The Oracle Relational Algebra Package

Table of Contents

TABLE OF CONTENTS **1**

PACKAGE OVERVIEW **2**

DESCRIPTION 2
USING THE PACKAGE 2
RELATIONS 2
RELATION AND COLUMN NAMES 3

OPERATOR GUIDE **4**

PURPOSE 4
OPERATOR SYNTAX 4
FILTER *5*
PROJECT *6*
REDUCE *6*
GROUP *7*
TIMES *8*
COMPARE JOIN *8*
MATCH JOIN *9*
OUTER JOIN *10*
UNION, MINUS, INTERSECT, DIVIDE *11*
FULL MINUS *12*
ASSOCIATE (SETS) *13*

PACKAGE BEHAVIOR **15**

CONTROLLING PACKAGE BEHAVIOR 15
UNDERSTANDING PACKAGE ERRORS 17

APPENDIX A – PACKAGE SPECIFICATION **19**

APPENDIX B – ORACLE AND SQL*PLUS PRIMER **24**

APPENDIX C – SAMPLE SESSION & EXAMPLES **29**

A SAMPLE DATABASE 29
A SAMPLE SESSION 30
SOME OPERATOR EXAMPLES 31
MORE OPERATOR EXAMPLES 32

Package Overview

Description

This package was created in the fall of 1996 in response to the need for a relational algebra interface to the Oracle DBMS. It uses stored procedures written in PL/SQL to perform relational algebra operations. The operand and result relations are stored as populated database tables with primary keys. The relational algebra language implemented is fully described in the book, "Mastering Database Analysis" by John Carlis and Scott Krieger.

Using The Package

The package is a collection of stored procedures and functions which can be called from the SQL*Plus prompt on any Oracle database where the package is installed. The package can supports multiple users, each working from their own schema space. Each user can only work on relations for which they are the owner (those which exist in their schema).

The operator guide (next section) lists each operator and its arguments. Notice that some operators (such as GROUP) may be overloaded (can be called with a varying number of parameters). You can also view the specification for a particular function or procedure in SQL*Plus with the DESC command. Examples of the operators and a sample session are included in Appendix C.

The package is limited in its user interface and syntax, being bound to the functionality and syntax conventions used by Oracle SQL*Plus and PL/SQL. The best way to learn the syntax is to study the examples provided and to grab an Oracle book. If you are a novice Oracle user, you will want to look at the Oracle and SQL*Plus Primer (Appendix B) for information on Oracle, SQL*Plus and SQL commands and syntax.

Each operator has at least one corresponding function in the package (e.g. FILTER_RA is the function used to perform a FILTER operation). Each function call creates a result relation in the database. In addition, the function call returns a string containing the name of the relation created. Composition of functions can be achieved by using the result relation name returned by one function as the operand relation to another. The `ops.go()` procedure is used to catch the final relation name and print it. If your operator fits nicely on one line you can use the following syntax to execute your query:

```
SQL  exec ops.go ( - place query here - )
```

However, due to a "feature" of SQL*Plus, you must use a dummy block surrounding the call to ops.go() in order to use multiple lines to state your query. The dummy block is as follows (don't forget the backslash!):

```
SQL  DECLARE
SQL  BEGIN
SQL    ops.go( - place query here - );
SQL  END;
SQL  /
```

Relations

Keep in mind that every function returns a result relation, not just the last function in a query. Look at the following query:

```
SQL  DECLARE
SQL  BEGIN
SQL  ops.go(
        ops.project_ra(
          ops.filter_ra('creature','c_type=''person'''','person'),
            'c_id,'id_of_person'));
SQL  END;
SQL  /
```

The displayed output of this query will be the statement 'Result Relation: ID_OF_PERSON', but in fact two relations were created: PERSON and ID_OF_PERSON. PERSON was created in the call to filter_ra. The name of the result relation (PERSON) was passed to project_ra as the name of the operand relation. ID_OF_PERSON was created as a result of the call to project_ra.

Understanding that a relation has exactly one primary key (identifier) is very important. **The package will not perform properly unless each relation that you create has a proper primary key.** The good news is that each operator returns a result relation with the proper primary key.

Using tables instead of views has advantages and disadvantages. The disadvantage is speed. Physically creating and populating a table takes considerable time. However, we use tables instead of views for two very important reasons:

- A table can have a primary key, a view cannot;
- A view updates as data is changed in the operand relations - this is unacceptable when "debugging" queries.

The default action of the package is to fail when you specify a result relation name of a table that already exists. You can change this behavior using the overwrite() command, which is fully described in the Package Behavior portion of this guide.

It is important to understand how the package creates result relations to be effective in debugging your code. Most of the standard operators create a result relation in two separate steps. First the table is created and then the primary key is assigned. Therefore, it is possible for the table to be created successfully but for the primary key assignment to fail.

The commands to view the structure of a relation, the data in a relation, or a list of all relations in your user space are described in the Oracle and SQL*Plus Primer (Appendix B).

Relation and Column Names

Oracle enforces various rules on the naming of relations and columns that you need to be aware of. A table or column name (unfortunately called an identifier in the Oracle vocabulary) can be only 30 characters in length. Also, a name can have use letters, numbers, the underscore, dollar sign, and pound sign. A name cannot include spaces. The first letter of a name must be a letter. Names are not case sensitive.

The solution to this problem is not to use "fake" names for your relations like 't1', but instead to develop a naming standard and stick to it. So, instead of naming your relation 'swimming_creature_with_blue_hair_and_red_nose' perhaps you use 'SwimCreat_BHair_RNose' or something similar. Resist the temptation to use names that have no meaning!

Note: *Some operators attempt to "clean_up" the relation and column names that you input to avoid easily prevented errors. This is done by truncating names to 30 characters and replacing spaces with underscores. The package warns you when this occurs. Clearly, you are better served by passing correct names, rather than relying on the package to fix your names for you.*

Operator Guide

Purpose

The purpose of this section is to provide detailed instructions to the user on each of the relational algebra operators included in the package. The guide focuses on the meaning and syntax of each operator's arguments rather than the behavior of the operator, which is explained at length in the book "Mastering Database Analysis".

Note: *All of the operators have some syntactic elements in common. To avoid re-writing or re-referencing the same information many times, the guide is written in a manner that assumes that you have read the operators in order, so that only syntax which is new is introduced for each operator.*

Operator Syntax

The following information will be provided for each operator:

- Normal Specification: The specification of the arguments for the operator. Optional arguments are in italics.
- Package Specification: The specification for the function or functions that correspond to the operator.
- Argument Descriptions: A detailed description of each argument. Only arguments that have not already been described will be presented for each operator.
- Example: An example of a legal call to the operator. For a more extensive set of examples refer to Appendix C.

FILTER

FILTER (Operand Relation, Condition Clause, Result Relation)

```
FUNCTION filter_ra
  (  relation_a             VARCHAR,
     condition              VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;
```

Operand Relation: This argument takes as input a string (delimited by single quotes) containing the name of a relation. The name of the relation must be 30 characters or less and the relation must be owned by the user calling the operator. The relation name should be identical to the Oracle table name, so no spaces or other illegal characters can be used. The name is not case sensitive.

Condition Clause: This argument takes as input a string containing the selection criteria for each row. The string must be less than 2000 characters in length. The condition must be written in such a way as to be directly substitutable into a WHERE clause in an Oracle SQL statement. This means that you are limited to using commands recognized by Oracle SQL (including user defined functions) and must follow proper SQL syntactic rules. Remember that if the condition clause needs to contain single quotes, the string you pass to the operator must use two single quotes in place of the single quote. For example, if you want your condition in SQL to read *WHERE c_type='person'* you would pass the string *'c_type=''person'''* as the condition argument to the FILTER operator.

Result Relation: This argument is identical in syntax to the Operator Relation.

Example: `ops.filter_ra('creature','c_type=''person''','person')`

PROJECT

PROJECT (Operand Relation, Projected Column List, Result Relation)

```
FUNCTION project_ra
  (  relation_a             VARCHAR,
     column_list            VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;
```

Projected Column List: This argument takes as input a string containing a comma delimited list of column names and/or statements for computing new columns. In addition, each column or computational statement may have a new column name (alias) specified for it. An alias is necessary with any computed column and can also used to rename an existing column. A column name must correspond exactly with the name of a column in the operand relation. A computational statement must be formed using standard SQL syntax. An alias must adhere to column naming standards (30 characters or less, no special characters, etc). An alias is placed in front of the column name or computational statement, using an equals sign as a separator. A space before or after an equals sign or comma is optional.

The package also includes a "helper" function called allbut(). You pass it a relation name and a list of columns and it returns a list of all the columns in the relation *excluding* those you specify.

Examples:

```
ops.project_ra('creature','c_id, cid2 =c_id*2, c_type','creature_plus_cid2')

ops.project_ra('creature',ops.allbut('reside_town'),'creature_without_town_data')
```

REDUCE

REDUCE (Operand Relation, ID Column List, *Non-ID Column List*, Result Relation)

```
FUNCTION reduce_ra
  (  relation_a             VARCHAR,
     id_column_list         VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION reduce_ra
  (  relation_a             VARCHAR,
     id_column_list         VARCHAR,
     nonid_column_list      VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;
```

ID Column List: This argument is identical in syntax to the Projected Column List in Project. The columns in the ID Column List will serve as the identifier of the result relation.

Non-ID Column List: *[Optional]* Same syntax as the ID Column List. The columns in the Non-ID Column List will be present in the result relation but will not be part of the identifier.

Example: `ops.reduce_ra('creature','c_name','type=c_type','c_name_plus_type')`

GROUP

GROUP (Operand Relation, *Over Column List*, *Carry Column List*, Aggregate List, Result Relation)

```
FUNCTION group_ra
  (  relation_a             VARCHAR,
     over_column_list       VARCHAR,
     carrying_column_list   VARCHAR,
     aggregate_column_list  VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION group_ra
  (  relation_a             VARCHAR,
     over_column_list       VARCHAR,
     aggregate_column_list  VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION group_ra
  (  relation_a             VARCHAR,
     aggregate_column_list  VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;
```

The GROUP operator can be called with a varying number of arguments. If there are no carrying columns you may want to use the second syntax rather than passing a NULL for the carrying column list argument in the first syntax. Likewise for the third syntax, where there are no carrying columns or over columns.

Over Column List: *[Optional]* This argument takes as input a string containing a comma delimited list of column names from the operand relation. These are the columns which you want to group over. No column aliases are allowed in this list.

Carry Column List: *[Optional]* This argument takes as input a string containing a comma delimited list of column names from the operand relation. No column aliases are allowed in this list. These are the columns which you want to "carry" along with the over columns. Due to how the operator works, carry columns are actually treated no differently from over columns. Therefore, if you specify a column that is not act as a "carry" column, the operator will treat it as a “over” column (giving you the wrong result). For this reason, be very careful using carried columns.

Aggregate Column List: This argument takes as input a string containing a comma delimited list of aggregate functions. You must use the standard Oracle SQL aggregate functions in standard SQL syntax. Each aggregate function can (and should) be aliased using the "equals" notation introduced in Project.

Examples:

```
ops.group_ra('achievement','c_id','c_name','min_prof=min(score), max_prof=max(prof)',
'creature_plus_min_max_score')

ops.group_ra('achievement','c_id','avg_prof=avg(score)','creature_plus_avg_score')

ops.group_ra('achievement','avg_prof=avg(score)','avg_score')
```

TIMES

TIMES (Operand Relation A, Operand Relation B, Result Relation)

```
FUNCTION times_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;
```

Operand Relation A & Operand Relation B: These arguments take as input a string containing the name of a relation (just like in the unary operators). However, in some binary operators you can (and often should) specify a relation alias. The alias is useful for two reasons: 1) You may have the same relation serving as both operand relation inputs, so you *must* specify aliases for both. 2) You may want to shorten the name of the relation for ease or want to shorten the name of resulting columns when the relation name is prepended*. A relation alias is specified the same as a column alias, by placing the alias name and an equals sign before the relation name.

**** Note:*** *The package supports automatic name resolution. Naming resolution is necessary when two columns (one from each operand relation) share the same name, meaning one of them has to be renamed in the result relation. The various methods of naming resolution are described in Package Behavior section.*

Example: `ops.times_ra('creature','s = skill','creature_skill_pair')`

COMPARE JOIN

COMPARE JOIN (Operand Relation A, Operand Relation B, Condition Clause, Result Relation)

```
FUNCTION cjoin_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     condition          VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;
```

Condition Clause: This argument takes as input a string containing the join criteria in a format similar to that of the Filter operator (see above). However, in a join criteria you will need to reference columns from both tables. There are two things to remember about referencing columns: 1) If the same column appears in both relations you will need to reference the column by prefixing it with the relation name or alias followed by a period. 2) If a column will be automatically renamed to avoid naming conflicts, you need to reference the column by its original name, not the new one.

Examples:

```
ops.cjoin_ra('creature','skill','reside_town != origin_town', 'diff_town_creature_skill_pair')

ops.cjoin_ra('c1 = creature','c2 = creature','c1.c_id = c2.c_id','same_creature_pair')
```

MATCH JOIN

MATCH JOIN (Operand Relation A, Operand Relation B, A Column List, B Column List, *Result Column List*, Result Relation)

```
FUNCTION mjoin_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION mjoin_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;
```

A Column List & B Column List: These arguments represent the columns being joined on in relation A and relation B, respectively. Each list is a comma delimited list of column names without aliases. The lists must have the same number of columns. These lists are used to create the join condition automatically, so reference column names the same as you would in a join condition clause (see previous page).

Result Column List: *[Optional]* This list is identical in syntax to the A & B Column Lists, but is necessary only if one or more of the columns will be renamed as a result of the operation. You must specify result names for all of the columns being joined on, so the result column list must have the same number of columns as the A column list and the B column list.

Example:

```
 ops.mjoin_ra('c=creature','a=achievement','c_id,reside_town','c_id,test_town','c_id,town',
'same_town_ach_plus_creature')
```

OUTER JOIN

OUTER JOIN (Operand Relation A, Operand Relation B, A Column List, B Column List, *Result Column List*, Result Relation)

```
FUNCTION ojoin_left_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_left_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_right_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_right_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_both_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_both_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;
```

Arguments are identical to Match Join. Examples are in the same syntax as Match Join.

UNION, MINUS, INTERSECT, DIVIDE

UNION (Operand Relation A, Operand Relation B, Result Relation)
MINUS (Operand Relation A, Operand Relation B, Result Relation)
INTERSECT (Operand Relation A, Operand Relation B, Result Relation)
DIVIDE (Operand Relation A, Operand Relation B, Result Relation)

```
FUNCTION union_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION minus_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION intersect_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION divide_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;
```

Operand Relation: The same as an operand relation in a unary (not binary) operator. This means that aliases are not allowed.

Note: *The columns must be in the same physical order in both operand relations for the set operators to work properly. It is always safest to use a project on each operand relation prior to performing a set operator to ensure that the columns match properly.*

Example: `ops.intersect_ra('achievement','aspiration','achievement_and_aspiration')`

FULL MINUS

FULL MINUS (Operand Relation A, Operand Relation B, *A Column List*, *B Column List*, Result Relation)

```
FUNCTION full_minus_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION full_minus_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;
```

A Column List & B Column List: *[Optional]* These arguments represent the columns that would be present in the A and B relations if a normal Minus were taking place. Each list is a comma delimited list of column names without aliases. The lists must have the same number of columns. The columns need not have the same names, as the columns in relation A are always used in the result. If these lists are not populated they default to being the id of the B relation.

Example:

```
ops.go(ops.full_minus_ra('achievement',ops.project_ra(ops.filter_ra('creature','c_type=''person''
','person'),'c_id','c_id_of_person'),'non-person_achievement'));
```

ASSOCIATE (SETS)

ASSOCIATE (Target Qualifier, Target Element Qualifier, Target Relation, Target Element Relation, Element Relation, Pattern Element Relation, Pattern Relation, Result Relation)

```
FUNCTION assoc_ra
  (  target_qual         VARCHAR,
     target_element_qual VARCHAR,
     target_rel          VARCHAR,
     target_element_rel  VARCHAR,
     element_rel         VARCHAR,
     pattern_element_rel VARCHAR,
     pattern_rel         VARCHAR,
     result_relation     VARCHAR
  ) RETURN VARCHAR;

FUNCTION assoc_ra
  (  target_qual         VARCHAR,
     target_rel          VARCHAR,
     target_element_rel  VARCHAR,
     element_rel         VARCHAR,
     result_relation     VARCHAR
  ) RETURN VARCHAR;
```

Target Qualifier: This argument is either a DEMONS-ZA letter string (single-quoted) or a Counts expression (single-quoted) involving any of the counts QCOUNT (qualifying count, the count of detail items present in both the target-detail relation and the pattern-detail relation for a given target instance), NQCOUNT (non-qualifying count, the count of detail items present in the target-detail relation but not in the pattern-detail relation, for a given target instance), MQCOUNT (missing-from-qualifying count, the count of detail items present in the pattern-detail relation but not in the target-detail relation, for a given target instance) or EXCOUNT (exact pattern-detail count, the count of detail items present in the pattern-detail relation for a given target instance).

Target Element Qualifier: CURRENTLY NOT USED. This argument is a placeholder for a Range Associate expression, to be implemented in a future version. For this release, this argument should be NULL.

Target Relation: This argument is a string containing the name of the relation holding the target instances, e.g. creature.

Target Element Relation: This argument is a string containing the name of the relation holding the target element instances, e.g. achievement or creature_skill_pair.

Element Relation: This argument is a string containing the name of the lookup relation holding the element instances, e.g. skill. It is not needed for a basic DEMONS-ZA or counts expression Associate query, and in these cases can be specified as NULL. If an Element relation is specified and a Weight column is present, the Associate output will contain weighting calculations for the target-pattern pairs generated.

Pattern Element Relation: This argument is a string containing the name of the relation holding the pattern-element instances, e.g. job_skill.

Pattern Relation: This argument is a string containing the name of the relation holding the pattern relation, e.g. job.

Examples:

```
-- DEMONS-ZA associate
ops.go(ops.assoc_ra('EM',NULL, 'creature','achievement',NULL,'job_skill','job',
'creature_job_pair_with_creature_achieving_every_or_more_than_the_skills_for_a_job'));


-- counts expression associate
ops.go(ops.assoc_ra('((QCOUNT >= MQCOUNT) AND (NQCOUNT > 0))',NULL,
'creature','achievement',NULL,'job_skill','job',
'creature_job_pair_with_creature_achieving_as_many_or_more_skills_than_they_are_missing_and_no_no
nqualifying_skills_for_a_job'));


-- weighted associate with modified element relation holding a weight for importance of each
skill
ops.go(ops.assoc_ra('EM',NULL, 'creature','achievement','skill','job_skill','job',
'same_as_first_example_but_with_weighted_results'));


-- self associate
ops.go(ops.assoc_ra('OS',NULL, 'creature','achievement',NULL,
'creature_creature_pair_where_first_creature_has_some_or_overlapping_skills_with_second_creature'
));
```

Package Behavior

Controlling Package Behavior

Certain behaviors of the package are configurable by the user. These behaviors include what type of messages and other information are displayed, whether or not a result relation is created, and whether or not an existing relation is automatically overwritten by the package. The user can change the settings for these behaviors using a set of procedure calls in the package, but the settings revert to their default setting each time the user logs in. You can put these procedure calls into your oracle startup script as well, if you wish for the same settings each time you log in.

Displayed Output

The package produces a number of different types of output for display to the user. The package allows you to specify which of these types of output are printed on the screen. There are six types of output:

Normal	Normal messages give the minimum amount of information necessary to follow the operation of the package operators. An example of a normal message is displaying the name of the result relation of each query passed to the go() function.
Warning	Warning messages indicate that some unexpected action was taken in order to continue execution. These messages do not necessarily mean that the operator failed to produce the correct result, just that something unusual happened that should be investigated. Examples include having to clean up relation or column names and working on a relation with no rows.
SQL	Each operator generates a set of SQL code that is executed to produce the result relation. Enabling the SQL display allows you to see the SQL statement that was produced. This is very helpful when trying to learn how relational algebra compiles to SQL or when you want to tune the SQL and run it yourself later.
Debug	Debug messages are displayed are major breakpoints in the code to help identify. You should never need to display these messages.
Timing	Timing messages are used solely for benchmarking, usually in addition to Debug messages. You should never need to display these messages.
Error	Error messages indicate that the package has failed to execute the specified command. Error messages are *always* displayed. A full description of error messages in given later in this section.

By default, normal messages and warning messages are displayed and SQL code, debugging messages, and timing messages are not. Error messages are always displayed. You can set a different display behavior using the display() command. The specification and syntax is as follows, where flag is expecting a 1 (enable) or a 0 (disable).

display(normal_flag, warning_flag, sql_flag, debug_flag, timing_flag)

```
SQL exec ops.display(1,1,0,0)
```

Note: *Oracle generates output using a buffer-based system. This has two consequences. First, if your output exceeds 2000 characters (in one script, operation, etc.) the DBMS will return an error. This means that you should watch how much output you are enabling, particularly on long queries. Second, the output is not displayed until the termination of the operation. Therefore debug statements will not show up as they are executed but only when the operation completes (or fails).*

Execution

An operator creates a result relation by generating an SQL command to execute. The display command just described is used to specify whether or not the SQL code is displayed. You can also set whether or not that code is executed. This of course means that no result relations are created as a result of your code. Note that this also limits you to executing one command at a time, since the result relation of one operator will not be available as the operand to the next operator and the operation will fail. The only reason to turn execution off is to test code or to produce SQL statements which you want to manually refine before executing. The command to change this behavior is execution(flag), using 1 to enable execution and 0 to disable execution.

```
SQL exec ops.execution(1)
```

Overwrite

Enabling this feature instructs the package to automatically drop and recreate a table if it has the same name as the result relation you are trying to create. This is sometimes very valuable (such as when you are debugging your query and keep needing to drop the "failed" tables and re-execute the query). However, for safety the default setting is to abort and return an error when finding an existing relation of the same name. The command to change this behavior is overwrite(flag), using 1 to enable overwriting and 0 to disable overwriting.

```
SQL exec ops.overwrite(1)
```

Naming Resolution

Automatic naming resolution is one of the most powerful features available in the package, but it is also one of the most complex issues to understand. When executing "Times-like" operators you can alter the way in which the package attempts to resolve column naming conflicts. Column naming conflicts occur when a column in one operand relation has the same name as a column in the other operand relation, but both columns need to appear in the result relation (meaning at least one needs to be renamed). The easiest way to avoid dealing with the packages automatic naming resolution is to make sure that all of the column names are unique (across both relations) before calling the operator.

If the package finds a naming resolution conflict it can deal with it in one of five ways:

1	**Fail**	The package returns an error if a naming conflict occurs. This is the only sure way to ensure that no columns get renamed without your explicit instructions.
2	**Minimal**	The package does the minimal amount necessary to resolve the naming conflict, in this case renaming one of the two columns in conflict. The package always renames the column from the A relation by prepending the relation name or alias of the A relation to the column name, separated by the package default delimiter (see Set Delimiter below).
3	**Normal**	The package renames both of the conflicting columns by prepending their respective relation names or aliases.
4	**Left**	The package renames every column in the A relation by prepending the relation name or alias.
5	**Full**	The package renames every column in both relation by prepending their respective relation names or aliases.

Note that for one call to an operator it is possible that one naming resolution may produce a successful result while another may fail. Make sure you carefully think out the consequences of using a particular technique. Keep in mind that prepending the relation name to one column name might result in a conflict with another existing column 0- a situation the package does not attempt to deal with. Again, the easiest way to avoid difficulties with automatic naming resolution is to rename the columns yourself before executing the operation.

You can set the naming resolution technique using the resolution(technique_number) procedure. The argument is an integer (1-5), corresponding to the numbers in the list of techniques above. Notice that the higher the number the more columns are renamed. The default technique is "normal" (3). For example, to set the resolution technique to "Minimal", you would type:

```
SQL exec ops.resolution(2)
```

Set Delimiter

In various places in the package, such as when prepending table or alias names or when replacing illegal characters in a name, a delimiter (the default is the underscore) is used to construct a legal name. You can set any string of up to three characters (use one if possible) as your standard delimiter. You may also choose to have no delimiter by passing NULL or a blank string as the argument. Use this command only if underscore presents problems for you. The command is set_delimiter(new_delimiter).

```
SQL exec ops.set_delimiter('+')
```

Understanding Package Errors

The package has some built in error handling. The intent is to trap the most common errors returned when using the package and provide useful error messages to the user, instead of the sometime cryptic Oracle errors. Other errors are detected but the package has no valuable information to add. Finally, some errors may be completely undetected by the package and the DBMS will return the error directly to the user.

Error Types

Errors which are handled by the package will return an error message similar to the following:

```
*
ERROR at line 1:
ORA-20001: RA-VALID-02: Relation CREATURE does not exist.
ORA-06512: at "RASTAMAN.OPS_UTIL", line 457
ORA-06512: at "RASTAMAN.OPS_UTIL", line 598
ORA-06512: at line 1
```

The ORA-20001 error code comes from the Oracle DBMS and states that a user-defined exception has occurred. All package errors will have this error number. The RA-VALID-02 is a package assigned error code and the message that follows it is generated by the package. The remaining lines (beginning with the ORA-06512) specifies the location in the code where the error occurred, which will not be helpful to you as you do not have access to the code.

A "unexpected" error which is trapped by the package will have an identical output, but the package error code will always have 00 as the last two digits, the message will say 'Unknown Error:' and an Oracle error number and message will be provided, as shown here:

```
*
ERROR at line 1:
ORA-20001: RA-VALID-00: Unknown Error: ORA-01403: no data found
ORA-06512: at "RASTAMAN.OPS_UTIL", line 457
ORA-06512: at "RASTAMAN.OPS_UTIL", line 606
ORA-06512: at line 1
```

A completely unhandled error will return an Oracle error number and message, without any ORA-20001 error or package error number, as shown here:

```
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "RASTAMAN.OPS_UTIL", line 594
ORA-06512: at line 1
```

Understanding the Error

If you receive an error from the package with a package error message it is very likely that the error was a result of user error. Use the error message to help you solve the problem. Other errors are also typically a result of user errors in syntax or specifying "invalid" operations. If you think you have found an error in the package itself you can leave a message on the Oracle Relational Algebra Package homepage.

Appendix A – Package Specification

```
**********************************
RELATIONAL ALGEBRA PACKAGE FOR ORACLE - RELEASE 2.3.2
Copyright 1996-2014 Scott Krieger, John Carlis & Paul Wagner;
Also Regents of the University of Minnesota

Any use, modification, or distribution of this package outside of the CSCI 5702
course without the permission of the author is strictly prohibited.  All remote
copies of the package must be removed at the conclusion of the 5702 course.

To use:
  Type 'set serveroutput on size 1000000'  - this allows for package output

  Execute your queries in the following format (don't forget the /):

  DECLARE
  BEGIN
    ops.go(  );
  END;
  /

**********************************
*/


/* ***************
   GLOBAL VARIABLES
   ***************

ra_display:    Specifies whether or not display, warning, SQL, debug, and timing
ra_warning     statements (respectively) are displayed to the user.  Values are
ra_sql         changed with the DISPLAY() procedure.
ra_debug
ra_timing

ra_execute:    Specifies whether or not the code generated by the package
               should be executed.  Value is changes with the EXECUTE_ON and
               EXECUTE_OFF procedures.
ra_overwrite:  Specifies whether or not an existing relation is automatically
               overwritten (removed) by a newly produced relation.
ra_delimiter:  Specifies the delimiter to be placed between a table name and a
               column name when naming resolution is being handled by the
               package.
ra_resolution: Specifies the automatic naming resolution strategy to be used when
               executing TIMES-like operators.

*/

  ra_display       BOOLEAN := TRUE;
  ra_warning       BOOLEAN := TRUE;
  ra_sql           BOOLEAN := FALSE;
  ra_debug         BOOLEAN := FALSE;
  ra_timing        BOOLEAN := FALSE;
  ra_execute       BOOLEAN := TRUE;
  ra_overwrite     BOOLEAN := FALSE;
  ra_delimiter     VARCHAR(3) := '_';
  ra_resolution    INTEGER := 3;  -- (1-Fail, 2-Minimal, 3-Normal, 4-Left, 5-Full)


/* *****************
   INTERFACE COMMANDS
   *****************
*/


PROCEDURE go (result_relation VARCHAR);
PROCEDURE display (normal_flag INTEGER, warning_flag INTEGER,
```

```
                  sql_flag INTEGER, debug_flag INTEGER, timing_flag INTEGER);
PROCEDURE execution (execution_flag INTEGER);
PROCEDURE overwrite (overwrite_flag INTEGER);
PROCEDURE set_delimiter (new_delimiter VARCHAR);
PROCEDURE resolution (resolution_option INTEGER);

/* *****************
   GLOBAL FUNCTIONS
   *****************

   These functions are defined and used within the package, but may also be
   useful for general purpose use in SQL functions, and so are made available
   to the user for non-package usage.
*/

FUNCTION allbut (relation VARCHAR, excluded_list VARCHAR) RETURN VARCHAR;
FUNCTION allcols (relation VARCHAR) RETURN VARCHAR;


 /* ************************
    STANDARD UNARY OPERATORS
    ************************

    The standard unary (one operand relation) operators are:
      Filter, Project, Reduce and Group
*/

FUNCTION filter_ra
  (  relation_a             VARCHAR,
     condition              VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

/* Select maintained for compatability with release 2.3.1 and earlier */
FUNCTION select_ra
  (  relation_a             VARCHAR,
     condition              VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION project_ra
  (  relation_a             VARCHAR,
     column_list            VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION reduce_ra
  (  relation_a             VARCHAR,
     id_column_list         VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION reduce_ra
  (  relation_a             VARCHAR,
     id_column_list         VARCHAR,
     nonid_column_list      VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION group_ra
  (  relation_a             VARCHAR,
     over_column_list       VARCHAR,
     carrying_column_list   VARCHAR,
     aggregate_column_list  VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION group_ra
  (  relation_a             VARCHAR,
     over_column_list       VARCHAR,
     aggregate_column_list  VARCHAR,
```

```
     result_relation        VARCHAR
  ) RETURN VARCHAR;

FUNCTION group_ra
  (  relation_a             VARCHAR,
     aggregate_column_list  VARCHAR,
     result_relation        VARCHAR
  ) RETURN VARCHAR;


 /* ************************
    STANDARD BINARY OPERATORS
    ************************

    The standard binary (two operand relations) operators are:
      Union, Intersect, Minus, Divide, Times, Compare Join, Match Join,
      and Outer Join (right, left, and outer), FullMinus
*/

FUNCTION times_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION cjoin_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     condition          VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION mjoin_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION mjoin_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_left_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_left_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_right_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
```

```
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_right_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_both_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION ojoin_both_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_col_list    VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION union_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION minus_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION intersect_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION divide_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION full_minus_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

FUNCTION full_minus_ra
  (  relation_a         VARCHAR,
     relation_b         VARCHAR,
     a_col_list         VARCHAR,
     b_col_list         VARCHAR,
     result_relation    VARCHAR
  ) RETURN VARCHAR;

/* ************************
    ADVANCED OPERATORS
    ************************
```

```
    The advanced operators are:
      Associate (Sets; formerly HAS); supporting basic DEMONS-ZA or counts expression associate;
efficient Self Associate, weighted Associate, Range Associate, and Vary Associate)
*/

-- general Associate (DEMONS-ZA, counts, weighting, range, vary)
FUNCTION assoc_ra
  (  target_qual         VARCHAR,
     target_element_qual VARCHAR,
     target_relation     VARCHAR,
     target_element_rel  VARCHAR,
     element_rel         VARCHAR,
     pattern_element_rel VARCHAR,
     pattern_rel         VARCHAR,
     result_relation     VARCHAR
  ) RETURN VARCHAR;

-- self Associate (optimized)
FUNCTION assoc_ra
  (  target_qual         VARCHAR,
     target_relation     VARCHAR,
     target_element_rel  VARCHAR,
     element_rel         VARCHAR,
     result_relation     VARCHAR
  ) RETURN VARCHAR;


END;
```

Appendix B – Oracle and SQL*Plus Primer

Introduction

The intent of this primer is to give you a very quick introduction on the commands available in SQL*Plus, the client of the Oracle DMBS. In addition, it will give you some insight into how to execute SQL queries in Oracle and the system tables you have access to. The information contained in this primer should be sufficient for you to do any of the examples shown in the accompanying book, "Mastering Database Analysis". This document is in no way intended to replace the documentation that Oracle provides or to teach you SQL syntax. Please refer to Oracle's documentation or other appropriate references if you have further questions.

SQL in Oracle

Here are some quick tips about using SQL*Plus and SQL in Oracle:

- SQL statements (SELECT, ALTER, etc.) must end with a semicolon, but SQL*Plus commands (like SPOOL and SET) do not.
- You will probably want to write you code in an editor rather than in SQL*Plus. If you need to make a little tweak to the previous command, the function is ch/*old-string*/*new-string* to do a search & replace.
- The aggregate functions for GROUP that you are likely to use are AVG, COUNT, MAX, MIN, SUM. There is also GLB, LUB, STDDEV, and VARIANCE.

Datatypes and Comparisons

Names A table or column name (unfortunately called an identifier in the Oracle vocabulary) can be only 30 characters in length. Also, a name can have use letters, numbers, the underscore, dollar sign, and pound sign. A name cannot include spaces. The first letter of a name must be a letter. Names are not case sensitive.

Strings Strings are delimited by single quotes, not double quotes. If the string you are passing as an argument requires the use of a single quote (e.g. to produce a condition such as: WHERE c_type='person'), use two single quotes in place of the single quote. Look at the following example. Notice that the condition string is enclosed in single quotes, and the string *person* is enclosed with two single quotes.

```
ops.filter_ra('creature','c_type=''person''','person_creature')
```

Dates Dates are passed enclosed in single quotes as well. The proper syntax is 'DD-MON-YYYY', where MON is the three letter abbreviation for the month (e.g. 25-SEP-1998).

Conversion It is possible to convert between datatypes in an SQL statement. The relevant commands are TO_CHAR, TO_DATE, and TO_NUMBER. Read the Oracle documentation for a complete description.

NULL When comparing for NULL in an SQL statement you do not use "WHERE column = NULL" or "WHERE column != NULL". Instead use "WHERE column IS NULL" or "WHERE column IS NOT NULL".

Creating Tables

Creating a Table

To create a table in Oracle you use the CREATE TABLE command, like in the example below. The exact formatting is not important, only that you have a comma-delimited list of columns with their datatypes.

```
CREATE TABLE creature
( c_id          INTEGER,
  c_name        VARCHAR(20),
  c_type        VARCHAR(20),
  reside_town   VARCHAR(2)
);
```

Another way to create a table is via a query, using the syntax CRATE TABLE *table_name* AS *query*. An example is shown below. This is the construct that the package uses to create result relations.

```
CREATE TABLE person AS SELECT * FROM creature WHERE c_type='person';
```

Specifying the Identifier

Remember, a relation must have one and only one unique *identifier* (called a the *primary key* by Oracle), which may be comprised of multiple columns. Oracle has an arbitrary limit of columns involved in an identifier - at one time the limit was 7.

There are a few different ways to assign the primary key for a relation. You can specify the primary key in the CREATE TABLE statement itself . The different methods are shown here:

One Column Primary Key

```
CREATE TABLE creature
( c_id          INTEGER     PRIMARY KEY,
  c_name        VARCHAR(20),
  c_type        VARCHAR(20),
  reside_town   VARCHAR(2)
);
```

Multi-Column Primary Key

```
CREATE TABLE achievement
( c_id        INTEGER,
  s_code      VARCHAR(1),
  score       INTEGER,
  test_town   VARCHAR(2),
  PRIMARY KEY (c_id, s_code)
);
```

You can also specify the primary key after the table has been created using the ALTER TABLE *table_name* ADD PRIMARY KEY (*list_of_columns_in_key*), as shown in the next example. This is the construct that the package uses to assign identifiers to result relations.

```
ALTER TABLE achievement ADD PRIMARY KEY (c_id, s_code);
```

Populating a Table

To populate a table you must add rows one at a time using the INSERT command. The syntax is INSERT INTO *table_name* VALUES (*list_of_values)* . The order of the values must correspond to the physical order of the columns in the CREATE TABLE statement.

```
INSERT INTO creature VALUES (1,'Bannon','person','p');
INSERT INTO creature VALUES (2,'Myers','person','a');
```

Schema and Table Management

Action	Command	Output
List tables in schema	SELECT TABLE_NAME FROM USER_TABLES; Note that TABLE_NAME is the actual column name, not a variable for an actual table name.	SQL> SELECT TABLE_NAME FROM USER_TABLES; TABLE_NAME ------------------------------ ACHIEVEMENT ASPIRATION CREATURE SKILL TOWN
Describe table structure	DESC *table_name* Unless you have added additional constraints to your tables, the NOT NULL will tell you which columns are part of the primary key of the table.	SQL> DESC creature Name Null? Type --------------- -------- ---- C_ID NOT NULL NUMBER(38) C_NAME VARCHAR2(40) C_TYPE VARCHAR2(40) RESIDE_TOWN VARCHAR2(2)
Display table contents	SELECT * FROM *table_name*	SQL> SELECT * FROM creature; C_ID C_NAME C_TYPE RESIDE_TOWN ---- ---------- ---------- ------------ 1 Bannon person p 2 Myers person a 3 Dougherty person b 4 Neff person c 5 Mieska person d 6 Carlis person p 7 Kermit frog h 8 Godzilla monster t
Drop existing table	DROP TABLE *table_name*; You will need to do this anytime you rerun a query using the relational algebra package in order to drop the tables previously created with the same name. You should also drop tables once you no longer need them to conserve disk space.	SQL> DROP TABLE creature;
Delete rows from table	DELETE FROM *table_name* WHERE *condition*;	SQL> DELETE FROM creature WHERE c_id=9;
View function	DESC *package.function*	SQL> DESC ops.filter_ra Argument Name Type In/Out Default? ------------ ---------- ------ -------- RELATION1 VARCHAR2 IN CONDITION VARCHAR2 IN RESULT_NAME VARCHAR2 IN

Controlling Output Appearance

To enable package output to the screen

Use the following command to enable message to be displayed in SQL*Plus (you should put this in your Oracle startup script). You can specify any size you want (in bytes) up to 1 million.

```
SET SERVEROUTPUT ON SIZE 1000000
```

To format SQL output to look nice:

COLUMN *column_name* FORMAT *type_and_length*

For a text column, use "a#" for the type and length. "A" stands for ASCII and you replace the # with the number of characters you want displayed. The result will be truncated to the size you specify. The format specification is used **before** you run the SQL command. The formatting applies to any column of the specified name for the duration of the session. Here are the commands used to produce the formatting in the example above:

```
SQL COLUMN c_name FORMAT a15
SQL COLUMN c_type FORMAT a10
SQL COLUMN reside_town FORMAT a12
```

For a numeric column, you must specify a "template" for the result. You use a '9' to indicate a number and a '0' to represent a number of a blank leading or trailing position. Check out the examples below:

```
FORMAT:     9      09     9.0

RESULT:   C_ID    C_ID    C_ID
          ----    ----    ----
             1      01     1.0
             2      02     2.0
             3      03     3.0
             4      04     4.0
             5      05     5.0
             6      06     6.0
             7      07     7.0
             8      08     8.0
```

Note that these commands only change the appearance of the data, it does not change the underlying representation (c_id is still an integer).

Setting other formatting variables in SQL*Plus:

Use the Oracle (SQL*Plus) documentation for a summary of the other variables you can set. Some other useful commands in order to improve output appearance are SET PAGESIZE 0 (which makes the heading print only once) and SET LINESIZE # (which sets the number of characters in a line).

Spooling and Script Files

To spool output to a file:

Use the following command to begin spooling the output of SQL*Plus to a file:

SPOOL *pathname*

Use the following command to end spooling:

SPOOL OFF

To execute a script file

You can save your queries in an SQL script file (with the .sql extension) and execute it using the @ command, as in the following example:

```
SQL @path/myscript.sql
```

You can place comments in a script file using -- for single line comments or using /* and */ for multi-line comments.

```
-- This is a comment in a script file
/* This is a
   multi-line comment */
```

Appendix C – Sample Session & Examples

A Sample Database

Here are some sample relations that you may recognize from the book. These relations are intended to show you how to create relations in Oracle, as well as to give you some relations to start testing the operators on. Please keep in mind that the relations change throughout the book, so running operators on these relations may result in slightly different results than those in the book.

A similar set of commands is available as a SQL script and is included with the RA package.

```
CREATE TABLE town
( t_id    VARCHAR(2)   PRIMARY KEY,
  t_name  VARCHAR(20)
);

INSERT INTO town VALUES ('a','Anoka');
INSERT INTO town VALUES ('b','Bemidji');
INSERT INTO town VALUES ('bl','Blue Earth');
INSERT INTO town VALUES ('c','Chaska');
INSERT INTO town VALUES ('d','Duluth');
INSERT INTO town VALUES ('em','Embarrass');
INSERT INTO town VALUES ('e','Edina');
INSERT INTO town VALUES ('h','Hollywood');
INSERT INTO town VALUES ('p','Phily');
INSERT INTO town VALUES ('s','Swampville');
INSERT INTO town VALUES ('t','Toyko');

CREATE TABLE creature
( c_id          INTEGER      PRIMARY KEY,
  c_name        VARCHAR(20),
  c_type        VARCHAR(20),
  reside_t_id   VARCHAR(2)   REFERENCES town(t_id)
);

INSERT INTO creature VALUES (1,'Bannon','person','p');
INSERT INTO creature VALUES (2,'Myers','person','a');
INSERT INTO creature VALUES (3,'Neff','person','b');
INSERT INTO creature VALUES (4,'Neff','person','c');
INSERT INTO creature VALUES (5,'Mieska','person','d');
INSERT INTO creature VALUES (6,'Carlis','person','p');
INSERT INTO creature VALUES (7,'Kermit','frog','h');
INSERT INTO creature VALUES (8,'Godzilla','monster','t');


CREATE TABLE skill
( s_code          VARCHAR(1)    PRIMARY KEY,
  s_description   VARCHAR(20),
  origin_t_id     VARCHAR(2)    REFERENCES town(t_id)
);

INSERT INTO skill VALUES ('A','float','b');
INSERT INTO skill VALUES ('E','swim','b');
INSERT INTO skill VALUES ('O','sink','t');
INSERT INTO skill VALUES ('U','walk on water','em');
INSERT INTO skill VALUES ('Z','gargle','p');


CREATE TABLE achievement
( c_id        INTEGER       REFERENCES creature(c_id),
  s_code      VARCHAR(1)    REFERENCES skill(s_code),
  score       INTEGER,
  test_t_id   VARCHAR(2)    REFERENCES town(t_id),
  PRIMARY KEY (c_id, s_code)
```

```
);

INSERT INTO achievement VALUES (1,'A',1,'a');
INSERT INTO achievement VALUES (1,'E',3,'a');
INSERT INTO achievement VALUES (1,'Z',3,'p');
INSERT INTO achievement VALUES (2,'A',3,'b');
INSERT INTO achievement VALUES (3,'A',2,'b');
INSERT INTO achievement VALUES (3,'Z',1,'p');
INSERT INTO achievement VALUES (4,'A',2,'c');
INSERT INTO achievement VALUES (4,'E',2,'c');
INSERT INTO achievement VALUES (5,'Z',3,'d');
INSERT INTO achievement VALUES (7,'E',1,'s');
INSERT INTO achievement VALUES (8,'O',1,'t');

CREATE TABLE aspiration
( c_id        INTEGER       REFERENCES creature(c_id),
  s_code      VARCHAR(1)    REFERENCES skill(s_code),
  score       INTEGER,
  test_t_id   VARCHAR(2)    REFERENCES town(t_id),
  PRIMARY KEY (c_id, s_code)
);

INSERT INTO aspiration VALUES (1,'A',1,'a');
INSERT INTO aspiration VALUES (1,'E',3,'b');
INSERT INTO aspiration VALUES (1,'Z',1,'bl');
INSERT INTO aspiration VALUES (2,'A',3,null);
INSERT INTO aspiration VALUES (3,'A',2,'b');
INSERT INTO aspiration VALUES (3,'Z',2,'bl');
INSERT INTO aspiration VALUES (4,'E',2,'c');
INSERT INTO aspiration VALUES (5,'Z',3,'d');
INSERT INTO aspiration VALUES (6,'Z',3,'e');
INSERT INTO aspiration VALUES (7,'E',3,'s');
INSERT INTO aspiration VALUES (8,'O',1,'t');

COMMIT;
```

A Sample Session

Here is a short sample session that takes steps through logging in to Oracle, executing a operator, and viewing the results. This session assumes that you have already created the CREATURE relation from the schema presented above.

```
% sqlplus

SQL*Plus: Release 3.3.2.0.0 - Production on Mon Sep 28 13:04:24 1998

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Enter user-name: myteam@test
Enter password:

Connected to:
Oracle7 Server Release 7.3.2.2.0 - Production Release
PL/SQL Release 2.3.2.2.0 - Production

SQL> set serveroutput on size 1000000

SQL> desc creature
 Name                           Null?    Type
 ------------------------------ -------- ----
 C_ID                           NOT NULL NUMBER(38)
 C_NAME                                  VARCHAR2(10)
 C_TYPE                                  VARCHAR2(10)

SQL> select * from creature;
```

```
      C_ID C_NAME     C_TYPE
---------- ---------- ----------
         1 Bannon     person
         2 Myers      person
         3 Dougherty  person
         4 Neff       person
         5 Mieska     person
         6 Carlis     person
         7 Kermit     frog
         8 Godzilla   monster

8 rows selected.

SQL>  DECLARE
  2   BEGIN
  3   ops.go(ops.filter_ra('creature','c_type=''person''','person'));
  4   END;
  5  /
Executing: CREATE TABLE MYTEAM.person AS  SELECT * FROM MYTEAM.CREATURE WHERE c_type='person'
Executing: ALTER TABLE MYTEAM.person ADD PRIMARY KEY (C_ID)

PL/SQL procedure successfully completed.

SQL> desc person
 Name                           Null?    Type
 ------------------------------ -------- ----
 C_ID                           NOT NULL NUMBER(38)
 C_NAME                                  VARCHAR2(10)
 C_TYPE                                  VARCHAR2(10)

SQL> select * from person;

     C_ID C_NAME     C_TYPE
--------- ---------- ----------
        1 Bannon     person
        2 Myers      person
        3 Dougherty  person
        4 Neff       person
        5 Mieska     person
        6 Carlis     person

6 rows selected.

SQL> exit
```

Some Operator Examples

This document contains a few examples showing the syntax of various operators in the package. It is intended to give you a feel of the syntax, rather than a complete battery of examples showing each variation of each operator.

The examples are taken from the book, but exact results will vary due to the changes in the Creature, Achievement, and Skill relations throughout the book.

You will notice that coming up with a good formatting technique for your code is essential - just like in any other program. Also, notice that later examples use the results of the earlier operators as operands.

```
ops.reduce_ra(
    ops.filter_ra('achievement','s_code = ''A''','floating_creature'),
    'c_id','s_code, score','FloatCreatureId_AchData')

ops.group_ra('achievement','s_code, proficiency','ach_cnt=count(*)',
             'SCodeProfPair_AchCnt')

ops.reduce_ra(
    ops.filter_ra('achievement','proficiency = 1','Good_Achievement'),
```

```
    'c_id',NULL,'Id_GoodAchievingCreature')

ops.reduce_ra(
    ops.filter_ra('achievement','proficiency = 3','Poor_Achievement'),
    'c_id',NULL,'Id_PoorAchievingCreature')

ops.union_ra('Id_GoodAchievingCreature','Id_PoorAchievingCreature',
             'Id_GoodOrPoorAchievingCreature')

ops.reduce_ra(
    ops.filter_ra(
        ops.times_ra('F=achievement','S=achievement','Achievement_Pair'),
        'F_S_code = ''A'' AND S_S_code = ''E'' AND F_C_id = S_C_id',
        'SameCreatureFloatSwimAchPair'),
    'C_id = F_C_id',NULL,'Id_FloatAndSwimCreature')
```

More Operator Examples

This set of queries is designed to be an extensive test of all of the relational algebra operators and will give you an example of the syntax of calling each operator. Notice, the result names are NOT the correct logical names for the result relations - they are intended to aid in debugging. Notice that some of the operations are identical, but have varying syntax - in these cases the first version is the "preferred" syntax and the subsequent versions are used to test the robustness of the parsing code.

This set of queries (along with the corresponding drop table commands) is available as a SQL script and is included with this documentation.

```
ops.go(ops.filter_ra('creature','c_type = ''person''','t_s1'));

ops.go(ops.project_ra('creature','c_id, c_type','t_p1'));
ops.go(ops.project_ra('creature','c_id, cid2 = c_id*2, c_type','t_p2'));
ops.go(ops.project_ra('creature','c_id,cid2=c_id*2','t_p3'));

ops.go(ops.reduce_ra('creature','c_type','t_r1'));
ops.go(ops.reduce_ra('creature','type = c_type',NULL,'t_r2'));
ops.go(ops.reduce_ra('creature','c_name, type = c_type',NULL,'t_r3'));
ops.go(ops.reduce_ra('creature',' name =c_name','type= c_type','t_r4'));

ops.go(ops.group_ra('creature','c_cnt=count(*)','t_g1'));
ops.go(ops.group_ra('creature','c_cnt=count(*), c_avg = avg(c_id)','t_g2'));
ops.go(ops.group_ra('creature','c_type','type_cnt = count(*)','t_g3'));
ops.go(ops.group_ra('achievement','score, s_code','ss_cnt = count(c_id)','t_g4'));
ops.go(ops.group_ra('creature','c_id','c_name, c_type','one = count(*)','t_g5'));

ops.go(ops.times_ra('creature','skill','t_t1'));
ops.go(ops.times_ra('creature','a = achievement','t_t2'));
ops.go(ops.times_ra('creature','c = creature','t_t3'));
ops.go(ops.times_ra('c1 = creature','c2 = creature','t_t4'));

ops.go(ops.cjoin_ra('creature','skill','reside_t_id != origin_town','t_cj1'));
ops.go(ops.cjoin_ra('c1 = creature','c2 = creature','c1.c_id < c2.c_id','t_cj2'));

ops.go(ops.mjoin_ra('creature','skill','reside_t_id,'origin_t_id,'town','t_mj1'));
ops.go(ops.mjoin_ra('creature','a = achievement','c_id','c_id','t_mj2'));
ops.go(ops.mjoin_ra('c = creature','a = achievement','c_id, reside_t_id,'c_id,test_t_id,'c_id ,
town', 't_mj3'));
ops.go(ops.mjoin_ra('c1 = creature','c2 = creature','c_type','c_type','t_mj4'));

ops.go(ops.ojoin_left_ra('creature',
           ops.group_ra('achievement','c_id','a_cnt=count(*)',
           't_lojtmp1'),'c_id','c_id','t_loj1'));

ops.go(ops.ojoin_right_ra(
           ops.filter_ra('creature','c_type=''person''','t_lojtmp2'),
           'achievement','c_id','c_id','t_loj2'));
```

```
ops.go(ops.ojoin_both_ra(
           ops.group_ra('creature','reside_t_id,'skill_cnt=count(*)','t_bojtmp11'),
           ops.group_ra('skill','origin_t_id,'creature_cnt=count(*)','t_bojtmp12'),
           'reside_t_id,'origin_t_id,'town','t_boj1'));

ops.go(ops.ojoin_both_ra(
           ops.reduce_ra(
               ops.group_ra('creature','reside_t_id,'skill_cnt=count(*)','t_bojtmp21'),
               'town=reside_t_id,'skill_cnt','t_bojtmp22'),
           ops.reduce_ra(
               ops.group_ra('skill','origin_t_id,'creature_cnt=count(*)','t_bojtmp23'),
               'town=origin_t_id,'creature_cnt','t_bojtmp24'),
           'town','town','t_boj2'));

ops.go(ops.union_ra(ops.project_ra('achievement','c_id, s_code','t_utmp1'),
            ops.project_ra('aspiration','c_id, s_code','t_utmp2'),'t_u1'));
ops.go(ops.minus_ra('achievement','aspiration','t_m1'));
ops.go(ops.intersect_ra('achievement','aspiration','t_i1'));
ops.go(ops.divide_ra(
               ops.filter_ra(ops.project_ra('creature','c_id','t_dtmp1'),'c_id<4','t_dtmp2'),
                           ops.project_ra('achievement','c_id, s_code','t_dtmp3'),'t_d1'));

ops.go(ops.full_minus_ra('achievement',
       ops.project_ra(
                 ops.filter_ra('creature','c_type=''person''','t_omtmp11'),
                       'c_id','t_omtmp12'),
             't_om1'));

ops.go(ops.full_minus_ra('achievement',
       ops.reduce_ra(
                 ops.filter_ra('creature','c_type=''person''','t_omtmp21'),
                       'pcid=c_id',NULL,'t_omtmp22'),
             'c_id','pcid','t_om2'));

ops.go(ops.assoc_ra('EM',NULL, 'creature','achievement',NULL,'job_skill','job',
          't_as1'));

ops.go(ops.assoc_ra('((QCOUNT >= MQCOUNT) AND (NQCOUNT > 0))',NULL,
        'creature','achievement',NULL,'job_skill','job','t_as2'));

ops.go(ops.assoc_ra('EM',NULL, 'creature','achievement','skill','job_skill','job',
't_as3'));

ops.go(ops.assoc_ra('OS','creature','achievement',NULL,
't_as4'));

ops.go(ops.assoc_ra('ES', '((s_code >= s_code_lo) AND (s_code <= s_code_hi))',
      'creature', 'achievement', NULL, 'job_skill_range', 'job', 't_as5'));

ops.go(ops.assoc_ra('EMOS', '((s_code = s_code2) AND (score >= score_lo)
        AND (score <= score_hi))','creature', 'achievement', NULL, 'job_skill_score_range',
        'job', 't_as6'));
```