Britton Lee Host Software

IDL REFERENCE MANUAL

(R3v5m2)

February 1988
Part Number 205-1235-003

Printed **February 1988.**

This document **supersedes all previous documents. This edition is intended for use with software release number 3.5 and future software releases, until further notice.**

Table of Contents

Notational Conventions

The following conventions are employed in the synopses throughout this manual:

Words in **boldface** should be entered exactly as they appear.

Words in roman face should be replaced with a value of the user's choice.

Square brackets "[]" indicate optional elements.

Braces "{}" enclose lists from which the user must select an element.

Vertical bars "|" separate choices.

Parentheses "()" are to be entered literally.

Ellipses "..." indicate that the preceding items may be repeated one or more times.

For a detailed description of the error messages generated by IDL, consult the *Message Summary (IDL Version)*.

PART I

INTRODUCTION TO IDL

Introduction to IDL

This part provides an introduction to IDL intended for data processing professionals interested in learning to use IDL to access data stored on a Britton Lee database server.

All Britton Lee database servers are designed to store and manipulate databases built on the relational model, which means that the data in the database is stored in tables or relations. A relation is organized horizontally into tuples and vertically into attributes. The tuples represent individual entities in the relation while the attributes describe characteristics associated with those entities.

The first chapter in this part explains how to invoke and exit the idl program. The rest of this part covers three general topics: manipulating, defining, and controlling access to data stored in relations.

Data manipulation refers to the part of a query language which extracts data from an existing relation and modifies existing relations by appending new data, changing the values of data, and deleting data.

Data definition refers to the part of a query language which creates, alters and deletes the structure of database objects such as relations, views, and stored commands.

Data authorization refers to the part of a query language which authorizes access to database objects for individual users and groups of users.

This part does not describe all the IDL commands, nor does it completely describe the commands which it does cover. For a complete description of every IDL command, consult Part II of this manual.

This part does not cover special features of IDL used for embedding IDL in procedural programming languages such as C. The applications programmer who needs to use embedded IDL should consult the *RIC User's Guide*.

The examples in this section use a hypothetical database called "books". The relations in "books" database are listed below.

AUTHOR RELATION		
authnum	first	last
1	alice	adams
2	herman	melville
3	brian	kernighan
4	dennis	ritchie
5	dh	lawrence
6	william	shakespeare
7	doug	adams

TITLE RELATION			
docnum	title	onhand	pubnum
1	moby dick	6	2
2	the c programming language	8	3
3	macbeth	12	1
4	superior women	3	2
5	fantasia of the unconscious	6	1
6	so long and thanks for all the fish	7	1

PUBLISHER RELATION			
pubnum	name	city	phone
1	penguin	london	441-301-9898
2	signet	new york	212-755-8400
3	prentice-hall	englewood cliffs	201-254-6300
4	south end	boston	617-445-3223

AUTHTTL RELATION	
authnum	docnum
1	4
2	1
3	2
4	2
5	5
6	3
7	6

PRICE RELATION			
docnum	**year**	**amount**	**distrib**
1	87	2.95	western
2	87	22.95	berkeley technical
3	87	2.50	cal-west
4	87	4.95	cal-west
5	87	4.95	bookpeople
6	87	2.50	western

Executing the IDL Program

ENTERING IDL

To invoke IDL enter

idl

On UNIX systems, IDL commands must be entered in lower case. On most other systems, IDL commands are case-insensitive.

If you have successfully invoked IDL, you will see displayed a numeral followed by a right parenthesis as in

1)

This is the IDL prompt.

In interactive IDL, all IDL commands or sequences of IDL commands must be terminated by the keyword **go** or a semicolon (;). If you are using IDL statements in one of the embedded query languages, terminate all commands with a semicolon (;). If you are using IDMLIB subroutines to query a database from within a program, use no command terminators.

In order to execute any IDL commands other than a **range** or **set** statement, you must first open a database. The following command opens the "books" database.

1) open books;

To invoke IDL and open the "books" database with a single command, enter

idl books

If the specified database does not exist or if you do not have permission to open it, IDL displays this information and exits.

EXITING IDL

If the IDL prompt is displayed, you can exit IDL by entering

3) exit;

If the prompt is not currently displayed and you wish to exit, the <BREAK> function on your system will usually produce the IDL prompt.

Data Manipulation

Data manipulation refers to the ability to examine the data in one or more relations and to modify existing relations by appending new data, deleting data, or changing the value of one or more attributes in specified tuples.

The **retrieve** command is used to examine or query the database, the **append**, **delete**, and **replace** commands to modify the database.

The **range** statement is used to associate a range variable with a relation. The **retrieve**, **replace**, and **delete** commands require that the relations they manipulate be referenced through range variables.

RANGE

The **range** statement associates a variable of the user's choice with a relation. The essential parts of a **range** statement are

- the variable name
- the relation to be associated with the variable name.

The following **range** statement associates the range variable "t" with the "title" relation.

```
1) range of t is title;
```

A **range** statement may specify an optional owner name, preceded by a colon (:) to distinguish the relation being associated from other relations of the same name belonging to other users. The command

```
1) range of t is title:susie;
```

associates "t" with the "title" relation which is owned by user "susie". If no owner name is specified, it is assumed that the range variable refers to a relation owned by the user executing the command. If such an object does not exist, it is assumed that the range variable refers to a relation owned by the DBA.

A range variable is always associated with the most recent **range** statement that defined it. The sequence

```
1) range of t is title;
1) range of t is author;
1) range of t is publisher;
```

leaves "t" associated with the "publisher" relation. A range variable is associated with its relation until it is used in another **range** statement or until the **idl** session terminates.

RETRIEVE

The **retrieve** command retrieves specified data from one or more relations. Used interactively, it displays its results in a relation consisting of the requested tuples and attributes at the user's terminal.

The essential part of any **retrieve** statement is the parenthesized *target-list* which consists of specifications of the relation(s) to be accessed, through use of a range variable, and the attributes to be displayed.

The order in which the *targets* are specified in the query determines the order in which they will appear, from left to right, at the terminal.

Thus the basic form of the **retrieve** statement is

retrieve (*target-list*)

A specified target may have various forms. It may be

- an attribute name prefaced by a range variable,
- a "result domain name = attribute name" prefaced by a range variable,
- the value returned by an aggregate or function,
- a "result domain name = value" returned by an aggregate or function,
- the keyword **all**,
- any arithmetic expression.

The object referenced by a range variable may be a relation or a view.[1] The parentheses enclosing the *target-list* are mandatory.

The following query illustrates the simplest form of the **retrieve** statement. It queries the database for the values of the attributes named "first" and "last" in all the tuples in the "author" relation. The **range** statement is necessary to establish the association between the range variable "a" and the "author" relation, unless this association has been established by a previous **range** statement.

[1]Views are described in the chapter on data definition.

```
1) range of a is author
2) retrieve(a.first, a.last);
```

first	last
alice	adams
herman	melville
brian	kernighan
dennis	ritchie
dh	lawrence
william	shakespeare
doug	adams

The word **all** is used to specify all of the attributes in a relation. The entire "author" relation consists of three attributes. The following command retrieves all of the attributes in the relation.

```
1) retrieve (a.all);
```

authnum	first	last
1	alice	adams
2	herman	melville
3	brian	kernighan
4	dennis	ritchie
5	dh	lawrence
6	william	shakespeare
7	doug	adams

It is also possible to specify result domain names which differ from the original attribute names in the relation displaying the retrieved data. The following command retrieves data from the "last" attribute in the "author" relation, but labels the selected domain "surname" in the result.

```
1) retrieve (surname = a.last);
```

surname
adams
melville
kernighan
ritchie
lawrence
shakespeare
adams

In addition to this basic format, there are several optional specifications which can be added to control

- the restrictions to apply for retrieving tuples (the *qualification*)
- the order in which the tuples should be displayed
- whether duplicate tuples should be ignored.

Where Clause

In order to specify that only some of the tuples in a relation should be retrieved, the query must indicate the conditions governing the retrieval of tuples. This set of conditions, called the *qualification*, consists of one or more comparisons between terms which evaluate to true or false. Each comparison is expressed by one of the following relational operators.

Symbol	*Meaning*
=	(equal to)
!=	(not equal to)
<>	(synonym for !=)
>	(greater than)
>=	(greater than or equal to)
<	(less than)
<=	(less than or equal to)

When a relational operator is applied to character data, the comparison is governed by ASCII or EBCDIC order, depending on which character set was specified when the database was created. Blanks at the ends of character strings are ignored for comparison purposes.

The format for a **where** clause is the keyword **where** followed by the conditions limiting the retrieval.

The following query requests tuples from the "author" relation in which the value of the "authnum" attribute is 2.

```
1) retrieve (a.all)
2) where a.authnum = 2;
```

authnum	first	last
2	herman	melville

The next query requests data from tuples in which the value of the "last" attribute is 'adams'. The constant value 'adams' must be enclosed in single or double quotation marks because it is being compared to an attribute of the type character string.[2]

[2]The types of attributes are discussed in more detail in the chapter on data definition.

```
1) retrieve (a.first, a.last)
2) where a.last = 'adams';
```

first	last
alice	adams
doug	adams

Unique

The **unique** modifier is used to specify that only zero or one tuple should be retrieved, even if more tuples meet the conditions of the *qualification*.

```
1) retrieve unique (a.last)
2) where a.last = 'adams';
```

last
adams

The **unique** modifier applies to the entire *target-list*. The command

```
1) retrieve unique (a.first, a.last)
2) where a.last = 'adams';
```

first	last
alice	adams
doug	adams

retrieves two tuples from the "author" relation, not one, because there is no duplication in the relation of the combined values for "first" and "last".

Multiple Conditions

If the *qualification* governing the **retrieve** statement is based on more than one condition, the relationship between the conditions can be expressed using the **and** and **or** operators. The following query uses the **and** operator to request all the tuples in which the value of the "last" attribute is 'adams' and the value of the "first" attribute is not "alice". In order to be retrieved, a tuple must satisfy both of these conditions.

```
1) retrieve (a.all)
2) where a.last = 'adams' and a.first != 'alice';
```

authnum	first	last
7	doug	adams

The same query could be expressed using the **not** keyword instead of the != relational operator.

```
1) retrieve (a.all)
2) where a.last = 'adams' and not a.first = 'alice';
```

authnum	first	last
7	doug	adams

The next query uses the **or** operator to retrieve the tuples in which the value of the "last" attribute is 'adams' or the value of the "first" attribute is not 'alice'. In this case, a tuple must satisfy only one of the conditions, not both, in order to be retrieved.

```
1) retrieve (a.all)
2) where a.last = 'adams' or a.first != 'alice';
```

authnum	first	last
1	alice	adams
2	herman	melville
3	brian	kernighan
4	dennis	ritchie
5	dh	lawrence
6	william	shakespeare
7	doug	adams

The **or** operator is useful when one is not certain of the precise value of a field on which a condition is based.

```
1) retrieve (a.all)
2) where a.first = 'herman' or a.first = 'herbert';
```

authnum	first	last
2	herman	melville

A query can combine several conditions in a single *qualification*. When **and** and **or** are used in the same query, the **and** operator takes precedence over the **or** operator. Parentheses can be used to override this precedence as illustrated below.

```
1) retrieve (a.all)
2) where (a.first = 'herman' or a.first = 'herbert')
4) and (a.last = 'melville' or  a.last = 'de melville');
```

authnum	first	last
2	herman	melville

Patterns

Patterns are used to indicate a string value in which all of the characters are not specified. The asterisk (*) is used in the character string to represent a substring of zero or more characters. The question mark character (?) is used to represent a single character.

The following query retrieves the "first" and "last" attributes for all tuples in which the value of the first character in the "first" attribute is "d".

```
1) retrieve (a.first, a.last)
2) where a.first = 'd*';
```

first	last
dennis	ritchie
dh	lawrence
doug	adams

The following query retrieves the tuple for a title in which two individual letters are not specified.

```
1) range of t is title;
2) retrieve (t.all)
3) where title = 'm?by d?ck';
```

docnum	title	onhand	pubnum
1	moby dick	6	2

Aggregates

There are a number of aggregate operators which can be used in queries to aggregate values supplied as arguments. These values may be attribute names or general arithmetic expressions. The following query demonstrates the effects of the **count**, **avg**, **max**, **min** and **sum** aggregates when applied to the "onhand" attribute of the "title" relation.

```
1) range of t is title;
1) retrieve
2) (count = count(t.onhand),
3) average = avg(t.onhand),
4) largest = max(t.onhand),
5) smallest = min(t.onhand),
6) total = sum(t.onhand));
```

count	average	largest	smallest	total
6	7	12	3	42

Order By

Normally the tuples fetched by a **retrieve** statement appear in an order determined by the database server software. The user can specify the order in which tuples should be displayed with the **order by** clause. The default order is ascending (lowest to highest), but descending (highest to lowest) can be specified with a **d** preceded by a colon (:). Both numeric and string type expressions can be used to order retrieved data. The following query specifies that the tuples be displayed in ascending order based on the value of the "last" attribute.

```
1) retrieve (a.first, a.last)
2) order by a.last;
```

first	last
alice	adams
doug	adams
brian	kernighan
dh	lawrence
herman	melville
dennis	ritchie
william	shakespeare

The next query specifies that the retrieved tuples be displayed in descending order based on the value of the "authnum" attribute.

```
1) retrieve (a.authnum, a.first, a.last)
2) order by a.authnum:d
3) where a.last != 'a*';
```

authnum	first	last
6	william	shakespeare
5	dh	lawrence
4	dennis	ritchie
3	brian	kernighan
2	herman	melville

In the next query the **order by** clause is used in retrieving data from the "title" relation to display the data ordered by the value of the "pubnum" attribute, and within that ordering by the value of the "onhand" attribute.

```
1) retrieve (t.pubnum, t.onhand, t.docnum)
2) order by t.pubnum, t.onhand;
```

pubnum	onhand	docnum
1	6	5
1	7	6
1	12	3
2	3	4
2	6	1
3	8	2

Joins

A join is a mechanism for relating data from multiple relations within a single query. When relations are joined, the **where** clause specifies a relationship, known as a "joining condition", between the tuples from which data is to be retrieved.

The following query retrieves data from the "title" and "onhand" attributes in the "title" relation and from the "name" attribute in the "publisher" relation. The joining condition is

"where t.pubnum = p.pubnum"

```
1) range of p is publisher;
1) retrieve (t.title, p.name, t.onhand)
2) where t.onhand < 7
3) and t.pubnum = p.pubnum;
```

title	name	onhand
fantasia of the unconscious	penguin	6
moby dick	signet	6
superior women	signet	3

The relation containing the joining condition need not be referenced in the *target-list*. The next query retrieves data from the "first" and "last" attributes of the "author" relation and from the "title" attribute of the "title" relation. The joining condition

"where l.authnum = a.authnum and l.docnum = t.docnum"

references a third relation, "authttl", through its range variable "l". The "authttl" relation consists only of attributes corresponding to key attributes in the "author" and "title" relations. This type of relation is called an associative relation. Its function is to enable a join in which the entities represented in two relations are related such that each tuple in one relation may be related to any number of tuples in the other relation, and vice-versa. Its use is applicable here, where a single title may be associated with multiple authors, and a single author may be

associated with several titles.

```
1) range of l is authttl;
1) retrieve (a.first, a.last, t.title)
2) where l.authnum = a.authnum
3) and l.docnum = t.docnum;
```

first	last	title
herman	melville	moby dick
brian	kernighan	the c programming language
dennis	ritchie	the c programming language
william	shakespeare	macbeth
alice	adams	superior women
dh	lawrence	fantasia of the unconscious
doug	adams	so long and thanks for all the fish

By Clause

The **by** clause is used to retrieve multiple values from an *aggregate*, one value for each group referenced in the **by** clause.

The following query uses the **sum** aggregate operator and the **by** clause to retrieve the total number of books on hand by publisher.

```
1) retrieve (total = sum(t.onhand by t.pubnum),
2) p.pubnum, p.name)
3) where p.pubnum = t.pubnum;
```

total	pubnum	name
25	1	penguin
9	2	signet
8	3	prentice-hall

This is to contrast with an aggregate which applies to the relation as a whole as in

```
1) retrieve (total = sum(t.onhand);
```

total
42

APPEND

The **append** command adds one or more tuples to a relation. This command can be used to append new data directly from the terminal or to append data from another relation.

For entering literal data from the terminal, the essential parts of an **append** command are specification of the relation to which the data is to be appended, the attributes to be appended, and the values of the attributes. The basic form of the **append** command is

append to *relation name* **(***attribute names* **=** *values***)**

The following command appends a new tuple to the "author" relation.

```
1) append to author
2) (authnum = 8, first = 'charles', last = 'dickens');
```

If the values for all of the attributes are not known, specify the attribute names and values which are known. Unspecified attributes are assigned zeros for numerics and blanks for character strings. These attributes can later be modified with the **replace** command when the values are available.

The next command appends a new tuple to the "title" relation. The value for the "docnum" attribute is an expression which evaluates to the next consecutive number in the attribute. The "onhand" attribute is omitted from the **append** command and is thus given a value of 0.

```
1) append to title
2) (docnum = max(a.docnum) + 1,
3) title = 'a tale of two cities',
4) pubnum = 1);
```

Data can also be appended from another relation. For example, assume that there is a relation called "modernauthor" which has three attributes named "fname" "lname" and "num". The following command appends to the "modernauthor" relation the existing data from the "first" and "last" attributes in the "author" relation. The value for the "authnum" attribute in the "modernauthor" relation is not specified, so it is assigned zeros in all the tuples.

```
1) range of a is author;
1) append to modernauthor
2) (fname = a.first, lname = a.last)
3) where a.authnum = 1
4) or a.authnum = 3
5) or a.authnum = 4
6) or a.authnum = 7;

1) range of m is modernauthor;
1) retrieve (m.all);
```

num	fname	lname
0	alice	adams
0	brian	kernighan
0	dennis	ritchie
0	doug	adams

REPLACE

The **replace** command changes the values of one or more attributes in the specified tuples in the specified relation. The conditions qualifying which tuples are to be replaced are specified in a **where** clause. If there is no **where** clause, all of the tuples in the relation are modified.

The basic form of the **replace** command is

replace *range variable* (*attribute names* = *values*)
where *specified conditions*

The following command changes the value of the "first" attribute of the "author" relation from 'doug' to 'douglas'.

```
1) range of a is author;
1) replace a (first = 'douglas')
2) where a.first = 'doug' and a.last = 'adams';
```

More than one attribute value can be replaced by a single **replace** command. The following command replaces two attributes in the "title" relation.

```
1) range of t is title;
1) replace t (title = 'hamlet', onhand = 8)
2) where t.docnum = 8;
```

The next command has no **where** clause. It increases by 5 the value of the "onhand" attribute in all of the tuples of the "title" relation.

```
1) range of t is title;
1) replace t (onhand = t.onhand + 5);
```

It is possible to replace the values in a relation by fetching them from another relation. The following command replaces the values of the "authnum" attribute in the "modernauthor" relation with the values that are used for equivalent tuples in the "author" relation.

```
1) range of m is modernauthor;
1) range of a is author;
2) replace m (num = a.authnum)
3) where a.first = m.fname and a.last = m.lname;

1) retrieve (m.all);
```

num	fname	lname
1	alice	adams
3	brian	kernighan
4	dennis	ritchie

DELETE

The **delete** command deletes entire tuples from the specified relation. It should be used with extreme caution, because without a **where** clause, the **delete** command deletes all of the tuples in a relation.

The basic form of the **delete** command is

delete *range variable*
where *specified conditions*

The following command deletes all of the tuples in the "title" relation in which the "onhand" attribute has a value less than 1.

```
1) range of t is title;
1) delete t
2) where t.onhand < 1;
```

The next command deletes all of the tuples in the "title" relation. After the command is executed, the relation would still exist, but it would have no data in it.

```
1) delete t;
```

Data Definition

Data definition refers to the ability to create, alter, or delete database objects such as relations, views, or stored commands.

The examples in this section assume that the user has been granted the necessary permissions to create database objects and indices in the "books" database.

This section and the next contain references to certain system relations, specifically to "descriptions", "relation", and "users". These are relations which are automatically created for every database by the system in order to manage the database.

CREATE

The **create** command creates a new relation in the open database. The command specifies the name of the relation and the names and *types* of its attributes, using the mnemonics indicated below.

The following mnemonic is used for character data.

c	character strings, specify length

The following mnemonics are used for numeric data.

i4	four-byte integers
i2	two-byte integers
i1	one-byte integers
f8	eight-byte floating-point numbers
f4	four-byte floating-point numbers
bcd	binary-coded decimal integers, specify length
bcdflt	binary-coded decimal floating point numbers, specify length

The following mnenomic is used for binary data.

bin	binary strings, specify length

The **c**, **bcd**, **bcdflt**, and **bin** mnemonics must be followed by an integer specifying the number of characters or bytes to allocate for the attribute, as in **c10** for a character attribute with a maximum length of ten characters or **bcd6** for a bcd attribute with a maximum length of 6 bytes.

The **bin**, **bcd**, and **bcdflt** mnemonics may be prefixed with the character **u** (for uncompressed) if leading and trailing zeros are to be retained. The **c** mnemonic may be prefixed with the character **u** if trailing blanks are to be retained. If a **u** is not specified for these types, trailing blanks and trailing and leading zeros are stripped.

The following command creates a new relation named "price", with four attributes named "docnum", "year", "amount", and "distrib". The "docnum" attribute is a two-byte integer field; the "year" attribute is a

one-byte integer field; the "amount" attribute is a binary-coded decimal floating point field with a maximum length of six digits; the "distrib" attribute is a character field with a maximum length of twenty characters.

```
1) create price
2) (docnum = i2,
3) year= i1,
4) amount = bcdflt8,
5) distrib = c20);
```

A **retrieve** command on "price" shows the empty relation.

```
1) range of p is price;
1) retrieve (p.all);
```

docnum	year	amount	distrib

CREATE INDEX

An index is a directory which relates the physical location of each tuple in a relation to the value of a specified attribute or group of attributes in the relation. The purpose of an index is to provide a direct access path to data when a query references the attribute(s) specified in the **create index** command. The creation of indices can greatly decrease access time if a relation is often searched on the basis of a particular attribute or set of attributes, because indices eliminate the need to scan all the data during a search.

There are two kinds of indices, **clustered** and **nonclustered**. If neither kind is specified in the **create index** command, a **nonclustered** index is created by default.

Clustered Index

A **clustered index** often provides faster access than a **nonclustered index** but requires that the data be sorted on the value of the indexed attribute(s). There can be only one **clustered index** for a single relation. That single index may, however, be on multiple attributes.

The following command creates an index on the "docnum" attribute of the "title" relation.

```
1) create clustered index
2) on title (docnum);
```

Nonclustered Index

A **nonclustered index** usually provides slower access than a **clustered index**, though faster access than a sequential scan of all the data. It does not require that the data in the relation be sorted. Up to 250 **nonclustered** indices can be created on a single relation.

The following command creates a **nonclustered index** on the combined "last" and "first" attributes of the "author" relation.

```
1) create nonclustered index
2) on author (last, first);
```

Unique Index

Both **clustered** and **nonclustered** indices can be specified **as unique**. This prevents duplicate attribute values from being introduced into the relation.

The following command creates a **unique nonclustered index** on the "authnum" attribute in the "author" relation.

```
1) create unique nonclustered index
2) on author (authnum);
```

After creation of this **index**, if a user tries to add a tuple in which the value of the "authnum" attribute is the same as the value of the "authnum" attribute for a tuple which already exists in the relation, an error message will be generated and the entire **replace** or **append** command aborted.

CREATE VIEW

A view is a virtual relation composed of parts of one or more base relations or other views. The view itself does not actually contain data, but it reflects the data contained in its underlying base relations. Views are manipulated and protected like relations, except that they cannot be modified unless the modification can unambiguously be applied to a base relation. Views are useful for defining subsets of relations, based on a selection of attributes, tuples or both. They are also useful for restricting access to certain parts of a relation.

The **create view** command specifies the name of the view and a *target-list* consisting of attributes prefaced with the appropriate range variables indicating the sources of the data to be accessed by the view.

The following command creates a view named "instock" consisting of data from the "title", "author" and "price" relations.

```
1) range of t is title
2) range of a is author
3) range of p is price
4) range of l is authttl;
1) create view instock
2) (t.docnum, t.title, a.last, p.amount)
3)      where l.docnum = t.docnum
4)      and l.authnum = a.authnum
5)      and t.docnum = p.docnum
6)      and t.onhand > 0;
```

The view can now be queried as though it were a relation. It is possible to permit a user to query the "instock" view without permitting that

user to query all the attributes in the base relations.[3]

```
1) range of i is instock;
1) retrieve (i.all)
2) where i.author = 'lawrence'
3) and i.title = 'fantasia*';
```

docnum	title	last	amount
5	fantasia of the unconscious	lawrence	4.95

DEFINE and EXECUTE

The **define** command creates an object called a stored command. A stored command is a sequence of data manipulation commands, such as **retrieve**, **append**, **replace**, or **delete**, which can be referenced collectively by the stored command's name. Because the stored command exists in a parsed and partially processed form on the database server, it is usually faster to execute a stored command than to execute its constitutent commands individually.

When a stored command is created, formal parameters are indicated by the parameter name prefaced by a dollar sign ($). When the stored command is executed, real values are substituted for the formal parameters.

The following command creates a stored command named "addauthor" which consists of an **append** command and a **retrieve** command. The formal parameters for the first and last names are indicated by "$f" and "$l".

```
1) /* This adds an author's name to the "author" relation. */
2) define addauthor
3)      range of a is author
4)      append to author                    /* add the name */
5)      (authnum = max(a.authnum) + 1, first = $f, last = $l)
6)      retrieve (a.all)                /* confirm it is there */
7)      where a.authnum = max(a.authnum)
8) end define;
```

A stored command is executed using the **execute** command.

```
1) execute addauthor
2) with f = 'pat', l = 'barker';
```

authnum	first	last
8	pat	barker

[3]Permitting access is discussed in the chapter on data authorization.

DESTROY

The **destroy** command removes an object, such as a relation, view, or stored command, from the database. If a view or stored command is dependent upon the object being destroyed, that view or stored command must be destroyed first.

The following command removes the "modernauthor" relation.

```
1) destroy modernauthor;
```

The next command removes the "instock" view.

```
1) destroy instock;
```

DESTROY INDEX

The **destroy index** command removes an index from a relation. The command identifies the index to be destroyed by its name and its characteristics: whether it is **clustered** or **nonclustered**, and whether it is **unique**.

The following command destroys the **unique nonclustered index** on the "authnum" attribute in the "author" relation.

```
1) destroy unique nonclustered index
2) on author(authnum);
```

AUTO-ASSOCIATE

When an object is created with the **create**, **create view**, or **define** commands, its name is automatically recorded in the system relation "relation" along with a unique identification number stored in the "relid" attribute of this relation.

The **associate** command is also automatically executed when an object is created. This command records information about a relation or attribute in the system relation "descriptions". The object being described is identified by its unique "relid" which is associated with the object's name as it was recorded in the "relation" system relation. The text of the command which created the object, including comments, is appended to the "text" attribute of the "descriptions" system relation.

When an object is removed from the database with the **destroy** command, references to it in the "relation" and "descriptions" relations are also removed.

This automatic association feature makes it possible to retrieve information about an object, such as the types of the attributes of a relation or the constituent commands of a stored command, knowing only the name of the object.

The following query requests a description of the "price" **relation**.

```
1) range of d is descriptions;
1) retrieve (d.text)
2) where d.relid = rel_id("price");
```

text
create price (docnum = i2, year = i1, amount = bcdflt8, distrib = c20)

The next query requests a description of "addauthor".

```
1) range of d is descriptions;
1) retrieve (d.text)
2) where d.relid = rel_id("addauthor");
```

text
/* This adds an author's name to the "author" relation. */ define addauthor range of a is author append to author /* add the name */ (authnum = max(authnum) + 1, $f, $l) retrieve (a.all) /* confirm it is there */ where a.authnum = max(a.authnum) end define

Data Authorization

When a database object is created, its creator, who is also its owner, automatically has permission to read, write to, and in the case of a stored command, execute, the object while all other users are automatically denied these privileges. In order to make the object accessible to other users, the owner of the object must specifically permit access using the **permit** command. Similarly, the owner of an object may deny certain users or all users specific types of access using the **deny** command.

PROTECT MODES

The types of access which can be permitted and denied are referred to as protect_modes. The protect_modes which apply to the objects described in this section are listed below. There is a complete list of protect_modes under "Protect_Mode" in the "General Concepts" section.

Protect_Mode	*IDL Command*
read	**retrieve, create view**
write	**append, delete, replace**
execute	**execute**
create	**create, define**
create index	**create index**

If, for example, a user is permitted read access of a relation, but not write access, that user may issue **retrieve** commands on that relation, but not **append**, **replace**, or **delete** commands.

PERMIT

The **permit** command gives access to an object to a user or group of users. The user, or group of users, is identified by the name by which he or she is known to the database server. These names are found in the "users" system relation in the open database.

The **permit** command specifies the protect_mode being permitted, the object name to which the privilege applies, and the user(s) to whom the privilege is being given.

The following command gives write privileges on the "price" relation to "susie". This permits her to modify this relation.

```
1) permit write
2) of price
3) to susie;
```

Access can be limited for certain attributes of an object. The following command permits the "salesfolk" to read the "book" and "price" attributes of the "instock" view. They are not permitted to read other attributes in this view. The group "salesfolk" has been defined in the system relation "users".

```
1) permit read
2) of instock (book, price)
3) to salesfolk;
```

The following command permits all users to read all attributes in the "title" relation.

```
1) permit read
2) of title;
```

DENY

The **deny** command prevents access to objects. Its syntax is the same as that for the **permit** command.

The following command denies read privileges on the "title" relation to all users.

```
1) deny read
2) of title;
```

The next command ensures that susie and jason are the only users who can execute the "addauthor" stored command.

```
1) deny execute
2) of addauthor;
1) permit execute
2) of addauthor
3) to susie, jason;
```

PART II

IDL COMMANDS

Introduction to IDL Commands

This part is a reference for accessing Britton Lee's database servers using IDL commands. It describes all of the IDL commands which can be executed interactively by a user running the **idl** program on a host system.

All of the examples in this manual are given for interactive IDL. To adapt the examples for embedded query languages such as RIC or for writing programs which incorporate IDL statements using IDMLIB, consult the appropriate User's Guide.

The **idl** program reads any system and user profile files which may exist before reading user IDL input. These profile files may contain any IDL commands or front-end commands. They often are used to execute front-end commands which configure IDL according to a particular set of needs. See the host-specific reference material for IDL for information on creating user profile files in a particular host environment.

Comments enclosed by the characters /* and */ may be included anywhere in IDL input.

SEE ALSO

idl(1I) in *Host Software Specification* (UNIX systems)
IDL in *Command Summary* (other systems)

abort transaction

DESCRIPTION

Abort transaction aborts the current transaction (atomic sequence of IDL commands). All logical effects of the transaction are undone.

EXAMPLE

The **abort transaction** in this example causes the **delete** command to be backed out and the three tuples restored.

```
1) range of e is emp;
1) begin transaction;
1) delete e where e.lastname = "Croft";

3 tuples deleted

1) abort transaction;
```

MESSAGES

illegal command (IDM.E45)
The user has not sent previously a **begin transaction** command.

SEE ALSO

begin transaction, end transaction

append [to] rel_name (target-list) [**where** qualification]

DESCRIPTION

The **append** command adds new tuples to the relation or view referenced by *rel_name*. Each target in the *target-list* contains an attribute name and the value to be assigned to that attribute in the new tuple.

Although each new tuple is appended in its entirety, it is not necessary to specify values for all of the attributes. If all of the attributes in the relation are not specified in the *target-list*, default values are assigned for the unspecified attributes. The default values are blanks for character attributes and zeros for numeric attributes. To assign values other than the default values to these attributes for tuples which have already been appended, use the **replace** command.

The database server normally checks for overflow and division by zero in a *target-list* or *qualification*. A user may specify that checking should be turned off, or duplicate tuples should be ignored, by using the **set** command.

To copy a large amount of data from a host data file to a relation, use the host utility **idmfcopy**.

PERMISSIONS

The user must have **write** permission on all the attributes of a relation in order to **append** to it.

EXAMPLES

Appending data directly from the terminal:

This command adds one tuple to the "parts" relation.

```
1) append to parts (name = "handle", quan = 10);
```

Appending data from another relation:

This command adds one tuple to the "newparts" relation for every tuple in the "parts" relation, taking the value of the "name" attribute from each tuple in the "parts" relation and assigning the value "10" to the "quan" attribute of each tuple added to the "newparts" relation.

```
1) range of p is parts;
1) append to newparts (name = p.part, quan = 10);
```

Appending with *qualification* and type conversion:

The following command appends one tuple to the "newparts" relation for each tuple in the "parts" relation in which the value of the "number" attribute is greater than 10. The value of "num" in the "newparts"

relation gets the value of "number" from the "parts" relation, but since "num" is a character attribute, and "number" is an integer attribute, the value of "number" is converted from integer to character using the **string** function.

```
1) range of p is parts
2) append to newparts (num = string(6, p.number))
3)              where p.number > 10;
```

MESSAGES

out of space (IDM.E42)
: No more tuples can be added because the database is out of free space. The database should be extended, the transaction log dumped, or relations within the database destroyed.

quota exceeded (IDM.E4)
: When the relation was created a quota was given. The addition of this tuple would cause the relation to exceed the quota.

not found (IDM.E6)
: The named relation or attribute was not found.

wrong type specified for attribute (IDM.E12)
: If a conversion from character to integer or numeric (or vice-versa) is necessary, it must be explicitly stated in the **append** command.

tuple too large (IDM.E61)
: A tuple is larger than the maximum size (2000 bytes). The tuple is not appended.

view not updatable (IDM.E60)
: User attempted to **append** to a view which is not updatable.

SEE ALSO

audit, delete, replace, retrieve, set
"Functions", "Rel_Name", "Qualifications", "Target-Lists"
idmfcopy(1I) in *Host Software Specification*
IDMFCOPY in *Command Summary*

associate {object_name | range_var.att_name}
[[**with**] string1[, string2]]

DESCRIPTION

The **associate** command adds or replaces information in the system relation "descriptions". This relation is used to associate one or more textual descriptions with an object. The *object_name* can refer to a relation, view, file, or stored command. The *range_var.att_name* refers to an attribute through a range variable. An entry in the "descriptions" relation might look like this:

attid	relid	key	text
0	29033	I1	Relation listing all parts
3	29033		Attribute for quantity on hand

If only an *object _name* is specified, the entry in "descriptions" pertains to the entire object. This is illustrated in the first tuple of the example entry above. In this case, the "relid" in the "descriptions" relation gets the value of the "relid" for that object as it is recorded in the system relation "relation". The "attid" attribute in the "descriptions" relation gets a value of zero.

If an attribute is specified by a *range_var.att_name*, the description refers only to that attribute. This is illustrated in the second tuple of the example entry above. In this case, the "attid" in the "descriptions" relation gets the value of the "attid" for that attribute as it is recorded in the system relation "attribute".

The *string1*, if specified, is appended to the "text" attribute of the "descriptions" relation. The *string2*, if specified, is appended to the "key" attribute of the "descriptions" relation. If entries already exist for "text" or "key", they are replaced by the new values. Both *string1* and *string2* must be entered as quoted character strings.

The function of the optional "key" attribute is user-defined. It is frequently used as a sequential line number for descriptions in the "text" attribute. For example, the following sequence of **associate** commands appends a four-tuple description of the "myrel" relation.

```
1) associate myrel with "This is my very own","M1";
1) associate myrel with "relation which has","M2";
1) associate myrel with "only two attributes","M3";
1) associate myrel with "called num and name","M4";
```

The description of the "myrel" relation can then by retrieved ordered by the "key" attribute:

```
1) range of d is descriptions;
2) range of r is relation;
3) retrieve(d.text)
4) order by d.key
5) where r.name = "myrel"
6) and r.relid = d.relid;
```

If neither *string1* nor *string2* is supplied, all of the tuples in the "descriptions" relation which apply to the specified object or attribute are deleted and nothing is added.

If the **associate** command references a relation that, on all three keys, is already in the data dictionary, the description is replaced; otherwise it is appended.

If a string is longer than one line and wrap-around is not desired, precede each carriage return with a backslash.

The keyword **with** is optional.

AUTO-ASSOCIATE

The **associate** command is automatically executed whenever a **create**, **create view**, or **define** command is executed. The full text of the command, including any comments enclosed within the characters /* and */ which precede or are contained within the command, is appended to the "text" attribute of the "descriptions" relation. This feature provides automatic documentation of relations, views, and stored commands.

If several objects are created in one command string (before a **go** or semicolon is entered), all of the command texts are associated with the first object created by the command string. For example, the command string

```
1) create x .....
2) create y .....
3) create z .....;
```

automatically associates the entire text with the "text" attribute in the "descriptions" relation for the "relid" identifying relation "x". The commands

```
1) create x .....;
1) create y .....;
1) create z .....;
```

on the other hand, associate each **create** command with the "relid" of the object it created.

If the text of a command creating a relation, view or stored command exceeds 4000 bytes in length, it will overflow the space allocated for it in the "text" attribute of the "descriptions" relation. To prevent this from occurring when entering long commands, the user can turn off the auto-associate feature by invoking **idl** with the **-a** or **/noassociate** flag, or turn auto-associate off and then on again using **%associate**.

PERMISSIONS

The user must be the owner of the object referenced in the command.

EXAMPLES

To associate a description with the "parts" relation:

```
1) associate parts "Relation listing all parts";
```

or

```
1) associate parts with "Relation listing all parts";
```

To add the information that the attribute "number" has an index:

```
1) range of p is parts;
1) associate p.number
2)      "Has a clustered index on number", "I1";
```

"I1" is a user-assigned key.

MESSAGES

illegal command (IDM.E45)
Cannot be used in a transaction.

permission denied (IDM.E43)
Must be owner of object.

not found (IDM.E6)
The object or attribute does not exist.

SEE ALSO

range, **retrieve**
"Object_Name"
%associate

audit [[**into**] rel_name] (target-list) [**where** qualification]

DESCRIPTION

The **audit** command creates a human-readable audit report from the transaction log or from a copy of it (i.e., the output of a **dump transaction**). It produces a formatted listing of the log in the order in which modifications to the database took place.

A simple **audit** command returns its output to the host, while an **audit into** command stores its output in a new relation specified by *rel_name*. For **audit into**, the name selected for *rel_name* must be unique.

The *qualification* and *target-list* are limited to the attributes listed below.

Attribute	*Meaning*
time	time of the update, in 60ths of a second since midnight
date	date of the update, in days from a date set by *idmdate*
user	user who made the modification
xtid	the "tid" of the tuple concerned
relid	the id of the relation involved
number	internal transaction number
type	type of update
value	data that was changed

See the entry for "Target-Lists" for a description of how *target-list* values are bound to program variables.

The "value" attribute is reserved for transaction logs. It may appear in the *target-list* but not in the *qualification*. It is used in the **audit** command to access all of the attributes of the relation whose modification is recorded in the transaction log. When the *target-list* is based on the "value" attribute, only one relation may be audited.

The interpretation of the "type" attribute is as follows:

Type	*Meaning*
00	null
12	stop update
14	split
16	begin query
17	replace begin
18	replace old
19	replace duplicate
1A	append duplicate
1C	end query
1D	abort query
1E	checkpoint
1F	safepoint
B4	root
C3	append
C4	delete
C7	destroy
C8	create index
CD	permit
CE	deny
D1	tuple
D2	abort transaction
D4	begin transaction
D5	end transaction
E1	define
EB	dump transaction
EE	define program

PERMISSIONS

For **audit into**, the user must have **create** permission in the open database.

EXAMPLES

This query displays a report of all activity in the "parts" relation during the last two days. The audit report is generated from the transaction log "transact".

```
1) range of t is transact;
1) audit (t.type, t.date)
2)     where t.relid = rel_id("parts")
3)     and t.date > getdate - 2;
```

The following command stores in the relation "inv_audit" a record of the type, date, and value of all the changes that were made to the relation "inventory". The audit report is generated from "log5".

```
1) range of l is log5;
1) audit into inv_audit (l.type, l.date, l.value)
2)       where l.relid = rel_id("inventory");
```

MESSAGES

incorrect number of logs
: Only one transaction log should be specified for this operation. One and only one variable can correspond to a log in the command.

incorrect use of value
: The "value" attribute can only appear in the *target-list* and no functions can be applied on it.

bad log
: An attempt was made to access a transaction log with a log from a different database.

permission denied
: User must have **read** permission on all attributes.

illegal command
: User must have **create** permission to use **audit into** command. The **audit** and **audit into** commands are illegal in a transaction.

SEE ALSO

append, delete, replace, retrieve
"Qualifications", "Target-Lists"
idmdump(1I) in *Host Software Specification*
IDMDUMP in *Command Summary*

begin transaction

DESCRIPTION

The **begin transaction** command introduces a sequence of IDL commands which are to be treated as a single command.

When **begin transaction** is used, the commands following **begin transaction** do not take effect until an **end transaction** command has been given.

Transactions are used to ensure consistency in a database. For example, in a bank money can be moved from one account to another by subtracting an amount from the balance of one account and adding it to another. If, after the update was subtracted and before the update was added, someone looked at the balances, it would appear as though money were either spontaneously generated or spontaneously lost. If the system went down between the two updates, the error could be made permanent.

This problem can be solved with a transaction. Although a transaction is composed of a sequence of commands, it is treated as an atomic operation; it is performed completely or not performed at all.

A transaction is also appropriate if the user wants to observe the effects of the constituent commands before they are committed. If the commands are put into a transaction and the user sees that the changes are undesirable, the changes can be backed out with an **abort transaction** command.

In interactive IDL, the **begin transaction** command must be immediately followed by the control keyword **go** or a semicolon. The only commands permitted within a transaction are: **abort transaction**, **append**, **begin transaction**, **delete**, **end transaction**, **range**, **replace**, **retrieve**, and **sync**.

Nested transactions are permitted, but all levels are considered to be part of the parent transaction. An **abort transaction** in a nested sequence will back out all changes back to the top level, as demonstrated below:

```
1) begin transaction;     /* start clean */
1> append ...;            /* level 1 */
1> end transaction;       /* committed */
1) begin transaction;     /* start again */
1> delete ...;            /* level 1 */
1> begin transaction;     /* nested */
1> replace ...;           /* level 2 */
1> end transaction;       /* not committed yet */
1> abort transaction;     /* back out everything since "start again" */
```

EXAMPLES

Completed transaction:

```
1) range of c is customers;
1) begin transaction;
1> replace c (c.balance = c.balance - 100)
2>   where c.name = "debtor";

1 tuple affected

1> replace c (c.balance = c.balance + 100)
2>   where c.name = "creditor";

1 tuple affected

1> end transaction;
1)
```

For extra security, this sequence may be placed into a stored command and permission to write to the original balances may be denied, while permission to execute the stored command may be granted.

Aborted transaction:

```
1) range of e is employees;
1) begin transaction;
1> delete e where e.lastname = "Croft";

3 tuples deleted

1> abort transaction;
WARNING: line 1: Transaction aborted
1)
```

In this example, the user wanted to **delete** a tuple in the "employees" relation where the employee's last name was "Croft". The effects of the **delete** command were displayed as "3 tuples deleted". The user was not aware that there was more than one employee with the last name of "Croft". To remove the change, the user issued the **abort transaction** command, which returned the data to its original state.

```
1) begin transaction;
1> delete e where e.lastname = "Croft" and
2>   e.firstname = "Traci";

1 tuple deleted

1> end transaction;
1)
```

Here, the user added another qualifier and observed that the effect of the **delete** command was "1 tuple deleted". As this was the desired effect, an **end transaction** command was given to commit the change in the database.

MESSAGES

must perform open first (IDM.E46)
The user must be in a database to start a **begin transaction**.

SEE ALSO

abort transaction, **define**, **end transaction**

create rel_name (att_name=type[, att_name=type ...]) [**with** option_list]

create rel_name ([partition_name] (att_name=type[, att_name=type ...])
[**with** option_list] [,[partition_name] (att_name=type
[, att_name=type ...]) [**with** option_list]] ...) [**with** option_list ...]

DESCRIPTION

The **create** command sets up an empty relation in the database which is currently open.

The second form shown in the synopsis is given to provide compatibility with future Britton Lee products.

The attributes comprising the new relation are specified in a list of one or more *att_name = type* pairs.

A *type* is composed by concatenating a predefined mnemonic representing the type of the data in the attribute with the maximum length of the attribute, as in "c20" for a character attribute of 20 characters or "i4" for an integer attribute of 4 bytes. The length is specified as number of characters for character attributes and as number of bytes for numeric attributes. For a list of the predefined mnemonics and a detailed description of the various *types* to which they refer, consult the section "Types".

Once a relation has been created, its basic structure cannot be altered. If it becomes desirable to change the structure of an existing relation, as in adding or removing attributes or changing the type of an attribute, a new relation must be created, and the data from the original relation appended to it. The **logging** status of a relation can be changed with the **extend** command.

The relation is initially created with no indices. If the relation is heavily used, a **clustered index** should be created for it either as soon as it has grown to several blocks of data or when the initial loading of data has been completed.

When **create** is executed, the **associate** command is automatically executed also, with the full text of the **create** command entered by the user inserted as the "text" portion of the description entered by the **associate** command into the "descriptions" relation. This feature provides automated documentation of relations.

OPTIONS

quota = n

The **quota** option specifies the maximum size, in 2048-byte data blocks, that the relation may attain. If no **quota** is specified, the relation will be allowed to grow until the database is full.

logging [= { **0** | **1** }]

If set to 1, this option specifies that the transaction log "transact" is to be updated whenever the relation is modified. If set

to 0 the transaction log is not maintained, but changes to the relation are recorded in the temporary system relation "batch". If the **logging** option is used but neither 0 nor 1 is specified, the default is 1.

PERMISSIONS

The user must have **create** permission in the database in which the relation is being created.

EXAMPLE

This command creates the "parts" relation with attributes "name" (a 20-character field), "cost" (an 8-digit bcd field), and "quan" (a two-byte integer field). It will be allowed to grow to a maximum size of 50 data blocks. All changes to the relation will be recorded in the system relation "transact".

```
1) create parts (name = c20, cost = bcd8,
2)     quan = i2) with logging, quota = 50;
```

MESSAGES

out of space on disk (IDM.E42)
The allocation could not be made because the disk was full.

tuple too wide (IDM.E61)
A command tried to add a tuple which was wider than the maximum allowable tuple width of 2000 bytes.

illegal command (IDM.E45)
It is illegal to create relations inside a transaction or stored command. It is illegal to create relations without **create** permission.

SEE ALSO

associate, **create database**, **create index**, **create view**, **destroy**, **extend**
"Att_Name", "Rel_Name", "Types"

create database dbname [**with** options]

DESCRIPTION

The **create database** command creates a database which contains only the system relations.

When a database is created, the system relation "host_users" is initialized with one tuple allowing access to the creator of the database. The creator is therefore the owner and DBA of the database.

PERMISSIONS

The **create database** command must be executed from the "system" database, and the user must have permission in the "system" database to create a database.

OPTIONS

demand = nblocks [**on**] "diskname"

The **demand** option specifies the number of 2K blocks to allocate for the database. The *nblocks* must be an integer.

A zone is a group of cylinders, with the number of cylinders per zone set when the disk is formatted. Zone sizes range from 128 to 254 blocks. The "bpz" attribute in the "disks" in the "system" database indicates the zone size for all disks attached to the database server.

Since database allocations are only made in whole numbers of zones, the program will round *nblocks* up to the first whole number of zones, allocate that number, and display the number of blocks actually allocated at the user's terminal.

The database will not be allowed to grow beyond the size allocated. If the size which the database was originally allocated is insufficient, the user may increase its size with the **extend database** command.

If a "diskname" is specified, the allocation is made on the specified disk; otherwise the allocation is made on any disk that has sufficient space.

If no **demand** is specified, the default allocation is one zone size.

The **demand** option can be repeated many times to specify how much of the database is to be placed on a given disk. The phrase

with demand=1000 on "disk1", demand=250 on "disk2"

requests that the database be allocated 1000 blocks on "disk1" and 250 blocks on "disk2".

logblocks = nblocks [**on** "diskname"]

This option specifies the number of blocks to allocate for the transaction log. If no value is specified, the default is one zone. The number of blocks actually allocated is rounded up to the first whole number of zones.

The number of blocks specified with this option is in addition to the demand for the rest of the database. A **disk** may be specified.

disk = "diskname"

The **disk** option specifies the disk for the database or the transaction log, depending on whether the disk allocation option is immediately preceded by the **demand** or **logblocks** option. The default is any disk which has sufficient space. The specification

```
with demand = 3000, disk = "abc",
logblocks = 1000, disk = "efg"
```

requests 3000 blocks on disk "abc" for the database and 1000 blocks on disk "efg" for the transaction log.

Portions of a database may be allocated to different disks by listing several pairs of **demand**=*nblocks*, **disk**=*name* options specifying how much of the database is to be located on a given disk.

The order of the options is significant. The order

```
with demand=1000, disk="abc"
```

requests 1000 blocks on disk "abc" for the database whereas

```
with disk="abc", demand=1000
```

requests one zone (the default demand) on disk "abc" and 1000 blocks on any available disk (the default disk).

ascii

This option specifies that the ASCII character set is to be used to store character data in the database. This is the default.

ebcdic

This option specifies that the EBCDIC character set is to be used to store character data in the database.

EXAMPLES

This command creates the database "documents" with a size limit of 7500 blocks. It will reside on "disk1". If there are fewer than 7500 blocks of free space on "disk1", the database will be created with as much space as is available on "disk1".

```
1) open system;
1) create database documents
2)     with demand = 7500 on "disk1";

7500 blocks allocated
```

This command creates a database with 1000 blocks, rounded up to the nearest disk zone size, on any available disk(s). Character data in "db" will be stored in the EBCDIC character set.

```
1) open system;
1) create database db with demand = 1000, ebcdic;

1000 blocks allocated
```

The following command creates a database on the two disks "diska" and "diskb". If neither disk had any free space, the database would not be created.

```
1) open system;
1) create database test
2)     with demand = 2500 on "diska",
3)     demand = 2500 on "diskb";

5000 blocks allocated
```

MESSAGES

illegal command (IDM.E45)
: The user must have permission to create a database, and must have the "system" database open. This command cannot be used in a stored command or in a transaction.

already exists (IDM.E2)
: Database names must be unique.

SEE ALSO

deny, **destroy database**, **extend database**, **permit**
"Dbname", "Options"

create [unique] [nonclustered | clustered] index
[on] rel_name (att_name[, att_name ...]) **[with** options]

DESCRIPTION

Indices are used to provide fast access to data. If tuples in a relation are often searched on the basis of a particular attribute, it is appropriate to create an index on that attribute to reduce access time. The index specifies a particular attribute or set of attributes called keys on which a relation will be searched. For example, if a relation represents a telephone book, one could create an index on the attributes "lastname, firstname". This would speed up the search when data in the telephone book is accessed with a *qualification* based on the "lastname, firstname" attributes.

Indices can be defined as **clustered** or **nonclustered**, and **unique** or non-unique. If none of these are specified, the index is created as **nonclustered** and non-unique by default.

A **clustered index** provides faster access than **nonclustered**, but requires that the data in the relation be stored in an order governed by the key to the index. On creation of a **clustered index**, the data in the relation is sorted according to the values of the attribute(s) specified for the index, and a modified B*-tree index is built. Only one **clustered index** is permitted for a single relation. When the index is created, all existing indices on that relation are destroyed unless the **recreate** option is specified. In addition, when the **clustered index** is created, duplicate tuples (identical in all attributes) are deleted. The maximum size for the keys of a **clustered index** is 252 bytes.

A **nonclustered index** does not physically reorganize the data. Up to 250 nonclustered indices may be created for a single relation. The maximum size for the keys of a **nonclustered index** is 248 bytes.

A **unique index** can be created for relations in which the indexed attributes must be unique. For example, social security numbers are supposed to be unique for all individuals. If a **unique index** has been created for the "social security number" attribute, the user is not permitted to assign to a tuple a social security number which already appears in another tuple in the relation. A **unique index** may be **clustered** or **nonclustered**.

When a **unique index** is being created, the **create index** command is aborted if the database server detects any duplicate values among the indexed attributes. If a **unique index** already exists on a relation and a user tries to modify the indexed relation such that the indexed attributes would no longer be unique, the offending **append**, **replace**, or *copy in* command is aborted. The **delete_dups** option can be used to prevent commands which introduce duplicate keys from aborting.

OPTIONS

delete_dups

If **delete_dups** is specified for a **unique clustered index**, and duplicate values on the indexed attributes are found in the relation while the data is being sorted, as many tuples as necessary are deleted in order to make the index unique. A warning message is displayed, but the **create index** command is not aborted. This option has no effect on a **unique nonclustered index** at the time that the index is being created.

However, if a **unique clustered index** or a **unique nonclustered index** was created with the **delete_dups** option, and a user tries to modify the relation such that the indexed attributes would no longer be unique, the modification does not occur (i.e. the tuple in question is not added or modified). The user is informed that the duplicate was not appended or replaced, but the entire **append** or **replace** command is not aborted. This effect can also be achieved by setting option 6 for the execution of the modification, if the index was not originally created with **delete_dups**.

fillfactor = m

When a **clustered index** is sorted, the relation is written to disk. The **fillfactor** value specifies the percentage of the blocks to be filled when the relation is written to the disk in sorted form. A **fillfactor** can range from 1 (1% of the block is to be filled) to 100 (the block is to be completely filled). The default **fillfactor** is 100. Relations that are known to have a high potential for growth should have a small **fillfactor** specified so the data can be kept physically clustered for as long as possible. If a relation has become scattered (blocks containing data which should be in sort order are spread over several cylinders), I/O time will become large with respect to average read time. When this situation becomes apparent, the **clustered index** should be created again (the old one is automatically destroyed) and a new **fillfactor** specified.

skip = n

The **skip** option indicates the number of blank blocks to leave between data blocks. This option can be used to provide room for growth.

recreate

The **recreate** option deallocates empty pages which were allocated for the creation of a **clustered index**. If **recreate** is specified, the data is not resorted and any **nonclustered** indices on the relation are not destroyed. When the **recreate** option is used, the keys must be the same as the keys of the original index.

nosort

This option specifies that a **clustered index** is to be created on data which is already sorted by the index keys. This option greatly increases the speed with which an index can be created for sorted data. If the **nosort** option is specified and the data is not sorted, an error message is displayed and the index is not created. The user must then create the index without the **nosort** option.

PERMISSIONS

The user must have **create index** permission for the relation and be the owner of the relation.

EXAMPLES

This command causes the "parts" relation to be sorted on (name, number), written on the disk in blocks 40% full, and a B*–tree index created for the (name, number) pairs. When a query specifies (name) or (name, number), only the index and the exact blocks needed are read, not the entire relation.

```
1) create clustered index on parts (name, number)
2)      with fillfactor = 40;
```

The "parts" relation already has a **clustered index** (from the example above). The next command creates a **nonclustered index** on "number" to simplify access to the "parts" relation when "number" alone is specified. It is a **unique** index to enforce the requirement that no two part numbers may ever be the same. If a user tries to modify the relation so that the uniqueness of the "number" attribute were not preserved, the entire **append** or **replace** command is aborted.

```
1) create unique nonclustered index on
2)      parts (number);
```

The next command creates the same type of index as the preceding one. The difference is that if a user tries to modify the relation so that the uniqueness of the "number" attribute were not preserved, the modification would not occur, but the entire command would not be aborted. Instead, a message would inform the user of the modification which was not executed.

```
1) create unique nonclustered index on
2)    parts (number)
3)    with delete_dups;
```

The next command deallocates any unused data pages in the "parts" relation and resets any pointers in the index that point to the deallocated pages. The data is not resorted and the **nonclustered index** on "number" is not destroyed.

```
1) create clustered index on parts (name, number)
2)    with recreate;
```

MESSAGES

illegal command (IDM.E45)
: Only the user who created the relation can create an index on it. The user must have **create index** permission. This command cannot be used in a transaction.

index exists (IDM.E29)
: An index with exactly the same characteristics exists.

out of space (IDM.E42)
: The space for the index is counted in the space for the database.

index too large (IDM.E66)
: The size of an index exceeds the maximum size allowed.

illegal nosort (IDM.E209)
: Unordered data was found in a command specifying the **nosort** option. When the **nosort** option is specified, the tuples must already be ordered by the index keys. This index may be created by not using the **nosort** option.

SEE ALSO

create, destroy index, set
"Att_Name", "Options", "Rel_Name"

create view object_name (target-list) [**where** qualification]

DESCRIPTION

The **create view** command sets up a virtual relation, one that is not a physical entity, but is composed of attributes from one or more relations (called base relations) or other views. A view looks like a relation when it is accessed, but in actuality it never has any data stored in it. It is similar to a temporary relation which is built from its base relations whenever it is accessed. For this reason, when the base relations from which the view is constructed are modified, the modification is reflected in the view. Thus, views are automatically updated.

If the *target-list* is not specified, the attributes in the view will have the same names as the attributes in the base relations. The *target-list* need not be specified unless an attribute in the view is derived from a value more complex than a simple attribute, or the resulting view would have more than one attribute with the same name, or the user wishes to assign new names to the attributes in the view.

A view is often created to access data from multiple relations, to access just a subset of a relation, or to restrict access to certain attributes in a relation. A user can be denied access to a base relation but permitted access to a view built from selected attributes from that base relation.

Views are similar to relations in some aspects. A view may be protected, retrieved, and destroyed in the same manner as a relation. Because a view does not actually contain any data, generally speaking, a view cannot be modified by the **append**, **replace**, and **delete** commands unless the attributes of the view are all simple copies of the attributes of a single base relation. If this is the case, the updates to the view will be reflected in the base relation as well.

Views are recorded in the system relation "query". Since a view is dependent on its base relations, a user cannot destroy a base relation without first destroying any views that refer to it. View definitions may not be "copied" to another database, such that an equivalent view would exist on the other database, referencing similar base relation. If it is desirable to use a single view definition in more than one database, save the view definition in a text file on the host system and use the IDL pseudo-command **%input** to create it in both databases.

When **create view** is executed, the **associate** command is automatically executed also, with the full text of the **create view** command entered by the user inserted as the "text" portion of the description entered by the **associate** command into the "descriptions" relation. This feature provides automated documentation of views.

PERMISSIONS

The creator of the view must have **read** permission on the base relations used to create the view.

EXAMPLE

This command creates a view of parts which need to be reordered. It is composed of two attributes from the "parts" relation and two attributes from the "vendors" relation.

```
1) range of p is parts
2) range of v is vendors
3) create view reorder (p.num, p.name, v.vendor, v.address)
4)     where p.num = v.num and p.qty < 10;
```

MESSAGES

permission denied (IDM.E43)
The user does not have access rights to the view.

illegal command (IDM.45)
This command cannot be used inside a transaction.

SEE ALSO

associate, create, deny, destroy, permit, retrieve
"Object_Name", "Qualfications", "Target-Lists"

define query_name command [command ...] **end define**

DESCRIPTION

The **define** command creates a stored command (also called a stored query). A stored command is a sequence of one or more IDL commands which can be referenced collectively by the *query_name*.

If a sequence of IDL commands is often executed, it is advisable to **define** a stored command for that sequence. Because a stored command is kept in a parsed and partially processed form, it will run faster than would the constituent commands executed individually.

A stored command should also be created if it is desirable to impose *protect-modes* for the stored command which differ from those on the constituent commands. A user can be granted permission to **execute** a stored command without permission to execute all of the consitutent commands individually.

Only certain commands can be used in a stored command. They are:

append
abort transaction
begin transaction
delete
end transaction
range
replace
retrieve
set

Options 1 through 17 are legal inside a stored command. If any of these are **set** inside the stored command, option 15 (use) must have been set prior to defining the stored command which contains the **set** options.

The keyword **go** and the semicolon may never be used in the body of a stored command.

When a stored command is defined, formal parameters can be used in place of constants. A formal parameter has the syntax of a *name* prefixed with a dollar sign ($). Later, when the stored command is executed, the user supplies the values to be substituted for the formal parameters. The order in which the parameters are sorted is discussed in the entry for the **execute** command.

A stored command, once defined, cannot be modified. If a change in the command is desired, a new stored command must be defined.

When **define** is executed, the **associate** command is automatically executed also, with the full text of the **define** command entered by the user inserted as the "text" portion of the description entered by the **associate** command into the "descriptions" relation. This feature provides automated documentation of stored commands.

EXAMPLE

To **define** a stored command:

```
1) define additem
2)     range of i is items
3)     append to items(salesman = $name, amt = $amount)
4)     retrieve(i.salesman, i.amt) where i.amt = $amount
5) end define;
```

To **execute** this stored command:

```
1) additem with name = "barbara",
2)     amount = 47;
```

or

```
1) additem(47, "barbara");
```

or

```
1) execute additem(47, "barbara");
```

In the last two examples, the value 47 is substituted for the "amount" parameter and the value "barbara" is substituted for the "name" parameter. When the attribute name is not explicitly stated in the **execute** command, the values must be listed in this order because of the alphabetical ordering of the parameter names ("a" before "n").

MESSAGES

already exists (IDM.E2)
: A relation, file, view, or stored command has the *query_name* given. All named objects must be unique for each user.

stored command or program too big (IDM.E65)
: The internal representation of the stored command occupies more than 14KB.

illegal command (IDM.E45)
: This command cannot be performed in a transaction.

may not be used in a stored command (IDM.E41)
: The specified command may not be used in a stored command.

SEE ALSO

associate, **destroy**, **execute**, **set**
"Query-Name"

delete range_var [**where** qualification]

DESCRIPTION

The **delete** command removes one or more tuples from a relation. If there is no *qualification* specified, all of the tuples in the relation are removed.

PERMISSIONS

The user must have **write** permission for all the attributes of the relation.

EXAMPLES

This command deletes all the tuples in the "parts" relation in which the value of the "qty" attribute is less than 1.

```
1) range of p is parts
2) delete p where p.qty < 1;
```

The delete command can be extremely powerful. This command deletes every tuple in the "parts" relation.

```
1) range of p is parts
2) delete p;
```

MESSAGES

permission denied (IDM.E43)
The user does not have **write** permission for all attributes of the relation.

view not updatable (IDM.E60)
The relation is really a view and the view is not updatable.

SEE ALSO

append
"Qualifications", "Range_Var"

deny protect_mode [**of** object_name] [**to** user[, user ...]]
deny protect_mode **of** rel_name (att_name[, att_name ...]) [**to** user[, user ...]]

DESCRIPTION

The **deny** command denies a specified type of access to a specified object to a specified *user* or group of *users*. Protections imposed with the **deny** command are recorded in the system relation "protect".

The *user* may be a user name or a group name. A group is any entry in the system relation "users" for which the "uid" is equal to the "gid". If no *users* are specified, the protection applies to all *users*.

When an object is first created, the *protect_modes* are set so that the creator of the object is permitted all types of access while other *users* are denied all types of access.

The *object_name* for which access is being denied may be a relation, view, file, or stored command. If no object is specified, the protection applies to all objects.

A **deny** command overrides any previous **permit** commands which contradict it.

The DBA may also **deny** permission to use the **create**, **create database**, and **create index** commands and to use database server tape.

The *protect_modes* which may be denied are listed under "Protect_Modes" in Part III of this manual.

PERMISSIONS

Only the owner of an object or the DBA may **deny** access.

EXAMPLES

This command specifies that everyone may read the data in the "parts" relation except "george", "harvey" and "mary".

```
1) permit read of parts;
1) deny read of parts to george, harvey, mary;
```

This command denies **write** permission on the "descript" attribute of the "parts" relation to the entire group "clerks". The "clerks" have been previously defined as a group in the system relation "users". Other attributes of the "parts" relation may still be writeable by "clerks".

```
1) deny write of parts (descript) to clerks;
```

MESSAGES

user not found (IDM.E6)
: The *user* specified is not in the "users" relation for this database.

bad protection mode (IDM.E73)
: The protection mode does not make sense with the rest of the command.

not owner (IDM.E44)
: Only the owner of an object or the DBA may **deny** permissions on it. For a view or stored command, the user must be the owner of the relations affected to **deny** permissions.

illegal command (IDM.E45)
: Cannot be in transaction. The protection mode does not make sense with the rest of the command.

SEE ALSO

create, **create view**, **define**, **permit**
"Att_Name", "Protect_Modes", "Rel_Name", "Users"

destroy object_name[, object_name ...]
destroy (target-list) [**where** qualification]

DESCRIPTION

The **destroy** command eliminates relations, views, files, and stored commands. It removes the entire object from the database, and frees its space for use by another object within the database.

If there are views or stored commands that depend on the relation or view to be destroyed, they must be destroyed first.

The first form of the **destroy** command is used when an entire object is to be destroyed with no *qualification*.

The second form requires a range variable and can take a *qualification* of the objects to be destroyed.

PERMISSIONS

Only the owner of the object or the database administrator can **destroy** an object.

EXAMPLES

This command destroys the "parts" and "products" relations.

```
1) destroy parts, products;
```

This command makes use of the system relation "relation". The "relation" relation contains information about each relation in the database, including the relation names and owners. This use of the **destroy** command destroys all relations owned by the user.

```
1) range of r is relation;
1) destroy (r.name) where (r.owner = userid);
```

MESSAGES

not owner (IDM.E44)
: Only the owner or DBA may destroy an object.

has dependencies (IDM.E72)
: There are dependent objects that must be destroyed first.

is open (IDM.E5)
: An object that is being accessed may not be destroyed.

not found (IDM.E6)
: The object could not be found.

illegal command (IDM.E45)
: This command cannot be issued from within a transaction.

SEE ALSO **create, create view, define, destroy database**
"Object_Name", "Qualifications", "Range_Var", "Target-Lists"

destroy database dbname[, dbname ...]

DESCRIPTION

The **destroy database** command removes the specified databases from the system and frees the space that was allocated for them. It destroys all relations and files in the specified database(s).

The database to be destroyed cannot be open at the time that the **destroy database** command is executed. The command must be executed from the "system" database.

The "system" database cannot be destroyed with the **destroy database** command.

PERMISSIONS

To destroy a database, the user must be the owner of the database (as specified in the "system" database relation "databases") or the owner of the "system" database.

EXAMPLE

This command destroys the database "inventory" and frees all disk space which was allocated to it.

```
1) destroy database inventory;
```

MESSAGES

not owner (IDM.E44)
: Only the owner of the database, or the owner of the "system" database, can destroy the database.

is open (IDM.E5)
: Someone is using the database.

illegal command (IDM.E45)
: Cannot use in transaction. The command must be executed from the "system" database.

SEE ALSO

create database, **destroy**
"Dbname"

destroy [unique] [nonclustered | clustered] index
[on] rel_name (att_name[, att_name ...])

DESCRIPTION

The **destroy index** command removes an index from a relation. This might be desirable if the index is seldom used to free the space occupied by its B*-tree for other applications and to eliminate the overhead of updating it whenever the tuple attributes that it indexes are updated.

The index is identified by its description: whether it is **unique**, **clustered** or **nonclustered**, and by the attributes that it indexes.

PERMISSIONS

The user must be the owner of the relation.

EXAMPLES

This command destroys the index on (name, number) for the "parts" relation. Initially the relation remains sorted on (name, number) as it was when it had its indices, but subsequent to the destruction of the indices, new data is appended at the end of the relation.

```
1) destroy clustered index
2)     on parts (name, number);
```

This command destroys the **unique nonclustered index** on the "number" attribute of the "parts" relation.

```
1) destroy unique nonclustered index
2)     on parts (number);
```

MESSAGES

not owner (IDM.E44)
The user must be the creator of the relation.

illegal command (IDM.E45)
Cannot use in transaction.

not found (IDM.E6)
The named relation or attributes were not found.

index does not exist (IDM.E30)
The index does not exist as specified (the clustering or the arrangement of attributes is incorrect).

SEE ALSO

create index
"Att_Name", "Rel_Name"

end transaction

DESCRIPTION

The **end transaction** command ends an atomic sequence of commands that that was initiated with a **begin transaction**. The results of the transaction are made permanent.

EXAMPLES

See **begin transaction**.

MESSAGES

illegal command (IDM.E45)
Must be used after a **begin transaction** command.

SEE ALSO

abort transaction, begin transaction

[**execute**] query_name [**with**] [name = constant[, name = constant ...]]
[**execute**] query_name [**with**] [constant[, constant ...]]

DESCRIPTION

The **execute** command executes the stored command ***query_name***, which was previously created with the **define** command.

The keyword **execute** may be omitted, provided that *query_name* does not conflict with the name of any IDL command.

The *constants* specify values to be substituted for the formal parameters supplied in the definition of the stored command.

If the *name = constant* form is used, the *name* must correspond to the name of a formal parameter as it was specifed in the **define** command. The *name = constant* assignments may be given in any order.

For example, if a stored command "mycommand" were defined as

```
1) define mycommand
2)     append to emps(name = $empname,
3)     num = $empnum, dept = $deptnum)
4) end define;
```

an **execute** command could look like

```
1) execute mycommand with empname = Smith,
2)     empnum = 2456, deptnum = 102;
```

If the *constant* form (no explicit *name*) is used, values are assigned based on the alphabetic order of the names of the formal parameters. For example, to execute "mycommand" using this form and obtain the same results as in the example above, "mycommand" would have to be invoked as

```
1) execute mycommand 102, Smith, 2456;
```

When this form is used, the order in which the values are listed is crucial, because the mapping of values to formal parameters is determined by the alphabetic ordering of parameter names. The digits in parameter names are considered characters, not numbers, so the parameters $1, $2, $3, $10, $20 sort as $1, $10, $2, $20, $3.

It is not necessary to enclose string constants in quotation marks if they contain only alphabetic, numeric, and underbar characters.

PERMISSIONS

The user needs only **execute** permission for the stored command, if the creator of the stored command owns all of the objects referenced by the stored command.

EXAMPLES

Assume that the stored command "update" has been defined as follows:

```
1) define update
2) append to expend (salesman = $name,
3)     amt = $amount, time = gettime,
4)     date = getdate)
5) end define;
```

This stored command can be executed as follows:

```
1) execute update with name = "mike",
2)      amount = 44;
```

or

```
1) update 44, "mike";
```

In the second example, the arguments must be given in this order because the alphabetic ordering of the parameters is "$amount", "$name". The keyword **execute** is optional because "update" does not conflict with any IDL command.

MESSAGES

not found (IDM.E6)
: The command was not found in the current database.

missing parameter (IDM.E23)
: The user has tried to execute a stored command without entering required parameters. A parameter was sent that was not in the stored command.

too many parameters (IDM.E36)
: Exceeded number of parameters in the stored command.

permission denied (IDM.E43)
: The user must have **execute** permission on the stored command and appropriate permissions for the commands comprising the stored command.

other messages
: Since executing the stored command causes other commands to be executed, they may give error messages. Consult the MESSAGES section under the appropriate command.

SEE ALSO

define
"Constants", "Query-Name"

exit

DESCRIPTION

The **exit** command exits the IDL parser. The **exit** command may be used anywhere in a command.

If the **exit** is issued inside a transaction, the user is warned that the transaction has been interrupted and all pending commands have been aborted.

extend rel_name [**with logging** [= {**0** | **1**}]]

DESCRIPTION

The **extend** command controls transaction logging of the relation *rel_name*.

If **logging** is set to to 1, the transaction log "transact" is to be updated whenever the relation is updated. If **logging** is set to 0, "transact" is not maintained, and updates are recorded in the system relation "batch". If the **logging** option is used but neither 0 nor 1 is specified, the default is 1.

PERMISSIONS

The user must be the owner of the relation.

EXAMPLE

1) extend unimportant with logging = 0;

MESSAGE

not relation (IDM.E70)
Only relations can be extended.

permission denied (IDM.E43)
Only the DBA can turn logging on.

can't extend system relation: %s (IDM.E81)
System relations cannot be extended.

SEE ALSO

create
"Options", "Rel_Name"

extend database dbname [**with** options]

DESCRIPTION

The **extend database** command increases or decreases the allocation for the database *dbname*. Since allocation is made by whole zones only, the number of blocks actually allocated is rounded up to the next multiple of the number of blocks per zone.

The *options* are the same as for the **create database** command except that the **demand** may be negative if deallocation is desired. Only entirely freeable zones, those containing no pages which are either used or demanded, are removed from the database. If a **disk** option is specified with a negative **demand** option, storage is deallocated only from freeable zones on the specified disk(s). If no **disk** option is specified, deallocation is from zones belonging to the database which reside on any disk(s).

The actual number of blocks allocated or deallocated is displayed at the terminal.

If both positive and negative demands are made in the same **extend database** command, the negative demands are processed first.

A database may be extended while others are using it.

The **extend database** command must be executed from the "system" database.

If the options are omitted, **extend database** increases the value of the **demand** option by one zone on any available disk.

OPTIONS

See **create database**.

PERMISSIONS

The user must be the owner of the database being extended.

EXAMPLES

This command increases the size of the "accounts" database by 2000 blocks.

```
1) open system;
1) extend database accounts
2)     with demand = 2000;
```

This command removes from the "accounts" database a total of 3500 blocks from "diska" and "diskb" and allocates 1500 blocks on "diskc".

```
1) open system;
1) extend database accounts
2)     with demand = -3500, disk = "diska", disk = "diskb",
3)     demand = 1500, disk = "diskc";
```

MESSAGES

illegal command (IDM.E45)
: Must be in "system" database. Must not be in transaction. Must have **create database** permission.

permission denied (IDM.E43)
: Must be owner of database.

out of space on disk (IDM.E42)
: There is no more room to extend the database on the specified disk.

SEE ALSO

create database, **destroy database**
"Dbname", "Options"

open dbname

DESCRIPTION

The **open** command opens a database for activity. The opened database will remain open until the user enters another **open** command specifying a different database, or until the IDL session is terminated.

To execute any IDL commands other than **range** or **set**, a database must first be opened.

PERMISSIONS

There must be an entry for the user's host id in the database's system relation "host_users".

EXAMPLE

```
1) open vino;
1) append to kinds
2)      (type = "chardonnay", color = "white");
```

MESSAGES

not found (IDM.E6)
: The database does not exist.

permission denied (IDM.E43)
: The user does not have an entry in the system relation "host_users".

database is locked (IDM.E53)
: The administrator is temporarily locking out users to do maintenance.

SEE ALSO

create database, **retrieve**
"Dbname"

permit protect_mode [**of** object_name] [**to** user[, user ...]]
permit protect_mode **of** rel_name (att_name[, att_name ...]) [**to** user[, user ...]]

DESCRIPTION

The **permit** command permits access to an object to a specific *user* or a group of *users*. The *user* may be a user name or a group name. A group is any entry in the system relation "users" for which the "uid" is equal to the "gid". If no *user* is specified, the permission applies to all *users*.

By default, access is permitted to the owner of an object and denied to other *users* when the object is created. To allow other *users* access to an object, the owner must explicitly **permit** such access. The *object_name* may refer to a relation, view, file, or stored command.

The DBA may also **permit** use of the **create**, **create database**, and **create index** commands and of database server tape.

Access to a view or stored command implies access to all objects that the view or stored command references only if the owner of those objects and the view or stored command is the same.

The *protect_modes* permitted are listed under "Protect_Modes". A **permit** command supersedes any previous **deny** commands which contradict it.

PERMISSIONS

The user must be the owner of a relation, view, or stored command in order to control permission over it. If permission is granted for a database, the user must be the DBA or owner of the database.

EXAMPLES

Permit on a relation:

The *user* "george" can read the "parts" relation.

```
1) permit read of parts to george;
```

Permit on an attribute:

The *users* "bill" and "sharon" can write to the "quan" attribute of the "parts" relation.

```
1) permit write of parts (quan)
2) to bill, sharon;
```

Permit of a stored command:

The *user* "dave" and all *users* in the group "managers" are the only *users* permitted to execute the stored command "getsum".

```
1) deny execute of getsum;
1) permit execute of getsum to managers;
1) permit execute of getsum to dave;
```

Permit with no object specified:

The *user* "gloria" may create relations in the open database.

```
1) permit create to "gloria";
```

Permit for all *users*:

When no *user* is specified, all *users* of the open database are permitted to create relations.

```
1) permit create;
```

MESSAGES

unknown user (IDM.E6)
: The system relation "users" for the currently open database must include the *user* (or group) specified.

not owner (IDM.E44)
: Only the owner or the DBA may grant permissions on an object.

not found (IDM.E6)
: The object or attribute specified was not found.

illegal command (IDM.E45)
: Cannot be done from transaction. Illegal protection mode for an object.

bad protection mode (IDM.E73)
: The protection mode does not make sense for the object.

result variable does not exist
: Object was not specified where it was needed.

SEE ALSO

create, **define**, **deny**
"Att_Name", "Object_Name", "Rel_Name", "Users"

range of range_var **is** rel_name [**with** options]

DESCRIPTION

The **range** statement associates a variable name supplied by the user with the name of a relation or view. Several commands require that a relation be referenced through a range variable rather than the relation name. The **retrieve**, **replace**, and **delete** commands all require a range variable while **append** and **truncate** require the relation name.

The user may use up to sixteen range variables in a single query.

OPTIONS

minlock

This option specifies minimum locking, in which data may be retrieved from the relation identified by *range_var* while another user is modifying the relation. This may result in the retrieval of some tuples that have been affected by a command and some that have not. The **minlock** option is useful in situations in which this type of inconsistency is not a problem and where other users' activities would interfere with simple retrievals were the option not used.

fulllock

This option specifies a full locking. It guarantees that any data retrieved with the specified range variable will reflect either completely or not at all the effects of other users' transactions. The **fulllock** option is the default if no options are specified.

dindex = *n*

This option specifies that the relation or view is to be accessed using the specified index. The **clustered index** is always index 0, and others are numbered from 1 to 15. The numbers of the indices correspond to the "indid" attribute of the "indices" relation for the database. If the **dindex** option is used, the **dorder** option is also required. If the **dindex** is omitted, the database server decides which index would be most efficient. Unless the join is extremely complicated (involves four or more relations), it is usually preferable to let the database server choose the index.

dorder = *n*

This option is used to specify the order in which relations should be processed when two or more relations are joined in a *qualification*. When the **dorder** option is omitted, the database server decides in which order to process relations. Unless the join is extremely complicated (involves four or more relations), it is usually preferable to let the database server choose the order.

EXAMPLES

The range statement below associates the range variable "p" with the relation "products". The **retrieve** command uses the range variable "p".

```
1) range of p is products;
1) retrieve (p.name);
```

The next statement associates the variable "p" with the relation "products" which is owned by user "bill". This is to distinguish the relation from other relations called "products" which may be owned by other users. Several users may own completely different relations with the same name in the same database. If the owner's name is not specified then the object is presumed to be owned by the current user or by the DBA.

```
1) range of p is products:bill;
```

A range variable is associated with the relation in the last range statement defining it. Here the variable "t" is bound to the relation "parts" at the end of this sequence.

```
1) range of t is temp;
1) range of t is newtemp;
1) range of t is parts;
```

The following query uses the **dindex** and **dorder** options to establish a plan for accessing the "small", "medium", and "large" relations.

```
1) range of s is small with dindex = 0, dorder = 1;
1) range of m is medium with dindex = 0, dorder = 2;
1) range of l is large with dindex = 4, dorder = 3;
1) retrieve (s.desc, m.name, l.quan)
2) where s.pos < 10
3) and s.num = m.num
4) and m.type = l.type;
```

This means:

(1) First, go through "small", searching for tuples in which the "pos" attribute is less than 10. Access "small" through its **clustered index**, which is on "pos".

(2) Second, from among those tuples retrieved above, go through "medium" searching for matches between "m.num" and "s.num". Access "medium" through its **clustered index**, which is on "num".

(3) Among those tuples retrieved above (in which "s.pos" is less than 10 and "s.num" equals "m.num") go through "large" looking for matches between "m.type" and "l.type". Access "large" through its fourth **nonclustered index** which is on "type".

MESSAGES None.

SEE ALSO **delete**, **replace**, **retrieve**
"Options", "Range_Var", "Rel_Name"

reconfigure

DESCRIPTION

The **reconfigure** command updates the configuration of the database server according to the contents of the "system" database relation "configure".

This command may only be issued from the "system" database.

PERMISSIONS

The user must be the DBA of the "system" database.

MESSAGES

illegal command (IDM.E45)
The user was not in the "system" database or the user is not the DBA of the "system" database.

SEE ALSO

IDM Installation Guide
idmconfig(1i) in *Host Software Specification*
IDMCONFIG in *Command Summary*

replace range_var (target-list) [**where** qualification]

DESCRIPTION

The **replace** command replaces the value of one or more attributes in zero or more tuples of a relation.

The *target-list* may reference literal values or values in attributes in other relations.

PERMISSIONS

The user must have **write** permission on the attributes to be replaced.

EXAMPLES

Qualification involving a single relation:

The following commands change the "name" attributes for all tuples in the relation "parts" for which the "name" fields begin with a "t" to the value "electronic".

```
1) range of p is parts;
1) replace p (name = "electronic")
2)     where p.name = "t*";
```

Qualification involving multiple relations:

This command changes the value of the "cost" attribute for each tuple in the "parts" relation in which the following conditions prevail: (1) the value of the "name" attribute in the "parts" relation equals the value of the "part" attribute in a tuple in the "products" relation and (2) the "name" attribute in that tuple in the "products" relation has the value "TV". The purpose of this command is to increase by 10% the cost of each part that is used in manufacturing a TV. No modification is made to the "products" relation.

```
1) range of p is parts;
1) range of pr is products;
1) replace p (cost = p.cost + p.cost / 10)
2)     where p.name = pr.part and pr.name = "TV";
```

MESSAGES

permission denied (IDM.E43)
: User must have **write** permission on the relation.

not found (IDM.E6)
: The specified relation or attribute was not found.

wrong type attribute (IDM.E19)
: An expression that the user specified for a target could not be converted to the type of the requested attribute.

view not updatable (IDM.E60)
: The view cannot be updated because the result of such an update could not be unambiguously resolved.

SEE ALSO **append, audit, delete, pattern, range**
"Qualifications", "Range_Var", "Target-Lists"

reset

DESCRIPTION

The **reset** command resets the command buffer without sending anything to the database server. It is useful for throwing away erroneous commands.

The **reset** command may be entered anywhere in a command.

EXAMPLE

```
1) range of c is coump
2) retrieve (c.all)
3) where c.salary > 2000 and reset
1)
```

Here the user has typed three lines before realizing that "count" is misspelled. Entering **reset** causes the input to be ignored and the line number to be reset to 1.

retrieve [unique] [[into] rel_name] (target-list)
[order by order_spec[, order_spec ...]] **[where** qualification]

DESCRIPTION

The **retrieve** command is used for fetching data from the database server. The simple **retrieve** returns the data to the host. The **retrieve into** command sends data to a newly created relation containing the attributes specified in the *target-list*. It is an error to **retrieve into** an existing relation.

The user can reference up to 15 relations in one **retrieve** command, if the relations are all in the same database.

If a *target-list* is used, it is necessary to use a *range_var* to specify *targets*.

The **unique** option specifies that duplicate tuples are to be removed in the result. Duplicate tuples are defined here as tuples that are equal in all attributes.

Order By

The optional **order by** clause specifies the order in which the returned tuples are sorted. The direction of the ordering can be specified with **a** or **asc** for ascending and **d** or **desc** for descending. The default is ascending order. The *order_spec* can be either

target[:direction]

or

expression[:direction]

If the *expression* is an integer *i*, the output are sorted by the *i*th item in the *target-list*. The query

```
1) retrieve (x.num, x.name, x.quan)
2)     order by 3;
```

displays its results ordered by the value of the "quan" attribute, the 3rd element in the target-list.

The attribute(s) by which data is to be ordered must be referenced with a range variable if the attribute is not explicitly used in the *target-list*. For example,

```
1) retrieve (x.name, c.cost)
2)     order by name;
```

will work because "name" is in the *target-list* as "x.name" but

```
1) retrieve (x.all)
2)     order by name;
```

will not work because "name" is not explicitly referenced in the *target-list*. In this case, the specification must be

```
1) retrieve (x.all)
2)     order by x.name;
```

To copy a large amount of data from a relation on the database server to a host file, use the host utility **idmfcopy**.

PERMISSIONS

For a **retrieve**, the user must have permission to **read** all the domains in the query. For a **retrieve into**, the user must also have **create** permission.

EXAMPLES

Retrieve ordered by *target-name*:

In the following example, the database server first calculates the average value of the "cost" field in the relation "parts". Then the database server accumulates the "name" and "cost" attributes of the tuples that contained a "cost" greater than the average. These are sorted by the value in the "cost" attribute, largest value first, and sent to the host where the data is displayed at the terminal.

```
1) range of p is parts;
1) retrieve (p.name, p.cost)
2)   order by cost:d
3)   where p.cost > avg (p.cost);
```

Retrieve ordered by *expression*:

This command retrieves all the attributes in the "accounts" relation, sorting them in descending order by the value of the difference between the "assets" attribute and the "liabilities" attribute. Duplicate tuples are not displayed in the result.

```
1) range of a is accounts;
1) retrieve unique (a.all)
2)     order by (a.assets - a.liablities):d;
```

Retrieve ordered by *target* specified by position in *target-list*:

This command retrieves four attributes from the "parts" relation, ordered by the "name" attribute (the second element in the *target-list*).

```
1) range of p is parts;
1) retrieve (p.num, p.name, p.cost, p.quan)
2)     order by 2;
```

Retrieve into:

This creates a new relation "exp_parts" in the open database composed of the "name" and "cost" attributes from the "parts" relation. The data from those attributes is copied into the new relation from every tuple in which the value of the "cost" attribute exceeds the average cost of all the parts in the relation.

```
1) range of p is parts;
1) retrieve into exp_parts(p.name, p.cost)
2)     order by cost:d
3)     where p.cost > avg (p.cost);
```

Ordering by more than one attribute:

In this example, the data is sorted by "group" and within each group it is sorted by "name".

```
1) retrieve (b.all)
2) order by b.group, b.name;
```

MESSAGES

permission denied (IDM.E43)
: The user must have **read** permission on all domains in the query.

not found (IDM.E6)
: The named attribute or relation was not found.

SEE ALSO

append, **audit**, **create**, **range**
"Qualifications", "Rel_Name", "Target-Lists"
idmfcopy(1I) in *Host Software Specification*
IDMFCOPY in *Command Summary*

set {option-number | option-name}[, {option-number | option-name} ...]

DESCRIPTION

This command enables certain options for IDL commands. The *option-number* or *option-name* must be chosen from the following list. *Option-names* may be in upper or lower case.

1 format
Set format before query. This option is **set** by database server software and cannot be **unset**.

2 names
Send result names. This option is **set** by database server software and cannot be **unset**.

3 overflow
Ignore overflow and use largest number instead.

4 divzero
Ignore division by zero and use largest number instead.

5 perform
Send elapsed execution time (wall clock). Do not set 5 if 11 is set.

6 duplicate
Delete tuples with duplicate keys which are generated by modifications to the relation.

7 round
Abort on rounding of bcdflt.

8 underflow
Ignore exponent underflow and use zero instead.

9 badbcd
Ignore bad bcd data from host or file and use zero instead.

11 time
Return dedicated time (database server CPU time). Do not set 11 if 5 is set.

12 nocount
Supress count of tuples effected when displaying query results.

13 "tape"
Use database server tape. If the *option-name* is used here, it must be quoted. This option can not be set from a user program.

14 protect
Allow DBA of the "system" database to access any database as DBA.

15 use
This is for options **set** within a stored command. To enable options at execution time, option 15 must be set prior to defining the stored command. Then, the options are enabled when the stored command is executed.

16 dumpwait
Wait for execution of command while a read-only dump is in progress.

17 fastagg
Process aggregates using faster method, with possible loss of accuracy in the result. If this option is **set**, queries may return inconsistent results.

18 crossjoin
Process joins using an older method. This may improve performance for certain queries which (1) join several small relations with one large relation, (2) but do not join the small relations with each other, (3) and have very few qualifying tuples in each small relation, (4) and can use a selective index to access the large relation.

33 resp
Return response time (in 60ths of a second) from when the DBP gets the command to when it sends the last of the results.

34 cpu
Return CPU use (in 60ths of a second).

37 inp
Return the time the dbin spent waiting for input from the start of the command (in 60ths of a second).

38 mem
Return the time the dbin spent waiting for memory after receiving a command (in 60ths of a second).

39 cpuw
Return the time the dbin spent waiting for the DBP or DAC when it had CPU work to do (in 60ths of a second).

40 disk
Return the time spent waiting for the disk (in 60ths of a second).

41 tapew
Return the time spent waiting for the tape (in 60ths of a second).

42 outw
Return the time spent waiting for the host to read its output (in 60ths of a second).

43 block
Return the time spent blocked on another dbin (in 60ths of a second).

44 dac
Return the time spent in the DAC or the simulation routines if there is no DAC in the system (in 60ths of a second).

45 outc
Return the time spent waiting for an output buffer (in 60ths of a second).

46 hits
Return the number of times a disk page was found in memory.

47 reads
Return the number of disk reads performed by this dbin.

48 tperrs
Return the number of soft tape errors.

49 qrybuf
Return the number of bytes of query buffer used.

60 plan
Return the query processing plan.

EXAMPLE

The following command causes execution time to be displayed at the user's terminal following each IDL command.

```
1) set perform;
1) range of k is kinds;
1) retrieve (k.all) where k.body = "full";
```

type	color	flavor	body
zinfandel	red	dry	full
port	red	sweet	full

```
2 tuples affected
-- 850 ms --
```

MESSAGES

option already set
: The option was already **set** by default or by a previous **set** command.

cannot set/unset "tape" option
: The database server tape option may not be **set** in **this** context. To **set** this option from a user program, use the **itaddopts** or itapeopts interface instead.

SEE ALSO

unset

sync

DESCRIPTION

This command creates a checkpoint in the open database or, if no databases are open, a checkpoint in all databases which are currently active. Any disk blocks that may have temporarily been kept in volatile RAM are written out to disk.

EXAMPLE

1) sync;

truncate rel_name[, rel_name ...]

DESCRIPTION

The **truncate** command deletes all tuples from a relation. It takes a relation name, rather than a range variable, as its argument.

This command is the functional equivalent of the **delete** command except that **truncate** can empty several relations with a single command. The deleted tuples are not recorded in a transaction log, so it is not possible to audit the tuples which were removed.

The **truncate** command may be executed from within a stored command, but it may not be used inside a transaction because it is not possible to back out the deletions.

PERMISSIONS

Only the owner of the relation being truncated or DBA may issue this command.

EXAMPLE

1) truncate oldparts, oldinvoices;

MESSAGES

not owner (IDM.E44)
User is not the owner of the relation or the DBA.

not relation (IDM.E70)
Only relations can be truncated.

illegal command (IDM.E45)
The command cannot be executed inside a transaction.

relation is unavailable (IDM.E26)
Another user is accessing the relation.

system relation (IDM.E57)
System relations cannot be truncated.

SEE ALSO

delete
"Rel_Name"

unset {option-number | option-name} [, {option-number | option-name} ...]

DESCRIPTION

This command disables options previously implemented with a **set** command. For a list of the *option-numbers* and *option-names* for options which can be **unset**, consult the entry for **set**.

EXAMPLE

1) unset perform;

IDL commands will no longer display the time they have taken to execute on the database server.

MESSAGES

option does not exist (IDM.E78)
: The specified option is already unset by default or by a previous **unset** command.

cannot set/unset "tape" option
: The database server tape option may not be set in this constext. To set this option from a user program, use the itaddopts or itapeopts interface instead.

SEE ALSO

set

PART III

IDL GENERAL CONCEPTS

Introduction to IDL
General Concepts

This part of the manual describes various components of an IDL command, such as *expression* or *qualification*, which may appear as arguments in a number of different IDL commands. The definition and use of these components are described here.

Aggregates

An *aggregate* has the following syntax:

```
aggregate_operator(expression
    [by expression1 [, expression2 ... ]]
    [where qualification])
```

The aggregate operators in IDL are:

Aggregate Operator	*Returns*
sum()	sum of all elements
sum unique()	sum of all unique elements
sumu()	same as sum unique
count()	count of elements
count unique()	count of unique elements
countu()	same as count unique
avg()	average of elements
avg unique()	average of unique elements
avgu()	same as avg unique
once()	returns one and only one value; if more or less than one value is found, then an error results
once unique()	once of unique elements
onceu()	same as once unique
any()	0 if no elements; 1 if one or more elements
max()	maximum of elements
min()	minimum of elements

The **sum**, **avg**, **sum unique**, and **avg unique** aggregate operators are available only with those data types that have addition (integer, bcd, or bcdflt). The other aggregate operators are available on all data types.

A simple *aggregate* with no **by** clause returns a single value as in

```
1) range of p is pricings;
1) retrieve (avgprice = avg(p.price));
```

avgprice
7.56

which computes a single tuple with one domain called "avgprice", the value of which is the average price of all of the wines in the "pricings" relation. This type of *aggregate* can be modified with an optional **where** clause:

```
1) range of p is pricings;
1) retrieve (avgprice = avg(p.price
2)       where p.year = 1982));
```

avgprice
8.02

This computes the average price of the 1982 wines in "pricings" relation.

An *aggregate* with an optional **by** clause, returns multiple values, one for each group identified by the **by** clause. This query yields a separate count value of each type of wine in the "wines" relation.

```
1) range of w is wines
2) retrieve (num = count(w.onhand by w.type), w.type);
```

num	type
1	beauclair
2	burgundy
5	cabernet sauvignon
1	chablis
4	chardonnay
4	chenin blanc
3	fume blanc
1	gamay beaujolais
1	grenache rose
7	johannisberg riesling
1	petite sirah
1	pinot chardonnay
1	scheurebe
6	zinfandel

EXAMPLES

The **sum** *aggregate* adds the attributes of several tuples and returns the result.

The "wines" relation in the "vino" database has an attribute named "onhand", which contains the number of cases of each wine available. The following query uses the **sum** aggregate operator to find the total number of cases on hand.

```
1) open vino;
1) range of w is wines;
1) retrieve (total = sum(w.onhand));
```

total
287

The following query retrieves only the tuples in which the "vintage" attribute is 1980 for calculation by the *aggregate*. This is done by including a *qualification* inside the parentheses.

```
1) retrieve (total80 =
2)       sum (w.onhand where w.vintage = 1980));
```

total80
82

The next query specifies a breakdown of how the information should be computed and displayed using the **by** clause. It retrieves the sum of the onhand attributes for each area.

```
1) retrieve (total =
2)       sum(w.onhand by w.area), w.area);
```

total	area
11	amador
42	california
3	lake
13	mendocino
11	monterey
169	napa valley
12	san benito
26	sonoma

The following query uses both the **by** and **where** clauses.

```
1) retrieve (total80 =
2)       sum(w.onhand by w.area where w.vintage = 1980),
3)       w.area);
```

total	area
0	amador
18	california
0	lake
4	mendocino
11	monterey
40	napa valley
12	san benito
9	sonoma

Queries containing *aggregates* can become quite complex. The following query retrieves the number, type, and total cost of all the wines displayed where the total cost of a wine is greater than the average of the total costs of all the wines:

```
1) range of w is wines;
2) range of p is pricings;
1) retrieve (w.winenum, w.type, total =
2)       sum(p.price * w.onhand by w.winenum
3)       where p.winenum = w.winenum))
4)       where sum (p.price * w.onhand by w.winenum
5)       where p.winenum = w.winenum)
6)       > avg (sum (p.price * w.onhand by
7)       w.winenum where p.winenum = w.winenum));
```

winenum	type	total
1	johannisberg riesling	22.50
3	grenache rose	42.00
5	beauclair	49.00
6	johannisberg riesling	102.00
7	chardonnay	12.00
9	gamay beaujolais	45.00
10	burgundy	90.00
11	johannisberg riesling	55.00
15	zinfandel	57.00
23	cabernet sauvignon	57.00
26	chardonnay	93.50
28	chardonnay	287.50
31	pinot chardonnay	29.70
33	johannisberg riesling	88.75

There may be more than one *expression* in the **by** clause, in which case separate *aggregates* are calculated for each combination of values in the **by** clause.

An *aggregate* with **by** clauses can be powerful and also extremely confusing. There is one important item to remember: the database server optimizes its queries heavily. Since duplicate tuples and tuple order are irrelevant, slightly different queries may produce different looking results simply because some algorithms the database server chooses cause duplicates to be deleted. The user can introduce some consistency by having tuples ordered and by using **retrieve unique** when retrieving *aggregates*.

Each aggregate operator available in IDL is briefly described on the following pages.

any()

The **any** operator returns 0 if none of the elements in its argument exist, 1 if at least one element exists. The choice of attributes among those comprising the relation being accessed is irrelevant.

In order to find out if any wines in the database date from before 1970:

```
1) retrieve (old = any(w.winenum
2)   where w.vintage < 1970));
```

old
0

avg(), avg unique()

The **avg** operator returns the average of all elements of its argument. All of the elements being averaged must be of type integer, bcd, or bcdflt. The **avg unique** operator returns the average of all of the unique elements of its argument.

For example, to find the winenumbers and cases on hand for all zinfandels where the number of cases on hand is less than the average number on hand for zinfandels:

```
1) retrieve(w.winenum, w.onhand)
2 )  where w.type = "zinfandel" and
3)   w.onhand < avg(w.onhand where w.type = "zinfandel");
```

winenum	onhand
4	1
38	3

count(), count unique()

The **count** operator returns the number of tuples in which its argument occurs. The **count unique** operator returns the number of tuples in which its argument occurs, excluding duplicate occurences of the element(s) being counted. For the **count** ***aggregate*** (but not the **count unique**), the choice of attributes among those comprising the relation being accessed is irrlevant.

This example counts all of the tuples in which the "vintage" attribute has a value of 1980:

```
1) retrieve(vintage80 = count(w.type
2)   where w.vintage = 1980));
```

vintage80
15

The following query counts all of the tuples in which the "vintage" attribute is 1980 but counts only once for each "type". For instance, for the three wines of 1980 vintage in which the "type" attribute has a value of "johannisberg riesling", there will be only one count. This is because the **count unique** is based on the "type" attribute.

```
1) retrieve(vintage80 = count unique(w.type
2)   where w.vintage = 1980));
```

vintage80
9

max()

The **max** operator returns the element with the maximum value. If the elements are character data types, the maximum is calculated on ASCII or EBCDIC order, depending on the character set associated with the database when it was created.

For example, to find the wine of which the greatest number of cases are in stock:

```
1) retrieve (w.winenum, w.type, w.onhand)
2)   where w.onhand = max(w.onhand);
```

winenum	type	onhand
28	chardonnay	23

min()

The **min** operator returns the element with the minimum value. If the elements are character data types, the minimum is calculated on ASCII or EBCDIC order, depending on the character set associated with the database when it was created.

For example, to find the least expensive wine in the database:

```
1) retrieve (p.winenum, p.price, w.type)
2)   where p.price = min(p.price)
3)   and p.winenum = w.winenum;
```

winenum	price	type
4	4.	sinfandel

once(), once unique()

The **once** operator returns one value if one occurrence of its argument exists. Otherwise it generates an error message. The **once unique** operator returns one value for one occurrence of a unique element.

```
1) retrieve (old_cab = once(w.winenum where
2)   w.vintage < 1978 and
3)   w.type = "cabernet sauvignon"));
```

old_cab

ERROR line 2: ONCE or ONCEU returned two values.

```
1) retrieve(old_napa_cab = once(w.winenum where
2)   w.vintage < 1978 and
3)   w.type = "cabernet sauvignon" and
4)   w.area = "napa valley"));
```

old_napa_cab
34

sum(), sum unique()

The **sum** operator returns the sum of all elements of its argument. All of the elements being summed must be of type integer, bcd, or bcdflt. The **sum unique** returns the sum of all of the unique elements of its argument.

SEE ALSO "Expressions", "Functions", "Qualifications", "Range_Var"

Att_Name

An *att_name* refers to an attribute of a relation. An *att_name* has the syntax of a *name*.

All the *att_names* in a database are listed in the system relation "attribute".

SEE ALSO

create, **create index**, **deny**, **destroy index**, **permit**
"Name", "Target-Lists"

Constants

A *constant* is a value that remains unchanged throughout the execution of a command. *Constants* are used in *expressions* and as arguments to the **execute** command. There are eight different types of *constants*:

Integer Constant

An integer constant is a sequence of decimal or hexadecimal digits. It may be preceded by "0o" or "0x" to indicate octal or hexadecimal values:

```
4          0o777
43         0x4E
```

Character Constant

A character constant is a sequence of characters enclosed in single or double quotation marks:

```
"Henry"    "a,b,c"
'x'        '123'
```

To include a single quotation mark (apostrophe) inside a character constant, either place the entire character constant in single quotation marks, and double the single quotation mark which is to appear inside the constant

```
'Britton Lee''s software'
```

or use double quotation marks around the character constant and a single quotation mark where it is to appear in the constant

```
"Britton Lee's software"
```

To include double quotation marks inside a character constant, either place the entire character constant in double quotation marks, and double the double quotation mark which is to appear inside the constant

```
"The word ""word"" is in double quotation marks."
```

or use single quotation marks around the character constant and double quotation marks around the part to be quoted

```
'The word "word" is in double quotation marks.'
```

BCD Constant

A BCD constant is a signed integer constant preceded by the character '#':

#1 **#104392684**
#-47 **#-4096**

BCDFLT Constant

A BCDFLT constant is a floating constant preceded by the character '#':

#1.0 **#-3.14e-47**
#-1. **#0.**

Parameter Constant

A parameter constant is a name preceded by a dollar sign ($). It can only be used inside an IDL **define** command. The parameter constant is replaced by a value when the stored command is executed. Its type is unspecified until execution time. Even though the value of a parameter constant can change, it is considered a constant because its value remains the same throughout the execution of a command.

Floating Constant

A floating constant is a signed integer constant followed by either a decimal point and digits, or by an 'E' or 'e' and a signed integer constant, or both. It may be preceded by "0f" for FLT4 or "0d" for FLT8. The magnitude and precision of a floating constant is system dependent.

24.4 **-0d3e100** **0f8.0211**

Binary Constant

A binary constant is represented by "0b" followed by a pair of hexadecimal digits:

0bA6 **0b88**

Substitute Constant

A substitute constant is a percent sign (%) followed by either a name or an integer. Substitute constants are used primarily as an intermediary form in the precompilation of embedded query languages, such as RIC, and hardly ever used in interactive IDL. They are used to substitute the value of a programming language variable into an IDL command.

SEE ALSO **execute**
"Expressions", "Qualifications", "Types"

Dbname

A *dbname* is the name of a database. It is listed in the "system" relation "databases" in the "system" database. A *dbname* has the syntax of a *name*.

All the data in a database server is contained in databases. The "system" database is a permanent database which contains data-dictionary relations that store information about all of the databases in the database server.

SEE ALSO

create database, **destroy database**, **extend database**, **open**
"Name"

Expressions

An *expression* yields upon evaluation a value or set of values. For example, the *expression* "43" or "a * b / c" yields a single value, while the *expression* "r.name" yields a set of values, one for each tuple in the relation described by the range variable "r". The set may contain no values at all.

An *expression* may be any of the following:

aggregate	
range_var.att_name	
constant	
function	
(expression)	
– expression	(integer, bcd, bcdflt types only)
expression + expression	(integer, bcd, bcdflt types only)
expression – expression	(integer, bcd, bcdflt types only)
expression * expression	(integer, bcd, bcdflt types only)
expression / expression	(integer, bcd, bcdflt types only)

Floating-point arithmetic is not supported in IDL. Multiplication and division have precedence over addition and subtraction, for example:

A + B * C = A + (B * C)

Every *expression* has an implied value type. The type of a constant *expression* is implied by the type of the constant. The type of an attribute is set when the relation is created. The type of a function or aggregate depends upon the particular function or aggregate.

The type of the result of a numeric *expression* involving more than one operand can be found in the table on the following page.

		Type of One Operand				
		i1	i2	i4	bcd31	bcdflt
Type of Other Operand	i1	i1	i2	i4	bcd31	bcdflt31
	i2	i2	i2	i4	bcd31	bcdflt31
	i4	i4	i4	i4	bcd31	bcdflt31
	bcd	bcd31	bcd31	bcd31	bcd31	bcdflt31
	bcdlft31	bcdflt31	bcdflt31	bcdflt31	bcdflt31	bcdflt31

The result of all bcd arithmetic is the full precision (31 digits). If any number in a calculation is bcdflt, the entire calculation will be performed to 31-digit precision. For example,

1) retrieve (a = #1./7);

returns

a
.1428571428571428571428571428571

SEE ALSO "Aggregates", "Qualifications", "Target-Lists", "Types"

Functions

The database server provides several predefined functions. These functions are divided into four categories:

- Arithmetic (**abs**, **mod**, etc.)
- String (**concat**, **substring**, etc.)
- Type conversion functions (**binary**, **string**, etc.)
- Database server functions (**rel_name**, **getdate**, etc.)

The syntax of an IDL function call is similar to that of traditional computer languages, except that the parentheses are omitted when there are no arguments:

no arguments:	funcname
one argument:	funcname(arg)
two arguments:	funcname(arg1, arg2)
three arguments:	funcname(arg1, arg2, arg3)

Most function arguments can be expressions of any appropriate type, except when the argument refers to a specific number of digits, characters, or bytes, in which case the argument must be an integer.

The predefined functions available in IDL are summarized on the next page, followed by a brief description of each function.

SUMMARY OF FUNCTIONS ON THE DATABASE MACHINE		
Category	*Function*	*Return Value*
arithmetic	**abs**(n) **mod**(n,d)	absolute value remainder of *n* divided by *d*
string or binary	**concat**(a,b) **substr**(pos,len,str) **substring**(pos,len,str)	concatenation of *a* and *b* substring of *str* same as **substr**
conversion	**int1**(n) **int2**(n) **int4**(n) [**fixed**] **binary**(n) **fbinary**(n) [**fixed**] **bcd**(len,n) **fbcd**(len,n) [**fixed**] **bcdflt**(len,n) **fbcdflt**(len,n) [**fixed**] **bcdfloat**(len,n) [**fixed**] **string**(len,n) **fstring**(len,n) **fchar**(len,n) [**fixed**] **char**(len,n) **bcdfixed**(prec,frac,n) **float4**(n) **flt4**(n) **float8**(n) **flt8**(n)	1-byte integer 2-byte integer 4-byte integer binary type same as **fixed binary**(n) [u] bcd type same as **fixed bcd**(len,n) [u] bcdflt type same as **fixed bcdflt**(len,n) same as **bcdflt**(len,n) [u] c type same as **fixed string**(len,n) same as **fixed string**(len,n) same as **string**(len,n) bcdflt type (rounded) 4-byte float same as **float4**(n) 8-byte float same as **float8**(n)
idm	**userid** **dba** **host** **gettime** **getdate** **databasename** **rel_name**(relid) **rel_id**(rel_name) **att_name**(relid,attid)	current user id in this database user id of DBA in this database host id time (i4) date (i4) name of open database relation name relation relid attribute name

abs(n)

The **abs** function returns the absolute value of its argument. The argument must be of type integer, bcd, or bcdflt. The result is of the same length as the argument for integers and 31 digits long for bcd and bcdflt.

att_name (relid,attid)

The **att_name** function returns the attribute name of the specified attribute. Each attribute in a database is uniquely identified by its *attid* and by the *relid* of the relation in which it occurs. These are listed in the system relation "attribute". The **att_name** function always returns a c12 value. If there is no attribute with the specified *attid* and *relid*, **att_name** returns blanks.

bcd(len,expr)

The **bcd** function converts *expr* to a bcd integer and returns a bcd number *len* digits long. The expression may be integer, character, bcd or bcdflt. If the expression is a bcdflt number, it is truncated toward zero (e.g., 6.6 becomes 6 and –6.6 becomes –6). The *len* must be an integer constant. For example,

```
1) retrieve (x = bcd(5, "123"));
```

returns

x
123

and

```
1) retrieve (x = bcd(4, "1234.56"));
```

returns

x
1234

The query

```
1) retrieve (x = bcd(3, "12345"));
```

generates the error message "Numeric overflow".

If *len* is zero, the following lengths (in digits) are used:

Argument Type	*Result Length*
i1	3
i2	4
i4	7
bcd (n)	n
bcdflt (n)	n
char (n)	n/2 + 2
bin (n)	n

bcdfixed(prec,frac,n)

The **bcdfixed** function returns a bcdflt value *prec* digits long and a maximum of *frac* digits to the right of the decimal point. The value of *n* is rounded toward even last digits. The *frac* and *prec* must be integer constants and *prec* cannot be smaller than *frac*; the numbers cannot be so large that there are not *frac* decimal places to round off. If *prec* is zero, a value for *prec* is determined from the *n* passed in. The *n* argument may be integer, character, bcd, or bcdflt.

bcdfixed (5,2,#.123) = .12000
bcdfixed (5,2,#.127) = .13000
bcdfixed (5,2,#.125) = .12000
bcdfixed (5,2,#.135) = .14000
bcdfixed (2,3,anything) = illegal
bcdfixed (3,3,#6./7) = .857
bcdfixed (3,3,#7./7) = overflow
bcdfixed (0,3,".1234") = .12300
bcdfixed (5,2,"768.534") = 768.534
bcdfixed (4,3,"123.45") = overflow
bcdfixed (8,2,"35.478") = 35.48
bcdfixed (7,3,100) = 100.0000
bcdfixed (5,2,#.1251) = .13000

bcdflt(len,expr)

The **bcdflt** function returns a bcdflt value *len* digits long. The expression may be integer, bcd, bcdflt, or character. Numbers are rounded if necessary.

1) retrieve (x = bcdflt(4, "123.45"));

returns

x
123.40

and

1) retrieve (x = bcdflt(5, "1234567.89"));

returns

x
1234600

If *len* is zero, the following lengths are used:

Argument Type	*Result Length*
i1	3
i2	4
i4	7
bcd(n)	n
bcdflt(n)	n
char(n)	n/2 + 2
bin(n)	n

binary(expr)

This function converts *expr* to type binary. The result is a binary value that is the internal representation of the *expr* passed in, whatever its type. This is, in effect, a relabeling of the argument data. All types may be passed in. The resulting length is the same as that of the argument passed in. When retrieved using IDL, binary targets are displayed in hexadecimal format.

concat(stra,strb)

The **concat** function returns the concatenation of the two strings passed in. It takes two character strings, strips all trailing blanks from the first string (all but one, if the string is all blank), strips all trailing blanks from the second string, and appends the second to the first. The **concat** function performs the same functions for binary strings, except trailing zero bytes are stripped instead of trailing blanks.

For instance,

```
1) retrieve (name = concat(emp.first,emp.last))
```

returns an employee's first and last names concatenated in the domain "name". Both strings must be character or both must be binary.

databasename

The **databasename** function returns the name of the currently open database, always a c12 value.

dba

The **dba** function returns an integer equal to the uid of the currently open database, as an i2 value. This is always the owner of the system relation "relation". The **dba** function always returns 1 in the "system" database.

fixed bcd(len,expr)
fixed bcdfit(len,expr)
fixed string(len,expr)
fixed binary(expr)

These are equivalent to **bcd**, **bdcfit**, **string**, and **binary**, except that the results are uncompressed. This difference is usually not significant. For example,

```
1) retrieve (fb=fixed bcdfit(5,".3"),
2)      b=bcdfit(5,".3"));
```

returns

fb	b
.30000	.3

float4(n)
float8(n)

These functions convert a flt4 to a flt8 and vice-versa. They cannot be used on any other type.

getdate

This function returns the number of days from an initial date. This initial date can be set to any value by the IDMDATE utility. When the time (reported by the **gettime** function) reaches the number of 60ths of a second in 24 hours, the time is reset to zero and the date (reported by **getdate**) is incremented by one. The date returned by **getdate** is represented in GMT.

gettime

The **gettime** function returns the time as the number of 60ths of a second since midnight as an i4 value. The following example provides the time in hours and minutes.

```
1) retrieve (hours = gettime/216000,
2)       minutes = mod(gettime,3600));
```

The time value is always fetched once for a command; it does not change over the course of a long retrieve.

host

The **host** function returns the host-id of the host through which the user is presently accessing the database server. It is only useful if there are multiple hosts connected to the database server.

int1(expr)
int2(expr)
int4(expr)

These functions convert their arguments to integers and return a value of i1, i2, or i4 type, respectively. The argument may be integer, character, bcd, or bcdflt.

mod(expr1,expr2)

The **mod** function returns the remainder when the first argument is divided by the second. It can only be used on integer or bcd expressions. For example,

```
1) replace emp (num_children =
2)       mod(emp.num_children,12)
```

takes the number of children an employee has (as specified in the relation "emp"), divides that number by 12, and stores the remainder in "num_children". The call **mod**(*n*,0) is defined to be equal to *n*. The sign of the first argument is the sign of the result. The sign of the second argument is ignored.

rel_id(object_name)

The **rel_id** function returns the relation identifier corresponding to the specified *object_name*. The *object_name* must be a character string.

The *object_name* may be followed by a colon followed by a user name to specify an object owned by another user (i.e. other than the user submitting the command). If the *object_name* is invalid, a zero is returned.

rel_name(relid)

The **rel_name** function returns the relation name corresponding to the relation identifier *relid*. Every relation in a database is uniquely identified by its *relid*. The definitions are listed in the "relation" relation in the system database. A value of type c12 is returned. If no relation with the specified *relid* exists, blanks are returned.

string(len,expr)

The **string** function converts *expr* to type character and returns a character string of length *len*. The *expr* can be of any type except float. If *len* is zero, a length is used based on *expr*:

Type of Expr	*Length of Result*
i1	4
i2	6
i4	11
bcd (n)	2n - 3
bcdflt (n)	2n - 3
char (n)	n
bin (n)	n

substring(pos,len,str)

The **substring** function extracts a string from a character or binary string expression. The result is a character or binary string of length *len*, containing the characters of bytes of *str* starting from position *pos*. The position of the first character of the string is 1. If the substring extends beyond the end of the *str*, the result is padded with blanks (for character) or zero bytes (for binary). For example,

```
1) retrieve (c=substring(3,4,"abcdefghi"),
2) b=substring(3,4,binary(12345678)));
```

returns

c	b
cdef	614E0000

userid

The **userid** function returns as an i2 value the database server user id of the current user. The user ids are recorded in the system relations "host_users" and "users" in the open database.

SEE ALSO "Expressions"

Name

A *name* is a sequence of one to twelve characters. The first character must be alphabetic and the remainder may be alphabetic, numeric and/or underbars. A *name* may or may not be case-sensitive, depending on the host environment. Valid names are:

host_users	**Keywords**
users	**keywords**
tx0174	**RS_232C**

Invalid names are:

sys$list	**821206**
rubber_cement	**6_dec_82**

SEE ALSO "Att_Name", "Dbname", "Object_Name", "Query-Name", "Rel_Name"

Object_Name

An *object_name* is the name of an object in a database. The objects in a database are listed in its system relation "relation". There are seven types of objects:

U - user relation
S - system relation
T - transaction log
C - stored command
P - stored program
V - view
F - file

The syntax of an *object_name* is

name[:owner]

where *name* is the name of the object and *owner* is the name of its owner, as stored in the system relation "users". If *owner* is not specified, the default owner is the current user. If no object belonging to the current user is found, the default is an object owned by the DBA.

An *object_name* may be a quoted string.

SEE ALSO

associate, **create view**, **deny**, **destroy**
"Dbname", "Name", "Query_Name", "Rel_Name"

Options

There are two kinds of options in IDL, those represented by an *option-number* or an *option-name* in the **set** and **unset** commands, and those designated by the **with** *options* syntax in the **create**, **create database**, **create index**, **extend database**, and **range** commands.

The options for **set** and **unset** are listed in the description of the **set** command.

The options preceded by the keyword **with** designate specific optional features with which a command may be invoked. Some options require a value that is a quoted string or an integer. The specific values which an *option* can take depend on the individual command and are described in the command descriptions.

SEE ALSO

create, **create database**, **create index**, **extend database**, **range**, **set**, **unset**

Protect_Modes

A *protect_mode* represents the type of access which can be permitted or denied a user for a particular object. Some *protect_modes* are applicable to relations, views, files, and attributes, others to stored commands and stored programs, and others to databases.

A privilege defined by a *protect_mode* is permitted or denied using its name, such as **read** or **create**, but it is identified in the "access" attribute of the system relation "protect" by a numeric value.

The following table maps the names and numeric values of *protect_modes*. The numbers in the *IDL* column of the table are the results of idl's conversion of database server values to signed 1-byte integers. These are the values displayed in a **retrieve** on the "protect" relation.

Mode	*Octal*	*Hex*	*IDL*	*Applies To*
read	0001	0x01	1	relations, views, files, attributes
write	0002	0x02	2	relations, views, files, attributes
all (read, write)	0003	0x03	3	relations, views, files, attributes
execute	0340	0xe0	-32	stored commands, stored programs
create	0306	0xc6	-58	this database (do not specify object)
create index	0310	0xc8	-56	this database (do not specify object)
create database	0313	0xcb	-53	system database (do not specify object)
read tape	0004	0x04	4	this database (do not specify object)
write tape	0010	0x08	8	this database (do not specify object)
all tape	0014	0x0c	12	this database (do not specify object)
dump	0344	0xe4		this database and transaction log (do not specify object)

SEE ALSO

deny, **permit**
"Users"

Qualifications

A *qualification* is a boolean expression used to specify tuples which meet certain criteria. It is that part of an IDL command that determines which particular tuples of a relation are to be affected by the command.

A *qualification* is may be used with a *target-list* to build a new relation from an existing relation.

A *qualification* is preceded by the keyword **where** and has one of the following forms:

(qualification)
not qualification
qualification **or** qualification
qualification **and** qualification
expression > expression
expression < expression
expression >= expression
expression <= expression
expression = expression
expression != expression
expression = pattern
expression != pattern
pattern = expression
pattern != expression

Relational operators (>, <, >=, <=, =, !=, <>) are supported by the database server for all data types. If the terms being compared contain characters, the comparison is governed by ASCII or EBCDIC order, depending on which character set was specified when the database was created. Blanks at the end of character strings are ignored for comparison purposes.

The command

```
1) range of e is employees
2) delete e where e.salary > 24000;
```

deletes from the employees relation all tuples representing employees with salaries over 24000. The clause "e.salary > 24000" is the *qualification*.

If the condition expressed by the *qualification* is never true (e.g., "1=2"), no tuples will be affected by the command. If the condition expressed by the *qualification* is always true, all of the tuples in the relation will be affected. This is the default, when no *qualification* is specified.

PATTERNS

A pattern-matching string may be used in a ***qualification*** to match a wide variety of character strings.

Pattern-matching strings can be formed using special characters which match characters other than themselves. Trailing blanks in uncompressed character attributes are not considered characters which can be matched.

These special characters are:

*	matches zero or more characters
?	matches any one character
[	begins a group of characters any one of which may be matched
]	ends the group of characters
\	escapes any of the above
- (dash)	specifies, within brackets "[ ]", a range of characters to match

Pattern-matching strings may appear only in the ***qualification*** of a command.

For example, to find the salary of all employees whose name starts with "J", use the command

```
retrieve (e.salary) where e.name="J*"
```

Any character string that contains an asterisk (*), question mark (?), or either bracket ([ ]) is considered a ***pattern***, unless the special character is escaped with a backslash (\). If a user wants to specify a literal asterisk, for example, the asterisk must be preceded with a backslash so that it will not be interpreted as a special character.

The table on the following page suggests, through the use of examples, the kinds of results produced by pattern-matching strings.

This pattern	*will match these strings*	*but not these strings*
"a*e"	"ae" "ace" "a3e" "abcde" "a2X.(#e"	"Ae" "aE" "bae"
"a?e"	"ace" "aQe" "a#e"	"ae" "abce"
"a[bcd]e"	"abe" "ace" "ade"	"aee" "aae" "abde"
"a[b-m]e"	"abe" "ace" "ade" "ame"	"aae" "ane" "aBe" "a-e"
"a*e"	"a*e"	"a\be" "abe" "ae"

The last example not a true *pattern* because the character '*' is to be interpreted as a literal asterisk, not as a special character specifying a match of zero or more characters; this is indicated by the backslash. The string is really a three-character constant consisting of the characters 'a', '*', and 'e'.

JOINS

A *join* is a mechanism for relating data from multiple objects in a single query. The *qualification* in the following query represents a *join* of the "x" and "y" relations.

```
1) range of x is x;
1) range of y is y;
1) retrieve (x.name, x.num, y.quan)
2) where x.num = y.num;
```

In this query, data from the "x" and "y" relations is retrieved only from those tuples in which the "num" attribute in "x" equals the "num" attribute in "y". If the "x" relation consists of:

x relation	
num	name
1	zinfandel
2	riesling
3	cabernet
4	chardonnay

and the "y" relation consists of

y relation	
num	quan
1	50
2	70
5	35
6	60

the query retrieves only

name	num	quan
zinfandel	1	50
riesling	2	70

A one-way *outer join* requests all the specified data from one relation, regardless of the whether the condition joining the other relation is true. Non-matching data from the other relation is assigned a default value of zero (0) for numeric data and blanks for character data.

A one-way outer join is indicated by an asterisk (*) attached to any of the allowable relational operators for a *qualification*. The asterisk is placed on the the same side of the relational operator as the relation from which all specified data is to be retrieved. Thus the query

```
1) retrieve (x.name, x.num, y.quan)
2) where x.num *= y.num;
```

retrieves all of the specified data from "x" and only the matching data from "y":

name	num	quan
zinfandel	1	50
riesling	2	70
cabernet	3	0
chardonnay	4	0

while

```
1) retrieve (x.name, y.num, y.quan)
2) where x.num =* y.num;
```

retrieves all of the specified data from "y" and only the matching data from "x":

name	num	quan
zinfandel	1	50
riesling	2	70
	5	35
	6	60

The database server does not support two-way outer joins.

SEE ALSO

append, **audit**, **create view**, **delete**, **destroy**, **replace**, **retrieve**
"Constants", "Target-Lists"

Query_Name

A *query_name* is the name of a stored command which is referenced by the **define** and **execute** commands.

The syntax of a *query_name* is

name[:owner]

SEE ALSO **define, execute**

Range_Var

A range variable or *range_var* represents one or more tuples in a relation. It has the syntax of a *name*. A *range_var* is considered a variable because the tuple it represents changes when the command is executed. Several IDL commands require that relations be accessed through a *range_var* rather than through the relation name. A *range_var* is declared in a **range** statement.

A specific attribute can be referenced by appending a period and the attribute name to the *range_var*.

The psuedo-attribute **all** can be appended to reference all attributes.

If a *range_var* "e" is declared as

```
1) range of e is employees;
```

then

```
1) retrieve (e.all);
```

accesses all of the tuples in the "employees" relation and

```
1) retrieve (e.num) where e.lastname = "Jones";
```

accesses all of the employee numbers in the "employees" relation for those employees whose last name is "Jones".

SEE ALSO

associate, **delete**, **range**, **replace**, **retrieve**
"Name"

Rel_Name

A *rel_name* is an *object_name* which refers to a relation, view, or transaction log.

The syntax of *rel_name* is the same as that of *object_name*.

A relation can be conceptualized as a table with rows and columns. The rows are called tuples (or records) and the columns are called attributes (domains or fields). Every attribute has a name and a declared data type (e.g., integer, character, etc.) and all values in the attribute must be of this type.

The order in which tuples are stored in the relation is arbitrary.

SEE ALSO

append, audit, create, create index, deny, destroy, extend, permit, range, retrieve into, truncate
"Att_Name", "Object_Name", "Qualification", "Target-Lists, "Types"

Target-Lists

A *target-list* is a list of *targets* separated by commas and enclosed in parentheses. The *targets* can have the following forms:

domain_name = expression
The name and value of the domain is explicitly stated as in

```
1) retrieve (total = sum(w.onhand));
```

range_var.att_name
Multiple values are accessed for each instance of the attribute referenced by *att_name* in the relation referenced by *range_var*:

```
1) range of e is employees
2) retrieve (e.name, e.phone);
```

*range_var.*all
The pseudo-attribute all yields all of the attributes of the referenced relation in the order in which they were created.

```
1) range of e is employees
2) retrieve (e.all) where t.name = "Smith";
```

When multiple targets are specified in a *target-list*, the *target-list* values are bound to associated program variables as illustrated below. If the relation indicated by "y" has three domains "y.q", "y.r", and "y.s", and the command is

```
retrieve (x.a, y.all, x.b)
```

the following bindings apply:

Target-List Position	*Value of Target*
1	x.a
2	y.q
3	y.r
4	y.s
5	x.b

SEE ALSO

append, **audit**, **create view**, **destroy**, **retrieve**
"Att_Name", "Expressions", "Qualifications", "Range_Var", "Rel_Name", "Types"

Types

Every attribute in a relation has a *type* which is set when the relation is created. The *type* of an attribute determines the values that the attribute can assume.

There are six value *types* available on the database server and each of these *types* can have a variety of lengths. The *types* are:

Name of Type	*Mnemonic*	*"Type" in "Attribute" Relation*
4-byte integer	i4	56
2-byte integer	i2	52
1-byte integer	i1	48
8-byte float	f8	60
4-byte float	f4	57
character	c	47
binary	bin	45
integer bcd	bcd	46
floating bcd	bcdflt	35

INTEGER

Integer attributes are stored as binary two's complement integers in one byte (-128 to +127), two bytes (-32,768 to +32,767), or four bytes (-2,147,483,648 to +2,147,483,647). Full four-function arithmetic and modulus and absolute value functions are supported for integer types.

FLOAT

Floating-point attributes are stored either as four bytes (f4) or eight bytes (f8). No arithmetic functions are available for them. The only comparison operations available are = and !=. Floating-point numbers may be stored and retrieved on the same machine; however, floating-point numbers written on one machine and read on another will probably not give predictable results.

CHARACTER

Character attributes are either compressed or uncompressed. Character compression is performed by deleting trailing blanks. For example, "c10" signifies a compressed character attribute that is a maximum of 10 characters long, and "uc19" signifies an uncompressed character attribute which is to be always stored as 19 characters (even if they are all blanks). The maximum length for a character attribute is 255 characters.

BINARY

A binary attribute is a binary string that is stored in the form in which it was received from the host system. Uncompressed binary strings are zero-filled to the length specified when the data is received from the host. Compressed binary strings have trailing zero bytes deleted. For example, "ubin5" means an uncompressed binary string of 5 bytes and "bin200" means a binary string with a maximum length of 200 bytes. The maximum length of a binary attribute is 255 bytes.

INTEGER AND FLOATING BCD

Integer and floating-point bcd attributes also are either compressed (variable-length) or uncompressed (fixed-length). The length is specified in number of digits. If an even number of digits is specified, the number is incremented by 1 so that the length is always odd. Compressed bcd attributes consume less storage than uncompressed because trailing zeros are dropped. Trailing zeros are left alone in uncompressed bcd attributes. The maximum length of a bcd or bcdflt is 31 digits. Bcd and bcdflt types can participate in arithmetic as if they were integers. The results of bcd arithmetic are always to a precision of 31 digits.

When a relation is created, each attribute *type* is declared by its mnemonic followed by its length, as in

```
1) create myrel (name=c10, count=i4, fraction=bcdflt8);
```

Every attribute of every relation in a database is listed in the system relation "attribute". The "name" attribute in this relation contains the attribute's name, the "type" attribute contains a numeric code representing its *type*, and the "length" attribute contains its length as an unsigned number. If tuples are retrieved from the "attribute" relation and the length appears to be a negative number, add 256 to get the correct length. For bcd and bcdflt, the recorded length represents the number of bytes (2 through 17) not the number of digits (1 through 31).

SEE ALSO

create
"Constants", "Expressions"

Users

A *user* is an individual or group of individuals with access to the database server. A *user* communicates with the database server through the intermediary of a host computer.

All *users* are identified through two identification numbers, a host-id and a host-user-id, which are provided by the host system. In the database server, the system relation "host_users" maps the host-id and the host-user-id into a single user-id. The system relation "users" maps the user-id to a user name and group.

The DBA assigns general access to new *users* by entering their identification data in the "host-users" and "users" relations. After a new *user* has been identified in these two relations, the DBA can assign specific access rights by user name or group name through use of the **permit** and **deny** commands.

EXAMPLE

We will add a new *user*, "karen", and assign her to group number 20. Assume that the host-id of the system "karen" works on is 3, and her host-user-id on that system is 301.

```
1) open system;
1) range of u is users;
1) append to users (
2)   name = "karen",
3)   gid = 20,
4)   id = max(u.id) + 1);
1) retrieve (u.stat, u.id, u.gid, u.name)
2)   where u.name = "karen";
```

stat	id	gid	name
0	321	20	karen

```
1) append to host_users (
2)   hid = 3,
3)   huid = 301,
4)   uid = u.id);
1) range of h is host_users;
1) retrieve (h.all)
2)   where u.name = "karen"
3)   and u.id = h.uid;
```

s1	hid	huid	uid
0	3	301	321

SEE ALSO

deny, permit

PART IV

IDL FRONT-END COMMANDS

Introduction to Front-End Commands

The IDL query language provides a set of front-end commands which can be invoked to govern certain aspects of an IDL session. These commands take effect immediately after they are invoked; unlike regular IDL commands, they are not buffered to a go or a semicolon.

All of the front-end commands must be invoked at the beginning of a line. All of the front-end commands begin with a percent symbol (%). All of the front-end commands may be abbreviated to any length, provided that the abbreviation results in an unambiguous command name.

This section describes the basic front-end commands which are available on all systems supported by Britton Lee host software. Some systems have an extended set of front-end commands. Consult the host software documentation for your particular environment for information concerning additional front-end commands which may be available on your system.

The front-end command %? lists all of the front-end commands described in this section.

%associate [on | off]

DESCRIPTION

%associate is used to turn the auto-associate feature on and off. The auto-associate feature automatically executes the **associate** command whenever a **create**, **create view** or **define** command is executed. It provides this automatic documentation of database objects unless the **idl** program was invoked with the **-a** or **/noassociate** flag for the session in which the object was created.

The **%associate** command may be used to suspend the automatic execution of the **associate** command for the remainder of the **idl** session or until the user wishes to turn it on again. This may be desirable if the command creating the object exceeds 4000 bytes, which would make it too large to fit into the command buffer.

If neither **on** nor **off** is specified, **%associate** turns auto-associate on.

EXAMPLE

```
1) %associate off          /* turn auto-associate off */
2) create myrel (          /* create a relation */
3)     . . .
4)     . . .
       . . .
97) );
1) %associate on           /* turn auto-associate on again */
```

SEE ALSO

associate

%continuation [character]

DESCRIPTION

This sets the continuation character to the value indicated by *character*. Lines ending with this continuation character are not sent directly to the parser.

If continuation mode has been set using **%continuation**, the **go** or semicolon is not recognized as an IDL command terminator. Instead, the first line of input which does not terminate with the continuation character terminates the command.

The value of *character* may not be a letter or digit. Valid continuation characters are:

! @ % ^ * ( ) + - = ~ ' | { } / ? < > , .

Any continuation character may be unset by invoking **%continuation** with no argument. If this is done, all lines are saved and the user must enter a semicolon (;) or the keyword **go** to indicate that the lines are to be submitted to the parser.

EXAMPLE

```
1) %continuation +          /* set continuation character */
2) range of p is parts +
3) append to parts(name = "tube", quan = 20) +
4) retrieve(p.name, p.quan) +
5) where p.name = "tube"  /* command ends here */
```

name	quan
tube	20

```
1) %continuation                    /* unset continuation character */
2) delete p where p.quan < 1;       /* go and semicolon reinstated */
```

%display text

DESCRIPTION **%display** sends *text* to standard output.

EXAMPLE
```
1) %display "Good Morning"
Good Morning
2)
```

%edit [filename]

DESCRIPTION

%edit with no argument edits the transcript of the IDL session. This is a useful tool for making a change in a series of IDL commands which have not yet been executed without having to re-enter the whole series of commands from the beginning. With a *filename*, **%edit** edits the specified file. Upon return to idl, **%edit** submits the file it has just edited as input.

The editor which is called is specified by the EDITOR parameter in the "params" file on the host system.

EXAMPLES

With a *filename*:

```
1) %edit cmd.file
```

Now "cmd.file" can be edited. The contents of "cmd.file" will be executed when the user leaves the editor.

Without a *filename*:

```
1) range of p is parts
2) append to parts (name =
3) %edit
```

This places the user in the editor editing a temporary file which looks like this:

```
range of p is parts
append to parts (name =
```

The contents of this file will be executed when the user leaves the editor.

SEE ALSO

params(5I) in *Host Software Specification*
params in *C Run-Time Library Reference*

%experience level

DESCRIPTION

%experience sets the user's experience level to *level*. The value of *level* controls the amount of detail which will be given in IDL error messages; the more elementary the *level*, the more detailed the message.

Values for *level* can be "beginner", "able", or "expert". Any other value is interpreted as "beginner". Values for *level* may be abbreviated and are not case-sensitive.

EXAMPLE

```
%experience beginner
```

%help

DESCRIPTION

%help lists all of the available front-end commands. **%?** is a synonym for **%help**.

EXAMPLE

```
1) %help
   HELP: Immediate Commands:
        associate -- auto-associate on (1) or off (0)
        continuation -- set continuation char
        display -- display user arguments
        edit -- edit session log or file
        experience -- change experience level
        ? -- print this list
        help -- print this list
        input -- input command file
        redo -- re-execute session log
        substitute -- set value x for %x usage
        showranges -- show current range variables
        trace -- set internal trace flag
```

%input [filename]

DESCRIPTION

%input specifies a command file from which **idl** can read its input. If the *filename* is not specified, the commands are read from standard input.

If a *filename* is specified, commands are read and executed until an **exit** or end-of-file is read, at which point **idl** reads from standard input.

The input file may contain comments enclosed by the characters /* and */. The IDL parser ignores all of the text between the /* */ pairs. The following is valid input to **idl**:

```
/* this is a comment */
range of p is parts
retrieve (p.name,  /* here is another comment */ p.quan);
```

EXAMPLE

```
%input "cmd.file"
```

%redo

DESCRIPTION

%redo resubmits the current **idl** session as input to **idl**.

EXAMPLE

```
1) range of p is parts;
1) retrieve (p.partnum, p.onhand);
```

partnum	onhand
1	25
2	30
3	48

3 tuples affected.

```
1) %redo
```

partnum	onhand
1	25
2	30
3	48

3 tuples affected.

%showranges

DESCRIPTION

%showranges displays the currently defined range variables.

EXAMPLE

```
1) range of p is parts;
1) range of pr is products;
1) %showranges

range of p is parts
range of pr is products
(2 range variables declared)

2)
```

%substitute name value

DESCRIPTION

%substitute assigns a specific *value* to *name*. Substitutions put place holders into an ITREE using the *%name* syntax in **idlparse**. Values may later be substituted into the tree without reparsing. The *value* may be quoted.

Since this command sets up a substitution, rather than a macro, there are restrictions on where the substitution can occur. Generally, substitutions can be used

- Wherever an *expression* can occur.
- As an *att_name* or *object_name* on the left side of an equals sign provided that the substitution is a character type.
- As the **with** part of an **associate** command.

%substitute can set character arguments to be used in pattern-matching strings, as long as the pattern-matching string is not used in a *target-list*.

To disable interpretation of a string containing a special character as a pattern-matching string, either precede the special character with a backslash as in

```
1) %substitute a "a\*b"
```

or follow the *value* argument with the word **char**, as in

```
1) %substitute a "a*b" char
```

EXAMPLES

```
1) %substitute a1 "hubcap"
2) %substitute a2 "20"
3) %substitute rel "parts"
4) append to %rel (name = %a1, quan = %a2);
```

SEE ALSO

idlparse(3I), **iesubst**(3I) in *Host Software Specification*
idlparse, **iesubst** in *C Run-Time Library Reference*

%trace tracespec

DESCRIPTION **%trace** invokes *tfset*(), with *tracespec* as its argument.

EXAMPLE **1) %trace IOTRAFFIC.10**

SEE ALSO **tf** (3I), **mapsym**(3I), **symfile**(5I) in *Host Software Specification*
tf, **mapsym**, **symfile** in *C Run-Time Library Reference*

PART V

APPENDICES

IDL Reserved Words

The following words are IDL reserved words, and may not be used otherwise in IDL commands unless they are quoted.

abort	**and**	**append**
associate	**audit**	**begin**
by	**clustered**	**create**
define	**database**	**delete**
deny	**destroy**	**end**
execute	**extend**	**fixed**
index	**into**	**is**
nonclustered	**of**	**on**
open	**or**	**order**
permit	**range**	**reconfigure**
replace	**retrieve**	**set**
sync	**tape**	**to**
trace	**transaction**	**truncate**
unique unset	**view**	
where with		

IDL Grammar

The following pages contain a formal description of the Intelligent Database Language (IDL) supported by Britton Lee Host Software.

```
IDL_program:
        statement_list
    |   execute_statement
    |   execute_statement statement_list
    |   /* null statement */
    ;

statement_list:
        statement
    |   statement_list statement
    ;

statement:
        stmt
    ;

stmt:
        abort transaction
    ;

stmt:
        append opt_TO subs_object_name (targets) opt_qualification
    ;

stmt:
        associate subs_object_name
    |   associate attribute
    |   associate subs_object_name opt_WITH string_constant
    |   associate attribute opt_WITH string_constant
    |   associate subs_object_name opt_WITH string_constant
            I_COMMA string_constant
    |   associate attribute opt_WITH string_constant I_COMMA
            string_constant
    ;

stmt:
        audit (targets) opt_qualification
    |   audit opt_INTO subs_object_name (targets) opt_qualification
    ;

stmt:
        begin transaction
    ;
```

```
stmt:
        create subs_object_name
        (
                opt_name (format_list) opt_with_clause
                [ , opt_name (format_list) opt_with_clause ]
        ) opt_with_clause
    ;

stmt:
        create database subs_object_name opt_with_clause
    ;

stmt:
        create opt_UNIQUE opt_CLUSTERED_or_NONCLUSTERED
                index opt_ON subs_object_name
                (attribute_NAMEs) opt_with_clause
    ;

stmt:
        create view subs_object_name (targets)
                opt_qualification
    ;

stmt:
        define object_name statement_list end define
    ;

stmt:
        delete name opt_qualification
    ;

stmt:
        deny protect_mode protect_object protect_attrs opt_user_list
    ;

stmt:
        destroy object_names
    |   destroy (targets) opt_qualification
    ;

stmt:
        destroy database object_names
    ;

stmt:
        destroy opt_UNIQUE opt_CLUSTERED_or_NONCLUSTERED
                index opt_ON subs_object_name (attribute_NAMEs)
    ;

stmt:
        end transaction
    ;
```

```
stmt:
        execute execute_statement
    ;

stmt:
        extend subs_object_name opt_with_clause
    ;

stmt:
        extend database subs_object_name opt_with_clause
    ;

stmt:
        open subs_object_name
    ;

stmt:
        permit protect_mode protect_object protect_attrs opt_user_list
    ;

stmt:
        range of name is rangeis opt_with_clause
    ;

stmt:
        reconfigure
    ;

stmt:
        replace name (targets) opt_qualification
    ;

stmt:
        retrieve opt_UNIQUE (targets) opt_order_list opt_qualification
    |   retrieve opt_UNIQUE opt_INTO subs_object_name (targets )
                opt_order_list opt_qualification
    ;

stmt:
        set I_CONSTANT_list
    ;

stmt:
        sync
    ;

stmt:
        trace tracetype constant
    ;
```

```
stmt:
        truncate object_names
    ;

stmt:
        unset I_CONSTANT_list
    ;

attribute:
        name I_PERIOD name
    ;

attribute_NAME:
        name
    ;

attribute_NAMEs:
        attribute_NAME_list
    ;

attribute_NAME_list:
        attribute_NAME
    |   attribute_NAME_list I_COMMA attribute_NAME
    ;

boolean_expression:
        ( boolean_expression )
    |   not boolean_expression
    |   boolean_expression and boolean_expression
    |   boolean_expression or boolean_expression
    |   expression relop expression
    ;

by_list:
        expression
    |   by_list I_COMMA expression
    ;

comma_with_option:
        I_COMMA with_option
    ;

constant:
        I_LEXCONSTANT
    |   substitution
    ;

const_term:
        constant
    |   I_LEXNAME
    ;
```

```
execute_statement:
        object_name param_list
    ;

expression:
        constant
    |   name
    |   parameter
    |   attribute
    |   - expression              %prec unary
    |   + expression              %prec unary
    |   (expression)
    |   expression + expression
    |   expression - expression
    |   expression * expression
    |   expression / expression
    |   name opt_UNIQUE (expression_list opt_by_clause opt_qualification )
    |   I_FIXED name (expression_list opt_qualification)
    ;

expr_list:
        expression
    |   expr_list I_COMMA expression
    ;

expression_list:
        expr_list
    ;

format_list:
        simple_format_list
    |   partitioned_format_list
    ;

format_spec:
        name I_EQ name
    ;

I_CONSTANT_list:
        const_term
    |   I_CONSTANT_list I_COMMA const_term
    ;

name:
        I_LEXNAME
    ;

named_param:
        name I_EQ expression
    ;
```

```
object_name:
        qname
    |   qname I_COLON qname
    ;

object_names:
        object_name_list
    ;

object_name_list:
        object_name_resdom
    |   object_name_list I_COMMA object_name_resdom
    ;

object_name_resdom:
        subs_object_name
    ;

opt_BY:
        /* empty */
    |   by
    ;

opt_by_clause:
        /* empty */
    |   I_BY by_list
    ;

opt_CLUSTERED_or_NONCLUSTERED:
        /* empty */
    |   clustered
    |   nonclustered
    ;

opt_direction:
        /* empty */
    |   I_COLON name
    ;

opt_INTO:
        /* empty */
    |   into
    ;

opt_name:
        /* empty */
    |   name
    ;
```

```
opt_ON:
        /* empty */
    |   on
    ;

opt_order_list:
        /* empty */
    |   order_list
    ;

opt_qualification:
    |   where boolean_expression
    ;

opt_TO:
        /* empty */
    |   to
    ;

opt_UNIQUE:
        /* empty */
    |   unique
    ;

opt_user_list:
        /* empty */
    |   I_TO user_list
    ;

opt_WITH:
        /* empty */
    |   with
    ;

opt_with_clause:
        /* empty */
    |   with with_list
    ;

order_list:
        order opt_BY expression opt_direction
    |   order_list I_COMMA expression opt_direction
    ;

parameter:
        I_LEXPARAM
    ;
```

```
param_list:
        opt_WITH (value_list)
    |   opt_WITH value_list
    |   opt_WITH (named_param)
    |   opt_WITH value_spec
    |   /* empty */
    ;

partition:
        (simple_format_list) opt_with_clause
    ;

partitioned_format_list:
        partition
    |   partitioned_format_list I_COMMA partition
    ;

protect_attrs:
        (object_name_list)
    |   /* empty */
    ;

protect_mode:
        read
    |   write
    |   all
    |   read tape
    |   write tape
    |   all tape
    |   create
    |   create database
    |   create index
    |   execute
    ;

protect_object:
        I_ON subs_object_name
    |   I_OF subs_object_name
    |   subs_object_name
    |   /* empty */
    ;

qname:
        I_LEXNAME
    |   I_LEXCONSTANT
    ;

rangeis:
        subs_object_name
    ;
```

```
relop:
        I_LOUT          %prec highest
    |   I_ROUT          %prec highest
    |   I_LGTOUT        %prec highest
    |   I_RGTOUT        %prec highest
    |   I_LGEOUT        %prec highest
    |   I_RGEOUT        %prec highest
    |   I_LLTOUT        %prec highest
    |   I_RLTOUT        %prec highest
    |   I_LLEOUT        %prec highest
    |   I_RLEOUT        %prec highest
    |   I_LNEOUT        %prec highest
    |   I_RNEOUT        %prec highest
    |   I_EQ
    |   I_GE
    |   I_GT
    |   I_LE
    |   I_LT
    |   I_NE
    ;

simple_format_list:
        format_spec
    |   format_list I_COMMA format_spec
    ;

string_constant:
        constant
    ;

substitution:
        % name
    |   % I_LEXCONSTANT
    ;

subs_object_name:
        object_name
    |   substitution
    ;

target:
        name I_EQ expression
    |   substitution I_EQ expression
    |   attribute
    ;

targets:
        target_list
    ;
```

```
target_list:
        target
    |   target_list I_COMMA target
    ;

tracetype:
        /* empty */
    |   I_ON
    |   I_DELETE
    ;

user_list:
        user_name
    |   user_list I_COMMA user_name
    ;

user_name:
        qname
    ;

value_list:
        value_spec I_COMMA value_spec
    |   value_list I_COMMA value_spec
    ;

value_spec:
        named_param
    |   expression
    ;

with_list:
        with_option
    |   with_list comma_with_option
    |   with_list I_ON string_constant
    ;

with_option:
        name
    |   name with expression
    ;
```

Index of Terms