Britton Lee Host Software

SQL REFERENCE MANUAL

(R3v5)

March 1988
Part Number 205-1344-003

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.

The information contained within this document is subject to change without notice. Britton Lee assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under license and may only be used or copied by the terms of such license.

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 SQL, consult the *Host Software Message Summary (SQL Version)*.

PART I

INTRODUCTION TO SQL

Introduction to SQL

This part provides an introduction to SQL intended for data processing professionals interested in learning to use SQL 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. A table is organized horizontally into rows and vertically into columns. The rows represent individual entities in the table while the columns describe characteristics associated with those entities.

In relational theory, the formal name for "table" is "relation;" the formal name for "row" is "tuple" and the formal name for "column" is "attribute".

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

Data manipulation refers to the part of a query language which extracts data from an existing table and modifies existing tables by inserting 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 tables, 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 SQL commands, nor does it completely describe the commands which it does cover. For a complete description of every SQL command, consult Part II of this manual.

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

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

AUTHOR TABLE		
authnum	first	last
1	alice	adams
2	herman	melville
3	brian	kernighan
4	dennis	richie
5	dh	lawrence
6	william	shakespeare
7	doug	adams
8	el	doctorow

TITLE TABLE			
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
7	ragtime	4	5

PUBLISHER TABLE			
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
5	random house	new york	212-288-1200

AUTHTTL TABLE	
authnum	docnum
1	4
2	1
3	2
4	2
5	5
6	3
7	6
8	7

PRICE TABLE			
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
7	87	3.95	bookpeople

Executing the SQL Program

ENTERING SQL

To invoke SQL enter

sql

In general, SQL commands are case-sensitive. Command names and keywords should be entered as they are shown in the synopses. User-defined identifiers such as database names and table names should be treated consistently. The identifiers

books
Books
BOOKS

are different.

Quoted strings are also case-sensitive, so that the following strings are all different:

"books"
"Books"
"BOOKS"

All SQL commands are terminated by a semicolon (;). Users who wish to emulate SQL-DS style input can do so by setting a continuation character using the front-end command **%continuation**, described in Part IV of this manual.

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

1)

This is the SQL prompt.

In order to execute any SQL commands, you must first open a database. The following command opens the "books" database.

1) open books;

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

sql books

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

EXITING SQL

If the SQL prompt is displayed, the user can exit SQL 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 SQL prompt.

Data Manipulation

Data manipulation refers to the ability to examine the data in one or more tables and to modify existing tables by inserting new data, deleting data, or changing the value of one or more columns in specified rows.

The **select** command is used to examine or query the database, the **insert**, **delete**, and **update** commands to modify the database.

SELECT

The **select** command extracts specified data from one or more tables. Used interactively, it displays its results in a table consisting of the requested rows and columns at the user's terminal.

The essential parts of any **select** statement are the specifications of

- the table(s) to be accessed
- the column(s) to be displayed (the *targets* or *target-list*).

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 **select** statement is

select *specified target(s)*
from *specified table(s)*

A specified target may have various forms. It may be

- a column name,
- a result column title = column name,
- the value returned by an aggregate or function,
- a result column title = value returned by an aggregate or function,
- an asterisk,
- any arbitrary arithmetic expression.

The specified table may be a table or a view.[1]

The following query illustrates the simplest form of the **select** statement. It queries the database for the values of the columns named "first" and

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

"last" in all the rows in the "author" table.

```
1) select first, last
2) from author;
```

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

The asterisk (*) is used to specify all of the columns in a table. The entire "author" table consists of three columns. The following command selects all of the data in the table.

```
1) select *
2) from author;
```

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

It is also possible to specify result column titles, which differ from the original column names, in the result table displaying the selected data. The following command selects data from the "last" column in the author table, but labels the selected column "surname" in the result.

```
1) select surname = last
2) from author;
```

surname
adams
melville
kernighan
ritchie
lawrence
shakespeare
adams
doctorow

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

- the conditions to apply in selecting the data (the *qualification*)
- the order in which the rows should be displayed
- whether duplicate rows should be ignored.

Where Clause

In order to specify that only some of the rows in a table should be selected, the query must indicate the conditions, or predicates, governing the selection of rows. This set of conditions, called the *qualification*, may consist 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)

A qualification is specified by the keyword **where** followed by the conditions limiting the selection.

The following query requests rows from the "author" table in which the value of the "authnum" column is 2.

```
1) select *
2) from author
3) where authnum = 2;
```

authnum	first	last
2	herman	melville

The next query requests data from rows in which the value of the "last" column is 'adams'. The constant value 'adams' must be enclosed in quotation marks because it is being compared to a column of the type character string.[2]

```
1) select last
2) from author
3) where last = 'adams';
```

last
adams
adams

Distinct

The **distinct** modifier is used to remove duplicate rows from the data returned by a query.

```
1) select distinct last
2) from author
3) where last = 'adams';
```

last
adams

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

```
1) select distinct first, last
2) from author
3) where last = 'adams';
```

first	last
alice	adams
doug	adams

selects two rows from the "author" table, not one, because there is no duplication in the table of the combined values for "first" and "last".

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

Multiple Conditions

If the *qualification* governing the **select** 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 rows in which the value of the "last" column is 'adams' and the value of the "first" column is not 'alice'. In order to be selected, a row must satisfy both of these conditions.

```
1) select *
2) from author
3) where last = 'adams' and first <> 'alice';
```

authnum	first	last
7	doug	adams

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

```
1) select *
2) from author
3) where last = 'adams' and not first = 'alice';
```

authnum	first	last
7	doug	adams

The next query uses the **or** operator to select the rows in which the value of the "last" column is 'adams' or the value of the "first" column is not 'alice'. In this case, a row must satisfy only one of the conditions, not both, in order to be selected.

```
1) select *
2) from author
3) where last = 'adams' or 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
8	el	doctorow

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

```
1) select *
2) from author
3) where first = 'herman' or 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) select *
2) from author
3) where (first = 'herman' or first = 'herbert')
4) and (last = 'melville' or  last = 'de melville');
```

authnum	first	last
2	herman	melville

Between Predicate

The **between** predicate is used to specify a field which falls within a certain range of values. The **not between** predicate is also valid. The following query uses the **between** predicate to select the rows in which the value of the "authnum" column is between 3 and 6 inclusive.

```
1) select *
2) from author
3) where authnum between 3 and 6;
```

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

This query is equivalent to

```
1) select *
2) from author
3) where authnum >= 3 and authnum <= 6;
```

The **between** predicate can also be applied to character data, in which case 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. The following query

requests rows for all authors whose last names fall alphabetically between 'flaubert' and 'tolstoy'. It is not necessary for the values indicating the range to be existing values in the table.

```
1) select *
2) from author
3) where last between 'flaubert' and 'tolstoy';
```

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

In Predicate

The **in** predicate is used to specify a field in a list of values. The predicate is satisfied if the value being compared is equal (or not equal if **not in** is specified) to any value in the list. The following query selects the rows in which the value of the "authnum" column is 4, 5, or 88.

```
1) select *
2) from author
3) where authnum in (4, 5, 88);
```

authnum	first	last
4	dennis	ritchie
5	dh	lawrence

The **in** operator can also be used with character data, as illustrated below.

```
1) select *
2) from author
3) where first in  ('brian', 'dennis');
```

authnum	first	last
3	brian	kernighan
4	dennis	ritchie

Like Predicate

The **like** predicate is used to indicate a string value in which all of the characters are not specified. The **not like** predicate is also valid. The percent symbol (%) is used in the character string to represent a substring of zero or more characters. The underscore character (_) is used to represent a single character.

The following query selects the "first" and "last" columns for all rows in which the value of the first character in the "first" column is 'd'.

```
1) select first, last
2) from author
3) where first like 'd%';
```

first	last
dennis	ritchie
dh	lawrence
doug	adams

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

```
1) select * from title
2) where title like '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 column 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" column of the "title" table.

```
1) select
2) count = count(onhand),
3) average = avg(onhand),
4) largest = max(onhand),
5) smallest = min(onhand),
6) total = sum(onhand)
7) from title;
```

count	average	largest	smallest	total
7	6	12	3	46

Order By

Normally the rows displayed by a **select** statement appear in an order determined by the database server software. The user can specify the order in which rows 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** or **desc**. Both numeric and string type expressions can be used to order selected data. The following query specifies that the rows be displayed in ascending order based on the value

of the "last" column.

```
1) select first, last
2) from author
3) order by last;
```

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

The next query specifies that the selected rows be displayed in descending order based on the value of the "authnum" column.

```
1) select authnum, first, last
2) from author
3) where last not like 'a%'
4) order by authnum d;
```

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

The **order by** clause is used below in selecting data from the "title" table to display the data ordered by the value of the "pubnum" column, and within that ordering by the value of the "onhand" column.

```
1) select pubnum, onhand, docnum
2) from title
3) order by pubnum, onhand;
```

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

Joins

A join is a mechanism for relating data from multiple tables within a single query. All the tables being joined in the query are listed in the **from** clause. When tables are joined, the **where** clause specifies a relationship, known as a "joining condition", between the rows from which data is to be selected.

If the joining tables have column names which are not unique among all the tables referenced by the **from** clause, the query must qualify the non-unique column names by prefacing them with their table names or with table labels.

The following query selects data from the "title" and "onhand" columns in the "title" table and from the "name" column in the "publisher" table. The joining condition is

"where title.pubnum = publisher.pubnum"

```
1) select title, name, onhand
2) from title, publisher
3) where onhand < 7
4) and title.pubnum = publisher.pubnum;
```

title	name	onhand
fantasia of the unconscious	penguin	6
moby dick	signet	6
superior women	signet	3
ragtime	random house	4

The next query selects data from the "first" and "last" columns of the "author" table and from the "title" column of the "title" table. The joining condition

"where authttl.authnum = author.authnum
and authttl.docnum = title.docnum"

references a third table, "authttl", which consists only of columns corresponding to columns in the "author" and "title" tables. This type of table is called an associative table. Its function is to enable a join in which the entities represented in two tables are related such that each row in one table may be related to any number of rows in the other table, 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) select first, last, title
2) from author, title, authttl
3) where authttl.authnum = author.authnum
4) and authttl.docnum = title.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
el	doctorow	ragtime

Group By Clause

The **group by** clause is used to apply an aggregate to a group of rows rather than to the table as a whole. The following query selects the total number of books on hand for each publisher.

```
1) select sum(onhand), pubnum
2) from title
3) group by pubnum;
```

sum(onhand)	pubnum
25	1
9	2
8	3
4	5

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

```
1) select sum(onhand)
2) from title;
```

sum(onhand)
46

When a **group by** clause is used, an optional **having** clause may be added to specify conditions to be met by the groups to be considered by the aggregate. The **having** and **group by** clauses are discussed in more detail in Part III under "Aggregates".

Subqueries

A qualification may refer to a value or set of values returned by a nested **select** statement, or subquery. Subqueries may be nested to any depth. They are usually used in the qualification of a **where** or **having** clause.

A subquery may be used on the right side of a relational operator in any qualification. The subquery, which is always enclosed in parentheses, is performed first, and its result is returned for use by the outer query. A simple subquery, with no modifier, returns a single result.

In the following query, the subquery first computes the average of the "onhand" column. This result is then used in the comparison in the **where** clause of the outer query. The entire query selects data from rows in which the "onhand" value is greater than the average of all the "onhand" values in the "title" table.

```
1) select title, pubnum, onhand
2) from title
3) where onhand >
4)       (select avg(onhand) from title);
```

title	pubnum	onhand
the c programming language	3	8
macbeth	1	12
so long and thanks for all the fish	1	7

A similar query could reference the "publisher" table in addition to the "title" table to supply the names of the publishers instead of the numbers. This requires a join of the two tables in the outer query.

```
1) select title, name, onhand
2) from title, publisher
3) where title.pubnum = publisher.pubnum
4) and onhand >
5)       (select avg(onhand) from title);
```

title	name	onhand
the c programming language	prentice hall	8
macbeth	penguin	12
so long and thanks for all the fish	penguin	7

If a *qualification* is modified by the keywords **any**, **all** or **in**, the subquery may return a set of values. If **all** is used, the condition is satisfied if the expression on the left side of the relational operator is true for all the values returned by the subquery. If **any** is used, the condition is satisfied if the expression on the left side of the relational operator is true for any of the values returned by the subquery. The keyword **in** is functionally equivalent to = **any**.

The following query uses a subquery with the **any** keyword to select data from any rows in the "publisher" table for publishers located in the same city as "signet".

```
1) select name, city
2) from publisher
4) where city = any
5)       (select city from publisher
6)       where name = 'signet');
```

name	city
signet	new york
random house	new york

Correlated Subqueries

A correlated subquery is a subquery which needs to reference a specific value in the row being examined in the outer query. It requires a correlation variable to establish the relationship between the tables in the inner and outer queries. The correlation variable signals the query to reevaluate the subquery once for every row in the outer query.

A correlation variable is a column name prefaced by a table label or a table name. A table label is a string which has been defined as an alias for a specific table at a different level of the query from that in which the correlation variable is used.

The following query defines 't' as a table label for the "title" table in the outer query. The correlation variable in the subquery is 't.pubnum'. For each row selected in the outer query, the subquery executes to establish whether the "onhand" value for the row selected in the outer query is greater than the average "onhand" value for titles by the publisher represented by the "pubnum" in the row currently being examined in the outer query.

```
1) select title, pubnum, onhand
2) from title t
3) where onhand >
4)       (select avg(onhand) from title
5)       where pubnum = t.pubnum);
```

title	pubnum	onhand
macbeth	1	12
moby dick	2	6

A similar query could reference the "name" attribute in the "publisher" table by joining the "publisher" and "title" tables in the outer query.

```
1) select title, name, onhand
2) from title t, publisher
3) where t.pubnum = publisher.pubnum
4) and onhand >
4)       (select avg(onhand) from title
5)       where pubnum = t.pubnum);
```

title	name	onhand
macbeth	penguin	12
moby dick	signet	6

Exists Predicate

The **exists** predicate tests for the existence of a row which satisfies a condition of a correlated subquery. The **not exists** predicate is also valid. When this predicate is used, the subquery returns only a value of true or false, true if at least one row is selected, false if no rows satisfying the subquery are selected.

The following query requests rows from the "publisher" table for which no corresponding "pubnum" exists in the "title" table.

```
1) select * from publisher p
2) where not exists
3)       (select * from title where pubnum = p.pubnum);
```

pubnum	name	city	phone
4	south end	boston	617-445-3223

INSERT

The **insert** command adds one or more rows to a table. This command can be used to insert new data directly from the terminal to an existing table, or to insert data from another table using a **select** statement.

For entering literal data from the terminal, the essential parts of an **insert** command are specification of the table to which the data is to be added and the values of the various fields. The basic form of the **insert** command is

insert into *specified table*
(*optional column names*)
values (*expressions*)

Specification of the column names is optional, but often provided for clarity. When column names are specified, there must be one value in the **values** clause for every column name listed, but all of the column names in the table need not be listed. Unlisted columns are assigned zeros for numerics and blanks for character strings. These columns can later be modified with the **update** command when the values are available.

If the optional column names are omitted, values for all of the columns in the table must be given. The values are inserted in the order in which the columns were specified when the table was originally created, in other words, the first value is assumed to apply to the first column, the second value to the second column, etc.[3] The examples here always specify the column names.

The following command inserts a new row into the "author" table.

```
1) insert into author
2) (authnum, first, last)
3) values (8, 'charles', 'dickens');
```

The next command inserts a new row into the "title" table. The value for the "docnum" column is an expression which evaluates to the next consecutive number in the column. The "onhand" column is omitted from the **insert** command and will thus be given a value of 0.

```
1) insert into title
2) (docnum, title, pubnum)
3) values (max(docnum) + 1, 'a tale of two cities', 1);
```

For entering multiple rows from one table into another use the form of the **insert** command which contains a **select** statement. The form of this version of the **insert** command is

insert into *specified table*
(*column names*)
select *specified columns* **from** *specified tables*
where *specified conditions*

Assume that there is a table called "modernauthor" which has three columns named "fname" "lname" and "num". The following command inserts into the "modernauthor" table the selected data from the "author" table.

[3] See the sections on the **create table** command in the chapter on data definition.

```
1) insert into modernauthor
2) (fname, lname)
3) select first, last
4) from author
5) where authnum in (1, 3, 4, 7);

4 rows affected

1) select *
2) from modernauthor;
```

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

UPDATE

The **update** command changes the values of one or more columns in the specified rows in the specified table. The conditions qualifying which rows are to be updated are specified in a **where** clause. If there is no **where** clause, all of the rows in the table are modified.

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

update *specified table*
set *column name = expression*
where *specified conditions*

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

```
1) update author
2) set first = 'douglas'
3) where first = 'doug' and last = 'adams';
```

More than one column value can be updated by a single **update** command. The following command updates two columns in the "title" table.

```
1) update title
2) set title = 'hamlet', onhand = 8
3) where docnum = 3;
```

The next command has no **where** clause. It increases by 5 the value of the "onhand" column in all of the rows of the "title" table.

```
1) update title
2) set onhand = onhand + 5;
```

It is possible to update the values in a table by fetching them from another table using an optional **from** clause specifying the table from which the new values are to be taken. The following command updates the "num" column in the "modernauthor" table with the values that are used for equivalent rows in the "author" table.

```
1) update modernauthor
2) from author
3) set num = authnum
4) where first = fname and last = lname;
```

DELETE

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

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

delete from *specified table*
where *specified conditions*

The following command deletes all of the rows in the "title" table in which the "onhand" column has a value less than 1.

```
1) delete from title
2) where onhand < 1;
```

The next command deletes all of the rows in the "title" table. After the command is executed, the table still exists, but it has no data in it.

```
1) delete from title;
```

Data Definition

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

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

This section and the next contain references to certain system tables which exist in every database, specifically to "batch", "transact", "descriptions", "relation", and "users". These are tables which are automatically created by the system in order to manage the database. For more information concerning the system tables, consult the *Database Administrator's Manual.*

CREATE TABLE

The **create table** command creates a new table in the open database. The command specifies the name of the table and the names and types of its columns.

The following type is used for character data.

char(*len*)	character strings

The following types are used for numeric data.

integer	four-byte integers
smallint	two-byte integers
tinyint	one-byte integers
float	eight-byte floating-point numbers
smallfloat	four-byte floating-point numbers
bcd(*len*)	binary-coded decimal integers
bcdflt(*len*)	binary-coded decimal floating point numbers

The following type is used for binary data.

binary(*len*)	binary strings

The **binary**, **bcd**, and **bcdflt** keywords may be prefixed with the word **fixed** if leading and trailing zeros are to be retained. The **char** keyword may be prefixed with the word **fixed** if trailing blanks are to be retained. If **fixed** is not specified for these types, trailing blanks and trailing and leading zeros are stripped.

The following command creates a new table named "price", with four columns named "docnum", "year", "amount", and "distrib". The "docnum" column is a two-byte integer field; the "year" column is a one-byte integer field; the "amount" column is a binary-coded decimal floating point field with a maximum length of six digits; the "distrib" column is a character field with a maximum length of twenty characters.

```
1) create table price
2)      (docnum smallint,
3)       year tinyint,
4)       amount bcdflt(6),
5)       distrib char(20));
```

A **select** command on "price" shows the empty table.

```
1) select *
2) from price;
```

docnum	year	amount	distrib

CREATE INDEX

An index is a directory which relates the physical location of each row in a table to the value of a specified column or group of columns in the table. The purpose of an index is to provide a direct access path to data when a query references the column(s) specified in the **create index** command.

The creation of indices can greatly decrease access time if a table is often searched on the basis of a particular column or set of columns, 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, but requires that the data be sorted on the value of the column(s) specified in the **create index** command. There can be only one **clustered index** for a single table. That single index may, however, be on multiple columns.

The following command creates an index on the "docnum" column of the "title" table.

```
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 of the data. It does not require that the data in the table be sorted. Up to 250 **nonclustered** indices can be created on a single table.

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

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

Unique Index

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

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

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

After creation of this **index**, if a user tries to add a row in which the value of the "authnum" column is the same as the value of the "authnum" column for a row which already exists in the table, an error message is generated and the entire **update** or **insert** command aborted.

CREATE VIEW

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

The **create view** command specifies the name of the view, the names of its columns, and a description of the data in the view as a **select** statement.

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

```
1) create view instock
2) (docnum, book, author, price) as
3)      select title.docnum, title, last, amount
4)      from title, author, price
5)              where authttl.docnum = title.docnum
6)              and authttl.authnum = author.authnum
7)              and title.docnum = price.docnum
8)              and onhand > 0;
```

The view can now be queried as though it were a table. It is possible to grant a user permission to query the "instock" view, without granting that user permission to query the base tables.[4]

[4]Granting permission is discussed in the chapter on data authorization.

```
1) select *
2) from instock
3) where author = 'lawrence'
4) and title like 'fantasia%';
```

docnum	book	author	price
5	fantasia of the unconscious	lawrence	4.95

STORE and START

The **store** command creates an object called a stored command. A stored command is a sequence of data manipulation commands, such as **select**, **insert**, **update**, 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 an ampersand (&). 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 **insert** command and a **select** 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" table. */
2) store addauthor
3)      insert into author              /* add the name */
4)      (authnum, first, last)
5)      values(max(authnum) + 1, &f, &l)
6)      select * from author            /* make sure it is there */
7)      where authnum = (select max(authnum) from author)
8) end store;
```

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

```
1) start addauthor
2) (f = 'pat', l = 'barker');
```

authnum	first	last
9	pat	barker

DROP

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

The following command removes the "modernauthor" table.

```
1) drop modernauthor;
```

The next command removes the "instock" view.

```
1) drop instock;
```

DROP INDEX

The **drop index** command removes an index from a table. The command identifies the index to be dropped 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" column in the "author" table.

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

AUTO-COMMENT

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

The **comment on** command is also automatically executed when an object is created. This command records information about a table or column in the system table "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 table. The text of the command which created the object, including comments, is inserted into the "text" column of the "descriptions" system table.

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

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

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

```
1) select text
2) from descriptions
3) where relid = table_id('price');
```

text
create table price (docnum smallint, year tinyint, amount bcdflt(6), distrib char(20))

The next query requests a description of "addauthor".

```
1) select text
2) from descriptions
3) where relid = table_id('addauthor');
```

text
/* This adds an author's name to the "author" table. */ store addauthor insert into author /* add the name */ (authnum, first, last) values(max(authnum) + 1, &f, &l) select * from author /* make sure it is there */ where authnum = (select max(authnum) from author) end store

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 grant these privileges using the **grant** command. Similarly, the owner, of an object may deny certain users or all users specific types of access using the **revoke** command.

PROTECT MODES

The types of access which can be granted and revoked are referred to as protect_modes. The protect_modes which apply to the objects described in this section are listed below.

Protect_Mode	*SQL Command*
read	**select, create view**
write	**insert, delete, update**
start	**start**
create	**create table, store**
create index	**create index**

If, for example, a user is granted read access on a table, but not write access, that user may issue **select** commands on that table, but not **insert**, **update**, or **delete** commands.

GRANT

The **grant** command permits 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 table in the open database. The keyword **public** represents all users.

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

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

```
1) grant write
2) on price
3) to susie;
```

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

```
1) grant read
2) (book, price)
3) on instock
4) to salesfolk;
```

The following command grants read privileges on all columns in the "title" table to all users.

```
1) grant read
2) on title
3) to public;
```

REVOKE

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

The following command revokes read privileges on the "title" table from all users.

```
1) revoke read
2) on title
3) from public;
```

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

```
1) revoke start
2) on addauthor
3) from public;
1) grant start
2) on addauthor
3) to susie, jason;
```

PART II

SQL COMMANDS

Introduction to SQL Commands

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

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

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

Comments enclosed by the characters /* and */ may be included anywhere any SQL input.

SEE ALSO

sql(1I) in *Host Software Specification* (UNIX systems)
SQL in *Command Summary* (other systems)

alter database db_name [**with** option_list]

DESCRIPTION

The **alter database** command increases or decreases the disk allocation for a database. The **demand**, **logblocks**, and **disks** options are the same as for the **create database** command except that **demand** or **logblocks** may be assigned a negative number.

A disk allocation may be increased while others are accessing the database.

If the options are omitted, **alter database** increases the allocation by one zone on any available disk.

OPTIONS

disk = "diskname"
Specifies the disk on which zones should be allocated or deallocated. The quotation marks are mandatory.

demand = nblocks [**on** "diskname"]
Specifies the number of 2K blocks to allocate (if ***nblocks*** is positive) or deallocate (if ***nblocks*** is negative). The number is rounded (away from zero) to the next zone size.

Only freeable zones are removed on deallocation. If the database is fragmented, few zones may be freed.

logblocks = nblocks [**on** "diskname"]
Specifies the number of blocks to allocate for the transaction log. If the optional diskname is specified and it differs from the previous disk, the old disk will continue to be used until the old transaction log is dumped. After the old transaction log has been dumped, the disk specified in the **alter database** command will be used for the new transaction log.

EXAMPLE

This command decreases the allocation for the "payroll" database by 20 blocks.

```
1) alter database payroll with demand= -20;
```

SEE ALSO

create database

alter table table_name [**with logging** [= {**0** | **1**}]]

DESCRIPTION

The **alter table** command changes the transaction **logging status** of a table.

OPTIONS

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

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

EXAMPLE

This command cancels transaction logging of the "unimportant" table.

```
1) alter table unimportant with logging = 0;
```

SEE ALSO

create table

```
audit [into table_name] { * | target-list }
                [from object_spec[, object_spec... ]]
                [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 table specified by *table_name*. The *qualification* and *target-list* can only refer to the columns listed below.

Column	*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 row id of the row concerned
relid	the table id of the table involved
number	internal transaction number
type	type of update (see table below)
value	data that was changed

The "value" column 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 columns of the table whose modification is recorded in the transaction log. Only one *object_spec* may be specified if the "value" field appears in the *target-list*.

The user must have **read** permission on all columns in the *target-list*. For the **audit into** command, the user must have **create** permission, and the name selected for *table_name* must be unique.

The values in the "type" column are interpreted as follows:

Type	*Meaning*
00	null
16	begin query
18	replace old
19	replace duplicate
1A	append duplicate
1C	end query
1D	abort query
1E	checkpoint
1F	safepoint
C3	insert
C4	delete
C5	update
C6	create
C7	drop
C8	create index
CD	grant
CE	revoke
D4	begin transaction
D5	end transaction
E1	store
EB	dump transaction

EXAMPLES

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

```
1) audit type, date
2) from transact
3) where relid = table_id('parts')
4) and date > getdate - 2;
```

This command stores, in the table "inv_audit", the changes that were made to the table "inventory". The audit report is generated from "log5".

```
1) audit into inv_audit type, date
2) from log5
3) where relid = table_id('inventory');
```

SEE ALSO

create table
"Object_Name", "Object_Spec", "Qualifications", "Target-Lists"
idmdump(1I) in *Host Software Specification*
IDMDUMP in *Command Summary*

comment on [**table**] object_name
[**is** string1[, string2]]

comment on column object_name.column_name
[**is** string1[, string2]]

DESCRIPTION

The **comment** command adds or replaces information in the system table "descriptions". This table is used to associate one or more textual descriptions with an object. The object is described in the table as a table-id/column-id pair. The "descriptions" table uses the IDL terminology "attid" for "column-id" and "relid" for "table-id".

Two rows in the "descriptions" table might look like this:

attid	relid	key	text
0	29033	I1	table listing all parts
3	29033		column for quantity on hand

If the **comment on table** version of the command is given, the entry in "descriptions" pertains to the named object. This object can be a table, view, stored command, or file. In this case, the "relid" in the "descriptions" table gets the value of the "relid" for that object as it is recorded in the system table "relation". The "attid" in the "descriptions" table gets a value of zero (0).

If the **comment on column** version of the command is given, the description refers only to the named column in the named object. In this case, the "attid" in the "descriptions" table gets the value of the "attid" for that column as it is recorded in the system table "attribute".

The *string1*, if specified, is inserted into the "text" column. The *string2*, if specified, is inserted into the "key" column. If entries already exist for "text" or "key", they are replaced by the new values. Both *string1* and *string2*, if specified, must be entered as quoted character strings.

The function of the optional "key" column is user-defined. It is frequently used as a sequential line number for descriptions in the "text" column. For example, the following sequence of **comment** commands appends a four-row description of the "mytable" table.

```
1) comment on table mytable is 'This is my very own','M1';
1) comment on table mytable is 'table which has','M2';
1) comment on table mytable is 'only two columns','M3';
1) comment on table mytable is 'called num and name','M4';
```

The description of "mytable" can then be selected ordered by the "key" column:

```
1) select text
2) from descriptions
3) where relid = table_id('mytable')
4) order by key;
```

If the **is** clause is omitted, any existing comment for the named object is removed.

The user must be the owner of the specified object.

AUTO-COMMENT

The **comment** command is automatically executed whenever a **create table**, **create view**, or **store** 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 inserted into the "text" column of the "descriptions" table. This feature provides automatic documentation of tables, views and stored commands.

If the text of a command creating a table, view or stored command exceeds 4000 bytes in length, it will overflow the space allocated for it in the "text" column of the "descriptions" table. To prevent this from occurring when entering long commands, the user can turn off the auto-comment feature by invoking **sql** with the **–c** or **/nocomment** flag, or turn auto-comment off and then on again by using the front-end command **%comment**.

EXAMPLES

To comment on the "parts" table:

```
1) comment on table parts is 'table listing all parts';
```

To add the information that there is an index on the "number" column with the user-assigned key "I1":

```
1) comment on table parts is
2) 'has a clustered index on number', 'I1';
```

To provide a verbal description of a view called "partview":

```
1) comment on partview is
2) 'select * from parts where pnum < 20', 'V';
```

To remove all descriptive text pertaining to the "parts" table:

```
1) comment on parts;
```

SEE ALSO

create table, **create view**, **drop**, **store**
%comment

commit [work]

DESCRIPTION

The **commit work** command ends the current transaction. A transaction is a logical sequence of SQL commands which are to be treated as a single command. All commands in a transaction are either executed, if a **commit work** command is issued, or not executed at all, if a **rollback work** command is issued. The **commit work** command commits all changes made to a database since the last **commit work**, **rollback work**, or, if neither has previously been issued, since the **autocommit** option was turned **off**.

If the **commit work** command is issued when **autocommit** is **on**, an error is reported and **autocommit** is then turned **off**. If **autocommit** is desired, be sure to **set** it **on** again.

EXAMPLE

```
1) set autocommit off
1> delete from stores
2> where storenum <= 0;

2 rows deleted

1> commit work;
```

In this example, the rows are not actually deleted until the **commit work** command is issued. If the user had decided that the **delete** was not desirable, a **rollback work** command would have undone the **delete**.

SEE ALSO

rollback work, **set**

create database dbname [**with** option_list]

DESCRIPTION

The **create database** command creates a new database that is empty except for the system tables.

On creation of a database, the system table "host_users" is initialized with one row allowing access only to the creator. The creator of a database is therefore the owner and DBA (database administrator) of the database.

The **create database** command is executable only from the "system" database. The user must have **create database** permission in the "system" database.

OPTIONS

Options, if specified, are separated by commas. The following options are available:

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

This option specifies the the number of 2K blocks assigned to the database. The *nblocks* must be an integer.

A zone is a group of cylinders, the precise number of cylinders per zone varying from disk to disk. The number of cylinders per zone is specified when the disk is formatted; zone sizes range from 128 to 254 blocks. The "bpz" column in the system table "disks" in the "system" database indicates the zone size for any disks attached to the database server.

Since database allocations are only made in whole numbers of zones, the number of blocks specified is rounded up to the first whole number of zones, the allocation made, and the number of blocks actually allocated displayed at the user's terminal.

The database is not allowed to grow beyond the size allocated. An error is reported if the database attempts to grow beyond this size. Use the **alter database** command to change the allocation for an existing database.

If a "diskname" is specified, the allocation is made on that disk; otherwise, the allocation is made on any disk which 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"]

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

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"

This 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 that has sufficient space. The "diskname" must be entered as a quoted character string. The specification

```
with demand = 3000, disk = "disk1",
logblocks = 1000, disk = "disk2"
```

requests 3000 blocks on "disk1" for the database and 1000 blocks on "disk2" 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

The commands

```
1) create database soo with demand = 3000 on "disk1";
```

and

```
1) create database soo with demand = 3000, disk = "disk1";
```

are equivalent. They create a database of 3000 blocks on "disk1".

The following command creates a database on "disk1" with a default demand of one zone. All character data in this database is stored in the EBCDIC character set.

```
1) create database soo with disk = "disk1", ebcdic;
```

SEE ALSO

alter database, **drop database**, **grant**, **revoke**

create [**unique**] [**clustered** | **nonclustered**] **index**
on object_name (column_name[, column_name ...])
[**with** option_list]

DESCRIPTION

Indices are used to provide fast access to data. If rows in a table are often searched on the basis of a particular column, it is appropriate to create an index on that column to reduce access time.

An index specifies a particular column or set of columns called keys on which a table is searched. For example, if a table represents a telephone book, one could create an index on the columns "lastname, firstname". This would speed up the search when data in the phone book is accessed with a qualification based on the "lastname" and "firstname" columns.

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 table be stored in an order governed by the key to the index. On creation of a **clustered index**, the data in the table is sorted according to the values of the column(s) specified for the index, and a modified B*-tree index is built. Only one **clustered index** is permitted for a single table. When the index is created, all existing indices on that table are destroyed unless the **recreate** option is specified. In addition, when the **clustered index** is created, duplicate rows (identical in all columns) 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 table. Attempting to create a **nonclustered index** which already exists is an error. The maximum size for the keys of a **nonclustered index** is 248 bytes.

A **unique index** can be created for tables in which values of all of the columns of the index 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" column, the user is not permitted to assign to a row a social security number which already appears in another row in the table. A **unique** index may be **clustered** or **nonclustered**.

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

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

OPTIONS

delete_dups

If **delete_dups** is specified for a **unique clustered index**, and duplicate values on the indexed columns are found in the table while the data is being sorted, as many rows 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 table such that the indexed columns would no longer be unique, the modification does not occur (i.e. the row in question is not added or modified). The user is informed that the duplicate was not inserted or updated, but the entire **insert** or **update** 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 table is written to disk. The **fillfactor** value specifies the percentage of each block to be filled when the table 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. Tables 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 table has become scattered (meaning that blocks containing data which should be in sort order are spread over several cylinders), I/O time will increase significantly. 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 when building a sorted table for creation of a **clustered index**. 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 table 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.

EXAMPLES

The following command causes the "parts" table to be sorted on (name, number), written to the disk in blocks 40% full, and a B*-tree index to be created for the (name, number) pairs. When the table is accessed with the "name, number" columns specified, the access is direct; only the index and the exact blocks needed are read, not the entire table.

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

The "parts" table already has a **clustered** index (from the previous example). The next command creates a **nonclustered** index on "number" to simplify access to the "parts" table 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 table so that the uniqueness of the "number" column were not preserved, the entire **insert** or **update** 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 table so that the uniqueness of the "number" column 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" table 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
2) parts (name, number)
3) with recreate;
```

SEE ALSO

create table, **drop index**, **set**

create table table_name (name type[, name type ...]) [**with** option_list]

create table table_name ([partition_name](name type[, name type ...])
[**with** option_list] [,[partition_name] (name type
[, name type ...]) [**with** option_list]] ...) [**with** option_list]

DESCRIPTION

The **create table** command creates an empty table in the open database.

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

A *type* is a mnemonic for a data type for a particular column. Please refer to the section "Types" in this manual for a list of the predefined mnemonics and a detailed description of the various *types* to which they refer.

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

When the table is created, it is empty, and no indices exist for it. If the table is heavily used, a **clustered index** should be created for it as soon as it has grown to several blocks of data or when the initial loading of data has been completed.

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

The user must have **create** permission to use this command.

OPTIONS

quota = n

This specifies the maximum size of the table in 2048-byte data blocks, excluding the index blocks. If no **quota** is specified, the table will be allowed to grow until it fills the database.

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

If set to 1, this option specifies that the transaction log "transact" is to be updated whenever the table is updated. If set to 0, the transaction log is not maintained, but changes to the table are recorded in the temporary the system table "**batch**". If the **logging** option is used but neither 0 nor 1 is specified, the default is 1.

EXAMPLE

This command creates a table named "parts" with columns "pname" (a 20-character field), and "quan" (an integer field). It may grow to a size of 50 data blocks, after which point an error message is displayed if further additions are attempted. The **logging** option causes all changes to the table to be recorded in the system table "transact".

```
1) create table parts
2) (pname char(20), quan integer)
3) with logging = 1, quota = 50;
```

SEE ALSO

alter table, **audit**, **comment**, **create index**, **drop**, **grant**, **revoke**
"Types" — p125
%comment

create view view_name [(col_name[, col_name ...])] **as** select_statement

DESCRIPTION

The **create view** command sets up a virtual table which is composed of parts of one or more tables (called the base tables) or other views. The *select_statement* portion of the command selects data to be accessed by the view.

If the column names are not specified, the columns in the view will have the same names as the columns in the base tables. The column names need not be specified unless the resulting view would have more than one column with the same name, or the creator of the view wishes to assign different names to the view columns.

Views may be protected and destroyed in the same manner as tables; they may be updated if the update can unambiguously be applied to a base table. If a view is an exact copy of a base table, it can be modified with **delete**, **insert**, and **update** commands with the net result of modifying the base table.

A view should be created when it is desirable to access data from more than one table collectively, or to restrict access to certain parts of a table.

Views are recorded in the system table "query". Since a view is dependent on its base tables, a user cannot destroy a base table 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 tables. 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 SQL pseudo-command **%input** to create it in both databases.

When **create view** is executed, the **comment** 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 **comment** command into the "descriptions" table. This feature provides automated documentation of views.

The user needs **read** permission on all the columns referenced to create a view, but not **create** permission.

If the creator of the view has access to the system tables "host_users" and "users" but is not the DBA, (s)he cannot grant access to the view to another user. This prohibition is to prevent uncontrolled proliferation of permission. Similar rules hold for stored commands.

EXAMPLES

This command creates a view "vparts" which ressembles a table with two columns, even though "parts" may have many more columns. Only rows in which "num" is greater than 20 are included in the view. The view has the same column names as the columns in the "parts" table from which they were derived.

```
1) create view vparts as
2)      select num, name
3)      from parts
4)      where num > 20;
```

This view is useful for seeing which indices exist in a database:

```
1) create view indexes
2) (inum, tab_name, col_count, uniq, del_dups) as
3)      select indid, table_name(relid), attcnt,
4)      abs(mod(stat, 2)), abs(mod(stat/4, 2))
5)      from indices;
```

SEE ALSO

comment, **create table**, **drop**, **grant**, **revoke**, **select**
"Aggregates", "Qualifications"
%comment, **%input**

delete from object_spec [label] [**where** qualification]

DESCRIPTION

The **delete** command removes one or more rows from a table. The user must have **write** permission for all the columns of the table. If the *qualification* is omitted, all rows are deleted from the table.

The *label* is specified only if a correlated subquery is used in the *qualification*. Correlated subqueries are described under "Qualifications".

EXAMPLES

To delete all rows in the "parts" table in which the "name" column has the value "TV":

```
1) delete from parts where name = 'TV';
```

To delete all rows from the "parts" table for which there are no parts on hand and no corresponding entry in the "suppliers" table:

```
1) delete from parts p
2) where onhand < 1
3) and not exists
4)      (select *
5)      from suppliers
6)      where pnum = p.num);
```

To delete every row in the "parts" table:

```
1) delete from parts;
```

SEE ALSO

drop, **grant**, **revoke**, **truncate**
"Qualifications"

drop object_name[, object_name, ...]

DESCRIPTION

The **drop** command eliminates tables, files, views, and stored commands from the database server. The entire object is removed from the database, and its space is freed for use within the current database. Only the owner of the object or the DBA can **drop** an object. If there are views or stored commands that depend on the table or view to be dropped, they must be dropped first. Appropriate entries in the system table "descriptions" are deleted when this command is invoked.

EXAMPLE

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

```
1) drop parts, products;
```

SEE ALSO

comment
"Object_Name"

drop database dbname[, dbname, ...]

DESCRIPTION

The specified databases are eliminated from the database server, and the space is freed for use by other databases. All tables and files in the dropped database are destroyed. The user must be the owner of the database or DBA of the "system" database to drop a database. The database to be dropped cannot be open at the time of the command.

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

EXAMPLE

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

```
1) drop database inventory;
```

SEE ALSO

create database

drop [unique] [clustered | nonclustered] index
on object_name (column_name[, column_name, ...])

DESCRIPTION

The **drop index** command removes an index from a table. 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 row fields that it indexes are updated.

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

The user must be the owner of the table, and the specified index must exist.

EXAMPLES

This command destroys the **clustered index** on (name, number). Initially the table remains sorted on (name, number). New data is appended to the end of the table.

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

This command destroys the **nonclustered index** on the "number" column of the "parts" table.

```
1) drop nonclustered index on parts (number);
```

SEE ALSO

create index

exit

DESCRIPTION

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

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

grant protect_mode [**on** object_name] **to** { user [, user ...] | **public** }
grant protect_mode [(col_name[, col_name ...])] **on** object_name **to** { user [, user ...] | **public** }

DESCRIPTION

The **grant** command permits access to an object to a specific *user* or a group of *users*. The *user* may be a user name, a group name, or the keyword **public**. A group is any entry in the system table "users" for which the "uid" is equal to the "gid". The keyword **public** designates 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 **grant** such access.

The *object_name* may represent a table, view, file, or stored command.

The *protect_modes* which may be granted are listed in the section "Protect_Modes" under "General Concepts".

A **grant** command supersedes any previous **revoke** commands which contradict it.

The user of the **grant** command must be the owner of an object to **grant** access to it. The DBA may also **grant** permission to use the **create**, **create database**, and **create index** commands and to use 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 owners of those objects and of the view or stored command are the same.

EXAMPLES

The following command permits "george" to read the "parts" table.

```
1) grant read on parts to george;
```

The following commands permit only the *users* in the "managers" group and "dave" to **start** the stored command "getsum".

```
1) revoke start on getsum from public;
1) grant start on getsum to managers;
1) grant start on getsum to dave;
```

The next command grants "bill" and "sharon" permission to write on the "quan" column of the "parts" table, but on no other attributes in this table.

```
1) grant write (quan) on parts
2) to bill, sharon;
```

To allow *user* "gloria" to create tables in the current database the DBA must issue

```
1) grant create to gloria;
```

To allow all *users* to create tables the DBA would issue

```
1) grant create to public;
```

NOTE

There is an earlier version of this command, with somewhat different syntax, which is still recognized for reasons of backward compatibility. However, with Host Software Release 3.6, the syntax described above will be the only syntax supported.

SEE ALSO

create database, **create index**, **create table**, **revoke**
"Protect_Modes", "Users"

ignore

DESCRIPTION

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

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

EXAMPLE

```
1) select coump(*)
2) from employees
3) where salary > 2000 and ignore
1)
```

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

insert into object_name [(column_name[, column_name...])] **values** (expression[, expression ...])
insert into object_name [(column_name[, column_name...])] select_statement

DESCRIPTION

The **insert** command adds one or more rows to a table or view.

When the **values** form is used, if the *column_names* are specified, the data is inserted into those columns. If the *column_names* are specified, there must be one *column_name* for each element of the **values** list. If *column_names* are not specified, the data is inserted into the columns in the named object in the order in which the columns are displayed from left to right when all of the columns of the table are selected. This reflects the order in which the columns were specified when the table was originally created.

If the *select_statement* form is used, data to be inserted is specified by the *select_statement*. Data may be selected from the object represented by *object_name* or from other objects. The number of values selected by the *select_statement* must be the same as the number of columns named (if any are specifically named) or the same as the total number of columns in the target object.

Options may be **set** to turn off certain forms of error checking and to ignore duplicates. See the **set** command for the list of options.

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

EXAMPLES

This command uses the **values** form with *column_names* to insert a row into the "parts" table. In the new row, the "name" column has the value "tube", and the "quan" column has the value 24.

```
1) insert into parts
2) (name, quan)
3) values ('tube', 24);
```

The following command uses the **values** clause without *column_names* to insert a row into the "parts" table, setting the first column to the value "tube" and the second column to the value 24.

```
1) insert into parts
2) values ('tube', 24);
```

This command uses the *select_statement* with *column_names*. For every row in "parts", it adds a new row to "newparts", setting the "name" column to the value of "part" and the "quan" column to the value of "onhand". "Part" and "onhand" are columns in the "parts" table; "name" and "quan" are columns in the "newparts" table.

```
1) insert into newparts
2) (name, quan)
3)      select part, onhand
4)      from parts;
```

This command uses the *select_statement* without *column_names*. For every row in "oldparts", it adds a new row to "newparts", setting the first column in "newparts" to the value of "name" in "oldparts" and the second column in "newparts" to the value of "onhand" in "oldparts".

```
1) insert into newparts
2)      select name, onhand
3)      from oldparts;
```

SEE ALSO

select, **set**
"Object_Name"
idmfcopy(1I) in *Host Software Specification*
IDMFCOPY in *Command Summary*

open name

DESCRIPTION

The **open** command opens the specified database for activity. A database must be open before any SQL commands are executed. The database remains open until the user opens a different database, or until SQL terminates.

The user is allowed to **open** a database if its system table "host_users" is empty, or if it contains a "guest" row, or if it contains a row that exactly matches the user's host id and host-users id.

EXAMPLE

1) open books;

reconfigure

DESCRIPTION

The **reconfigure** command updates the configuration of the database server according to the contents of the system table "**configure**" in the "system" database. The user must be the DBA of the "**system**" database, and the command may only be issued from the "**system**" database.

SEE ALSO

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

revoke protect_mode [**on** object_name]
from { user [, user ...] | **public** }

revoke protect_mode [(col_name[, col_name ...])]
on object_name **from** { user [, user ...] | **public** }

DESCRIPTION

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

The *user* may be a user name, a group name, or the keyword **public**. A group is any entry in the system table "users" for which the "uid" is equal to the "gid". The keyword **public** designates all *users*.

When an object is first created, the *protect-modes* are set so that the creator of the object is granted all types of access while other *users* are denied all types of access.

The *object_name* may be a table, view, file, or stored command. If no object is specified, the protection applies to all objects.

The *protect_modes* which may be revoked are listed in the section "Protect_Modes" under "General Concepts".

A **revoke** command overrides any previous **grant** commands which contradict it.

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

Only the owner of an object or the DBA may **revoke** permissions.

EXAMPLES

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

```
1) grant read on parts to public;
1) revoke read on parts from george, harvey, mary;
```

The following command denies entire group "clerks" **write** permission on the "descript" and "pnum" columns of the "parts" table. The "clerks" has been previously defined as a group in the system table "users". They may still write to other columns in the "parts" table.

```
1) revoke write (pnum, descript) on parts from clerks;
```

NOTE

There is an earlier version of this command, with somewhat different syntax, which is still recognized for reasons of backward compatibility. However, with Host Software Release 3.6, the syntax described above will be the only syntax supported.

SEE ALSO

create table, **grant**
"Protect_Modes", "Users"

rollback [work]

DESCRIPTION

The command **rollback work** aborts the current transaction. A transaction is an atomic sequence of SQL commands initiated by **a set autocommit off**. The **rollback work** command restores the database to its state prior to the last **commit work** command, if one has been issued, or since the **autocommit** option was turned **off**.

The user is informed when a **rollback work** command causes commands to be aborted.

If the **rollback work** command is issued when **autocommit** is **on**, an error is reported and **autocommit** is then turned **off**. If **autocommit** is desired, be sure to **set** it **on** again.

EXAMPLE

```
1) set autocommit off;
1> delete from stores
2> where storenum < 3;

2 rows deleted.

1> rollback work;
*** Warning: Work rolled back
```

The user decided that the **delete** was not desirable. The **rollback work** caused the deletion to be annulled. After the **rollback work** command has been issued, it is as if the **delete** command had never been issued. If the user had decided to allow the **delete**, a **commit work** command would have made the **delete** permanent.

SEE ALSO

commit work, **set**

```
select [ distinct | all ] [into table_name] { * | target-list }
       [from object_spec[, object_spec, ... ]]
       [where qualification]
       [group by column_name [having qualification]]
       [order by order_spec[, order_spec... ]]
```

DESCRIPTION

The **select** command is used for extracting data from the database server from one or more tables or views. The **select into** command sends data to a newly created table containing the columns specified in the *target-list*. It is an error to **select into** an existing table.

Duplicate Rows

Duplicate rows are ignored if **distinct** is specified; they are selected if **all** is specified. If neither is specified, the default is **all**.

Specifying Columns

The user must specify the data to be selected from the tables. If an asterisk (*) is specified, all columns in the table are selected. If a *target-list* is used, only the specified targets are selected. The formats for a *target-list* are listed under "Target-Lists".

When data is selected from an existing table **into** a new table, and the "*domain_name = expression*" form is used to specify the *target*, the *domain_names* (such as "name" and "date" in the query below) become the *column_names* in the new table. *Column_names* longer than 12 characters are truncated. If only a *column_name* is specified as the target, the *column_name* of the column in the new table is the same as that in the old table.

In the following query, the data for the "name" and "hiredate" columns is taken from the "employees" table; the data for the "date" and "gettime" columns is calculated by predefined database server functions which return the current date and time.

```
1) select into newhires
2) name = lname, date = getdate,
3) gettime, hiredate from employees
4)       where hiredate  > getdate - 30;
```

The resulting table, "newhires", has four columns (although "employees" may have more).

Specifying Tables

The most common way to identify tables or views from which data is to be selected is to use a **from** clause.

The following query specifies a table using a **from** clause.

```
1) select name, number
2) from stores;
```

Multiple tables may be listed in a **from** clause. When this method is used, the database server searches all the listed tables for the columns

named in the *target-list*. The following query selects columns from the "stores", "items", and "prices" tables. It assumes that the *column_names* in the *target-list* are unique among the specified tables.

```
1) select name, address, item, price
2) from stores, items, prices;
```

If the *column_names* are not unique, the resulting ambiguity must be resolved using another format. The following queries could be used to specify the "name" column from the "stores" table if more than one of the specified tables had a "name" column.

```
1) select stores.name, address, item, price
2) from stores, items, prices;
```

or

```
1) select st.name, address, item, price
2) from stores st, items, prices;
```

These two queries are equivalent. In the first example, the "name" to be selected is specified to be a column in the "stores" table by prepending the *table_name* to the *column_name* in the *target-list*. If there is a "name" column in the "items" table, it is ignored. In the second example, a *table_label* "st" is defined in the **from** clause and used to identify the "name" column in the *target-list*.

Table_labels provide a useful way to join a table with itself. The following query selects stores and prices of pink tub-and-sink combinations from all dealers in the "pricings" table.

```
1) select a.storenum, a.price + b.price
2) from pricings a, pricings b
3) where a.storenum = b.storenum
4)       and a.part  = 'tub'
5)       and b.part  = 'sink'
6)       and a.color = 'pink'
7)       and b.color = 'pink';
```

For more information about *table_labels*, refer to "Target-Lists" and to the discussion of correlated subqueries under "Qualifications".

Queries that do not access tables do not, obviously, need to specify tables. The query

```
1) select 'date', getdate;
```

simply returns the character string "date" and the current date.

Options

Each object specification in the **from** clause may be followed by a comma-separated list of special processing options. The entire list is enclosed in parentheses. The options are discussed under "Object_Spec".

Qualifications

The **where** clause specifies one or more conditions to apply in selecting data. If the **where** clause is omitted, all rows in the source object(s) are selected. See "Qualifications" for a detailed discussion of the **where** clause.

Special Clauses

The **group by** and **having** clauses are usually used with *aggregates*. See "Aggregates" for a detailed discussion of these special clauses.

Sorting Output

The **order by** clause causes the selected rows to be sorted by value of a specified expression. The syntax of an *order_spec* is:

```
{ column_name | column_number | expression }
[ a | asc | d | desc ]
```

The direction specifiers **a** and **asc** specify that the rows are to be sorted in ascending order, while **d** and **desc** specify descending order. If no direction is specified, the default order is ascending.

If a *column_name* is specified, the results are sorted by the values in that column.

```
1) select name, quan
2) from parts
3) order by partnum;
```

The named column does not have to be in the *target-list*.

If a column number *i* is used, output is sorted by the *i*th element in the target-list. If * is specified, it is sorted by the *i*th column which was specified when the table was created.

```
1) select num, name, quan from parts
2) order by 3;
```

Finally, any arbitrarily complex numeric expression may be used to order output:

```
1) select *
2) from accounts
3) order by (assets - liabilities) d;
```

EXAMPLES

The following query displays all the rows of the "stores" table in which the "city" field has the value "berkeley". The results are sorted by the "storenum" field in ascending order. Minimum locking allows the query to be processed although other users may be accessing the "stores" table.

```
1) select *
2) from stores (minlock)
3) where city = 'berkeley'
3) order by storenum;
```

The following query creates a new table named "newparts" with columns named "name", "num", "price", and "date". The values for these rows are selected from the "prices" and "parts" tables. Columns named "name" exist in both the "parts" and "prices" tables, so a *table_label* "pa" is used to specify that the reference is to the "name" column in the "parts" table. This query uses the system-supplied function **getdate**, which returns the date.

```
1) select into newparts
2) pa.name, num, prices, date = arrival
3) from parts pa, prices
4)      where num = partnum
5)      and arrival > getdate - 2;
```

SEE ALSO **create view**, **insert**, **update**
"Aggregates", "Functions", "Object_Name", "Object_Spec", "Qualifications", "Target-Lists"

set { option-name | option-number } [**on** | **off**]

DESCRIPTION

The **set** command enables certain options for SQL commands. The option can be specified by a name or a number. If neither **on** nor **off** is specified, the option is set **on**. Valid options are discussed below.

Autocommit Option

The **autocommit** option is used to specify whether commands are to be committed automatically. If **autocommit** is set **on**, changes to tables caused by SQL commands are processed or committed as soon as they are entered. When SQL starts up, this option is turned **on** automatically.

If **autocommit** is set **off**, commands are bundled into logical units of work called transactions. Multiple SQL commands are then handled as a single command or transaction.

When **autocommit** is **off** (a transaction is being built) the only legal SQL commands are

commit work
delete
insert
rollback work
select
start
update

A transaction begins when the user sets **autocommit off**, or after a **commit work** or **rollback work** command is issued. A transaction ends when a **commit work** or **rollback work** command is issued, or when **autocommit** is set **on** again. The work in the current transaction is not committed to a database until a **commit work** command is issued. All work in the current transaction is aborted if a **rollback work** command is issued.

The following sequence produces three transactions.

```
1) set autocommit off;          /* begins 1st transaction */
1> update.....
2> select.....
3> commit work;                 /* commits 1st transaction,
                                   begins 2nd transaction */
1> insert....
2> select.....
3> rollback work;               /* aborts 2nd transaction,
                                   begins 3rd transaction */

1> select.....
2> delete....
3> insert.....
```

```
4> commit work;              /* commits third transaction */
1> set autocommit on;        /* gets out of transaction mode */
1)
```

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 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 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 a **rollback work** command.

```
1) set autocommit off;
1> update customers
2>     set balance = balance - 100
3>     where name = 'debtor';

1 row affected

1> update customers
2>     set balance = balance + 100
3>     where name = 'creditor';

1 row affected

4> commit work;
1> set autocommit on;
1)
```

OTHER OPTIONS

A number of other options can be **set**, using either the *option-name* or *option-number*. They are listed following:

1 format
Set format before query.

2 names
Send result names.

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 rows with duplicate keys which are generated by modifications to the table.

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 rows affected 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 tables with one large table, (2) but do not join the **small** tables with each other, (3) and have very few qualifying rows in each

small table, (4) and can use a selective index to access the large table.

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

This command causes SQL to suppress the "Rows **affected**" **messages** that are usually displayed with query results.

```
1) set nocount on;
```

SEE ALSO

commit work, **rollback work**

start qname[(name = constant[, name = constant ...])]
start qname[(constant[, constant ...])]

DESCRIPTION

The **start** command executes the stored command *qname*, which was previously created with the **store** command.

The *constants* specify values to be substituted for the formal parameters 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 **store** command. For example, if a stored command "mycommand" were defined as

```
1) store mycommand
2)      insert into emps(name, num, dept)
3)      values(&empname, &empnum, &deptnum)
4) end store;
```

a **start** command could look like

```
1) start mycommand
2) (empname = 'Smith', empnum = 2456, deptnum = 102);
```

The *name* = *constant* assignments may be given in any order.

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) start 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.

A parameter to a stored command may be a *pattern* string as described in "PATTERNS" under "Qualifications". In such cases, the **like** predicate is used to specify the parameter, both in the **store** command which creates the stored command and in the **start** command which executes it. For example, assume a stored command defined as

```
1) store search
2)      select * from employees where name like &name
3) end store;
```

To invoke this stored command to find all employees whose names begin with the letter 'J', use

```
1) start search (like 'J%');
```

or

```
1) start search (name like 'J%');
```

EXAMPLE

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

```
1) store newitem
2)      insert into expend(salesman, amt, time, date)
3)      values (&name, &amount, gettime, getdate)
4) end store;
```

This stored command can be executed with

```
1) start newitem (name='mike', amount=44);
```

or

```
1) start newitem (44, 'mike');
```

In the last example, the value 44 is substituted for the "amount" parameter and the value "mike" is substituted for the "name" parameter. The values must be listed in this order because of the alphabetic ordering of the parameter names ("a" before "n").

SEE ALSO

comment, **drop**, **store**
"Constants", "Qualifications"

store qname command [command ...] **end store**

DESCRIPTION

The **store** command creates a stored command (also called a "stored query") in the database server. A stored command is a sequence of one or more SQL commands which can be referenced collectively by the *qname*.

The *command*s used in the stored command may include

commit work
delete
insert
rollback work
select
set
update

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

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 an ampersand ('&'). Later, when the stored command is executed by the **start** command, the user supplies the values to be substituted for the parameters.

Create permission is not required to define a stored command, but the creator of the stored command does need the permissions necessary for each command as though each constituent command were being entered manually.

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

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

EXAMPLES

This command creates a stored command named "additem".

```
1) store additem
2)      insert into items (iname, number)
3)      values (&name, &num)
4)      select (count = count(number))
5)      from items
6)      where number = &num)
7) end store;
```

The stored command "additem" could be invoked as follows:

```
1) start additem ('bertha', 46);
```

or

```
1) start additem
2) (name = 'bertha', num = 46);
```

SEE ALSO

comment, **drop**, **grant**, **revoke**, **set**, **start**
"Constants"
%comment, **%input**

sync

DESCRIPTION

The **sync** command initiates a checkpoint in the open database. Any disk blocks that may have (temporarily) been kept in volatile database server memory are written out to disk.

truncate table_name[, table_name ...]

DESCRIPTION

The **truncate** command deletes all rows from the named tables. The tables continue to exist, but they contain no data, as is the case when a table is first created. This command is the functional equivalent of **delete from** *table_name*, except that **truncate** can empty several tables with a single command.

EXAMPLE

1) truncate invoices;

This command removes all rows from the "invoices" table.

SEE ALSO

delete

```
update object_name [label]
                   [from object_spec[, object_spec ... ]]
                   set col_name = expression[, col_name = expression ... ]
                   [where qualification]
```

DESCRIPTION

The **update** command is used to change the values of one or more fields in one or more rows of a table or view. See **create view** for restrictions on updating views.

The *expression* may refer to values from other tables in which case the other tables involved are named in the **from** clause.

Columns that are to be updated must be explicitly named in the **set** clause. Columns that are not named are not changed.

The optional **where** clause may refer to the object being updated or to the objects listed in the **from** clause. The **where** clause may be used to select the rows to be updated, or to select data from other tables. If no **where** clause is specified, all rows in the table are updated.

EXAMPLES

This query changes the value of the "name" column for all rows in the "parts" table for which "name" has the value "transistor" to the value "electronic".

```
1) update parts
2) set name = 'electronic'
3) where name = 'transistor';
```

This query sets a new value in the "price" column in all rows in the "products" table in which the following conditions prevail: (1) the value of the "name" column in the "products" table equals the value of the "name" value in the "parts" table and (2) the value of the "name" column is "tube". The purpose of this command is to set the price of all tubes to be 10% higher than the purchase cost as reflected in the "cost" column of the "parts" table.

```
1) update products pr
2) from parts pa
3) set price = cost + cost/10
4) where pr.name = pa.name
5) and pa.name = 'tube';
```

SEE ALSO

create view, **insert**, **select**
"Object_Name", "Object_Spec", "Qualifications"

PART III

GENERAL CONCEPTS

Introduction to General Concepts

This part describes various components of an SQL command, such as *expression* or *qualification*, which may appear in a number of different SQL commands.

Some of these components are defined in terms of other components, all of which are described in this part.

Aggregates

Aggregates are used in queries and subqueries. The **aggregate** operators available in SQL are

Aggregate	*Returns*
sum(arg)	sum of all elements
sum(**distinct** arg)	sum of all distinct elements
count(arg)	count of elements
count(**distinct** arg)	count of distinct elements
avg(arg)	average of elements
avg(**distinct** arg)	average of distinct elements
once(arg)	returns one and only one value; if more or less than one value is found, returns an error
once(**distinct** arg)	once of distinct elements
any(arg)	0 if no elements exist, 1 if one or more elements exist
max(arg)	maximum of elements
min(arg)	minimum of elements

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

This query displays the average salary earned by employees in the toy department:

```
1) select avg(salary)
2) from employees
3) where dept = 'toy';
```

An *aggregate* can be modified by the addition of **where**, **group by**, and **having** clauses, all of which are discussed below. When the **group by** clause is used, the *aggregate* returns a result for each group.

WHERE

When an *aggregate* appears in a query, the **where** clause specifies the data to be treated by the *aggregate*. See the "Qualifications" section for a detailed description of the **where** clause.

An *aggregate* may appear inside a *qualification* only in a subquery. An *aggregate* in a subquery is illustrated below, in a query which selects the employees, other than John Smith, who make less than the average salary of all employees. The **avg** is the average of all salaries, including John Smith's:

```
1) select * from employees
2) where name <> 'smith, john'
3) and salary <
4)     (select avg(salary)
5)       from employees);
```

GROUP BY

Use the **group by** clause to apply an *aggregate* to groups of rows rather than to a table as a whole. The following query selects, for each department, the number of employees over 30 and the department name.

```
1) select count(*), dept
2) from employees
3) where age > 30
4) group by dept;
```

HAVING

The **having** clause is an optional *qualification* of the groups to be considered by the *aggregate* in a query which has a **group by** clause. For general information about *qualifications* see "Qualifications".

This query selects the department name, the average employee salary, and the number of employees in every department in which the average employee salary is higher than 2000:

```
1) select avg(salary), dept, howmany = count(*)
2) from employees
3) group by dept
3) having avg(salary) > 2000;
```

This query selects the department and average salary for the department with the highest average salary for its employees.

```
1) select dept, avg(salary)
2) from employees
3) group by dept
4) having avg(salary) >= all
5)     (select avg(salary)
6)      from employees
7)      group by dept);
```

The following query uses **group by** and **having** clauses even though there is no *aggregate* in the *target-list*. It selects the employees that belong to the department that has the highest average salary:

```
1) select name, salary, dept
2) from employees
3) group by dept
4) having avg(salary) >= all
5)     (select avg(salary)
6)      from employees
7)      group by dept);
```

A short description of each aggregate operator is given below.

any(expr)

Any returns 1 if at least one of the elements of its argument exists, nothing if none of the elements exist. The choice of the column specified in the *target-list* is irrelevant.

In order to find out if any wines in the database date from before 1986:

```
1) select old = any(winenum)
2) from wines
3) where vintage < 1986;
```

old
1

avg(arg), **avg**(**distinct** arg)

Avg returns the average of all elements of its argument. All of the elements being averaged must be of type **integer**, **bcd**, or **bcdflt**. **Avg distinct** returns the average of all of the distinct 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:

```
1) select winenum, onhand
2) from wines
3) where type = 'zinfandel'
4) and onhand <
5)       (select avg(onhand) from wines);
```

winenum	onhand
4	1
38	3

count(arg), **count(distinct arg)**

Count returns the number of rows in which its argument occurs. **Count distinct** returns the number of rows in which its argument occurs, excluding duplicate occurences of the element(s) being counted. For the **count** *aggregate* (but not **the count distinct**), the choice of the column specified in **the** ***target-list*** is irrelevant.

The **count** *aggregate* may be called with the asterisk (*) as the argument to count all the rows in the table. If multiple tables are being accessed in the query, the asterisk may be modified by a *table_name* or a *table_label*. The following queries both return the count of the rows in "stores":

```
1) select count(*) from stores;
```

```
1) select count(st.*) from stores st;
```

Count (distinct *) is not permitted. Distinct rows could be counted by using the **select** command, with the **distinct** modifier, to display the table followed by a count of the rows.

This query counts all of the rows in which the "vintage" field has a value of 1980:

```
1) select Vin80 = count(type)
2) from wines
3) where vintage = 1980;
```

Vin80
15

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

```
1) select Vin80 = count (distinct type)
2) from wines
3) where vintage = 1980;
```

Vin80
9

The **count** *aggregate* cannot count empty sets. **The** following query returns no rows at all, since there are **no wines** in the

database of vintage 1999.

```
1) select Vin99 = count(winenum)
2) from wines
3) where vintage = 1999;
```

Vin99

max(arg)

Max 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) select winenum, type, onhand from wines
2) where onhand =
3)       (select max(onhand) from wines);
```

winenum	type	onhand
28	chardonnay	23

min(arg)

Min 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 from the "pricings" relation, which gives the prices for all the wines in the database:

```
1) select p.winenum, p.price, w.type
2) from pricings p, wines w
3) where p.winenum = w.winenum and
4) p.price =
5)       (select min(price) from pricings);
```

winenum	price	type
4	4.	zinfandel

once(arg), **once(distinct** arg)

Once returns one value if one occurence of its argument exists. Otherwise it generates an error message. **Once distinct** returns one value for one occurence of a distinct element.

```
1) select Cab78 = once(winenum)
2) from wines
3) where vintage < 1978 and
4) type = 'cabernet sauvignon';
```

Cab78

ERROR line 2: ONCE or ONCE DISTINCT returned two values.

```
1) select NapCab78 = once(winenum)
2) from wines
3) where vintage < 1978 and
4) type = 'cabernet sauvignon' and
5) area = 'napa valley';
```

NapCab78
34

sum(arg), **sum(distinct** arg)

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

```
1) select sum(onhand) from wines;
```

sum (onhand)
190

SEE ALSO **audit**, **create database**, **create view**, **insert**, **select**, **update**
"Functions", "Qualifications"

Constants

A *constant* is a value that remains unchanged. *Constants* are used in expressions and as arguments to the **start** 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
43
0o777
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 database server'
```

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 database server"
```

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
#0.

Parameter Constant

A parameter constant is a name preceded by an ampersand "&". Such a constant can only be used inside a **store** command. The parameter constant is replaced by a value named in a **start** command. The parameter constant is similar to an argument to a subroutine. 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 (4-byte float) or "0d" for FLT8 (8-byte float). It may not begin with a decimal point.

24.4
-3e100
0f6.0211
0.333

The magnitude and precision of a floating constant are system dependent.

Binary Constant

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

0bA6
0be0a6ff

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 RSF and RSC, and hardly ever used in interactive SQL. They are used to substitute the value of a programming language variable into an SQL command.

SEE ALSO "Expressions", "Types"

Expressions

An *expression* yields a value or set of values. *Expressions* such as "43" or "a * b / c" yield a single value, while the *expression* "r.name" yields a set of values, one for each row in the table referenced by the *table_label* "r". The set may contain no values at all.

An *expression* may be any of the following:

aggregate	
column_name	
table_name.column_name	
table_label.column_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 SQL. Addition, subtraction, multiplication, division and negation may be used with integer, bcd and bcdflt types. Multiplication and division have precedence over addition and subtraction, for example:

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

Precedence may be forced by the use of parentheses.

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

The type of the result of an *expression* involving more than one operand can be found on the graph on the following page:

Type of Other Operand	Type of One Operand: i1	i2	i4	bcd31	bcdflt
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) select (a = #1./7);
```

returns

a
.142857142857142857142857142857

SQL prints a warning when a number has been rounded and some precision lost.

SEE ALSO

create table, **select**
"Aggregates", "Functions", "Types"

Functions

SUMMARY OF FUNCTIONS ON THE DATABASE MACHINE		
Category	*Function*	*Return Value*
arithmetic	**abs**(n)	absolute value
	mod(n,d)	remainder of n/d
string	**concat**(a,b)	concatenation of a and b
	substring(p,l,s)	substring of s
conversion	**integer**(n)	n, converted to integer
	smallint(n)	n, converted to smallint
	tinyint(n)	n, converted to tinyint
	binary(n)	n, converted to binary
	[**fixed**] **bcd**(l,n)	n, converted to bcd
	[**fixed**] **bcdflt**(l,n)	n, converted to bcdflt
	[**fixed**] **char**(l,n)	n, converted to char
	bcdfixed(p,f,n)	n, converted to bcdflt (rounded)
	float(n)	n, converted to float
	double(n)	same as float(n)
	smallfloat(n)	n, converted to smallfloat
	real(n)	same as smallfloat(n)
	string(n)	n, converted to string
database server	**userid**	current user-id
	dba	user-id of database DBA
	host	id of host computer
	gettime	database server time (integer)
	getdate	database server date (integer)
	databasename	name of open database
	table_name(r)	name of table
	table_id(s)	id of table
	col_name(r,a)	name of column

FUNCTION DESCRIPTIONS

userid
Returns the database server user-id for the current user, as a value of type smallint.

dba
Returns the database server user-id for the database administrator for the currently open database, as a value of type smallint.

host
Returns the database server host-id originating the current command, as a value of type smallint.

gettime
Returns the number of 60ths of a second since midnight. The value will initially be wrong after the database server has been brought on line. The time can be set with the utility program IDMDATE.

getdate
Returns the number of days from an initial value. This value can be initialized 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.

databasename
Returns the name of the currently open database.

table_id (name)
Expects a quoted character string (for *name*) as its argument. The return value is the id number of the object.

table_name (id)
Expects an id number of an object and returns its name.

abs (num)
Returns the absolute value of its argument.

[**fixed**] **binary** (arg)
Converts *arg* to binary type. The *arg* can be of any type. Does not change the value of *arg* in any way.

integer (arg)
Converts its argument to a four-byte integer.

smallint (arg)
Converts its argument to a two-byte integer.

tinyint (arg)
Converts its argument to a one-byte integer.

float (arg)
Converts a four-byte floating point number to an eight-byte floating point number.

smallfloat (arg)
Converts an eight-byte floating point number to a four-byte floating point number.

[**fixed**] **bcd** (precision,expression)
Converts *expression* to a bcd value with the specified *precision*. For example

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

returns

x
123

and

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

returns

x
1234

The query

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

generates the error message "Numeric overflow".

[**fixed**] **bcdflt** (precision,expression)

Converts *expression* to a bcdflt value with the specified *precision*. The query

```
1) select x = bcdflt(4, "123.45");
```

returns

x
123.40

and

```
1) select x = bcdflt(5, "1234567.89");
```

returns

x
1234600

[**fixed**] **char** (len,expression)

Converts *expression* to a char variable of length *len*. If *len* is zero, the length of the result will depend on the data type of *expression*. If **fixed** is specified, the result is blank-padded to the given length. Otherwise *len* is simply the maximum allowable length of the result.

string (len,expression)

Converts *expression* to a character string of length *len*. The expression can be of any type except float. If *len* is zero, a length is used based on the type of the *expression* as indicated below.

Type of Expression	*Length of Result (in bytes)*
i1	4
i2	6
i4	11
bcd(n)	2n - 3
bcdflt(n)	2n - 3
char(n)	n
bin(n)	n

mod (expr1,expr2)

Returns the remainder when the *expr1* is divided by *expr2*. The **mod** function can only be used on integer or bcd *expressions*.

concat (str1,str2)
Takes the character strings ***str1*** and ***str2***, strips all trailing blanks from both strings, and appends ***str2*** to ***str1***. Also works for binary data, stripping zero-bytes instead of blanks.

col_name (table_id,col_id)
Returns the name of a column when given an id number for the table and for the column.

substring (pos,len,str)
Extracts a string from a character or binary string *expression*. The result is a character or binary string of length *len*, containing the characters or bytes of *str* starting from the position *pos*. For instance,

```
substring(3, 4,'abcdefghi')
```

returns

```
cdef
```

bcdfixed (precision,fraction,expr)
Converts *expr* to a bcdfit number with a maximum of *precision* digits and a maximum of *fraction* significant fractional digits, rounding the value if necessary. For example,

```
1) select x = bcdfixed(5, 2, "768.534");
```

returns

x
768.53

and

```
1) select x = bcdfixed(8, 2, "35.478");
```

returns

x
35.48

The query

```
1) select x = bcdfixed(4, 3, "123.45");
```

generates the error message "Numeric overflow".

SEE ALSO **audit**, **create view**, **insert**, **select**, **update**
"Qualifications", "Types"

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 "Object_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 table "relation". There are seven types of objects:

U - user table
S - system table
T - transaction log
C - stored command
P - stored program
V - view
F - file

The syntax of an *object_name* is

[<owner>.] <name>

where *name* is the name of the object and *owner* is the name of its owner, as stored in the system table "users". If *owner* is not specified, the default is the current user. If no object belonging to the current user is found, the default is an object owned by the DBA.

A *table_name* and a *view_name* are subsets of *object_name*.

SEE ALSO

comment, **drop**, **drop index**, **grant**
"Name", "Object_Spec"

Object_Spec

An *object_spec* specifies the object of a **from** clause in an **audit**, **select**, or **update** command.

It has the following syntax:

object_name [table_label] [options_list]

The *object_name* may refer to a table, view, or transaction log. Refer to "Object_Name" for the syntax of an *object_name*.

A *table_label* is used to specify a particular table when multiple tables are referenced in a query and a column reference is ambiguous, to join a table with itself, or to express a correlated subquery. The use of *table_labels* is demonstrated under the **select** command, in the discussion of correlated subqueries under "Qualifications" and under "Target-Lists".

Each object named in the **from** clause may be followed by a comma-separated list of special processing options enclosed in parentheses. The available options are:

minlock
: This specifies minimum locking, in which data in the table may be accessed while the table is being accessed by another user. This may result in the access of some rows 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 queries were the option not used.

fulllock
: This option specifies a full locking. It guarantees that any data accessed 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 table 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" column of the "indices" table 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 tables), it is usually preferable to let the database server choose the index.

dorder = n
: This option is used to specify a plan for the **order** in which tables should be processed when two or more tables are joined in a *qualification*. When the **dorder** option is omitted, the

database server decides in which order to process the tables. Unless the join is extremely complicated (involves four or more tables), it is usually preferable to let the database server choose the order.

The following query permits the user to select data from the "parts" table while other users may be updating it.

```
1) select partnum, partname, quan
2) from parts (minlock)
3) where quan > 10;
```

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

```
1) select desc, name, quan
2) from small (dindex = 0, dorder = 1),
3) medium (dindex = 0, dorder = 2),
4) large (dindex = 4, dorder = 3)
5)          where small.pos < 10
6)          and medium.num = small.num
7)          and medium.type = large.type;
```

This means:

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

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

(3) Among those rows selected above (in which "small.pos" is less than 10 and "small.num" equals "medium.num") go through "large" looking for matches between "medium.type" and "large.type". Access "large" through its fourth **nonclustered index** which is on "type".

SEE ALSO **audit**, **select**, **update**
"Object_Name"

Protect_Modes

A *protect_mode* represents the type of access which can be granted to or revoked from a *user* for a particular object. Some *protect_modes* are applicable to tables, views, files, and columns, others to stored commands and stored programs, and others to databases.

A *protect_mode* is granted or revoked using its name, such as **read** or **create**, but it is identified in the "access" column of the system table "protect" by a numeric value.

The following table maps the names and numeric values of each *protect_mode*. The numbers in the *SQL* column of the table are the results of **sql**'s conversion of database server values to signed 1-byte integers. These are the values displayed in a **select** on the "protect" table.

Mode	*Octal*	*Hex*	*SQL*	*Applies To*
read	0001	0x01	1	tables, views, files, columns
write	0002	0x02	2	tables, views, files, columns
all [privileges]	0003	0x03	3	tables, views, files, columns
start	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

grant, **revoke**
"Users"

Qualifications

The conditional selection of data in the **select**, **update**, **create view**, **insert**, **delete**, and **audit** commands is controlled by an optional clause in one of the following formats:

where *qualification*
or
having *qualification*

The **having** form is used only for a *qualification* that selects groups of rows, defined by a **group by** clause, which are to be considered for calculation by an *aggregate*.

SIMPLE CONDITIONS

A *qualification* is one or more boolean conditions consisting of comparisons between terms which evaluate to true or false. The terms may be made up of column names, constants, arithmetic expressions, functions, and nested *select_statements*. The following query contains a simple *qualification*:

```
1) select * from parts
2) where pnum = 132;
```

This displays every row in the "parts" table where the value of the "pnum" field is 132. "pnum = 132" is the *qualification*.

The allowable relational operators in a *qualification* are

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

Any of these relational operators can be modified by an asterisk (*), which is the outer join operator. The meaning of this operator is described below in the sub-section labeled "JOINS".

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.

MULTIPLE CONDITIONS

Multiple conditions may be linked with the keywords **and** and **or**.

The following query displays the rows from the "wines" table for 1980 pinot noirs.

```
1) select * from wines
2) where type = 'pinot noir'
3) and vintage = 1980;
```

The following query selects all cabernet sauvignons of vintages 1980 and 1981. Parentheses may be used to group conditions. The **and** operator has a higher precedence than the **or** operator, so the parentheses are necessary here.

```
1) select * from wines
2) where (vintage = 1980 or vintage = 1981)
3) and type = 'cabernet sauvignon';
```

SPECIAL OPERATORS

The keyword **not** can be used to negate any condition. This query selects all wines in the "wines" table except the 1980 merlots.

```
1) select * from wines
2) where not (type = 'merlot'
3) and vintage = 1980);
```

The special operator **between** can be used to determine whether a value falls within a given range.

```
1) select * from wines
2) where vintage between 1980 and 1985;
```

An equivalent query would be:

```
1) select * from wines
2) where vintage >= 1980
3) and vintage <= 1985;
```

Another special operator, **in**, determines whether a value appears in a given set of literal values:

```
1) select * from wines
2) where type in ('merlot', 'zinfandel', 'pinot noir');
```

An equivalent query is:

```
1) select * from wines
2) where type = 'merlot'
3) or type = 'zinfandel'
4) or type = 'pinot noir';
```

JOINS

A join usually exists when more than one table is referenced in a condition, although there are cases involving self-joins of columns within a single table. The discussion of pink tub-and-sink in the "select" entry illustrates a use of a self-join.

The *qualification* in the following query represents a simple join of the "redwines" and "redquans" tables.

```
1) select name, redwines.num, quan
2) from redwines, redquans
3) where redwines.num = redquans.num;
```

In this query, data from the "redwines" and "redquans" tables is selected only from those rows in which the "num" column in "redwines" equals the "num" column in "redquans". If the "redwines" table consists of:

redwines table	
num	name
1	zinfandel
2	merlot
3	cabernet
4	pinot noir

and the "redquans" table consists of

redquans table	
num	quan
1	50
2	70
5	35
6	60

the query selects only

name	num	quan
zinfandel	1	50
merlot	2	70

A one-way outer join requests all the specified data from one table, regardless of the whether the condition joining the other table is true. Non-matching data from the other table 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 same side of the relational operator as the table from which all specified data is to be reported. Thus the query

```
1) select name, redwines.num, quan
2) from redwines, redquans
3) where redwines.num *= redquans.num;
```

selects all of the specified data from "redwines" and only the matching data from "redquans":

name	num	quan
zinfandel	1	50
merlot	2	70
cabernet	3	0
pinot noir	4	0

while

```
1) select name, redquans.num, quan
2) from redquans, redwines
3) where redwines.num =* redquans.num;
```

selects all of the specified data from "redquans" and only the matching data from "redwines":

name	num	quan
zinfandel	1	50
merlot	2	70
	5	35
	6	60

The database server does not support two-way outer joins.

PATTERNS

A *pattern* is a special class of character constant which can be used in a *qualification* involving string comparisons. A *pattern* differs from a regular character constant in that it contains special characters, called meta-characters, which match characters other than themselves. Trailing blanks in fixed character fields are not considered characters which can be matched.

The meta-characters are

_	Matches any single character.
%	Matches zero or more characters.

The **like** operator is used to express a boolean condition in which one of the expressions being compared is a ***pattern***. The syntax for a *qualification* containing a *pattern* is

column_name [**not**] **like** pattern [**escape** escape_character]

The optional escape clause is for specifying an ***escape_character*** which can be used in front of a meta-character to indicate that the meta-character should be interpreted literally. The *escape_character* must be a single character and may not be a backslash.

Both the *pattern* and the *escape_character* must be quoted.

This query selects the salaries of all employees whose names start with "J":

```
1) select name, salary
2) from employees
3) where name like 'J%';
```

This query uses a *pattern* to select all the part names which have underbar characters in them. The exclamation point is declared as the *escape_character* and is used here to indicate that the underbar is to be interpreted literally, not as a meta-character.

```
1) select name
2) from parts
3) where name like '%!_%' escape '!';
```

This query selects the names and salaries of all employees whose names do not contain the percent character ('%'). It uses '|' as the escape chararacter:

```
1) select name, salary
2) from employees
3) where name not like '%|%%' escape '|';
```

The table on the following page is provided to suggest, through the use of examples, the kinds of results produced by various uses of the meta-characters.

This table assumes that the tilde (~) has been specified as the *escape_character*.

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%_e"	"ace" "axQe" "a#e"	"ae" "bce"
"a~%e"	"a%e"	"a~be" "abe" "ae"

The last example is not a true *pattern* because it contains no meta-characters. The character "%" is to be interpreted as a literal percent character, not as a meta-character; this is indicated by the *escape_character* preceding it. The string is really a three-character constant consisting of an "a", a percent character, and an "e".

SUBQUERIES

A *qualification* may contain a nested subquery. One value may be compared to another value returned by a nested *select_statement*. Subqueries may be nested to any depth. Subqueries must be enclosed in parentheses. Subqueries may not contain **order by** clauses.

A simple, unmodified subquery is a *select_statement* which returns a single value which can then be compared with an expression supplied by the outer query.

This query selects all the stores that sell part 10. An error message is displayed if more than one "storeid" is returned by the subquery.

```
1) select * from stores
2) where number =
3)     (select storeid
4)      from pricings
5)      where partnum = 10);
```

As in a non-nested **select** command, **all** or **distinct** may be specified in a subquery. The specification of **distinct** removes duplicate rows from the result. If neither is specified, the default is **all**. The query

```
1) select * from emps
2) where name =
3)       (select ename from dept
4)        where dname = 'toy');
```

produces an error message if more than one row satisfies the subquery, even if the values returned for "ename" are identical. If the subquery is stated with the "distinct" modifier,

```
1) select * from emps
2) where name =
3)       (select distinct ename from dept
4)        where dname = 'toy');
```

and there are duplicate rows which satisfy the subquery, only one row is returned to the outer query and no error message is produced. If, however, more than one row satisfies the subquery but those rows are not duplicates, which would be the case if there were several different "enames" associated with the toy department, the error message is displayed.

Modified subqueries may return more than one result. The keyword **in** is used for subqueries which return more than one value. The condition evaluates to true if it is true for any of the values returned by the subquery.

This query selects information for all parts used in making televisions where there are fewer than 5 of those parts on hand.

```
1) select * from parts
2) where pnum in
3)    (select partnum
4)     from products
5)     where name = 'TV'
6)     and onhand < 5);
```

Another way to build a subquery which returns more than one value is to modify the relational operator with the keywords **any** or **all**. The modifier **some** is a synonym for **all**. These modifiers behave as follows:

expression > **all** (subquery)

The subquery may return more than one value. The condition is true if the *expression* is greater than every value returned by the subquery. In other words, the condition is true if the *expression* is greater than the maximum value returned by the subquery.

expression = **all** (subquery)
The condition is true if every value returned by the subquery is equal to the value of *expression*. The condition is false if the subquery returns any value that doesn't equal the value of the *expression*.

expression > **any** (subquery)
The subquery may return more than one value. The condition is true if the *expression* is greater than any of the values returned by the subquery. In other words, the condition is true if the *expression* is greater than the minimum value returned by the subquery.

expression = **any** (subquery)
This is equivalent to: *expression* **in** (*subquery*)

CORRELATED SUBQUERIES

Correlated subqueries return one value for every row considered in the outer query in situations in which the same table is referenced at different levels of the query. A correlation must be established between the inner and outer queries with a "correlation variable", which is a variable with a *table_label*. The *table_label* is defined at a different level of the subquery from the level in which the correlation variable is used.

The query below lists the employees who make more than the average salary in their departments. The entire query considers each employee, one at a time, and for each employee calculates the average salary in that employee's department through the subquery.

The *table_label* "e" is defined in the outer query to represent the "employees" table. It is then used in the correlation variable "e.dept" which correlates the outer query with the subquery.

```
1) select * from employees e
2) where salary >
3)     (select avg(salary)
4)      from employees
5)      where dept = e.dept);
```

This query describes as "overpaid" the highest-paid employee in each department. It uses the correlation variable "emp.dept" to correlate the "employees" table in the outer query with the table being accessed by the *qualification* in the subquery.

```
1) update employees emp
2) set descrip = 'overpaid'
3) where salary =
4)     (select max(salary)
5)      from employees
6)      where dept = emp.dept);
```

A correlated subquery may also be used in a **having** clause, as demonstrated below.

This query selects the department with the highest average salary. The average salary is compared to the averages of all other departments in the table for each department in the outer query. The **all** modifier allows the subquery to return more than one value for each group in the outer query.

```
1) select dept, avg(salary) from employees e
2) group by dept
3) having avg(salary) > all
4)     (select avg(salary)
5)      from employees
6)      group by dept
7)      having dept <> e.dept);
```

EXISTS

A special form of subquery predicate is the **exists** subquery. The *qualification* format is

where [**not**] **exists** (subquery)

The condition is true if one or more of the specified rows exists. If the **not** keyword is used, the condition is true if none of the specified rows exist.

The **exists** predicate is useful only when the subquery is correlated.

This query selects all salesmen for whom there are no sales records.

```
1) select * from salesmen s
2) where not exists
3)     (select *
4)      from salesrec
5)      where snum = s.empnum);
```

SEE ALSO

audit, **create view**, **delete**, **insert**, **select**, **update**
"Aggregates", "Expressions", "Functions"

Target-Lists

A *target-list* is a list of *targets* separated by commas. The *targets* can have the following forms:

expression

A *target* may be an *expression*:

```
1) select 2345 * 976 / 24;
```

or

```
1) select salary + integer(#0.10 * salary)
2) from employees;
```

domain_name = expression

The name and value of the domains to be selected can be explicitly stated as in

```
1) select zinfs = wines.winenum
2) where wines.type = 'zinfandel';
```

The name "zinfs" will be the title of the display of all of the values described by the qualified *expression* "wines.winenum where wines.type = 'zinfandel'".

The *expression* in a *target* can take any of the forms described under "Expressions", but usually a *target* references a database object, such as a column. Below are some examples of *targets* which specify database objects:

column_name

When *targets* are specified by *column_name* alone, the *target-list* must be followed by a **from** clause which indicates the tables from which the columns are to be accessed:

```
1) select name, number, qty
2) from items;
```

If multiple tables are being accessed using this method, the *column_names* in the *target-list* must be unique. For example, the query

```
1) select name, number, qty, cost
2) from items, prices;
```

will work only if "name", "number", "qty" and "cost" each appear in only one of the tables "items" or "prices", but not both. If *column_names* in the *target-list* are not unique, it is necessary to use another format for specifiying *targets*.

table_name.column_name

The values are accessed for the columns referenced by *column_name* from the table referenced by *table_name*:

```
1) select items.number, items.name, stores.name
2) from items, stores;
```

table_label.column_name

A *table_label* is a character string which is defined to represent a table:

```
1) select pr.cost, pr.num
2) from prices pr;
```

The clause "prices pr" is the definition of the *table_label* "pr", which is defined as a label for the relation "prices". *Table_labels* are commonly used in specifying *targets* for queries qualified by correlated subqueries and self-joins. Correlated subqueries are discussed at greater length under "Qualifications", and self-joins are discussed under **select**.

*table_name.**
*table_label.**
***

* is a pseudo-domain which yields all of the columns of the referenced table.

```
1) select * from employees
2) where 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 table indicated by "y" has three domains "y.q", "y.r", and "y.s", and the command is

```
select x.a, y.*, x.b
from x, y
```

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 **audit**, **create view**, **select**
"Expressions", "Qualifications"

Types

The following data types are supported on the database server. The names given for each *type* are those which should be used in specifying the *types* of columns in the **create table** command.

For converting data of various *types* refer to the data-conversion routines described in the "Functions" section.

integer
Four-byte integer, stored in binary two's-complement format.

smallint
Two-byte integer, stored in binary two's-complement format.

tinyint
One-byte integer, stored in binary two's-complement format.

[**fixed**] **char** [(len)]
Character string. If **fixed** is specified, blank-padding takes place, and the string is always stored with the specified length. Otherwise, trailing blanks are stripped and the specified length is regarded as a maximum. The maximum length for a **char** value is 255 characters. If no length is specified, a length of 1 (one) is assumed.

[**fixed**] **binary** (len)
Binary data consists of binary strings that are stored as they are received from the host computer. If **fixed** is specified, zero-padding takes place, and the string is always stored with the specified length. Otherwise, trailing zeros are stripped and the specified length is regarded as a maximum. The maximum length for a **binary** value is 255 bytes.

[**fixed**] **bcd** (len)
Binary Coded Decimal. The length specified is the number of digits, so the real length in bytes is (*len*/2)+2. If **fixed** is not specified, trailing zeros are stripped. If a literal **bcd** is being used in a command, it must be prefaced by a score (#).

[**fixed**] **bcdflt** (len)
Binary Coded Decimal Floating-Point. The length specified is the number of digits, so the real length in bytes is (*len*/2)+2. The *len* determines the precision. Trailing zeros are stripped if **fixed** is not specified. If a literal **bcdflt** is being used in a command, it must be prefaced by a score (#); for example,

```
1) update pricings
2) set price = #2.95
3) where price = #2.50;
```

real
Four-byte floating-point number. The database server can only store and retrieve floating-point numbers; it does no floating-point arithmetic.

smallfloat
This is a synonym for **real**.

double
Eight-byte floating-point number. The database server can only store and retrieve floating-point numbers; it does no floating-point arithmetic.

float
This is a synonym for **double**.

Every column in every table in a database is listed in the system table "attribute". The "name" column in this table contains the column's name, the "type" column contains a numeric code for the *type*, and the "length" column contains its length as an unsigned number. If rows are selected from the "attribute" table 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).

The table below maps the numeric codes for the "type" column in the "attribute" table to their respective *types*.

Code	*Type*
56	integer
52	smallint
48	tinyint
47	char
45	binary
46	bcd
35	bcdflt
57	real *or* smallfloat
60	double *or* float

SEE ALSO **create table**
"Constants", "Functions"

Users

A *user* is an individual or group of individuals with access to the database server. *Users* communicate 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 table "host_users" maps the "host-id" and the "host-user-id" for each user into a single "user-id". The "users" table is the system table which 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" tables. After a new *user* has been identified in these two tables, the DBA can assign specific access rights by user name or group name through use of the **grant** and **revoke** commands.

EXAMPLE

The following commands 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) insert into users (name, gid, id)
2) values ('karen', 20, max(users.id) + 1);
1) select *
2) from users
3) where name = 'karen';
```

stat	id	gid	name	passwd
0	321	20	karen	

```
1) insert into host_users (hid, huid, uid)
2)      select 3, 301, id
3)              from users
4)              where name = 'karen';
1) select hu.*
2)      from host_users hu, users
3)      where hu.uid = users.id
4)      and users.name = 'karen';
```

s1	hid	huid	uid
0	3	301	321

SEE ALSO

grant, **revoke**

PART IV

FRONT-END COMMANDS

Introduction to Front-End Commands

The SQL query language provides a set of front-end commands which can be invoked to govern certain aspects of an SQL session.

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 which is not described here. 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 **%?** displays a list of all of the front-end commands described in this section.

SEE ALSO

Britton Lee Host Software — IBM VM/CMS: SQL Terminal User's Guide

%comment [on | off]

DESCRIPTION

%comment is used to turn the auto-comment feature **on and off**. The auto-comment feature automatically executes the **comment** command whenever a **create table**, **create view** or **store** command **is** executed. It provides this automatic documentation of database objects **unless** the **sql** program was invoked with the **–c** or **/nocomment** flag for the session in which the object was created.

The **%comment** command may be used to suspend the automatic execution of the **comment** command for the remainder of the **sql** session or until the user wishes to turn auto-comment on again. This may be desirable if the command creating the object exceeds 4000 bytes, which is too large to fit into the command buffer.

If neither **on** nor **off** is specified, **%comment** turns auto-comment on.

EXAMPLE

```
1) %comment off              /* turn auto-comment off */
2) create table mytable      /* create a table */
3) (   . . .
4)     . . .
       . . .
97) );
1) %comment on               /* turn auto-comment on again */
```

SEE ALSO

comment

%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 semicolon (;) is not recognized as the SQL command terminator. Instead, the first line of input which does not terminate with the specified 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 (;) to indicate that the lines are to be submitted to the parser.

EXAMPLE

```
1) %continuation -                    /* set continuation character to - */
1) insert into parts(name, quan) -
2) values ('washer', 24) -
3) select parts.name, parts.quan -
4) where parts.name = 'washer'        /* command ends here */
```

name	quan
washer	20

```
1) %continuation                      /* unset continuation character */
2) delete from parts
3) where quan < 1;                    /* ; reinstated as terminator */
```

%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 SQL session. This is a useful tool for changing and resubmitting a series of commands. With a *filename*, it edits the specified file. Upon return to SQL from the editor, **%edit** submits the file it has just edited as input to SQL.

The editor which is called is specified by the EDITOR parameter in the "params" file on the host system.

EXAMPLE

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) insert into parts
2) (name, quan)
3) %edit
```

This places the user in the editor editing a temporary file which looks like this:

```
insert into parts
(name, quan)
```

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 the *level* specified. The value of *level* controls the amount of detail which will be given in SQL error messages; the more elementary the *level*, the more detailed the message.

Values for *level* can be "beginner", "able", or "expert". These values can be abbreviated and are not case sensitive. Any other value will be interpreted as "beginner".

EXAMPLE

1) %experience beginner

%help

DESCRIPTION

%help lists all of the available front-end commands. **%?** is a synonym for **%help**.

EXAMPLE

```
1) %help
  HELP: Immediate Commands:
        comment -- auto-comment 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
        trace -- set internal trace flag
```

%input [filename]

DESCRIPTION

%input specifies a file from which SQL can read its input.

If a *filename* is specified, commands are read and executed until an **exit** or end-of-file is read, at which point SQL will read from standard input.

If a *filename* is not specified, the commands are read from standard input.

The input file may contain comments begun with the characters /* and terminated with the characters */. The SQL parser ignores all of the text between the /* */ pairs. The following is valid input to SQL:

```
/* this is a comment */
select name, quan from parts /* another comment */
where /* yet another comment */ quan < 4;
```

EXAMPLE

1) %input cmd.file

%redo

DESCRIPTION **%redo** resubmits the current SQL session as input to SQL.

EXAMPLE

```
1) select partnum, onhand
2) from parts;
```

partnum	onhand
1	25
2	30
3	48

3 rows affected.

```
1) %redo
```

partnum	onhand
1	25
2	30
3	48

3 rows affected.

%substitute name value

DESCRIPTION

%substitute assigns a specific *value* to *name*. Substitutions put place holders into an ITREE using the *%name* syntax in *sqlparse*. Values may later be substituted into the tree without reparsing. The ***value*** **argument** 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 a *column_name*, provided that the substitution is a character type.
- As an *object_name*, provided that the substitution is a character type.
- As the **is** part of a **comment** command.

%substitute can set character arguments to be used in pattern-matching strings, if 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) insert into %rel
5) values(%a1, %a2);

SEE ALSO

sqlparse(3I), **iesubst**(3I) in *Host Software Specification*
sqlparse, 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) in *Host Software Specification* (UNIX systems)
tf in *C Run-Time Library Reference* (other systems)

PART V

APPENDICES

SQL Reserved Words

The following words are SQL reserved words, and may not be used otherwise in SQL commands.

all	alter	and
as	audit	between
by	clustered	column
comment	commit	create
database	delete	distinct
drop	end	exists
fixed	from	grant
group	having	in
index	insert	into
is	like	nonclustered
not	off	on
open	or	order
program	reconfigure	revoke
rollback	select	set
start	store	sync
table	tape	to
trace	truncate	unique
update	values	view
where	with	

SQL Grammar

The following pages contain a formal description of the version of SQL supported by Britton Lee Host Software. The notational conventions follow those in the rest of this manual except for the following:

Curly braces are used for grouping, so

A { B | C D } E

matches A B E or A C D E.

The plus sign is used to indicate one or more of the elements in curly braces, so

{ X },+

means one or more Xs separated by commas.

The asterisk is used to indicate zero or more of the elements in curly braces, so

{ X },*

means zero or more Xs separated by commas.

```
SQL_program:
        { statement }*

statement:
        alter database dbname
                [ with option_list ]

statement:
        alter table object_name
                [ with option_list ]

statement:
        audit [ into object_name ] target_list [ from from_list ]
                [ where bool_expr ]

statement:
        comment on [ table ] object_name [ is comment_strings ]
    |   comment on column qualified_column_spec
                [ is comment_strings ]

statement:
        commit [ work ]

statement:
        create database database_name [ with option_list ]

statement:
        create [ unique ] [ clustered | nonclustered ]
                index on object_name ( column_name_list )
                [ with option_list ]
statement:
        create table object_name ( format_list )
                [ with option_list ]

statement:
        create table object_name ([partition_name]( format_list )
                [ with option_list ]
                [ ,[partition_name](format_list) [ with option_list ]] *)

statement:
        create view object_name [ ( column_name_list ) ]
                as subquery

statement:
        delete from object_declaration [ where bool_expr ]

statement:
        drop { object_name },+

statement:
        drop database { database_name },+
```

```
statement:
        drop [ unique ] [ clustered | nonclustered ]
                index on object_name ( column_name_list )

statement:
        grant protect_mode [ [ ( column_list ) ] on object_name ]
                to {user_list | public}

statement:
        insert into object_name [ ( column_name_list ) ]
                values ( expr_list )
    |   insert into object_name [ ( column_name_list ) ] subquery

statement:
        open database_name

statement:
        reconfigure

statement:
        revoke protect_mode [ [ ( column_list ) ] on object_name ]
                from {user_list | public}

statement:
        rollback [ work ]

statement:
        select [ distinct | all ] [ into object_name ] target_list
                [ from from_list ]
                [ where bool_expr ]
                [ group by expr_list ]
                [ having bool_expr ]
                [ order by order_list ]

statement:
        set { name | constant } [ on | off ]

statement:
        start query_name [ ( { value_spec },+ ) ]

statement:
        store query_name statement_list end store

statement:
        sync

statement:
        trace constant [ on | off ]

statement:
        truncate { object_name },+
```

```
statement:
        update object_declaration [ from from_list ]
                set target_list [ where bool_expr ]

aggname:
        any
    |   avg
    |   count
    |   max
    |   min
    |   once
    |   sum

bool_expr:
        ( bool_expr )
    |   not bool_expr
    |   bool_expr and bool_expr
    |   bool_expr or bool_expr
    |   expression relop expression
    |   expression relop [ all | any | some ] ( subquery )
    |   exists ( subquery )
    |   expression [ not ] in ( subquery )
    |   expression [ not ] in ( { expression },+ )
    |   expression [ not ] between expression and expression
    |   expression [ not ] like_predicate

column_name_list:
        { column_name },+

comment_strings:
        string [, string ]

constant:
        LEXCONSTANT
    |   substitution

database_name:
        name
    |   substitution

expr_list:
        { expression },+
```

```
expression:
        constant
    |   parameter
    |   qualified_column_spec
    |   column_name
    |   - expression
    |   + expression
    |   ( expression )
    |   expression + expression
    |   expression - expression
    |   expression * expression
    |   expression / expression
    |   { count | any } ( * )
    |   aggname ( [ distinct | all ] { expression },+ )
    |   [ fixed ] funcname ( { expression },+ )

format_list:
        { name [ fixed ] format_type [ ( length ) ] },+

format_type:
        integer
    |   smallint
    |   tinyint
    |   char
    |   binary
    |   bcd
    |   bcdflt
    |   float
    |   real
    |   smallfloat
    |   double

from_list:
        { object_declaration }, +
```

```
funcname:
        abs
    |   mod
    |   concat
    |   substring
    |   integer
    |   smallint
    |   tinyint
    |   binary
    |   bcd
    |   bcdflt
    |   char
    |   string
    |   bcdfixed
    |   float
    |   smallfloat
    |   userid
    |   dba
    |   host
    |   gettime
    |   getdate
    |   databasename
    |   table_name
    |   table_id
    |   col_name
    |   double
    |   real

like_predicate:
        like string [ escape char ]
    |   like parameter

name:
        LEXNAME

object_declaration:
        object_name [ ( option_list ) ]
    |   object_name object_tag [ ( option_list ) ]

object_name:
        name
    |   owner.name
    |   substitution

object_tag:
        name

option_list:
        { name [ = expression ] [ on string ] },+
```

```
order_list:
        { expression [ ascending | descending | asc | desc
                | a | d ] },+

parameter:
        { & | $ } LEXNAME

protect_mode:
        { read | write | all } [ tape ]
    |   create
    |   create { database | index }
    |   start
    |   all [ privileges ]

query_name:
        name
    |   owner.name

qualified_column_spec:
        object_name . { column_name | * }
    |   object_tag.{ column_name | * }

relop:
        = | >= | > | <= | < | !=
    | *= | =* | *< | >* | *>= | >=*
    | *< | <* | *<= | <=* | *!= | !=*

string:
        substitution
    |   " { character }* "
    |   ' { character }* '

subquery:
        select [ distinct | all ] { expression | * }
                [ from from_list ]
                [ where bool_expr ]
                [ group by expr_list ]
                [ having bool_expr ]

substitution:
        % { LEXNAME | integer }

target_list:
        { target_resattr },+

target_resattr:
        name = expression
    |   substitution = expression
    |   expression
    |   *
```

```
user_list:
        { user_name },+

value_spec:
        [ name = ] expression
    |   [ name ] like_predicate

LEXCONSTANT:
        string
    |   & name
    |   [ # | Oo | Ox | Ob ] { digit },+ [ . { digit },+
    |   [ # | Of | Od ] { digit },+ [ . | e ] { digits }, +
```

Index of Terms