in English calls PPM or SPM; this is the default and no special denotation is necessary. You can make use of a wildcard (*) operator with English words if you wish. Entering "gorb*yelt" would locate "Gorbachev had a meeting with Yeltsin". The asterisk (*) will locate up to 80 characters per asterisk noted.

REX makes it possible to look for fixed or variable length regular expressions of any kind and is integrated into the Metamorph search routine so that you can mix and match words and regular expressions. You signal REX by putting a forward slash (/) in front of the word or expression, and REX will be called by Metamorph to handle that string, utilizing all the rules of REX syntax.

XPM allows you to specify an “almost right” pattern which you are unsure of, so that you can find approximately what you have specified. XPM is also integrated into the search procedure and can be mixed in with PPM word searches and REX regular expressions; you signal XPM with a percent sign (%) denoting the percentage of proximity to the entered pattern you desire.

NPM allows you to look for numeric quantities in text which may have been expressed in English. NPM does number crunching through all possible numbers found in the text to locate those numbers which are in the specified range of desired numbers. It is generally used in combination with some other search item, such as a unit. NPM is signalled with a pound sign (#) preceding the numeric quantity you wish to match.

The heart of Metamorph’s ability to encompass so many functions and subroutines so effectively, in a way which produces quick results for the user in acceptable response time, is its exceedingly fast search algorithms. Other bodies of technology have attempted to create small replicas of a few of the functions in Metamorph, but none of this can be successful if it cannot be done fast enough to get plentiful and accurate search results.

Metamorph on its own has been benchmarked on some fast Unix machines at around 4.5 million characters per second internal throughput rate. The speed and accuracy of the pattern matching techniques employed make possible Metamorph’s versatile and flexible operation.

5.2 The Natural Language Question

The natural language question is an easy way to phrase a search query, where the APICP flag `keepnoise` has been left off. Type in a question which contains the elements you are looking for in the text. There are really only these few rules to keep in mind.

- Use only the important words in your query. Metamorph assumes every important concept in your question must be matched to qualify it as a hit.
- Pronouns are meaningless to Metamorph; rephrase them as the nouns that they actually represent.
- Idiomatic expressions should be avoided if possible, unless they are legitimately important to the sense of the question. Metamorph may or may not be aware of the true meaning of such a phrase.

An example of a “good” question to ask would be:

```
Have there been any corporate takeovers?
```

An example of a less desirable question would be:

```
What information is there on corporate takeovers?
```

The reason is that in this second question, the concept “information” must be stated in the text along with the concept “corporate” and “takeover” to be considered a match, probably excluding relevant responses; any hit from a news article is implicitly understood to be information without so stating. All you really need are the two important concepts “corporate” and “takeover”.

Another “good” question might be:

```
Were there any corporate takeovers in Germany?
```

or

```
Have there been any power struggles in the Near East?
```

Where idiomatic expressions or phrases exist as entries in the Equivalence File, they will be meaningfully processed as a whole, whether so marked or not. The question parser will check for phrase entries in an entered query. If the word grouping is not known to be a phrase, the words will be processed separately, according to their individual associations. If it is important to you that the words of such an expression or phrase be processed as one unit and Metamorph does not recognize it as such, mark it as a phrase by putting it in quotes.

Once your question is entered, the question parser will select the important words and denote them as root words, to be expanded into sets from the known associations in the Equivalence File. These words and all their equivalences are in the main passed to PPM to be searched for simultaneously while determining where answers to your question might lie.

As a rule, unless escaped with a backslash (\), hyphens are stripped from the words on the query line before expanded to include their associations in the Equivalence File and sent to the search engine. Once passed to PPM or SPM occurrences of those words as separated by either hyphens or white space will be located.

5.3 Single/Multiple Items, Keywords, Phrases, Wildcard Patterns (*)

It isn’t required that you ask a question. Any search item can be entered on the query line. The simplest search would be to enter one keyword, like “bear”. All matches containing just the word *bear* (subject to morpheme processing, if turned on) will be retrieved.

Equivalences (from the thesaurus) for query words may also be searched for, in one of two ways. First, equivalences can be turned on for *all* terms by setting *keepeqvs* (Vortex/tsql) or *Synonyms* (Search Appliance) on. Second, equivalences can be toggled (reverse the *keepeqvs* setting) for *individual* query terms with the tilde (“~”) operator.¹

For example, with default settings² (*keepeqvs* off), the query “~bear arms” will find all equivalences to the word “bear” – i.e. “cub”, “endure”, “carry” etc. – but only the single term “arms”. If we turned *keepeqvs* on, the exact same query would find only the single word “bear” (tilde now toggles *equivs* to off) but all equivalences for the word “arms”.

¹Both of these actions are subject to enabling by the *alequivs* setting in Vortex/tsql.

²In the Metamorph API (unlike Vortex, tsql and the Search Appliance) equivalences are on by default, so the following actions would be the opposite of what is described.

To look for a specific set of equivalences for a keyword – instead of equivalences derived from the thesaurus – enter them in parentheses, separated by commas (with no spaces). E.g. “(bolt, fastener, screw)” would find any of “bolt”, “fastener”, or “screw”. Note that wildcards (see below) are disabled in parenthetical lists, however morpheme processing is still done if turned on.

Entering more than one keyword on the query line will be interpreted as 2 search items, as delimited by a space character, unless it is a phrase known by the Equivalence File. To link any words together as a phrase you need only put it in quotes. For example, “Alabama Representative” must find those two words in that sequence, as a phrase. Such a phrase can be entered as a new entry in the Equivalence File, and specific names of Alabama Representatives could be associated as a set. Thereafter the quotes would not be required on the query line for it to be processed as a single search item.

A wildcard ‘*’ can be used along with an English word to extend a rooted pattern by up to 80 characters per asterisk ‘*’. For example, “Pres*Bush” would locate “President George Bush”. More than one asterisk ‘*’ may be used. Such an item which includes an asterisk is matched by a special operator which is part of SPM, the Single Pattern Matcher which looks for single items.

A wildcarded item can be searched for in intersection with other search items as well. For example: “Pres*Reagan campaign” would locate the sentence “**President Ronald Reagan won the election in November.**”

A wildcard operator ‘*’ means just that: “anything” before or after the string to which it is rooted. If you occasionally find that the morpheme processing rules for a given word are not treating it correctly, you can substitute a wildcard to locate the word in a different way. Even though “property” will also find “properties” through morpheme processing, “prop*” will find the word “properties” for different reasons. “prop*” will also find “proper” and “propane”, which morpheme processing would intelligently exclude.

5.4 Intelligent Intersection Search

The intelligent intersection search is used to locate logical intersections of equivalence sets within defined textual delimiters, a sentence being a good default to use for a context weighted application.

Where the APICP flag `keepnoise` has been turned on, Metamorph is directed to keep all words entered as part of the query whether noise or not. With noise words being retained as search words, remember not to use extraneous words or punctuation unless it is meant to be part of a designated search item. The same type of search is being used where noise is being filtered out, but applied only to the non-noise words.

Specify the words you want to look for in your query, separated by space. No punctuation or other designation is required. In this query:

```
life death disease
```

you will get hits containing intersections of occurrences from the “life”, “death”, and “disease” equivalence sets as well as the morphological constructs connected with those words.

You are signalling the program to look for intersections of each of the sets you have specified on the query line. If not otherwise marked this indicates a PPM search (or SPM search if there are no equivalences) where

Metamorph's set logic, morpheme processing, equivalence handling, and so forth is called for, for each of the words you enter on the line as delimited by spaces. Preceding a word with a tilde (~) signifies you want the root word only without equivalences, calling SPM.

You can also specify a REX expression by preceding it with a forward slash (/), or an XPM expression by preceding it with a percent sign (%), or an NPM expression by preceding it with a pound sign (#), or a phrase by putting it in quotes, or a wildcard pattern by appending or preceding a word with an asterisk (*). Each such entry, as well as words, terms or acronyms, as delimited by spaces, will be understood as a separate set for which intelligent intersections will be looked for. Logic operators plus (+) or minus (-) can be assigned to any of these search item sets.

5.5 String Search (Literal Pattern Match)

To get Metamorph to do a literal string (pattern matching) search type a slash '/' preceding the string. If you want to enter a whole line to be viewed as one string, put it in quotes, with the forward slash inside the quotes. Example:

```
"/Uncle Sam's soldiers"
```

This query will go and get each place in the textfiles being queried where that phrase is located, exactly as entered. Anything to the right of the slash, including a space before the word if you enter it so, will be considered part of the string; so don't enter a period or a space unless you want to look for one.

In the above example, "Uncle Sam's soldiers", you would get the same result whether a slash was entered or not, since there are no known equivalences for the phrase "Uncle Sam's soldiers". However, if you compare the following:

```
"/frame of mind"    (as compared to)  "frame of mind"
```

you will see that the Equivalence File has some equivalences associated with the phrase "frame of mind". To cut off those equivalences and just look for the pattern "frame of mind" you could insert the forward slash (/) as the first character of the phrase inside the quotes. (You could accomplish the same thing more efficiently by preceding the phrase with a tilde ~.)

When you denote a slash (/), remember that you're signalling the Metamorph Engine to use REX, bypassing the usual English word processing that goes on where PPM is the algorithm most often in use. REX can sometimes be more direct when such a task is all that is required.

While you can use a forward slash (/) in front of any fixed length pattern as is herein discussed, REX has many more uses which involve special characters. If such characters are part of your string and are therefore being inappropriately interpreted, use the backslash (\) to escape those characters inside your string; e.g., "43\ . 4" would indicate a literal decimal point (.), rather than its special meaning inside a REX expression.

5.6 Fixed and Variable Length Regular Expressions (REX)

5.6.1 REX Description

REX stands for Regular EXpression Pattern Matcher. REX gives you the ability to match ranges of characters, symbols, and numbers, as well as to selectively designate how many of each you wish to look for. By combining such pattern designations into what is called a “Regular Expression” you can look for such things as phone numbers, chemical formulas, social security numbers, dates, accounting amounts, names entered in more than one form, ranges of years, text formatting patterns, and so on.

As REX is also supplied with the Taxis package as a separate utility, the next chapter is devoted to a detailed account of all of REX’s syntax and features, and can be studied to learn how to designate complex regular expressions and how to use it to accomplish tasks outside a normal search environment.

While a complete understanding is not required for the casual searcher, the better you understand how to describe expressions using REX syntax, the more you will be able to make use of it, in or outside Taxis or a Metamorph search application.

REX can be used in the following ways, where the same rules of syntax apply to all:

- REX expressions can be entered as query items, following a forward slash /.
- REX expressions can be entered in a query following a w/ designation to dynamically define a special pattern to delimit your Metamorph query.
- REX expressions can be used as part of functions in a Vortex program, in the use of the functions `rex` and `sandr` (search and replace) and the start and end delimiter expressions `sdexp` and `edexp`.
- REX can be used as a stand-alone utility outside of a Taxis or Metamorph application, to do such things as change file formats with Search and Replace, search through excerpted report files to pull out specific items of interest, or to create lists of headers.

While this is not intended as a complete list, by way of example, some of the ranges of characters one can delineate with REX’s syntax follow.

| | |
|---------------------|---|
| <code>\alpha</code> | Matches any alpha character; [A-Z] or [a-z]. |
| <code>\upper</code> | Matches any upper case alpha character; [A-Z]. |
| <code>\lower</code> | Matches any lower case alpha character; [a-z]. |
| <code>\digit</code> | Matches any numeric character; [0-9]. |
| <code>\alnum</code> | Any alpha or any numeric character; [A-Z] or [a-z] or [0-9]. |
| <code>\space</code> | Any space character; [space,return,linefeed,tab,formfeed,vertical-tab]. |
| <code>\punct</code> | Any punctuation; [not control and not space and not alphanumeric]. |
| <code>\print</code> | Any printable character; [all of the above]. |
| <code>\cntrl</code> | Any control character. |
| <code>\R</code> | Respect case. |
| <code>\I</code> | Ignore case. |
| <code>\Xnn</code> | Matches hexadecimal. |

5.6.2 Examples

- `"/\alpha+==\alpha+":` Looks for one or more occurrences of a letter (i.e., any word) followed by one occurrence of a hyphen (designated by the equal sign (=), followed by one or more occurrences of a letter (i.e., a word); and as such, can be used to locate hyphenated words.
- `"/cost=\space+of=\space+living=":` Looks for the word “cost” followed by one or more of any space character (i.e., a space or a carriage return), followed by the word “of”, followed by one or more of any space character, followed by the word “living”; and as such, would locate the phrase “cost of living”, regardless of how it had been entered or formatted in terms of space characters.
- `"/\digit{1,6}\.=\digit{2}":` Looks for from 1 to 6 digits followed by a decimal point. ‘.’ is a special character in REX syntax and so must be preceded with a backward slash in order to be taken literally), followed by 2 digits; and as such would locate dollar type amounts and numbers with a decimal notation of 2 places.

5.6.3 Examples of Some Useful REX Expressions

- To locate phone numbers:

```
1?\space?(?\digit\digit\digit)?[\\-\space]?\digit{3}-\digit{4}
```

- To locate social security numbers:

```
\digit{3}-\digit{2}-\digit{4}
```

- To locate text between parentheses:

```
(=[^()]+)      <- without direction specification
or
>>([!]+)      <- with direction specification
```

- To locate paragraphs delimited by an empty line and 5 spaces:

```
>>\n\n\space{5}=!\n\n\space{5}+\n\n\space{5}
```

- To locate numbers in scientific notation; e.g., “-3.14 e -21”:

```
[+\\-]?\space?\digit+\\.?\digit*\space?e?\space?[+\\-]?\space?\digit+
```

You can formulate patterns of things to look for using these types of patterns. You can look for a REX expression by itself, or in proximity to another search item. Such a search could combine a REX expression in union with an intelligent concept search. For example, you could enter the following in a query input box:

```
"/\digit{2}%" measurement
```

The REX expression indicates 2 occurrences of any digit followed by a percent sign. “Measurement” will be treated as an English root word with its list of equivalences, and passed to PPM.

In this search, Metamorph will look for an intersection of both elements inside the specified delimiters, and may come up with a hit such as:

They **estimated** that only **65%** of the population showed up to vote.

where “estimated” was associated with “measurement”, and “65%” was associated with the pattern “\digit{2}%”.

5.7 Misspellings, Typos, and Pattern Approximations (XPM)

5.7.1 Searching for Approximations

In any search environment there is always a fine line between relevance and irrelevance. Any configuration aims to allow just enough abstraction to find what one is looking for, but not so much that unwanted hits become distracting. Speed is also an important consideration; one does not want to look for so many possibilities that the search is overly burdened and therefore too slow in response time.

If a spelling checker were run into every Metamorph search, not only would the general search time be greatly impeded, but a lot of what can be referred to as “noise” would deflect the accuracy, or relevancy of the search results. The aim of Metamorph is to allow maximum user control and direction of the search. Since there is no requirement to conform to any spelling standard, Metamorph is able to accept completely unknown words and process them accordingly: this includes slang, acronyms, code, or technical nomenclature. Even so, this does not deal with the issue of misspellings or typos.

Metamorph thoroughly handles this problem through the use of XPM, Metamorph’s ApproXimate Pattern Matcher. The intent behind XPM is that you haven’t found what you believe you should have found, and are therefore willing to accept patterns which deviate from your entered pattern by a specified percentage. The percentage entered on the query line is the percentage of proximity to the entered pattern (rather than the percent of deviation).

5.7.2 XPM Rules of Syntax

- The syntax for XPM is to enter a percent sign (%) followed by a two digit number from 01 to 99 representing percentage of proximity, followed by the pattern you wish, as: %72Raygun, to find “Reagan”. If no numbers are listed after the percent sign, the default of 80% will be used; as, %lithwania, looks for an 80% match and will find “Lithuania”.
- XPM is not a stand-alone tool; it can only be used from within a Metamorph query. It can be used in intersection with other designated words, expressions, or XPM patterns.
- You can designate a logic operator in front of an XPM pattern ‘+’ or ‘-’; do not leave a space between the operator and the percent sign; as: +%72Raygun. It does not need to be put in quotes.

- XPM is case insensitive just as are other Metamorph special searches.
- There is no way to search for an approximated REX expression; either REX or XPM will be called by Metamorph. Use XPM in front of words or otherwise fixed length patterns not inclusive of REX syntax.

Let us say you are looking for something that happened in Reykjavik last week, and you are almost certain it is in your file somewhere. There is such a reference in the demo news file which is shipped with the program, so you can try this. Perhaps you specified on the query line “event Rakechavick” but got no hit, as you were not spelling the name correctly. You can enter the same search, but call up XPM for the word you don’t know how to spell. For example:

```
event %64Rakechavick
```

This query will look for an intersection of the word “event” (plus all its equivalences) and a 64% approximation to the entered pattern “Rakechavick”. This will in fact successfully locate a hit which discusses “events” which occurred at “Reykjavik”.

When looking for this sort of thing, you can keep lowering the specified percentage until you find what you want. You’ll notice that the lower the specified proximity, the more “noise” you allow; meaning that in this case you will allow many patterns in addition to “Reykjavik”, as you are telling the program to look for anything at all which approximates 64% of the entered pattern.

Such a facility has many applications. Probably the most common use for XPM is when looking for proper nouns that have either unknown or multiple spellings. “Qadhafi” is an example of a name which is in the news often, and has several different completely accepted spellings. Someone for whom English is a second language can much more successfully search for things he cannot spell by calling up XPM when necessary with the percent (%) sign. And in instances where there are file imperfections such as human typos or OCR scanning oddities, XPM will call up the full range of possibilities, which can be very useful in catch-all batchmode searching, or otherwise.

5.8 Numeric Quantities Entered as Text (NPM)

NPM, the Numeric Pattern Matcher, is one of several pattern matchers that can be called by the user in sending out a Metamorph query. It is signified by a pound sign ‘#’ in the starting position, in the same way that the tilde ‘~’ calls SPM, a percent sign ‘%’ calls XPM, a forward slash ‘/’ calls REX, and no special character in the first position (where there are equivalences) calls PPM or SPM.

There are still many numeric patterns that are best located with a REX expression to match the range of characters desired. However, when you need the program to interpret your query as a numeric quantity, use NPM. NPM does number crunching through all possible numbers found in the text to locate those numbers which are in the specified range of desired numbers. Therefore where a lot of numeric searching is being done you may find that a math co-processor can speed up such searches.

Since all numbers in the text become items to be checked for numeric validity, one should tie down the search by specifying one or more other items along with the NPM item. For example you might enter on the query line:

```
cosmetic sales $ #>1000000
```

Such a search would locate a sentence like:

Income produced from **lipstick** brought the company **\$4,563,000** last year.

In this case “income” is located by PPM as a match to “sales”, “lipstick” is located by PPM as a match to “cosmetic”, the English character “\$” signifying “dollars” is located by SPM as a match to “\$”, and the numeric quantity represented in the text as “4, 563, 000” is located by NPM as a match to “#>1000000” (a number greater than one million). Another example:

```
cosmetic sales $ #>million
```

Even though one can locate the same sentence by entering the above query, it is strongly recommended that searches entered on the query line are entered as precise numeric quantities. The true intent of NPM is to make it possible to locate and treat as a numeric value information in text which was not entered as such.

You would find the above sentence even without specifying the string “\$”, but realize that the dollar sign (\$) in the text is not part of the numeric quantity located by NPM. There may be cases where it is important to specify both the quantity and the unit. For example, if you are looking for quantities of coal, you wouldn’t want to find coal pricing information by mistake. Compare these two searches:

```
Query1: Australia coal tons #>500
Query2: Australia coal $ #>500
```

The first would locate the sentence:

Petroleum Consolidated mined **1200 tons** of **coal** in **Australia**.

The second would locate the sentence:

From dividends paid out of the **\$3.5** million profit in the **coal** industry, they were able to afford a vacation in **Australia**.

Some units, such as kilograms, milliliters, nanoamps, and such, are understood by NPM to be their true value; that is, in the first case, 1000 grams. Use NPMP to find out which units are understood and how they will be interpreted. The carrot mark (^) shows where the parser stops understanding valid numeric quantities. Note that an abbreviation such as “kg” is not understood as a quantity, but only a unit; therefore, “5 kilograms” has a numeric quantity of 5000 (grams), where “5 kg” has a numeric quantity of 5 (kg’s).

Beware of entering something that doesn’t make sense. For example, a quantity cannot be less than 6 and greater than 10 at the same time, and therefore “#<6>10” will make the controlfile sent to the engine unable to be processed.

Do not enter ambiguity on the query line; NPM is intended to deal with ambiguity in the text, not in the query. The safest way to enter NPM searches is by specifying the accurate numeric quantity desired. Example:

```
date #>=1980<=1989
```

This query will locate lines containing a date specification and a year, where one wants only those entries from the 1980's. It would also locate dates in legal documents which are spelled out. Example:

```
retirement benefits age #>50<80
```

This query will locate references about insurance benefits which reference age 54, 63, and so on. Reflecting the truer intent of NPM, a sentence like the following could also be retrieved.

At **fifty-five** one is **awarded** the company's special **Golden Age** program.

In the event that a numeric string contains a space, it must be in quotes to be interpreted correctly. So, although it is strongly not recommended, one could enter the following:

```
revenue "#>fifty five"
```

With this, you can locate references like the following example.

Their corporate gross **income** was **\$1.4 million** before they merged with Acme Industrial.

Keep in mind that an NPM Search done within the context of Metamorph relies upon occurrences of intersections of found items inside the specified text delimiters, just as any Metamorph search. It is still not a database tool. The Engine will retrieve any hit which satisfies all search requirements including those which contain additional numeric information beyond what was called for.

In an application where Metamorph Hit Markup has been enabled, exactly what was found will be highlighted. This is the easiest way to get feedback on what was located to satisfy search requirements. If there are any questions about results, review basic program theory and compare to the other types of searches as given elsewhere in this chapter.

5.9 Designating Logic Operators (+) and (-)

Searches can be weighted by indicating those sets you "must include" with a plus sign (+) and those sets "not to include" with a minus sign (-). Those sets not so marked have the default designation of an equal sign (=), which means all such sets have an equal weight. The must include (+) and must not include (-) designations are outside the intersection quantity count; intersections are calculated based on the number of intersections of unmarked or equal (=) sets you are looking for.

In Metamorph terms we refer to an equally weighted set (=) as “set logic”; a “must include” set (+) as “and logic”; and a “must not include” set (–) as “not logic”. These definitions should not be confused with Boolean terms, as although the definitions overlap, they are not identical. Traditional “or” logic can be assigned by using the “@0” designation on the query line, denoting zero intersections of the unmarked sets.

When a (+) or (–) set is designated, remember that it applies to the whole set; not just the word you have marked. Example:

```
@1 disease blood skin +infection -bandaid
```

The above query specifying intersections at one (@1) means that you are looking for one intersection (@1) of anything from the set of words associated with “disease”, “blood”, and “skin”; and of those hits, you only want those containing something from the set of words associated with “infection”; but you would rule the hit out if it contained anything from the set of words associated with “bandaid”.

You can designate any set entered on the query line as ‘+’ or ‘–’; therefore this applies as much to wildcard (*), REX, XPM, NPM expressions, and macros, as it does to words. Example:

```
power struggle -%70Raygun
```

This finds all references to `power` and `struggle` (and their equivalences) but filters out any references to 70% approximation to the pattern “Raygun” (i.e., it would omit references to hits containing the word “Reagan”).

The important rule to remember about assigning ‘+’ or ‘–’ operators is that you cannot look for only minuses (–).

This chapter has attempted to cover the types of items which comprise a Metamorph query. Logic operators can be used to add special weighting to any of those things which will be viewed as single sets. Therefore you can assign a ‘+’ or ‘–’ to any of the types of query items that are described herein; and realize that with no other such marking, any search item is understood to be given an equal ‘=’ weighting.

Chapter 6

REX On Its Own

6.1 REX, The Regular Expression Pattern Matcher

REX is a very powerful pattern matcher which can be called internally within the Metamorph Query Language. It can also be used as an external stand-alone tool. Any pattern or expression which is outlined herein can be specified from within a Metamorph query by preceding it with a forward slash (/). Similarly, any **REX** pattern can be entered as Start or End Delimiter expression, or used otherwise in a Vortex script which calls for `rex`, there requiring no forward slash (/).

REX (for **R**egular **E**Xpression) allows you to search for any fixed or variable length regular expression, and executes more efficiently types of patterns you might normally try to look for with the Unix `Grep` family. **REX** puts all these facilities into one program, along with a Search & Replace capability, easier to learn syntax, faster execution, ability to set delimiters within which you want to search (i.e., it can search across lines), and goes beyond what is possible with other search tools.

It may be somewhat new to those who haven't previously used such tools, but you'll find that if you follow REX's syntax very literally and practice searching for simple, then more complex expressions, that it becomes quite understandable and easy to use. One thing to keep in mind is that REX looks both forwards and backwards, something quite different from other types of tools. Therefore, when you construct a REX expression, make sure it makes sense from a global view of the file; that is, whether you'd be looking forwards, or looking backwards.

A REX pattern can be constructed from a series of **F-REX'S** (pronounced "f-rex"), or Fixed length Regular EXpressions, where repetition operators after each subexpression are used to state how many of each you are looking for. Unless otherwise delineated, REX assumes you are looking for one occurrence of each subexpression stated within the expression, as specified within quotation marks.

To begin learning to use REX, first try some easy F-REX (fixed length) patterns. For example, on the query line, type in:

```
/life
```

This pattern "life" is the same as the pattern "life=", and means that you are asking the pattern matcher REX to look for one occurrence of the fixed length expression "life".

The equal sign ‘=’ is used to designate one occurrence of a fixed length pattern, and is assumed if not otherwise stated.

The plus sign ‘+’ is used to designate one or more occurrences of a fixed length pattern. If you were to search for “life+” REX would look for one or more occurrences of the word “life”: e.g., it would locate the pattern “life” and it would also locate the pattern “lifelife”.

The asterisk ‘*’ is used to designate zero or more occurrences of a fixed length pattern. If you were to try to look for “life*”, REX would be directed to look for zero or more occurrences of the word life (rather than 1 or more occurrences); therefore, while it could locate “life” or “lifelife”, it would also have to look for 0 occurrences of “life” which could be every pattern in the file. This would be an impossible and unsatisfactory search, and so is not a legal pattern to look for. The rule is that a ‘*’ search must be rooted to something else requiring one or more occurrences.

If you root “life*” to a fixed length pattern which must find at least one occurrence of something, the pattern becomes legal. Therefore, you could precede “life*” with “love=”, making the pattern “love=life*”. Now it is rooted to something which definitely can be found in the file; e.g., one occurrence of the word “love”, followed by 0 or more occurrences of the word “life”. Such an expression “love=life*” would locate “love”, “lovelife”, and “lovelifelife”.

If there is more than one subexpression within a REX pattern, any designated repetition operator will attach itself to the longest preceding fixed length pattern. Therefore a pattern preceding a plus sign ‘+’, even if it is made of more than one subexpression, will be treated as one or more occurrences of the whole preceding pattern. Use the equal sign ‘=’, if necessary, to separate these subexpressions and prevent an incorrect interpretation.

For example: if you say “lovelife*” (rather than “love=life*”) the ‘*’ operator will attach itself to the whole preceding expression, “lovelife”, and will therefore be translated to mean 0 or more occurrences of the entire pattern “lovelife”, making it an illegal expression. On the other hand, in the expression “love=life*”, REX will correctly look for 1 occurrence of “love”, followed by 0 or more occurrences of “life”.

Use the “-x” option in REX (from the Windows or Unix command line) to get feedback on how REX translates the syntax you have entered to what it is really looking for. In this way you can debug your use of syntax and learn to use REX to its maximum power. Using our same example, if you enter on the Windows or Unix command line:

```
rex -x ``love=life*``
```

you will get the following output back to show you how REX will interpret your search request:

```
1 occurrence(s) of : [Ll][Oo][Vv][Ee]
followed by from 0 to 32000 occurrences of : [Ll][Ii][Ff][Ee]
```

The brackets above, as: “[Ll]”, mean either of the characters shown inside the brackets, in that position; i.e., a capital or small ‘l’ in the 1st character position, a capital or small ‘o’ in the 2nd character position, and so on.

You can find REX syntax for all matters discussed above and delineated below, by typing in “REX” followed by a carriage return on the Windows or Unix command line.

6.2 REX Program Syntax and Options

REX locates and prints lines containing occurrences of regular expressions. Beginning and end of line are the default beginning and ending delimiters. If starting (`-s`) and ending (`-e`) delimiters are specified REX will print from the beginning delimiting expression to the ending delimiting expression. If so specified (`-p`) REX will print the expression only. If files are not specified, standard input is used. If files are specified, the filename will be printed before the line containing the expression, unless the "`-n`" option (don't print the filename) is used.

SYNTAX: `rex [options] expression [files]`

Where: options are preceded by a hyphen (`-`)
 expressions may be contained in quotes (`" "`)
 filename(s) comes last and can contain wildcards

OPTIONS:

| | |
|----------------------|--|
| <code>-c</code> | Don't print control characters, replace with space. |
| <code>-C</code> | Count the number of times the expression occurs. |
| <code>-l</code> | List file names (only) that contain the expression. |
| <code>-E"EX"</code> | Specify and print the ending delimiting expression. |
| <code>-e"EX"</code> | Specify the ending delimiting expression. |
| <code>-S"EX"</code> | Specify and print the starting delimiting expression. |
| <code>-s"EX"</code> | Specify the starting delimiting expression. |
| <code>-p</code> | Begin printing at the start of the expression. |
| <code>-P</code> | Stop printing at the end of the expression. |
| <code>-r"STR"</code> | Replace the expression with "STR" to standard output. |
| <code>-R"STR"</code> | Replace the expression with "STR" to original file. |
| <code>-t"Fn"</code> | Use "Fn" as the temporary file. (default: "rextmp") |
| <code>-f"Fn"</code> | Read the expression(s) from the file "Fn". |
| <code>-n</code> | Do not print the file name. |
| <code>-O</code> | Generate "FNAME@OFFSET,LEN" entries for mm3 subfile list. |
| <code>-x</code> | Translate the expression into pseudo-code (debug). |
| <code>-v</code> | Print lines (or delimiters) not containing the expression. |

- Each option must be placed individually on the command line.
- "EX" is a REX expression.
- "Fn" is file name.
- "STR" is a replacement string.
- "N" is a drive name e.g.: A, B, C ...

6.3 REX Expression Syntax

- Expressions are composed of characters and operators. Operators are characters with special meaning to REX. The following characters have special meaning: “\=+*?{ }, [] ^ \$. - !” and must be escaped with a ‘\’ if they are meant to be taken literally. The string “>>” is also special and if it is to be matched, it should be written “\>>”. Not all of these characters are special all the time; if an entire string is to be escaped so it will be interpreted literally, only the characters “\=?+*{ [^ \$. !>” need be escaped.
- A ‘\’ followed by an ‘R’ or an ‘I’ mean to begin respecting or ignoring alphabetic case distinction. (Ignoring case is the default.) These switches *do not* apply inside range brackets.
- A ‘\’ followed by an ‘L’ indicates that the characters following are to be taken literally up to the next ‘\L’. The purpose of this operation is to remove the special meanings from characters.
- A subexpression following ‘\F’ (followed by) or ‘\P’ (preceded by) can be used to root the rest of an expression to which it is tied. It means to look for the rest of the expression “as long as followed by ...” or “as long as preceded by ...” the subexpression following the \F or \P, but the designated subexpression will be considered excluded from the located expression itself.
- A ‘\’ followed by one of the following ‘C’ language character classes matches that character class: alpha, upper, lower, digit, xdigit, alnum, space, punct, print, graph, cntrl, ascii.
- A ‘\’ followed by one of the following special characters will assume the following meaning: n=newline, t=tab, v=vertical tab, b=backspace, r=carriage return, f=form feed, 0=the null character.
- A ‘\’ followed by Xn or Xnn where n is a hexadecimal digit will match that character.
- A ‘\’ followed by any single character (not one of the above) matches that character. Escaping a character that is not a special escape is not recommended, as the expression could change meaning if the character becomes an escape in a future release.
- The character ‘^’ placed anywhere in an expression (except after a ‘[’) matches the beginning of a line. (same as: \x0A in Unix or \x0D\x0A in Windows)
- The character ‘\$’ placed anywhere in an expression matches the end of a line. (\x0A in Unix, \x0D\x0A in Windows)
- The character ‘.’ matches any character.
- A single character not having special meaning matches that character.
- A string enclosed in brackets [] is a set, and matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in [a-z] or [0-9]. A ‘^’ occurring as the first character of the set will invert the meaning of the set. A literal ‘-’ must be preceded by a ‘\’. The case of alphabetic characters is always respected within brackets.

A double-dash (“--”) may be used inside a bracketed set to subtract characters from the set; e.g. “[\alpha--x]” for all alphabetic characters except “x”. The left-hand side of a set subtraction

must be a range, character class, or another set subtraction. The right-hand side of a set subtraction must be a range, character class, or a single character. Set subtraction groups left-to-right. The range operator “-” has precedence over set subtraction. Set subtraction was added in version 6.

- The ‘>>’ operator in the first position of a fixed expression will force REX to use that expression as the “root” expression off which the other fixed expressions are matched. This operator overrides one of the optimizers in REX. This operator can be quite handy if you are trying to match an expression with a ‘!’ operator or if you are matching an item that is surrounded by other items. For example: “x+>>y+z+” would force REX to find the “y’ s” first then go backwards and forwards for the leading “x’ s” and trailing “z’ s”.
- The ‘!’ character in the first position of an expression means that it is *not* to match the following fixed expression. For example: “start=!finish+” would match the word “start” and anything past it up to (but not including the word “finish”. Usually operations involving the “!” operator involve knowing what direction the pattern is being matched in. In these cases the ‘>>’ operator comes in handy. If the ‘>>’ operator is used, it comes before the ‘!’. For example: “>>start=!finish+finish” would match anything that began with “start” and ended with “finish”. *The “!” operator cannot be used by itself* in an expression, or as the root expression in a compound expression. NOTE: This “!” operator “nots” the whole expression rather than its sequence of characters, as in earlier versions of REX.

Note that “!” expressions match a character at a time, so their repetition operators count characters, not expression-lengths as with normal expressions. E.g. “!finish{2,4}” matches 2 to 4 characters, whereas “finish{2,4}” matches 2 to 4 times the length of “finish”.

6.4 REX Repetition Operators

- A regular expression may be followed by a repetition operator in order to indicate the number of times it may be repeated.

NOTE: Under Windows the operation “{X,Y}” has the syntax “{X-Y}” because Windows will not accept the comma on a command line. Also, N occurrences of an expression implies infinite repetitions but in this program N represents the quantity 32768 which should be a more than adequate substitute in real world text.

- An expression followed by the operator “{X,Y}” indicates that from X to Y occurrences of the expression are to be located. This notation may take on several forms: “{X}” means X occurrences of the expression, “{X,}” means from X to N occurrences of the expression, and “{,Y}” means from 0 (no occurrences) to Y occurrences of the expression.
- The ‘?’ operator is a synonym for the operation “{0,1}”. Read as: “Zero or one occurrence.”
- The ‘*’ operator is a synonym for the operation “{0,}”. Read as: “Zero or more occurrences.”
- The ‘+’ operator is a synonym for the operation “{1,}”. Read as: “One or more occurrences.”
- The ‘=’ operator is a synonym for the operation “{1}”. Read as: “One occurrence.”

6.5 RE2 Syntax

In Taxis version 7.06 and later, on some platforms the search expression may be given in RE2 syntax instead of REX. RE2 is a Perl-compatible regular expression library whose syntax may be more familiar to Unix users than Taxis' REX syntax. An RE2 expression in REX is indicated by prefixing the expression with “\<re2\>”. E.g. “\<re2\>\w+” would search for one or more word characters, as “\w” means word character in RE2, but not REX.

REX syntax can also be indicated in an expression by prefixing it with “\<rex\>”. Since the default syntax is already REX, this flag is not normally needed; it is primarily useful in circumstances where the syntax has already been changed to RE2, but outside of the expression – e.g. a Vortex <rex> statement with the option `syntax=re2`.

Note that RE2 syntax is not supported on all `taxis -platform` platforms; where it is unsupported, attempting to invoke an RE2 expression will result in the error message “REX: RE2 not supported on this platform”. (Windows, most Linux 2.6 versions except `i686-unknown-linux2.6.17-64-32` are supported.)

6.6 REX Replacement Strings

NOTE: Any expression may be searched for in conjunction with a replace operation but the replacement string will always be of fixed size.

An expression will be defined one way when given to REX to search for; it will be defined another way when used with the ‘-r’ or ‘-R’ `Replace` options as the replacement string. The characters “?#{ }+” have entirely different meanings when used as part of the syntax of a replacement string, than when used as part of the syntax of expressions or repetition operators.

Replacement syntax is different from expression syntax, and is limited to the following delineated rules.

- The characters “?#{ }+\” are special. To use them literally, precede them with the escapement character “\”.
- Replacement strings may just be a literal string or they may include the “ditto” character “?”. The ditto character will copy the character in the position specified in the replace string from the same position in the located expression.
- A decimal digit placed within curly braces (e.g. {5}) will place that character of the located expression to the output.
- A “\” followed by a decimal number will place that subexpression to the output. Subexpressions are numbered starting at 1.
- The sequence “\&” will place the entire expression match (sans \P and \F portions) to the output. This escape was added in Taxis version 7.06.
- A plus character “+” will place an incrementing decimal number to the output. One purpose of this operator is to number lines.

- A “#” in the replace string will cause the character in that position to be printed in hexadecimal form.
- Any character in the replace string may be represented by the hexadecimal value of that character using the following syntax: \Xdd where “dd” is the hexadecimal value.

6.7 REX Caveats and Commentary

REX is a highly optimized pattern recognition tool that has been modeled after the UNIX family of tools: **GREP**, **EGREP**, **FGREP**, and **LEX**. Wherever possible REX’s syntax has been held consistent with these tools, but there are several major departures that may bite those who are used to using the **GREP** family.

REX uses a combination of techniques that allow it to surpass the speed of anything similar to it by a very wide margin.

The technique that provides the largest advantage is called “state anticipation or state skipping” which works as follows . . .

If we were looking for the pattern:

ABCDE

in the text:

AAAAABCDEAAAAAA

a normal pattern matcher would do the following:

```

ABCDE
  ABCDE
    ABCDE
      ABCDE
        ABCDE
          ABCDE
AAAAABCDEAAAAAA

```

The state anticipation scheme would do the following:

```

ABCDE
      ABCDE
AAAAABCDEAAAAAA

```

The normal algorithm moves one character at a time through the text, comparing the leading character of the pattern to the current text character of text, and if they match, it compares the leading pattern character +1 to the current text character +1, and so on . . .

The state anticipation pattern matcher is aware of the length of the pattern to be matched, and compares the last character of the pattern to the corresponding text character. If the two are not equal, it moves over by an amount that would allow it to match the next potential hit.

If one were to count the number of comparison cycles for each pattern matching scheme using the example above, the normal pattern matcher would have to perform 13 compare operations before locating the first occurrence; versus 6 compare operations for the state anticipation pattern matcher.

One concept to grasp here is: *“The longer the pattern to be found, the faster the state anticipation pattern matcher will be;”* while a normal pattern matcher will slow down as the pattern gets longer.

Herein lies the first major syntax departure: REX always applies repetition operators to the longest preceding expression. It does this so that it can maximize the benefits of using the state skipping pattern matcher.

If you were to give GREP the expression: `ab*de+`

It would interpret it as: an ‘a’ then 0 or more ‘b’ s’ then a ‘d’ then 1 or more ‘e’ s’.

REX will interpret this as: 0 or more occurrences of ‘ab’ followed by 1 or more occurrences of ‘de’.

The second technique that provides REX with a speed advantage is ability to locate patterns both forwards and backwards indiscriminately.

Given the expression: `“abc*def”`, the pattern matcher is looking for “Zero to N occurrences of ‘abc’ followed by a ‘def’”.

The following text examples would be matched by this expression:

```

abcbabcbabcbdef
def
abcbdef

```

But consider these patterns if they were embedded within a body of text:

```

My country 'tis of abcbabcbabcbdef sweet land of def, abcbdef.

```

A normal pattern matching scheme would begin looking for ‘abc*’. Since ‘abc*’ is matched by every position within the text, the normal pattern matcher would plod along checking for ‘abc*’ and then whether it’s there or not it would try to match ‘def’. REX examines the expression in search of the most efficient fixed length subpattern and uses it as the root of search rather than the first subexpression. So, in the example above, REX would not begin searching for ‘abc*’ until it has located a ‘def’.

There are many other techniques used in REX to improve the rate at which it searches for patterns, but these should have no effect on the way in which you specify an expression.

The three rules that will cause the most problems to experienced GREP users are:

1. Repetition operators are always applied to the longest expression.
2. There must be at least one subexpression that has one or more repetitions.
3. No matched subexpression will be located as part of another.

Rule 1 Example : `abc=def*` Means one ‘abc’ followed by 0 or more ‘def’ s’.

Rule 2 Example : `abc*def*` CANNOT be located because it matches every position within the text.

Rule 3 Example : `a+ab` Is idiosyncratic because ‘a+’ is a subpart of ‘ab’.

6.8 Other REX Notes

- When using REX external to Metamorph in Windows or Unix, it can occasionally occur that a very complicated expression requires more memory than is available. In the event of an error message indicating not enough memory, you can try the same expression using “REXL”, another version of REX compiled for large model. As “REXL” is slightly slower than “REX” it would not be recommended for use in most circumstances.
- The beginning of line (^) and end of line (\$) notation expressions for Windows are both identified as a 2 character notation; i.e., REX under Windows matches `\x0d\x0a` (carriage return [cr], line feed [lf]) as beginning and end of line, rather than `\x0a` as beginning, and `\x0d` as end.

Part III

Metamorph API Tools

Chapter 1

Metamorph Application Program Interface Overview

“Build the power of Metamorph into your application”

Metamorph is the only REAL concept based text retrieval product on the market today. In its stand-alone form it acts as a research assistant that provides its user with an easy query interface while under the hood it is performing some of the most complex text search techniques in the field.

What is a Metamorph Search?

Metamorph searches for some combination of “lexical sets” within the bounds of two lexical delimiters. The idea of using set logic is very important to the way the software works.

When people communicate to each other they rarely use exactly the same vocabulary when trying to communicate a common idea. Typically, concepts are communicated by stringing combinations of abstract meanings together to form a concise idea. For example, if you were trying to communicate the idea of a “nice person” to someone else you might use any of the following forms:

- nice guy
- pleasing chap
- agreeable character
- excellent human being
- exceptionally fine person

While there are subtle differences in each of these phrases, the underlying concept is the same. It is also worth noting that by themselves the individual words in each phrase carry little indication of the whole idea. In a much larger sense, people string these types of concepts to form heuristically larger and more complex communications. One ordering of heuristic classifications might be as follows:

- morpheme (a small token that can be built into a word)

- word
- phrase
- clause
- sentence
- paragraph
- chapter
- book
- collection

If you are searching for a concept within a body of text, you are actually searching for an intersection in meaning of your idea of what you are searching for with/and the body of information you are searching. Metamorph performs this search operation for you automatically.

Within Metamorph if you perform the query: *Are there power struggles in the Near East?*

Metamorph will find the individual “important” terms within your query. Then, it will look these terms up in a thesaurus that contains over 250,000 associations and it will expand each term to the set of things that mean approximately the same thing:

power : ability, jurisdiction, regency, sovereignty, ascendancy, justice, restraint, sway, authority, kingship, scepter, electrify, carte blanche, leadership, skill, clutches, majesty, strength, command, mastership, suction, control, mastery, superiority, domination, militarism, supremacy, dominion, monarchy, vigor, efficiency, nuclear, fission, weight, electricity, omnipotence, acquisition, energy, persuasiveness, capability, force, potency, faculty, hegemony, predominance, function, imperialism, preponderance, might, influence, pressure, reign,

struggle : battle, contest, combat, flounder, competition, strive, conflict, effort, exertion, experience, fight, scuffle, strife, attempt, cash, endeavor, flight, oppose, agonize, compete

Near East : Israel, Jordan, Lebanon, Saudi Arabia, Syria, Turkey, Kuwait, Iran, Iraq

After it has built these sets, it will search through the text you have designated for a place in the text that has all three of the concepts present within some defined boundary (i.e.; sentence, line, paragraph, page, chapter, etc.). So, if it was instructed to search by sentence it would be able to retrieve the following:

Iraq’s Saddam Hussein is being pressured by the U.N. to suspend his endeavors to annex Kuwait.

Please Note that Metamorph recognizes that “Near East” is a phrase that means the countries in the Near East and not the concepts of “near” and “east” individually. Also, it is not only looking for each of the words in the lists, but it is also looking for every word-form of the words in each of the lists.

1.1 Technical Description

Within Metamorph a “set” can be any one of four different types of text data:

- The set of words or phrases that mean the same thing.
- The set of text patterns that match a regular-expression.
- The set of text patterns that are approximately the same.
- The set of quantities that are within some range.

There are three types of operations that can be used in conjunction with any set:

| | |
|-------------|----------------------------------|
| INCLUSION | The set must be present. |
| EXCLUSION | The set must not be present. |
| PERMUTATION | X out of Y sets must be present. |

The set logic operations are performed within two boundaries:

- A starting delimiter (e.g.; the beginning of a sentence).
- An ending delimiter (e.g.; the end of a sentence).

Each type of set plays an important role in the real-world use of a text retrieval tool:

- The word-list pattern matcher can locate any word form of an entire list of English words and/or phrases.
- The regular-expression pattern matcher allows the user to search for things like dates, part numbers, social security numbers, and product codes.
- The approximate pattern matcher can search for things like misspellings, typos, and names or addresses that are similar.
- The numeric/quantity pattern matcher can look for numeric values that are present in the text in almost any form and allows the user to search for them generically by their value.

The Metamorph search engine will always optimize the search operations performed so that it will minimize the amount of CPU utilization and maximize the throughput search rate. At the heart of the Metamorph search engine lie seven of the most efficient pattern matchers there are for locating items within text. With the exception of the Approximate Pattern Matcher, all of these pattern matchers use a proprietary algorithmic technique that is guaranteed to out-perform any other published pattern matching algorithm (including those described by Boyer-Moore-Gosper and Knuth-Pratt-Morris).

Providing the user with set-logic to manipulate combinations of these set-types gives them the ability to search for just about anything that they might want to find in their textual information. The query tool in general can be as simple or sophisticated as the user wishes, with the simplest query being a simple natural-language question.

1.2 Some Search Examples and Explanations

Example 1:

Let's say that we want to search for any occurrence of An Intel 80X86 processor on the same line with the concept of "speed" or "benchmark" as long as the string "Motorola" is not present.

The query is: `+ /80=[1-4]?86 -/motorola speed benchmark`

Explanation:

A leading '+' means "this must be present".

A leading '-' means "this must not be present".

The '/' signals the use of a regular-expression.

' /80=[1-4]?86' will locate an '80' followed by an optional ('1' or '2' or '3' or '4') followed by an '86'. This will locate: 8086, 80186, 80286, 80386 or 80486.

' /motorola' will locate 'MOTOROLA' or 'Motorola' or 'motorola' (or any other combination of alphabetic cases).

' speed' will locate any word that means "speed".

' benchmark' will locate any word that means "benchmark".

The beginning and ending delimiting expressions would be defined as '\n' (meaning a new-line character).

The Metamorph search engine will now optimize this search and will perform the following actions:

- A:** Search for any pattern that matches ' /80=[1-4]?86'. When it is located do item (B).
- B:** Search backwards for the start delimiter '\n' (or begin of file/record whichever comes first).
- C:** Search forwards for the ending delimiter '\n' (or end of file/record whichever comes first).
- D:** Search for the pattern ' /motorola' between the start and end delimiters. If it is *not* located do item (E), otherwise go to item (A).
- E:** Search for the set of words that mean "benchmark". If a member is located do item (G), otherwise, do item (F).
- F:** Search for the set of words that mean "speed". If a member is located do item (G), otherwise, go to item (A).
- G:** Inform the user that a hit has been located.

Example 2:

Let's say we are searching an address and phone number list trying to find an entry for a person whose name has been apparently entered incorrectly.

The query: `"%60 Jane Plaxton" "%60 234 rhoads dr." /OH /49004`

Because our database is large, we want to enter as much as possible about what we know about Ms. Plaxton so that we decrease the number of erroneous hits. The actual address in our database looks as follows:

```
Jane Plxaton
243 Roads Dr.
Middle Town OH 49004
```

This is a little exaggerated for reasons of clarity, but what has happened is that the data-entry operator has transposed the 'x' and the 'a' in 'Plaxton' as well as the '4' and '3' and has also misspelled 'Rhodes'.

The query we performed has four sets:

```
A 60% approximation of:    "Jane Plaxton"
A 60% approximation of:    "234 rhoads dr."
The state string :          OH
The zip code string :       49004
```

The database records are separated by a blank line, therefore our start and end delimiters will be '`\n\n`' (two new-line characters).

The Approximate pattern matcher will be looking for the name and street address information and will match anything that comes within 60matcher will default to 80regular-expression pattern matcher will be looking for the state and zip-code strings. We are searching for three intersections of the four sets (this is the default action).

Example 3:

We are reading the electronic version of the Wall Street Journal and we are interested in locating any occurrence of profits and/or losses that amount to more than a million dollars.

The query: `+#>1,000,000 +dollar @0 profit loss gain`

The '+' symbol in front of the first two terms indicates that they must be present in the hit. The '@0' tells Metamorph to find zero intersections of the following sets. Put another way, only one of the remaining sets needs to be located.

The sets:

- Mandatory (because of the '+' symbol):
 - Any quantity in the text that is greater than one million.
 - Any word (or string) that means "dollar".
- Permutation (because of the '@0'):
 - Anything that means "profit".
 - Anything that means "loss".
 - Anything that means "gain".

We would probably define the delimiters to be either a sentence or a paragraph.

The following would qualify as hits to this query:

- Congress has spent 2.5 billion dollars on the stealth bomber.
- Lockheed Corp. has taken a four million dollar contract from Boeing.
- The Lottery income from John Q. Public last week was One Million Two Hundred and Fifty Thousand dollars and twenty five cents.

1.3 Potential Applications

The nature of our API makes it possible to use Metamorph as a generic text searching tool no matter where the text resides. Given the quantity of text that exists on most computers the potential variants are boundless, but here are some ideas:

- A method of searching the text field information in databases (relational or otherwise).
- Document management and control systems.
- Document/record classification systems.
- Real time text analysis.
- E-Mail services.
- Image classification/retrieval databases.
- Message traffic management.
- Educational/instructional aids.
- Executive information systems.
- Research analysis tools.

Chapter 2

The Programmers' Interface Overview

There are thousands of stand-alone Metamorph programs in the field today, and over time we have received many requests by application developers who would like to be able to embed our searching technology inside their particular application. It has taken us a long time to figure out how to provide a simple and clean method to provide a solution to their problems. We have tried to make it as easy as possible while providing the maximum power and flexibility.

All of the code that comprises Metamorph has been written in ANSI compliant 'C' Language. The source code to the API (only) is provided to the programmer for reference and modification. Metamorph has currently been compiled and tested on 22 different UNIX platforms, MS-DOS, and IBM MVS. The API can be ported by Thunderstone to almost any Machine/OS that has an ANSI compliant 'C' compiler.

The set of calls in the API are structured in a fashion similar to `fopen()`, `fclose()`, `ftell()`, and `gets()`, standard library functions. And just like you can have multiple files open at the same time, you can open as many simultaneous Metamorph queries as needed. (One reason you might do this is to have a different search in effect for two different fields of the same record.)

The API itself allows the software engineer to conduct a Metamorph search through any buffer or file that might contain text. There are two data structures that are directly involved with the API:

```
APICP      /* this structure contains all the control parameters */
MMAPI      /* this structure is passed around to the API calls */
```

The APICP structure contains all the default parameters required by the API. It is separate from the MMAPI structure so that its contents can be easily manipulated by the developer. An APICP contains the following information:

- A flag telling Metamorph to do suffix processing
- A flag telling it do prefix processing
- A flag that says whether or not to perform word derivations
- The minimum size a word may be processed down to

- The list of suffixes to use in suffix processing
- The list of prefixes to use in prefix processing
- A start delimiter expression
- An end delimiter expression
- A flag indicating to include the starting delimiter in the hit
- A flag indicating to include the ending delimiter in the hit
- A list of high frequency words/phrases to ignore
- The default names of the Thesaurus files
- Two optional, user-written, Thesaurus list editing functions
- The list of suffixes to use in equivs lookup
- A flag indicating to look for the within operator (w/)
- A flag indicating to lookup see references
- A flag indicating to keep equivalences
- A flag indicating to keep noise words
- A user data pointer

Usually the developer will have no need to modify the contents of this structure more than one time to tailor it to their application, but in some applications it will be very desirable to be able to modify its contents dynamically. Two calls are provided that handle the manipulation of this structure:

```
APICP * openapicp(void)           /* returns an APICP pointer */
APICP * closeapicp(APICP *cp)    /* always returns a NULL pointer */
```

The `openapicp()` function creates a structure that contains a set of default parameters and then returns a pointer to it. The `closeapicp()` function cleans up and releases the memory allocated by the `openapicp()` function. Between these two calls the application developer may modify any of the contents of the APICP structure.

There are five function calls that are associated with the actual API retrieval function; they are as follows:

```
MMAPI *openmmapi(char *query, APICP *cp)
int    setmmapi(MMAPI *mm, char *query)
char *getmmapi(MMAPI *mm, char *buf, char *endofbuf, int operation)
```



```
int    infommapi(MMAPI *mm, int index, char **what, char **where,
               int *size)

MMAPI *closemmapi(MMAPI *mm)
```

The `openmmapi()` function takes the set of default parameters from the `APICP` structure and builds an `MMAPI` structure that is ready to be manipulated by the other four functions. It returns a pointer to this structure.

The `setmmapi()` function is passed a standard Metamorph query (see examples) and does all the processing required to get the API ready to perform a search that will match the query. If the application program wishes to, it can define a function that will be called by the `setmmapi()` function to perform editing of the word lists and query items before the initialization is completed (this is not required).

The `getmmapi()` function performs the actual search of the data. All that is required is to pass the `getmmapi()` function the beginning and ending locations of the data to be searched. There are two operations that may be performed with the `getmmapi()` call; `SEARCHNEWBUF` and `CONTINUESEARCH`. Because there may be multiple hits within a single buffer, the `search-new-buf` command tells the API to locate the first hit, and then by using successive calls with the command `continue-search` you will locate all the remaining hits in the buffer.

The `infommapi()` function returns information about a hit to the caller; it will give the following information:

- Where the hit is located within the buffer.
- The overall length of the hit.
- For each set in the search that was matched:
 1. The query set searched for and located.
 2. The location of the set item.
 3. The length of the set item.
- The location and length of the start and end delimiters.

The `closemmapi()` function cleans up and releases the memory allocated by the `openmmapi()` call.

The last of the important calls in the API is the function that reads data in from files. While your application may not require this function, if files are being read in as text streams the use of this function is mandated.

```
int rdmmapi(char *buf, int n, FILE *fh, MMAPI *mm)
```

This function works very much like `fread()` with one important exception; it guarantees that a hit will not be broken across a buffer boundary. The way it works is as follows:

- A normal `fread()` for the number of requested bytes is performed.

- `rdmmapi()` searches backwards from the end of the buffer for an occurrence of the ending delimiter regular-expression.
- The data that is beyond the last occurrence of an ending delimiter is pushed back into the input stream. (The method that is used depends on whether an `fseek()` can be performed or not.)

2.1 The Metamorph 3 API Package

The Metamorph 3 API package consists of the following files:

```
api3.doc      - this documentation
lapi3.lib     - MS-DOS, for Microsoft 'C' large model
               library containing all api functions
libapi3.a     - Unix library containing all api functions
api3.h        - header to be included by any program using Metamorph 3 api
api3i.h       - header automatically included by api3.h
mmsg.h        - header automatically included by api3.h
api3.c        - source code to the top level api calls
apimsg.c      - source code to the default message handler
mmex1.c       - example source implementing a text search interface
mmex2.c       - example source implementing a database search interface
mmex2.dat     - example database for mmex2.c
readme.doc    - system specific and installation notes
```

The Metamorph 3 API uses bytes and strings of bytes for most of its character manipulations. “Byte” is defined, in the API header, as an unsigned 8 bit quantity (unsigned char) and is used to allow greater latitude in string contents.

All byte pointers are normal 'C' strings (pointer to an array of bytes, terminated by '\0').

All byte pointer lists are arrays of pointers to normal 'C' strings. Each list is terminated with an empty string `((byte *) "")`.

WARNING: All APICP strings, string list members, and pointer arrays will be freed by `closeapicp` if they are `!=NULL`. This includes the terminator `("")` in string lists.

The Metamorph API provides the following functions:

```
closeapicp() - control parameters interface
closemmapi() - cleanup
closemmsg()  - close the message file
fixmmsgfh()  - message control
getmmapi()   - search routine
informmapi() - hit information
openapicp()  - control parameters interface
openmmapi()  - initialization
```

putmsg() - message handler
rdmmapi() - synchronized read
setmmapi() - reinitialization

The minimum set of function calls you will use is:

closeapicp()
closemmapi()
getmmapi()
openapicp()
openmmapi()

The Metamorph 3 API needs 3K of stack space in addition to whatever the calling program uses.

Chapter 3

Metamorph 3 API functions

3.0.1 openapicp, closeapicp - Metamorph API control parameters interface

SYNOPSIS

```
#include <stdio.h>
#include "api3.h"

APICP * openapicp(void)

APICP * closeapicp(cp)
APICP * cp;
```

DESCRIPTION

Openapicp returns a pointer to a structure that contains all of the default parameters needed by the Metamorph API. Each of the members of the structure are initialized in a manner that will allow for simple modification of its contents by the calling program. Closeapicp frees all the memory allocated by openapicp and returns an APICP *) NULL.

The following describes how to modify each of the variable types within the APICP structure:

```
(byte)      : Direct assignment
              eg: cp->suffixproc=(byte)1;

(int)       : Direct assignment
              eg: cp->minwordlen=2;

(byte *)    : Free original pointer and assign new allocated pointer
              eg: free(cp->sdexp);
                  cp->sdexp=(byte *)malloc(strlen("string")+1);
                  strcpy(cp->sdexp,"string");

(byte **)   : Free original pointers and assign new allocated pointers
              eg: #define MYLISTSZ 3
                  static char *mylist[MYLISTSZ]={"new","list",""};
                  int i;
                  for(i=0;*cp->noise[i]!='\0';i++)
                      free(cp->noise[i]);
                  free(cp->noise[i]); /* free empty string at end */
                  free(cp->noise); /* free the array pointer */
                  cp->noise=(byte **)calloc(MYLISTSZ,sizeof(byte *));
                  for(i=0;i<MYLISTSZ;i++)
```

```

        {
            cp->noise[i]=(byte *)malloc(strlen(mylist[i])+1);
            strcpy(cp->noise[i],mylist[i]);
        }

int  (*)() : Direct assignment
        eg:  cp->eqedit=myeditfunction;

```

WARNING: The `closeapicp()` will free all variable pointers. Do not assign static data pointers or attempt to free any pointers placed in the `APICP` structure.

3.1 APICP Variable Definitions

The following fields are defined in the `APICP` structure:

- `suffixproc`
Do suffix stripping processing
- `prefixproc`
Do prefix stripping processing
- `rebuild`
Perform the morpheme rebuild check
- `incsd`
Include the start delimiter in the hit
- `inced`
Include the end delimiter in the hit
- `withinproc`
Look for within operator (`w/ . .`)
- `suffixrev`
Internal Thunderstone use: Strings in suffix list are reversed
- `minwordlen`
Minimum remaining length of a pre/suffix stripped word
- `intersects`
Number of intersections to be located in the hit
- `sdexp`
The start delimiter expression
- `edexp`
The end delimiter expression

- `query`
Query from user
- `set`
Array of sets of things being searched for, in equiv format; sets are in original query order
- `suffix`
The list of suffixes
- `suffixeq`
The list of suffixes for equivalence lookup
- `prefix`
The list of prefixes
- `noise`
The list of words that constitute “noise”
- `eqprefix`
The Path-filename of the main equiv file
- `ueqprefix`
The Path-filename of the user equiv file
- `see`
Lookup “see also” references
- `keepeqvs`
Keep equivalences
- `keepnoise`
Keep noise words
- `eqedit`
A user programmable equiv edit function
- `eqedit2`
A user programmable equiv edit function A user settable data pointer
- `denymode`
API3DENY... mode: how to deny query-protection-forbidden actions
- `al...`
Flags for allowing/denying query-protection actions
- `qmin..., qmax...`
Query-protection limits
- `defsuffrm`
Whether to remove a trailing vowel, or one of a trailing double consonant pair, after normal suffix processing, and if the word is still `minwordlen` or greater. This only has effect if suffix processing is enabled (`suffixproc` on and the original word is at least `minwordlen` long)

- `reqsdelim`
Flag indicating start delimiter must be present
- `reqedelim`
Flag indicating end delimiter must be present
- `olddelim`
Flag indicating old delimiter behavior should be used
- `withincount`
Value of integer N if within operator was “w/N”
- `phrasewordproc`
Phrase word processing mode (`API3PHRASEWORD... value`)
- `textsearchmode`
The TXCFF mode for text searches
- `stringcomparemode`
The TXCFF mode for string comparisons
- `setqoffs`
List of offsets into original user query, corresponding to `sets`
- `setqlens`
List of lengths in original user query, corresponding to `sets`
- `originalPrefixes`
List of set-logic, tilde, open-parenthesis, pattern-matcher character prefixes in original query, corresponding to `sets`; NULL-terminated
- `sourceExprLsts`
Each `sourceExprLsts` item corresponds to a `set` item, and is a list of source expressions/terms (before equivalence etc. processing) from original query for that set; NULL-terminated

NOTE: See Metamorph chapter 4 for detailed descriptions of what many of these variables do.

3.2 Application Notes

Generally speaking, the user program will have little need to modify the contents of the `APICP` structure returned by `openapicp()`. If the user wishes to permanently modify one or more of the default parameters it is far easier to directly edit and recompile the `api3.c` file.

The user `eqedit` and `eqedit2` functions are intended for those applications that wish to process the results of the command line/thesaurus lookup process before the remainder of the `open/setmmapi()` processing occurs. This has a similar role to the ‘EDIT’ knob inside the Metamorph user interface. For more information see the `openmmapi()` and `setmmapi()` documentation.

3.2.1 openmmapi, closemmapi - Metamorph API initialization and cleanup

SYNOPSIS

```
#include <stdio.h>
#include "api3.h"

MMAPI * openmmapi(query, cp)
char * query;
APICP * cp;

MMAPI * closemmapi(mm)
MMAPI * mm;
```

DESCRIPTION

`Openmmapi` performs the initialization required to perform a Metamorph query. It returns a pointer to a structure that will be required by the `getmmapi`, `setmmapi`, and `closemmapi` functions.

`Openmmapi` requires two parameters. The first parameter is the user's query. The query is a '`\0`' terminated string which has exactly the same syntax as a query would have within the Metamorph User Interface with the exception that there is no macro facility. Internally `openmmapi` calls `setmmapi` if the query is not `(char *)NULL`. If the query is `(char *)NULL` it is up to the programmer to call `setmmapi` before calling `getmmapi`. The second parameter is the `APICP` pointer returned by a successful call to `openapicp()`.

SEE ALSO

`setmmapi()`, `openapicp()`

3.2.2 setmmapi - Metamorph API reinitialization

SYNOPSIS

```
#include <stdio.h>
#include "api3.h"

MMAPI * setmmapi(mm, query)
MMAPI * mm;
char * query;
```

DESCRIPTION

`setmmapi()` takes a pointer to an open MMAPI and a query string. The query is a `'\0'` terminated string which has exactly the same syntax as a query would have within the Metamorph User Interface with the exception that there is no macro facility.

The query will be parsed using the APICP variables from the `openmmapi()` call, and following the rules described under “Query processing and Equivalence lookup”.

`setmmapi()`, or `openmmapi()` with a non-`(char *)` NULL query, must be called before making calls to `getmmapi()`.

DIAGNOSTICS

`setmmapi()` returns the mm pointer passed if successful or MMAPIPN if there was an error.

SEE ALSO

`openmmapi()`

3.2.3 getmmapi - Metamorph API search routine

SYNOPSIS

```
#include <stdio.h>
#include "api3.h"

char * getmmapi(mm,buf_start,buf_end,operation)
MMAPI * mm;
char * buf_start;
char * buf_end;
int operation;
```

DESCRIPTION

The `getmmapi()` is passed the `MMAPI *` returned by `openmmapi()` function and performs the actual search of the data pointed to by the `buf_start` and `buf_end` pointers. The operation parameter can be `buf_end`. Successive calls to `getmmapi()` with the operation be one of two values: `SEARCHNEWBUF` or `CONTINUESEARCH`. `getmmapi()` will return a `(char *)NULL` if it does not locate a hit within the buffer. If a hit is located it will return a pointer to the beginning of the hit.

If `getmmapi()` is called with the operation parameter set to `SEARCHNEWBUF`, it will begin its search at `buf_start` and search through the buffer until it locates a hit or until it reaches `buf_end`. Successive calls to `getmmapi()` with the operation parameter set to `CONTINUESEARCH` will locate all remaining hits within the bounds set by `buf_start` and `buf_end`.

Typically the sequence of events would look as follows:

```
{
    char *hit;
    char *my_buffer;
    int my_buf_size;

    MMAPI *mm;

    ...

    for(hit=getmmapi(mm,my_buffer,my_buffer+my_buf_size,SEARCHNEWBUF);
        hit!=(char *)NULL;
        hit=getmmapi(mm,my_buffer,my_buffer+my_buf_size,CONTINUESEARCH)
    )
    {
        /* process the hit here */
```

```
    }  
    ...  
}
```

SEE ALSO

`infrommapi()`

3.2.4 infommapi - Metamorph API hit information

SYNOPSIS

```
#include <stdio.h>
#include "api3.h"

int      infommapi(mm, index, what, where, size)
MMAPI   *mm;
int      index;
char    **what;
char    **where;
int      *size;
```

DESCRIPTION

After a hit has been located by the `getmmapi()` function, the calling program may get information about objects contained within the hit by passing the `MMAPI *` to the `infommapi()` function. This call can provide the following information:

- Location and length of the entire hit.
- Location and length of the start delimiter.
- Location and length of the end delimiter.
- For each set in the search that was matched:
 - The query set searched for and located.
 - The location of the set item.
 - The length of the set item.

The idea behind `infommapi()` is to provide the caller with a structured method for obtaining information about a hit that was located with the `getmmapi()` call. The index parameter and the return code are used to “walk” through the items that were located. Information about each item is placed into the variables pointed to by the `what`, `where` and `size` parameters. A return value of `-1` indicates a usage error, `0` indicates that the index is out of range, and `1` indicates that the index was in range and the data is valid.

Index values and what they return:

```
infommapi(mm, 0, &what, &where, &size)
what : Will be set to the query that was passed to the openmmapi()
      call.
```

where: Will point to the location of the hit within the buffer being searched.

size : Will be the overall length in bytes of the located hit.

```
infommapi(mm, 1, &what, &where, &size)
```

what : Will be set to the start delimiter expression in use.

where: Will point to the location of start delimiter.

size : Will be the overall length in bytes of the located delimiter.
size will be 0 and where will be (char *)NULL if the hit is at the beginning of the buffer or immediately after the previous hit.

```
infommapi(mm, 2, &what, &where, &size)
```

what : Will be set to the end delimiter expression in use.

where: Will point to the location of end delimiter.

size : Will be the overall length in bytes of the located delimiter.
size will be 0 and where will be (char *)NULL if the hit is at the end of the search buffer and no end delimiter was found in the buffer.

```
infommapi(mm, [3...n], &what, &where, &size)
```

what : Will point to the first "set" being searched for;

| set type | what points to |
|----------|------------------------------------|
| ----- | ----- |
| REX | A regular expression |
| NPM | The npm query expression |
| PPM | The root word of the list of words |
| XPM | The "approximate" string |

where: Will point to the buffer location of the set-element.

size : Will be the overall length in bytes of the located set-element.

EXAMPLE

```
{
  MMAPI  *mm;
  char   *what, *where;
  int    size, index;

  ...

  for (index = 0;
```

```

infommapi(mm, index, &what, &where, &size) == 1;
index++)
{
    switch (index)
    {
        case 0 :
            printf("The Query: %s\n", what);
            printf("The hit  :");
            for( ; size > 0; size--, where++) putchar(*where);
            putchar('\n');
            break;
        case 1 :
            printf("The start delimiter expression: %s\n", what);
            printf("The start delimiter located  :");
            for( ; size > 0; size--, where++) putchar(*where);
            putchar('\n');
            break;
        case 2 :
            printf("The end delimiter expression: %s\n", what);
            printf("The end delimiter located  :");
            for( ; size > 0; size--, where++) putchar(*where);
            putchar('\n');
            break;
        default:
            printf("set %d expression: %s\n", index - 2, what);
            printf("The set located  :");
            for( ; size > 0; size--, where++) putchar(*where);
            putchar('\n');
            break;
    }
}

...

}

```

SEE ALSO

getmmapi()

3.2.5 rdmmapi - synchronized read

SYNOPSIS

```
int      rdmmapi (buf, bufsize, fp, mp)
char     *buf;
int      bufsize;
FILE     *fp;
MMAPI    *mp;

bool freadex_strip8;
```

DESCRIPTION

| | |
|---------|--|
| buf | where to put the data |
| bufsize | the maximum number of bytes that will fit in buf |
| fp | the file to read from which must be opened binary ("rb") |
| mp | the Metamorph 3 API to synchronize for |

freadex_strip8 controls whether the high bit will be stripped from incoming data

This function works very much like `fread()` with one important exception; it guarantees that a hit will not be broken across a buffer boundary. The way it works is as follows:

1. A normal `fread()` for the number of requested bytes is performed.
2. `rdmmapi()` searches backwards from the end of the buffer for an occurrence of the ending delimiter regular expression.
3. The data that is beyond the last occurrence of an ending delimiter is pushed back into the input stream. (The method that is used depends on whether an `fseek()` can be performed or not.)

If the `freadex_strip8` global variable is non-zero the 8th bit will be stripped off all of the incoming data. This is useful for reading WordStar(C) and other files that set the high bit. Setting `freadex_strip8` incurs a speed penalty because every byte read gets stripped. Don't use this flag unless it is absolutely necessary.

`rdmmapi()` should be used any time you are doing delimited searches. An unsynchronized read can cause hits to be missed.

DIAGNOSTICS

`rdmmapi()` returns the number of bytes actually read into `buf` or `(-1)` if there was an error.

3.2.6 putmsg - handle a formatted message mmsgfh - FILE pointer for output mmsgfname - filename for output

SYNOPSIS

```
#include <stdio.h>
#include ``api3.h``
#include ``mmsg.h``
#include ``cgi.h``

int  putmsg(msgn,fn,fmt,...)
int  msgn;
char *fn;
char *fmt;

FILE *mmsgfh;
char *mmsgfname;
```

DESCRIPTION

msgn is the number of the message or (-1).
fn is the name of the function issuing the message or (char *)NULL.
fmt is the httpf() format (similar to printf() but extended).
... is the argument list for fmt if necessary.

These functions handle all output from the Metamorph API. The API reports its status periodically at points of interest. Each message has a number associated with it that indicates what type of message it is. Left alone the Metamorph API will generate message file output just like the Metamorph 3 product.

Messages consist of four basic parts:

1. the message number followed by a space
2. the text of the message
3. the name of the function issuing the message
4. a newline.

Message numbers are broken into various levels or types. The levels are grouped in hundreds. The levels are:

```

000-099  messages indicate total failure of the process
100-199  messages indicate potential failure or hazard to the process
200-299  messages are informative messages on the operation of a process
300-399  messages are hit information coming from a Metamorph 3 engine
400-499  messages are non-error messages coming from a mindex engine
500-599  messages about query/hit logic
600-699  query information/debugging info
700-999  undefined as yet (reserved)

```

Output formatting:

`putmsg()` will output `msgn` formatted with `%03d` if `msgn != (-1)`, followed by the results of `fmt` and its arguments if `fmt != (char *) NULL`, followed by `fn` formatted with “in the function: %s” if `fn != (char *) NULL`, followed by a newline. The output buffer is flushed to disk after every message so that any process reading the message file will always be able to get the latest messages.

Summary of formatting control:

```

to suppress msgn : pass -1
to suppress fn   : pass (char *) NULL
to suppress fmt  : pass (char *) NULL

```

Output destination:

`mmsgfh` and `mmsgfname` control where `putmsg()` will send its output. Each time `putmsg()` or `datamsg()` is called they will attempt to make `mmsgfh` point to a legal file, named by `mmsgfname`, and send their output there. Setting `mmsgfh` is accomplished by the function `fixmmsgfh()` (See ‘`putmsg()`’ extensions”). How it works is described below.

If `mmsgfh` becomes `(FILE *) NULL` or the name pointed to by `mmsgfname` changes `mmsgfh` will be closed, if it was not `(FILE *) NULL`, and reopened for binary append with the new `mmsgfname`. If the open fails `mmsgfname` will be set to point to "", the empty string, `mmsgfh` will be set to `stderr`, and a warning message will be issued via `putmsg()`. Only the first 127 characters in `mmsgfname` will be remembered between calls, so changes beyond that point will not be noticed.

If you want to set `mmsgfh` yourself and not have it changed set `mmsgfname` to `(char *) NULL`. This will preempt the checks described above. `mmsgfh` will, however, be checked for `(FILE *) NULL` and will be reset to `stderr` if it is.

The initial setting for `mmsgfh` is `(FILE *) NULL` and the initial setting for `mmsgfname` is `(char *) NULL`. This will, by default, cause all output to go to `stderr`.

EXAMPLE

call:

```

    putmsg(MERR, "parse expression", "invalid escapement");
output:
    000 invalid escapement in the function parse expression\n

call:
    putmsg(-1, "parse expression", "invalid escapement");
output:
    invalid escapement in the function parse expression\n

call:
    char *filename="myfile";
    putmsg(MERR+FOE, (char *)NULL, "can't open file %s", filename);
output:
    002 can't open file myfile\n

```

DIAGNOSTICS

`putmsg()` returns 0 for success or -1 if there was an error writing the output file. If there was an error the standard library variable `errno` may be checked for the reason. The output file will *not* be closed if there is an error.

NOTES

`putmsg()` may be overridden by writing your own function with the same name, arguments and return value so that messages can be handled in an application specific manner.

When `putmsg()` outputs a newline it will be the correct type for the host operating system (CRLF on MS-DOS, LF on Unix).

MS-DOS applications:

MS-DOS does not allow real multi-tasking so the contents of the message file will not become available to another program until the message file is closed (this is an MS-DOS limitation). To read the message file while a search is in progress you must access `mmsgfh` directly. If you move the `mmsgfh` file position by seeking, remember to reposition it to the end with `fseek(mmsgfh, 0L, SEEK_SET)` before allowing the Metamorph 3 API to continue.

SEE ALSO

`apimmsg.c`, `apimmsg.h` for specific message numbers and their macros

putmsg() extensions
putmsg() replacement

3.2.7 putmsg() extensions: fixmsgfh, closemsgfh

SYNOPSIS

```
void closemsg(void)
```

```
void fixmsgfh(void)
```

DESCRIPTION

These are some useful extensions to the `putmsg()` family of functions that provide more flexibility to programmers. They are not used directly by the Metamorph 3 API.

`closemsg()` closes the message file and should be called before exiting any application that uses `putmsg()` or the Metamorph 3 API. It may also be called in the middle of an application to flush the message file buffers to disk or force the message file to be reopened on the next call to `putmsg()`. If `mmsgfh` is `stderr`, it will not be closed, but the next `putmsg()` call will still force a reopen. It is safe to call `closemsg()` at any time because it will not attempt to close a file that is already closed or has never been opened.

`fixmsgfh()` is called by `putmsg()` before any output is attempted. It guarantees that `mmsgfh` points somewhere legal based on `mmsgfname`. See “Output destination” in the `putmsg()` description.

`fixmsgfh()` will probably not be needed by API users because `putmsg()` supplies ample output functionality.

If you use any of these functions and replace `putmsg()` you will have to write your own replacements for the extensions. All of the `putmsg()` functions are in the same object module within the library. Therefore, calling any one of the functions will cause all of them to be brought in by the linker, which will then cause a clash if you have your own version of `putmsg()`.

SEE ALSO

```
putmsg()  
putmsg() replacement
```

3.2.8 putmsg() replacement

DESCRIPTION

Normally all Metamorph 3 API output goes to `stderr` or a disk file. But, depending on your application, this may not be desirable. You may wish to send all messages to a special alternate window under a graphical environment or process the messages as they occur and take immediate action based on the type of message. You may also want to filter the messages so that only errors and warnings get displayed. Whatever your reason, `putmsg()` may be replaced.

Since `putmsg()` takes a variable number of arguments it must be written using `varargs` or the ANSI `stdarg`, if you prefer (see your 'C' manual). Only the `varargs` method will be documented here. The core is the same either way; the only variation is how you go about getting the function arguments.

There are three arguments that are always present. The first argument is a message number of type `(int)`. The second argument is a function name of type `(char *)`. The third argument is a `htpf()` format string of type `(char *)`. `htpf()` is a Thunderstone function similar to `printf()`, but with extended flags and codes: the `fmt` argument should always be printed with an `htpf()`-family function and not `printf()`-family because some messages may utilize these extended flags and codes. Any remaining arguments are as required by the `htpf()` format string.

`putmsg()` returns whether there was an error outputting the message or not. A return of 0 means there was not an error. A return of non-0 means there was an error.

All of the macros needed for `putmsg()` are in the header `"mmsg.h"`. `"api3.h"` automatically `#include's` `"mmsg.h"`. If you put `putmsg()` in its own source file just use `"mmsg.h"`. If you put `putmsg()` in the same file as Metamorph API calls or call the API from `putmsg()` use `"api3.h"`.

EXAMPLE

```
/*
** This implementation will *ONLY* output errors(MERR) and
** warnings(MWARN) .
** It will output "ERROR:" or "WARNING:" instead of a message number.
** It will always send its output to stderr.
** Function names will not be printed.
*/

#include <stdio.h>
#include <varargs.h>          /* for variable argument list handling */
#include "mmsg.h"             /* or "api3.h" */
#include "cgi.h"              /* for htvfpf() prototype */

int
```



```

putmsg(va_alist)                                /* args: msgn,funcname,fmt,... */
va_dcl      /* no semicolon allowed! - just the way varargs works */
{
va_list args;          /* for variable argument list handling          */
int      n;            /* the message number (may be -1)          */
char     *fn;          /* the function name (may be NULL)        */
char     *fmt;         /* the http type format string (may be NULL) */
int level;             /* message hundreds level */

                                /* get the fixed arguments */
va_start(args);          /* initialize variable argument list usage */
n  =va_arg(args,int      );          /* message number */
fn =va_arg(args,char *);          /* function name */
fmt=va_arg(args,char *);          /* http format */

if(n>=0){                    /* is there a message number */
    level=n-(n%100);        /* clear the tens and ones places */
                                /* to get the hundreds level */
    if(level==MERR || level==MWARN){ /* only do error or warning */
        if(level==MERR) fputs("ERROR: ",stderr);
        else            fputs("WARNING: ",stderr);
        if(fmt!=(char *)NULL){          /* is there message text */
            htvfpf(stderr,fmt,args); /* print the message content */
                                /* using the varargs version */
                                /* of httpf(): htvfpf() */
        }
        fputc('\n',stderr);          /* print the new line */
    }
}
va_end(args);              /* terminate variable argument list usage */
return(0);                 /* return OK */
}

```

NOTES

The `putmsg()` extensions need not be replaced if you are not going to use them because the API does not use them. If you do use any extensions you must also replace the ones that you use to avoid linker clashes.

SEE ALSO

`putmsg()`

apimmsg.c, mmsg.h

3.3 Query processing and Equivalence lookup

Query processing and equivalence lookup occur in `setmmapi()` and `openmmapi()` if `query!=(byte *)NULL`.

Control query parsing and equivalence lookup with the following APICP variables:

```
byte  *query
    : The user query interpret and get equivalences for.

byte  *eqprefix
    : The main equivalence file name.

byte  *ueqprefix
    : The user equivalence file name.

byte   see
    : Flag that says whether to lookup see references or not.

byte   keepeqvs
    : Flag that says whether to keep equivalences or not.

byte   keepnoise
    : Flag that says whether to keep noise words or not.

byte   withinproc
    : Flag that says whether to process the within operator (w/) or not.

byte   suffixproc
    : Flag that says whether to do suffix processing or not.

int    minwordlen
    : The smallest a word is allowed to get through suffix stripping.

byte **suffixeq
    : The list of suffixes.

byte **noise
    : The list of noise words.

int    (*eqedit)(APICP *)
    : Equivalence editor function.
```

```
int  (*eqedit2) (APICP *,EQVLST ***)
: Equivalence editor function.

void  *usr
: An arbitrary user data pointer.
```

NOTE: Also see Metamorph chapter 4 for further descriptions of these variables.

```
byte *query:
```

query is a pointer to a Metamorph query. This string typically comes directly from user input, but may be constructed or preprocessed by your program. All rules of a Metamorph query apply.

- REX patterns are prefixed by ' / ' .
- XPM patterns are prefixed by ' % ' .
- NPM patterns are prefixed by ' # ' .
- Required sets are prefixed by ' + ' .
- Exclusive sets are prefixed by ' - ' .
- Normal sets are prefixed by ' = ' or nothing.
- Intersection quantities are prefixed by ' @ ' .
- Equivalence lookup may be prevented/forced on an individual word or phrase by prefixing it with ' ~ ' .
- Commas will be treated as whitespace except when part of a pattern (REX, XPM, or NPM).
- Phrases or patterns with spaces in them that should be treated as a unit are surrounded by double quotes (' " ').
- Noise stripping is controlled by the keepnoise flag (see below).
- Equivalence lookup may be completely turned off by setting eqprefix to (byte *)NULL (see below). Turning off equiv lookup does not affect query parsing as described above.
- New delimiters may be specified using the within operator (w/).

```
byte *eqprefix:
```

This string contains the name of the main equivalence file. This typically includes the full path but may have a relative path or no path at all. The eqivs may be relocated or even renamed.

Default eqprefix "builtin" which refers to a compiled in equiv file.

This default may be permanently adjusted by changing the macro `API3EQPREFIX` in the `api3.h` header file and recompiling `api3.c` and replacing the resultant object file in the library.

Equivalence lookup may be completely turned off by setting `eqprefix` to `(byte *)NULL`. Sometimes it is not appropriate to get the associations from the `equiv` file or you may want to run your application without the disk space overhead of the `equiv` file which is very large (around 2 megabytes). Turning off `equiv` lookup does not affect query parsing as described previously.

```
byte *ueqprefix:
```

This string contains the name of the user equivalence file. This typically includes the full path but may have a relative path or no path at all. The `equivs` may be relocated or even renamed.

Default `ueqprefix` for Unix: `"/usr/local/morph3/eqvsusr"` Default `ueqprefix` for MS-DOS: `"c:\morph3\eqvsusr"`

This default may be permanently adjusted by changing the macro `API3UEQPREFIX` in the `api3.h` header file and recompiling `api3.c` and replacing the resultant object file in the library.

Equivalences in the user `equiv` file edit and/or override those in the main `equiv` file.

```
byte withinproc:
```

Process the within operator (`w/`). The within operator allows changing the start and end delimiters from the query line. The argument of the within operator may be one of the built in names, a number indicating character proximity, or a REX expression. The built in names are:

| Name | Meaning | Expression |
|-----------------|-----------|---|
| sent | Sentence | <code>\verb '[^\\digit\\upper][.?!][\\space'"]'</code> |
| para | Paragraph | <code>\verb '\\x0a=\\space+''</code> |
| line | Line | <code>\verb '\$'</code> |
| page | Page | <code>\verb '\\x0c'</code> |
| <code>\#</code> | Proximity | <code>\verb '{, #}' (where \# is the number of characters)</code> |

Any other string following the `"w/"` is considered a REX expression. When using a REX expression with the within operator both start and delimiters are set to the expression to set the end delimiter to a different expression specify another within operator and expression. e.g. `"power w/tag: w/$"` will set the start delimiter to `"tag: "` and the end delimiter to `"$"`.

By default both delimiters will be excluded from the hit when using a REX with the within operator. To specify inclusion use a `"W/"` instead of `"w/"`. You may specify different inclusion/exclusion for the end delimiter without repeating the expression if you wish to use the same expression for both. Simply use the `"W/"` or `"w/"` by itself for the end delimiter. e.g. `"power w/$$ W/"` will set both delimiters to `"$$"` but will exclude the start delimiter and include the end delimiter.

The default value for `withinproc` is 1. This default may be adjusted by changing the macro `API3WITHINPROC` in the `api3.h` header file and recompiling `api3.c`.

See also the section "Reprogramming the Within Operator".

byte see:

Lookup “see” references in the equiv file. The equiv file has “see” references much as a dictionary or thesaurus has. With this flag off “see” references are left in the word list as is. With it on, those references will be looked up and their equiv lists added to the list for the original word. This can greatly increase the number of equivs and abstraction for a given word. This is not needed in most cases.

The default value for see is 0. This default may be adjusted by changing the macro `API3SEE` in the `api3.h` header file and recompiling `api3.c`.

byte keepnoise:

Keep noise words. With this flag off any word in query, that is not part of a larger phrase, that is also found in the noise array will be removed from the list.

The default value for keepnoise is 1. This default may be adjusted by changing the macro `API3KEEPNOISE` in the `api3.h` header file and recompiling `api3.c`.

byte keepeqvs:

Invert normal meaning of `~`. With this flag on words will not normally have equivs. To get the equivs for a word use the `~` prefix.

The default value for keepeqvs is 1. This default may be adjusted by changing the macro `API3KEEPEQVS` in the `api3.h` header file and recompiling `api3.c`.

Setting keepeqvs to 0 does not eliminate looking for the equiv file. See the `eqprefix` variable for how to eliminate the equiv file completely.

byte suffixproc:

This is a flag that, if not set to 0, will cause the equiv lookup process to strip suffixes from query words and words from the equiv file to find the closest match if there is not an exact match. Words will not be stripped smaller than the `minwordlen` value (see below). This flag has a similar effect on the search process (see Metamorph section 4.3.3).

The default value for suffixproc is 1. This default may be adjusted by changing the macro `API3SUFFIXPROC` in the `api3.h` header file and recompiling `api3.c`.

byte **suffixeq:

This is the list of word endings used by the suffix processor if `suffixproc` is on (see the description of lists). The suffix processor also has some permanent built in rules for stripping. This is the default list:

' s ies

The default may be changed by editing the `suffixeq[]` array in the function `openapicp()` in the file `api3.c` and recompiling.

```
int minwordlen:
```

This only applies if `suffixproc` is on. It is the smallest that a word is allowed to get before suffix stripping will stop and give up.

The default value for `minwordlen` is 5. This default may be adjusted by changing the macro `API3MINWORDLEN` in the `api3.h` header file and recompiling `api3.c`. This flag has a similar effect on the search process (see Metamorph section 4.3.5).

```
byte **noise:
```

This is the default noise list:

| | | | | | |
|----------|------------|--------|-----------|-----------|----------|
| a | between | got | me | she | upon |
| about | but | gotten | mine | should | us |
| after | by | had | more | so | very |
| again | came | has | most | some | was |
| ago | can | have | much | somebody | we |
| all | cannot | having | my | someone | went |
| almost | come | he | myself | something | were |
| also | could | her | never | stand | what |
| always | did | here | no | such | whatever |
| am | do | him | none | sure | what's |
| an | does | his | not | take | when |
| and | doing | how | now | than | where |
| another | done | i | of | that | whether |
| any | down | if | off | the | which |
| anybody | each | in | on | their | while |
| anyhow | else | into | one | them | who |
| anyone | even | is | onto | then | whoever |
| anything | ever | isn't | or | there | whom |
| anyway | every | it | our | these | whose |
| are | everyone | just | ourselves | they | why |
| as | everything | last | out | this | will |
| at | for | least | over | those | with |
| away | from | left | per | through | within |
| back | front | less | put | till | without |
| be | get | let | putting | to | won't |
| became | getting | like | same | too | would |
| because | go | make | saw | two | wouldn't |
| been | goes | many | see | unless | yet |
| before | going | may | seen | until | you |
| being | gone | maybe | shall | up | your |

The default may be changed by editing the `noise[]` array in the function `openapicp()` in the file `api3.c` and recompiling.

```
void *usr:
```

This is a pointer that the application programmer may use as a method of passing arbitrary application specific information to the callback functions `(*eqedit)()` and `(*eqedit2)()`. This pointer is entirely under the control of the programmer. The Metamorph API does not reference it in any way except to set it to `(void *)NULL` in `openapicp()`.

3.4 Equivalence editing callbacks

During query processing, `setmmapi()` will call two user callback functions to perform editing on the query terms. The processing sequence is as follows:

1. parse the query and lookup terms in equiv file.
2. build eqvlist for `eqedit2`.
3. * call `(*eqedit2)()`.
4. check for empty or NULL list.
5. check for and remove duplication in set lists.
6. set intersections if not already set (`!0`).
7. build formatted sets for `(*eqedit)()` from eqvlist.
8. free eqvlist.
9. * call `(*eqedit)()`.
10. perform rest of internal setup.
11. return to caller.

`(*eqedit2)()` is the recommended method for implementing on the fly equiv editing because it is easier to use. `(*eqedit)()` is available for backwards compatibility.

```
int (*eqedit2)(APICP *, EQVLST **):
```

This function is always called after a successful equiv lookup and before the search begins. It is called with the current `APICP` pointer and a pointer to the list of eqivs generated by the query (see the description of lists). The list pointer may be reassigned as needed.

The return value from `(*eqedit2)()` determines whether to go ahead with the search or not. A return value of 0 means OK, go ahead with the search. A return value of anything else means ERROR, don't do

search. An `ERROR` return from `(*eqedit2)()` will then cause `setmmapi()` or `openmmapi()`, depending on where it was called from, to return an error. A `NULL` list from `(*eqedit2)()` is also considered an error.

There is one `EQVLST` for each term in the query. The array of `EQVLST`s is terminated by an `EQVLST` with the words member set to `(char *)NULL` (all other members of the terminator are ignored). The `EQVLST` structure contains the following members:

```
char    logic: the logic for this set
char **words: the list of terms including the root term
char **clas : the list of classes for 'words'
int     sz   : the allocated size of the 'words' and 'clas' arrays
int     used : the number used (populated) of the 'words' and 'clas'
               arrays, including the terminating empty string ("")
int     qoff : the offset into user's query for this set (-1 if unknown)
int     qlen : the length in user's query for this set (-1 if unknown)
char    *originalPrefix: set logic/tilde/open-paren/pattern-matcher
char    **sourceExprs: NULL-terminated list of source expressions for set
```

The `words` and `clas` arrays are allocated lists like everything else in the `APICP`, and are terminated by empty strings. The `sz` and `used` fields are provided so that editors may manage the lists more efficiently.

The `words` and `clas` lists are parallel. They are exactly the same length and for every item, `words[i]`, its classification is `clas[i]`.

The `originalPrefix` field (added in Taxis version 6) contains the set logic (“+”, “-”, “=”), tilde (“~”), open-parenthesis, and/or pattern-matcher characters (“/” for REX, “%” for XPM, “#” for NPM) present in the original query for this set, if any. It can be used in reconstructing the original query, e.g. if the terms are to be modified but set logic etc. should be preserved as given.

The `sourceExprs` field (added in Taxis version 6) contains a list of the source expressions or terms for the set, i.e. as given in the original query. For SPM queries, this will be a single word or phrase. For PPM queries given as parenthetical lists, this will be a list of the individual terms or phrases. For REX/NPM/XPM queries, this will be the expression (sans “/”/“#”/“%”). For single terms that are expanded by equivalence lookup, this will be the original single term, *not* the expanded list (as `words` will be) – because `sourceExprs` is from the source (original query), not post-equivalence-processing. Note also that `sourceExprs` is `NULL` (not empty-string) terminated. The `sourceExprs` array can be used in reconstructing or modifying queries.

The default function is the function `nulleqedit2` in `api3.c` which does nothing and returns 0 for OK.

```
int (*eqedit)(APICP *):
```

This function is always called after a successful equiv lookup and before the search begins. It is called with the current `APICP` pointer with the “set” list in the `APICP` structure set to the list of equivs generated by the query (see the description of lists).

The return value from `(*eqedit)()` determines whether to go ahead with the search or not. A return value of 0 means OK, go ahead with the search. A return value of anything else means `ERROR`, don't do

search. An `ERROR` return from `(*eqedit)()` will then cause `setmmapi()` or `openmmapi()`, depending on where it was called from, to return an error.

The format of the sets is:

```
{-|+|=}word[;class][,equiv][...]
```

Or:

```
{-|+|=}{/|%99|#}word
```

Where:

```
[ ]    surround optional sections.
{ }    surround required items to be chosen from.
|      separates mutually exclusive items between { }.
9      represents a required decimal digit (0-9).
word   is the word, phrase, or pattern from the query.
equiv  is an equivalent for word.
class  is a string representing the classification for the
       following words.
...    means any amount of the previous item.
```

Classifications in the default thesaurus (case is significant):

| | |
|------------------|------------------------|
| P = Pronoun | c = Conjunction |
| i = Interjection | m = Modifier |
| n = Noun | p = Preposition |
| v = Verb | u = Unknown/Don't care |

Words and phrases will be in the first format. Patterns will be in the second format.

```
=struggle;n,battle,combat,competition,conflict,compete;v,contest,strive
    battle, combat, competition, and conflict are nouns
    compete, contest, and strive are verbs
    struggle can be a noun or verb
```

```
=status quo;n,average,normality
    status quo, average, and normality are nouns
```

```
+Bush;P
    Bush is a pronoun
```

```
-/19\digit{2}
    a REX pattern to find "19" followed by 2 digits
```

```
=%80qadafi
    an XPM pattern to find qadafi within 80%

=#>500
    an NPM pattern to find numbers greater than 500
```

Remember that each of the “set” strings is allocated. So if you replace a set you must free the old one, to prevent memory loss, and use an allocated pointer for the replacement because it will get freed in `closepic()`, unless it is `(byte *)NULL`.

The “set” format must be totally correct for the search process to work.

The default is the function `nulleqedit` in `api3.c` which does nothing and returns 0 for OK.

3.5 User equivalence file maintenance

A user equivalence file contains equivalences in a manner similar to the main equiv file. The user equiv contains equivs that edit and/or replace equivs in the main equiv. It may also contain new equivs.

Make a user equiv file by creating an ASCII file containing your desired equiv edits. Then index that source file with `backref` program.

The user equiv source file has the following format:

- The root word or phrase is the first thing on the line.
- Hyphenated words should be entered with a space instead of a hyphen.
- Subsequent words/phrases (equivs) follow on the same line prefixed with edit commands (see below).
- Add optional classification information by appending a semicolon (;) and the class to the word to apply it to. Any specified classification is carried onto subsequent words until a new classification is entered.
- Lines should be kept to a reasonable length; around 80 characters for standard screen display is prudent. In no case should a line exceed 1K. Where more equivs exist for a root word than can be fit onto one line, enter multiple entries where the root word is repeated as the first item.
- There should not be any blank lines. Lines should not have any leading or trailing spaces. Words or phrases also should not have any leading or trailing spaces.
- A user equiv file may “chain” to another user equiv file by placing the key string “;chain;” followed by the name of the equiv source file to chain to on the first line of the source file. e.g.
`”;chain;c:\morph3\eqvsusr”` Equivs are looked up in the chained file before the current file so that a user may, for example, override system wide settings. Chains are resolved when the source file gets backreferenced.

3.6 Edit commands:

Comma (,) means add this word/phrase to the list of equivs for this root.

Tilde (~) means delete this word from the list of equivs for this root.

Equals (=) only applies for the first equiv specified for a root. It means replace this entry with the following entry. The first word after the equals is taken as the new root and the rest of the words are its equivs. If the equals is followed by a non-alphanumeric character the entire rest of the line is taken literally as a replacement for the original entry. This is a macro like facility that allows you to make a word mean a regular expression or other Metamorph special pattern match.

Once you have a user equiv source file index it with the backref command. This syntax is:

```
backref source_file indexed_file
```

Where `source_file` is the ASCII file you created. And `indexed_file` is the backreferenced and indexed file that you specify in the `ueqprefix` variable. To just index the source file without backreferencing it use the `-ll` option:

```
backref -ll source_file indexed_file
```

By convention the source file should have the same path and filename as the indexed file with an extension of `".lst"`. This is what the Metamorph user interface expects. For example: the source file for `"c:\morph3\eqvsusr"` would be `"c:\morph3\eqvsusr.lst"`

Sample user equiv file:

```
chicken,bird,seed,egg,aviary,rooster
seed;n,food,feed,sprout
ranch,farm,pen,hen house,chicken house,pig sty
Farmer's Almanac,research,weather forecast,book
rose,flower,thorn,red flower
water,moisture,dew,dewdrop,Atlantic Ocean
bee pollen,mating,flower,pollination,Vitamin B
grow,mature,blossom,ripen
abort,cancel,cease,destroy,end,fail,kill
abort,miscarry,nullify,terminate,zap
wish;n,pie in the sky,dream;v,yearn,long,pine
constellation~nebula~zodiac,big dipper
abandon=abandon,leave
galaxy=andromeda
slice=slice
lots=#>100
bush=/\RBush
```

3.7 Reprogramming the Within Operator

NOTE: The mechanism described here may be replaced with something different in a future version of the Metamorph API.

The symbolic expressions that the within operator knows about may be reprogrammed by the application developer. The within processor maintains two lists of symbolic expressions: a “standard” list and a “user” list. By default the standard list contains the symbols described elsewhere in this document (line/page/etc). The user list is empty by default.

The within processor processes the string after the "w/" in the following order:

- If the first character is a digit its a proximity count.
- If it matches something in the user list, use its expression.
- If it matches something in the standard list, use its expression.
- Otherwise it's taken literally as a rex expression.

Each list is made up of an array of MDPDLM structures

```
MDPDLM {
    char *name;
    char *expr;
    int   incsd;
    int   inced;
};
```

Where:

name is the name used in the query with the "w/" operator. expr is the rex expression associated with name. incsd is a flag indicating whether to include the start delimiter. inced is a flag indicating whether to include the end delimiter.

The array is terminated with a MDPDLM with the name member set to (char *)NULL.

The lists are manipulated with the mdpstd() and mdpusr() functions to control the standard and user lists respectively.

```
MDPDLM *mdpstd(MDPDLM *);
MDPDLM *mdpusr(MDPDLM *);
```

These functions set their respective lists to those provided by the argument. They return the previous lists. Any list may be MDPDLM_PN to suppress its processing. The list pointers are kept in a static global variable within the api library, so all subsequent within operators will be effected by any changes. The table is not copied, so the pointers passed must remain valid for the duration of all api usage.

Comparisons to name need only match for the length of name, thus allowing abbreviations. e.g. The following will both match for a name of “mess”: w/message, w/messy.

EXAMPLE

```
static MDPDLM mydelims[]={
    { "mess", "^From:"          ,1,0 },          /* add a new name */
    { "page", "-- \\digit+ --:" ,0,1 }, /* override an existing name */
    { CHARPN }
};

...
/*
    you could call
        mdpstd(MDPDLMPN);
    to suppress the standard names so that only the usr names
    would be recognized.
*/
mdpusr(mydelims);
...
setmmapi(mm,query);
...
```


Chapter 4

Low level Pattern Matchers

Programmers note: Do not use these functions unless you are sure you know why you need them. Thunderstone will not provide technical support involving the `((ab=normal ?)|mis)` use of these functions.

4.0.1 Low level Pattern Matching functions

SYNOPSIS

REX - Deterministic Regular Expression Matcher

```
FFS    *openrex(byte *s);
FFS    *closerex(FFS *fs);
FFS    *mknegexp(FFS *fs);
byte    *getrex(FFS *fs, byte *buf, byte *end, int operation);
int      rexsize(FFS *ex);
```

XPM - Approximate Pattern Matcher

```
XPMS    *openxpm(byte *s, int threshold); /* 0 < threshold < 101 */
XPMS    *closexpm(XPMS *xs);
byte    *getxpm(XPMS *xs, byte *buf, byte *end, int operation);
```

PPM - Parallel String Pattern Matcher

```
PPMS    *closeppm(PPMS *ps);
PPMS    *openppm(byte **sl);
byte    *getppm(PPMS *ps, byte *buf, byte *end, int operation);
```

SPM - Single String Pattern Matcher

```
SPMS    *openspm(char *s);
SPMS    *closespm(SPMS *fs);
byte    *getspm(SPMS *fs, byte *buf, byte *end, int operation);
int      spmhitsz(SPMS *fs);
```

NPM - Numeric Pattern Matcher

```
NPMS    *opennpm(char *s);
NPMS    *closenpm(NPMS *);
byte    *getnpm(NPMS *, byte *, byte *, int);
```

DESCRIPTION

These pattern matchers represent the core search algorithms that are present in the **Metamorph** program and API. Usage and syntax details are not presented here as they are described in great detail in the **Metamorph User Manual**. The programmer is encouraged to understand their purpose and usage before implementing their own software using these functions.

All of the pattern matchers behave in a similar fashion from a programming perspective. The `open____()` call initializes the matcher and returns a “handle” to it, and the `close____()` call `free()`s the memory associated with that object’s “handle”. If the `open____()` call fails for any reason it will return a cast pointer to `NULL`.

The arguments passed to the `open____()` call are as follows:

| Call | Parameters Type(s) | Example |
|------------------------|-----------------------|---|
| <code>openrex()</code> | Rex expression | <code>"\\x0d\\x0a+"</code> |
| <code>openxpm()</code> | any string, threshold | <code>"abc widgets", 80</code> |
| <code>openppm()</code> | a list of strings | <code>{"abc", "def", "ghi", ""};</code> |
| <code>opennpm()</code> | a numeric expression | <code>">1000<=million"</code> |
| <code>openspm()</code> | a string expression | <code>"abc*def"</code> |

In all cases, the `open____()` call is computationally intensive, as each algorithm makes extensive use of dynamic programming techniques. It is generally considered that the pattern matcher will be processing a great deal of information between it’s creation and destruction so that the creation overhead is justified by the dramatic increase in search rates.

If the pattern matchers are to be used in conjunction with one another then the programmer should optimize usage by dispatching the pattern matchers in order of relative speed. The following table enumerates the relative search rates:

| Pattern Types | Governing factors |
|---------------|--|
| SPM Fastest | longer strings are faster |
| REX . | longer expressions are usually faster |
| PPM . | shorter lists with the longest minimum <code>strlen()</code> |
| XPM . | shorter strings are faster |
| NPM Slowest | nothing makes a difference |

The `get____()` call is responsible for the location of *hits* within a buffer of information. All of the pattern matchers share a common parameter set for this operation:

(object pointer, byte *buf , byte *end, int operation)

object pointer The structure pointer that was returned by the `open____()` call.

byte *buf A pointer to the first character of the buffer to be searched ¹.

byte *end A pointer to one character past the last character in the buffer to be searched. (Usually obtained by the expression `buf+bufsize`).

int operation This will be one of the four possible operation codes:

`SEARCHNEWBUF`, `CONTINUESEARCH`, `BSEARCHNEWBUF`, or `BCONTINUESEARCH`.

¹byte is defined as an unsigned char

The `SEARCHNEWBUF` to locate the first occurrence of a hit within the delineated buffer, and the `CONTINUESEARCH` operation code indicates that the pattern matcher should locate the next (or successive) occurrence. A pointer to the matched pattern will be returned or, a `(byte *)NULL` if no match was located within the buffer for the given call.

The operation codes `BSEARCHNEWBUF` and `BCONTINUESEARCH` are only understood by the REX pattern matcher, and are used to search backwards from the `end` pointer towards the `buf` pointer. A non `NULL` return value will point to the beginning of the matched pattern.

Some information about the matched hits for each of the pattern matchers may be obtained by looking into that pattern matcher's structure. The following structure members are the only valid ones for an API programmer to use:

```
NPMS /* Numeric Pattern Matcher */
{
    int hitsz; /* the length of the matched quantity */
    double hy; /* the maximum numeric value of the matched string */
    double hx; /* the minimum numeric value of the matched string */
};

PPMS
{
    byte **slist; /* the strings being sought */
    int sn; /* the index of the located string within slist */
};

XPMS
{
    word thresh; /* threshold above which a hit can occur */
    word maxthresh; /* the max possible value a match may have */
    word thishit; /* the value of this match */
    word maxhit; /* max value located so far */
    byte maxstr[DYNABYTE]; /* the string with highest match value */
};

SPMS
{
    /* no usable members */
};

FFS /* aka REX */
{
    /* no usable members */
};
```

Hit length information for REX and SPM is available through calls to `rexsize()` and `spmhitsz()` respectively. Each of these functions return the length of the last hit located by a call to `get____()`. The

reason there are not similar calls available for the other pattern matchers is because their length is obtainable via the structure.

EXAMPLE

```
#include "api3.h"

/* this code breaks some rules in the interest of brevity */

main(argc,argv)
int    argc;
char **argv;
{
    void *pm=(void *)NULL;
    int    i;
    void (*close)();          /* pointer to the close function */
    void (*get)();            /* pointer to the get function */
    char ln[80];

    switch(*argv[1])          /* determine search type via leading character */
    {
        case '/' : pm=(void *)openrex(argv[1]+1);
                    get=getrex;
                    close=closerex;
                    break;
        case '#' : pm=(void *)opennpm(argv[1]+1);
                    get=getnpm;
                    close=closenpm;
                    break;
        case '%' : pm=(void *)openxpm(argv[1]+1,80);
                    get=getxpm;
                    close=closexpm;
                    break;
    }
    if(pm==(void *)NULL)      /* check to see if it opened */
        exit(1);

    while(gets(ln)!=NULL)
    {
        /* see if there hit on this line */
        if((*get)(pm,(byte *)ln,(byte *) (ln+strlen(ln)),SEARCHNEWBUF)
            !=(byte *)NULL)
            puts(ln);
    }
    (*close)(pm);
    exit(0);
}
```

```
}
```

Chapter 5

Windows Addendum

Those using non-Microsoft compilers should also read `MSFOPEN.H`.

The Metamorph Windows API is provided in the form of two DLLs and their associated export libraries:

| | |
|---------------------------|---------------------------|
| <code>MORPH.DLL</code> | <code>MORPH.LIB</code> |
| <code>MORPHMEM.DLL</code> | <code>MORPHMEM.LIB</code> |

Both DLLs must be available for Metamorph applications at run time. `MORPHMEM.DLL` contains all memory handling functions.

The entire Metamorph API was compiled large model with the PASCAL calling convention. All function calls are therefore PASCAL unless declared otherwise. `Putmsg()` uses the C calling convention since it has a variable number of arguments. All data passed to and from API functions must be FAR.

`Putmsg()` is handled a little differently under Windows since the version in the DLL can not be effectively replaced. The default behavior of `putmsg()` is to write to file handle 2. To change the output file call `setmmsgfname()` with the name of the file to write to. To change the output file to an already opened file call `setmmsgfh()` with the opened file handle. To change the message handling function completely call `setmmsg()` with a pointer to your message handling function. See the descriptions of these functions for more details.

Message handling should be setup before calling any Metamorph API functions that could fail so that messages will go to a known place.

See `MMEXW.C` for working examples. `MMEXW.MAK` is a Quick C 1.00 project file for `MMEXW`.

5.0.1 setmmsgfname - set the name of the message file

SYNOPSIS

```
#include "windows.h"
#include "stdio.h"
#include "api3.h"
#include "mmsg.h"

char FAR * FAR setmmsgfname(newfname)
char FAR *newfname;
```

DESCRIPTION

Setmmsgfname() will change the file that putmsg writes messages to. It returns its argument. The default is to write messages to file handle 2 (stderr).

EXAMPLE

```
APICP *acp;

...

setmmsgfname("msg.001");    /* set message file name to msg.001 */

...

acp=openapicp();            /* open mm api control parameters */
```

5.0.2 setmmsgfh - set the message file handle

SYNOPSIS

```
#include "windows.h"
#include "stdio.h"
#include "api3.h"
#include "mmsg.h"

int FAR setmmsgfh(newfhandle)
int newfhandle;
```

DESCRIPTION

Setmmsgfh() will change the file handle that putmsg writes messages to. It returns its argument. The default is to write messages to file handle 2 (stderr).

EXAMPLE

```
int fh;
OFSTRUCT mmsginfo;
APICP *acp;

...

/* open file msg.001 in "wb" mode */
fh=OpenFile("msg.001",&mmsginfo,
            (OF_CANCEL|OF_CREATE|OF_WRITE|OF_SHARE_DENY_NONE));

if(fh!=(-1)) setmmsgfh(fh); /* set message file handle if ok */

...

acp=openapicp(); /* open mm api control parameters */
```

5.0.3 setmmsg - set custom message handler function

SYNOPSIS

```
#include "windows.h"
#include "stdio.h"
#include "stdarg.h"
#include "api3.h"
#include "mmsg.h"

void FAR setmmsg(newfunction)
int (FAR *newfunction)(int msgn, char FAR *fn, char FAR *fmt,
                      va_list args);
```

DESCRIPTION

Setmmsg() will set the function to call to handle messages from putmsg(). The handler function will receive four arguments:

```
1: int msgn;          the message number          (same as putmsg())
2: char *fn;          the function name            (same as putmsg())
3: char *fmt;         the http format string       (same as putmsg())
4: va_list args;      the stdarg argument list as derived in putmsg()
by the va_start() call.
```

The args variable may be used like any va_list that has been va_start()'d. e.g. in a call to WVSPRINTF(). Do *not* call va_end() on args.

The handler function pointer must be the result of the Windows MakeProcInstance() call so that it can be called correctly from within the Metamorph API DLL. See your Windows SDK manual for details on MakeProcInstance(). You should not use MakeProcInstance() more than once for any given function if possible. Each call to it will use a little memory.

The return value of the handler function will be returned by putmsg() to the original caller.

EXAMPLE

...

```
/* a custom message handler */
int FAR PASCAL
```



```

msghandler(int n,char FAR *fn,char FAR *fmt,va_list args)
{
static char buf[256];          /* place to sprintf to */
char          *d;

    if(n>=0 && n<100)          strcpy(buf,"ERROR: ");
    else if(n>=100 && n<200) strcpy(buf,"WARNING: ");
    else                        strcpy(buf,"FYI: ");
    if(fmt!=(char *)NULL){
        d = buf + strlen(buf);
        htsnprintf(d, (buf + sizeof(buf)) - d, fmt, args);
    }
    if(fn!=(char *)NULL){
        strcat(buf," In the function: ");
        strcat(buf,fn);
    }

        /* display message in a standard windows message box */
    MessageBox(GetFocus(),(LPSTR)buf,(LPSTR)"Metamorph 3 Message",MB_OK);
    return(0);
}

...

HANDLE hInst;                  /* current instance */
FARPROC m;
APICP *acp;

...

putmsg(MINFO,(char *)NULL,"Default handler active");

m=MakeProcInstance(msghandler,hInst);

if (m!=(FARPROC) NULL) {

    setmmsg(m);

    putmsg(MINFO,(char *)NULL,"My custom handler active");

}

...

acp=openapicp();               /* open mm api control parameters */

```

Windows programs are generally case insensitive pascal calling sequence. Using the Metamorph API under Borland C with case sensitivity and C calling sequence as defaults requires a little adjustment.

Borland C options:

1. Windows large model exe.
2. `define_WINDOWS`.
3. Borland C++ source (detect C/C++ by extension).
4. Case sensitive exports as well as case sensitive link.

api3.h modifications:

1. Insert the “pascal” keyword before each function name in its prototype.
2. Redefine all functions as uppercase before their prototypes. (e.g.
`"#define rdnmapi RDMMAPI")`

Part V

Taxis Programs

0.0.4 `tsql` - Taxis Interactive SQL

SYNOPSIS

```
tsql [-a command] [-c [-w width]] [-l rows] [-hqm?] [-d database]
[-u username] [-p password] [-i file] ["SQL statement"]
```

DESCRIPTION

`Tsql` is the main program provided for interactive use of a Taxis database. You should either be in the database directory, or else specify it on the command line as the `-d <database>` option.

If a query is present on the command line than `tsql` will execute that statement, and display the results on stdout (the screen). If no query is present then queries will be accepted on stdin (the keyboard). Queries on stdin must be terminated by a semicolon. To exit from `tsql` you should produce EOF on it's stdin. On Unix systems this is usually done with Control-D, and on Windows with Control-Z.

`Tsql` also provides facilities for doing many administrative tasks.

Options

To aid in the use of `tsql` from scripts or as parts of a chain of commands there are many options available. The options are

- h** Don't display the column headers.
- q** Don't display the SQL prompt and copyright statement.
- c** Change the format to one field per line.
- w width** Make the field headings width characters long. 0 means use the longest heading's width.
- l rows** Limit output to the number of rows specified.
- u username** Login as username. If the `-p` option is not specified then you will be prompted for a password. The user must have been previously added to the system. This forces usage of the Taxis permission scheme. If this is not specified then the name defaults to PUBLIC. See Chapter 10 for more information.
- p password** Use password to log in.
- P password** _SYSTEM password to log in for admin.
- d database** Specify the location of the database.
- m** Create the named database. See `creatdb`.
- i file** Read SQL commands from file instead of stdin. Commands read this way will echo to stdout, whereas commands read from input redirection would not.

-r Read the default profile file.

-R profile Read the specified profile file.

-f Specify a format. One or two characters can follow with the following meanings.

t same as **-c** option

c default behaviour

any other character is a field separator character. This can be followed by **q** to suppress the quotes around the fields. To get quoted comma separated values you would use **-f ,**

-? Print a command summary.

Chapter 1

The Taxis Client API

1.1 Overview

This chapter provides a reference to the functions that you will need to use to write your own custom application that talks to Taxis.

1.1.1 Tesis Client Functions

SYNOPSIS

SQL interface

```
TEXIS *tesis_open(char *database, char *user, char *password);
TEXIS *tesis_open_options(void *, void *, void *, char *database, char *user, char *g
int tesis_prepare(TEXIS *, char *sqlQuery);
int tesis_execute(TEXIS *);
int tesis_param(TEXIS *, int paramNum, void *data, long *dataLen, int cType, int sqlT
FLDLST *tesis_fetch(TEXIS *, int stringsFrom);
int tesis_flush(TEXIS *tx);
int tesis_flush_scroll(TEXIS *tx, int nrows);
char **tesis_getresultnames(TEXIS *tx);
int *tesis_getresultsizes(TEXIS *tx);
int tesis_getrowcount(TEXIS *tx);
TEXIS *tesis_close(TEXIS *);
FLDLST *freefldlst(FLDLST *fl);
```

DESCRIPTION

The Tesis API provides the client program with an easy to use interface to any Tesis server.

The programmer is *strongly* encouraged to play with the example programs provided with the package before attempting their own implementation.

SQL interface This group of operations is for passing a user's query to the database for processing and subsequently obtaining the results.

1.1.2 FLDLST - Struct containing results

SYNOPSIS

```
typedef struct FLDLST
{
    int      n;                /* number of items in arrays */
    int      type [FLDLSTMAX]; /* data type */
    void     *data [FLDLSTMAX]; /* pointer to data */
    int      ndata[FLDLSTMAX]; /* number of els in data */
    char     *name [FLDLSTMAX]; /* name of field */
    int      ondata[FLDLSTMAX]; /* 2002-07-05 number of els in schema */
} FLDLST;
```

A FLDLST is a structure containing parallel arrays of information about the selected fields and a count of those fields. FLDLST members:

`int n` — The number of fields in the following lists.

`int *type` — An array of types of the fields. Each element will be one of the FTN_XXXX macros.

`void **data` — An array of data pointers. Each element will point to the data for the field. The data for a field is an array of types.

`int *ndata` — An array of data counts. Each element says how many elements are in the data array for this field.

`char **name` — An array of strings containing the names of the fields.

`SRCHLST *sl` — An array of SRCHLSTs containing information Metamorph searches, if any. Filled in, on request only, by `n_filllsrchlst()`.

`MMOFFS mmoff` — An array of Metamorph subhit offsets and lengths from Metamorph searches, if any. Filled in, on request only, by `n_filllsrchlst()`.

Possible types in FLDLST->type array:

FTN_BYTE — An 8 bit byte.

FTN_CHAR — A char.

FTN_DOUBLE — A double.

FTN_DATE — A long in the same form as that from the `time()` system call.

FTN_DWORD — A 32 bit dword.

FTN_FLOAT — A float.

FTN_INT — An int.

FTN_INTEGER — An int.

FTN_LONG — A long.

FTN_INT64 — An int64.

FTN_UINT64 — A uint64.

FTN_SHORT — A short.

FTN_SMALLINT — A short.

FTN_WORD — A 16 bit word.

FTN_INDIRECT — A char string URL indicating the file that the data for this field is stored in.

FTN_COUNTER — A ft_counter structure containing a unique serial number.

FTN_STRLST — A delimited list of strings.

The type may also be |ed with FTN_NotNullableFlag (formerly DDNULLBIT) and/or DDVARBIT. If FTN_NotNullableFlag is set, it indicates that the field is not allowed to be NULL. DDVARBIT indicated that the field is variable length instead of fixed length. When handling result rows these bits can generally be ignored.

1.1.3 `n_getindexcount()` - Hit information

SYNOPSIS

```
#include <sys/types.h>
#include "tstone.h"
int n_getindexcount (SERVER *se);
```

DESCRIPTION

This function will return the count of rows to be read from the index for the most recently prepared statement, if available.

CAVEATS

The return from this function is only valid under certain circumstances, which are when the index has been scanned to get a list of potentially matching records. In a join, the return value will be the number of matches in the inner table corresponding to the current outer row, not the number of outer table matches. The actual number of records returned may be significantly less if post processing is done to resolve some of the where clause.

The default behaviour of Taxis with a single relational operator and an index on the field is to walk the index as the rows are returned, which is faster at getting the initial rows out. Since it does not get all the matching rows from the index first, `n_getindexcount()` will return 0. This behaviour can be changed with the `bubble` property.

EXAMPLE

1.1.4 `taxis_open()`, `taxis_dup()`, `taxis_close()` - SQL interface

SYNOPSIS

```
TEXIS *taxis_open(char *database, char *user, char *password);  
TEXIS *taxis_dup(TEXIS *tx);  
TEXIS *taxis_close(TEXIS *tx);
```

DESCRIPTION

`taxis_open()` opens the database. It returns a valid TEXIS pointer on success or `TEXISPN` on failure. `TEXISPN` is an alias for `(TEXIS *)NULL`.

`taxis_dup()` creates a new TEXIS pointer to the same database as a currently valid handle. This saves much of the overhead of opening a new connection to the database. The returned handle is a clean TEXIS handle, and will not have a copy of the SQL statement from the copied handle. It returns a valid TEXIS pointer on success or `TEXISPN` on failure. `TEXISPN` is an alias for `(TEXIS *)NULL`.

`taxis_close()` closes the previously opened database. It always returns `TEXISPN`.

SQL statements are setup and executed with `taxis_prepare()`, `taxis_execute()`, and `taxis_fetch()`.

SEE ALSO

`taxis_prepare()`, `taxis_execute()`, and `taxis_fetch()`

1.1.5 taxis_prepare(), taxis_execute(), taxis_prepexec() - SQL interface**SYNOPSIS**

```
int taxis_prepare(TEXIS *, char *sqlQuery);
int taxis_prepexec(TEXIS *, char *sqlQuery);
int taxis_execute(TEXIS *);
```

DESCRIPTION

These functions perform SQL statement setup and execution for databases opened with `taxis_open()`. `taxis_prepare()` takes a `TEXIS` pointer from `taxis_open()` and a SQL statement.

These functions provide an efficient way to perform the same SQL statement multiple times with varying parameter data. `taxis_prepare()` will return 1 on success, 0 on failure.

`taxis_execute()` and `taxis_fetch()` or `taxis_flush()` would then be called to handle the results of the statement as with `taxis_prepare()`.

Once a SQL statement is prepared with `taxis_prepare()` it may be executed multiple times with `taxis_execute()`. Typically the parameter data is changed between executions using the `taxis_param()` function.

`taxis_execute()` will start to execute the statement prepared with `taxis_prepare()`. `taxis_execute()` will return 1 on success, 0 on failure.

Parameters should be set between `taxis_prepare()` and `taxis_execute()`. If there are no parameters you can also use `taxis_prepexec()`, which will call `taxis_prepare()` and `taxis_execute()` in one call.

EXAMPLE

```
SERVER *se;
TEXIS   *tx;
long    date;
char    *title;
char    *article;
int     tlen, alen, dlen;

...
if(!taxis_prepare(tx,"insert into docs values(counter,?,?,?);"))
    { puts("taxis_prepare Failed"); return(0); }
for( each record to insert )
{
```

```

...
date=...
dlen=sizeof(date);
title=...
tlen=strlen(title);
article=...
alen=strlen(article);
if(!tesis_param(tx,1,&date ,&dlen,SQL_C_LONG,SQL_DATE ) ||
    !tesis_param(tx,2,title ,&tlen,SQL_C_CHAR,SQL_LONGVARCHAR) ||
    !tesis_param(tx,3,article,&alen,SQL_C_CHAR,SQL_LONGVARCHAR));
    { puts("tesis_param Failed"); return(0); }
if(!tesis_execute(tx))
    { puts("tesis_execute Failed"); return(0); }
tesis_flush(tx);
}
...

```

EXAMPLE

```

TEXIS      *tx;
char      *query;
int       qlen;
FLDLST *fl;

...
if(!tesis_prepare(tx,
    "select id,Title from docs where Article like ?;"))
    { puts("tesis_prepare Failed"); return(0); }
for( each Article query to execute )
{
    query=...
    qlen=strlen(query);
    if(!tesis_param(tx,1,query,&qlen,SQL_C_CHAR,SQL_LONGVARCHAR))
        { puts("tesis_param Failed"); return(0); }
    if(!tesis_execute(tx))
        { puts("tesis_execute Failed"); return(0); }
    while((fl=tesis_fetch(tx,-1))!=FLDLSTPN)
    {
        ...
    }
}
...

```

SEE ALSO

`taxis_open()`, `taxis_fetch()`

1.1.6 tesis_fetch() - SQL interface

SYNOPSIS

```
FLDLST *tesis_fetch(TEXIS *tx, int stringsFrom);
```

DESCRIPTION

`tesis_fetch()` returns a `FLDLST` pointer to the result set. The `stringsFrom` parameter lets you control if results are automatically converted to text. An argument value of -1 will not convert any result, other numbers will start converting from that result column, so passing 0 will cause all fields to be converted.

Continue calling `tesis_fetch()` to get subsequent result rows. `tesis_fetch()` will return `FLDLSTPN` when there are no more result rows.

EXAMPLE

```
TEXIS      *tx;
FLDLST *fl;

...
if((tx=tesis_open())!=TEXISPN)      /* initialize database connection */
{
    ...
                                /* setup query */
    if(tesis_prepare(tx,
        "select NAME from SYSTABLES where CREATOR!='tesis';"
    )!=TEXISPN)
        while((fl=tesis_fetch(tx))!=FLDLSTPN)/* get next result row */
        {
            dispfields(fl);          /* display all of the fields */
        }
    ...
    tesis_close(tx);                /* close database connection */
}
...
```

SEE ALSO

`tesis_prepare()`, `tesis_execute()`

1.1.7 n_settexasparam(), texas_param(), n_resetparamtx() - SQL interface**SYNOPSIS**

```

int texas_param( tx, ipar, buf, len, ctype, sqltype)
SERVER *se;
TEXIS *tx;
int ipar;
void *buf;
int *len;
int ctype;
int sqltype;

int texas_resetparams( tx)
SERVER *se;
TEXIS *tx;

```

DESCRIPTION

These functions allow you to pass arbitrarily large or complex data into a SQL statement. Sometimes there is data that won't work in the confines of the simple C string that comprises an SQL statement. Large text fields or binary data for example.

Call `texas_param()` to setup parameters after `texas_prepare()` and before `texas_execute()`. If you have a statement you have already executed once, and you want to execute again with different data, which may have parameters unset which were previously set you can call `texas_resetparams()`. This is not necessary if you will explicitly set all the parameters. Place a question mark (?) in the SQL statement where you would otherwise place the data.

These are the parameters:

TEXIS *tx The prepared SQL statement.

int ipar The number of the parameter, starting at 1.

void *buf A pointer to the data to be transmitted.

int *len A pointer to the length of the data. This can be `(int *)NULL` to use the default length, which assumes a `'\0'` terminated string for character data.

int ctype The type of the data. For text use `SQL_C_CHAR`.

int sqltype The type of the field being inserted into. For varchar use `SQL_LONGVARCHAR`.

| Field type | sqltype | ctype | C type |
|------------|-----------------|---------------|------------|
| varchar | SQL_LONGVARCHAR | SQL_C_CHAR | char |
| varbyte | SQL_BINARY | SQL_C_BINARY | byte |
| date | SQL_DATE | SQL_C_LONG | long |
| integer | SQL_INTEGER | SQL_C_INTEGER | long |
| smallint | SQL_SMALLINT | SQL_C_SHORT | short |
| float | SQL_FLOAT | SQL_C_FLOAT | float |
| double | SQL_DOUBLE | SQL_C_DOUBLE | double |
| varind | SQL_LONGVARCHAR | SQL_C_CHAR | char |
| counter | SQL_COUNTER | SQL_C_COUNTER | ft.counter |

EXAMPLE

```

SERVER *se;
TEXIS      *tx;
char      *description;
char      *article;
int       lend, lena;

if(!taxis_prepare(tx,"insert into docs values(?,?);"))
    { puts("taxis_prepare Failed"); return(0); }
...
description="a really large article";
lend=strlen(description);
article=...
lena=strlen(article);
if(!taxis_param(tx,1,description,&lend,SQL_C_CHAR,SQL_LONGVARCHAR) ||
    !taxis_param(tx,2,article,&lena,SQL_C_CHAR,SQL_LONGVARCHAR));
    { puts("taxis_param Failed"); return(0); }
if(!taxis_execute(tx))
    { puts("taxis_execute Failed"); return(0); }
...

```

1.1.8 taxis_flush() - SQL interface

SYNOPSIS

```
int taxis_flush(tx);  
TEXIS      *tx;
```

DESCRIPTION

This function flushes any remaining results from the current SQL statement. Execution of the statement is finished however. This is useful for ignoring the output of `INSERT` and `DELETE` statements.

EXAMPLE

```
SERVER *se;  
  
...                                     /* setup query */  
if(taxis_prepare(tx,  
    "delete from customer where lastorder<'1990-01-01';"  
    )!=TEXISPN)  
    taxis_flush(tx);                   /* ignore result set */  
...
```

1.1.9 texit_flush_scroll() - SQL interface

SYNOPSIS

```
int texit_flush_scroll(tx,nrows);
TEXIS      *tx;
int      nrows;
```

DESCRIPTION

This function flushes up to `nrows` results from the specified SQL statement. This is useful for skipping a number of records from a `SELECT`.

The return will be the number of records actually flushed if successful, or if an error occurred then the return will be a negative number of `(-1 - rowsflushed)`. Reaching the end of the results is not considered an error, and will result in a return less than `nrows`.

EXAMPLE

```
SERVER *se;

...
/* setup query */
if(texit_prepare(tx,
    "select name from customer where lastorder<'1990-01-01';"
    )!=TEXISPN)
    texit_flush_scroll(tx,10);    /* ignore first 10 results */
...
```

1.2 Modifying the server

1.2.1 adduserfuncs - Adding to the server

SYNOPSIS

```
#include <sys/types.h>
#include "dbquery.h"

void adduserfuncs(fo)
FLDOP *fo;
```

DESCRIPTION

This section describes how to add user defined types and functions to the texit server. The file `aufex.c` in the `api` directory provides an outline of how to do this. The server calls the function `adduserfuncs` which gives you the opportunity to add data types, operators and functions to the server. There are three functions that are used for adding to the server. They are `dbaddtype`, `foaddfuncs` and `fosetop`.

Once you have created your own version of `aufex.c` you need to compile and link this file with the rest of the daemon objects. You must make sure that this file is linked before the libraries to make sure your function gets called. An example makefile is also included in the `api` directory which shows the needed objects and include paths to make a new daemon. After a new daemon has been created, make sure that it is running and not the standard daemon. See the documentation on `texitd` to see how to invoke it.

1.2.2 dbaddtype - Add a datatype

SYNOPSIS

```
int dbaddtype(name, type, size)
char *name;
int type;
int size;
```

DESCRIPTION

Parameters

name the new name for the type. It should not start with the string “var”, as that is reserved for declaring the variable size form of the datatype.

type an integer which is used to refer to the type in functions etc. For a user added type this number should be between 32 and 63 inclusive. This number should be unique.

size the size of one element of the datatype. When one item of this type is written to the database, at least size bytes will be transferred.

The function will return 0 if successful, and -1 if there is no room for more datatypes, or if a type with a different name already exists with the same type number.

EXAMPLE

```
typedef struct tagTIMESTAMP
{
    short          year;
    unsigned short month;
    unsigned short day;
    unsigned short hour;
    unsigned short minute;
    unsigned short second;
    unsigned long  fraction;
} TIMESTAMP;

dbaddtype("timestamp", 32, sizeof(TIMESTAMP));
```

1.2.3 foaddfuncs - Add functions

SYNOPSIS

```
#include <sys/types.h>
#include "dbquery.h"

int foaddfuncs(fo, ff, n)
FLDOP *fo;
FLDFUNC *ff;
int n;
```

DESCRIPTION

Foaddfuncs adds a function to the math unit of Taxis. The function can take up to five arguments, and returns a single argument. The function will be called with pointers to FLD structures. The return values should be stuffed into the pointer to the first argument.

The math unit takes care of all the required stack manipulation, so no stack manipulation is required in the function. The math unit will always pass the maximum number of arguments to the function.

Foaddfuncs takes an array of function descriptions as one of its arguments. The functions description function looks like

```
struct {
    char *name;                /* name of function */
    int (*func)();              /* handler */
    int minargs;                /* minimum # of arguments allowed */
    int maxargs;                /* maximum # of arguments allowed */
    int rettype;                /* return type */
    int types[MAXFLDARGS];     /* argument types, 0 means don't care */
} FLDFUNC;
```

Parameters

fo The math unit to add to.

ff Array of function descriptions to be added.

n The number of functions being added.

EXAMPLE


```
int
fsqr(f)
FLD *f;
{
    int      x;
    size_t   sz;

    x = *(ft_int *)getfld(f, &sz);      /* Get the number */
    x = x * x ;                          /* Square it */
    putfld(f, x, 1);                    /* Put the result */
    return 0;
}

static FLDFUNC  dbfldfuncs[]=
{
    {"sqr",    fsqr, 1, 1, FTN_INT, FTN_INT, 0, 0, 0, 0 },
};
#define NFLDFUNCS (sizeof(dbfldfuncs)/sizeof(dbfldfuncs[0]))

foaddfuncs(fo, dbfldfuncs, NFLDFUNCS);
```

This will add a function to square an integer.

1.2.4 foasetop - Add an operator

SYNOPSIS

```
#include <sys/types.h>
#include "dbquery.h"

int foasetop(fo, type1, type2, func, ofunc)
FLDOP *fo;
int type1;
int type2;
fop_type func;
fop_type *ofunc;
```

DESCRIPTION

Foasetop changes a binary operator in the math unit of Tesis. The function `func` will be called for all operations on (type1, type2). If the function does not know how to handle the specific operation, and `ofunc` is not NULL, `ofunc` can be called to handle the operation.

The function being called should look like

```
int
func(f1, f2, f3, op)
FLD *f1;
FLD *f2;
FLD *f3;
int op;
{
}
```

Parameters

fo The math unit to change.

type1 The type of the first operand.

type2 The type of the second operand.

func Pointer to the function to perform the operation.

ofunc Pointer to pointer to function. The pointer to function will be stuffed with the old function to perform the operation. This can be used to add some cases, and keep the old functionality in others. The return value should be 0 for success. The result of the operation should be put in `f3`.

EXAMPLE

```

#include <sys/types.h>
#include "dbquery.h"

fop_type o_ftich;          /* the pointer to the previous handler */

int
n_ftich(f1, f2, f3, op)
FLD      *f1;
FLD      *f2;
FLD      *f3;
int      op;
{
    TIMESTAMP      *ts;
    double          d;
    int             n;

    if (op != FOP_ASN)      /* We only know about assignment. */
        if (o_ftich)
            return ((*o_ftich)(f1, f2, f3, op));
        else
            return FOP_EINVAL;

    /* Set up the return field. */
    f3->type = FTN_TIMESTAMP;
    f3->elsz = sizeof(TIMESTAMP);
    f3->size = sizeof(TIMESTAMP);
    f3->n = 1;
    if(sizeof(TIMESTAMP) > f3->allocated)
    {
        void *v;

        v = malloc(sizeof(TIMESTAMP));
        setfld(f2, v, sizeof(TIMESTAMP));
        f3->allocated = sizeof(TIMESTAMP);
    }

    /* First 0 out all the elements */
    ts = getfld(f1, NULL);
    ts->year = 0;
    ts->month = 0;
    ts->day = 0;
    ts->hour = 0;

```

```

    ts->minute = 0;
    ts->second = 0;
    ts->fraction = 0;

/* Now read in the values */
    n = sscanf(getfld(f2, NULL), "%hu/%hu/%hd %hu:%hu:%hu%lf",
               &ts->month, &ts->day, &ts->year,
               &ts->hour, &ts->minute, &ts->second, &d);

/* Convert any fractional seconds into the appropriate number
   of billionths of a second.
*/
    if (n == 7)
        ts->fraction = d * 1000000000 ;
}

.
.
.
    fasetop(fo, FTN_TIMESTAMP, FTN_CHAR, n_ftich, &o_ftich);
.
.
.

```

This example adds an operator to allow the assignment of a `TIMESTAMP` field from a character string. See `dbaddtype` for the definition of `TIMESTAMP`.