Overview

See also

What is local SQL?

Local SQL is the subset of the SQL-92 specification used to access dBASE, Paradox, and FoxPro tables. On receving local SQL statements from front-end applications, the Borland Database Engine (BDE) translates the statements into BDE API functions.

The local SQL language set

The SQL statements fall into two categories: Data Manipulation Language (DML) and Data Definition Language (DDL).

DML consists of SQL statements used for retrieving, inserting, updating, and deleting table data. SELECT is a DML statement.

DDL consists of SQL statements used for creating, altering, and deleting tables, and for creating and deleting indexes. CREATE TABLE and DROP INDEX are DDL statements.

For a complete introduction to ANSI-standard SQL, see one of the many available third-party books.

Typographical conventions

See the Legend for definitions of the typographical conventions used in this help file for SQL syntax examples.

Note The information covered in this help file pertains to the 32-bit version of the BDE. Certain features discussed are not available in the 16-bit BDE, like nested SELECT queries.

{button ,AL(`bdedocs')} Other BDE online documentation

Legend

Typographical conventions

To ensure clarity in the SQL syntax examples and pseudo-code, the typographical conventions that follow are used throughout the local SQL help.

SQL statements

Each topic contains a pseudo-code prototype of an SQL statement or part of a statement that demonstrates the language element discussed. These prototype examples appear at the beginning of topics. Actual SQL statements appear within topics that demonstrate actual use of the language element discussed. Both the prototype and example statements appear in `Courier New` font.

While the local SQL language itself is case-insensitive (language elements and metadata object names), examples in this help file use the following convention to differentiate between language and metadata objects. All language elements appear in uppercase. Metadata names appear in lowercase. Correlation names appear in mixed case.

Optional elements

Language elements that are available, but that do not have to be used with an SQL statement for the statement to be valid appear in prototype syntax examples in brakets (**[** and **]**). For example, in the line below, the DISTINCT keyword is optional.

```
SELECT [DISTINCT] *
```

Syntax choices

When there is a choice between one of a number of possible syntax elements, such choices will be listed in prototype syntax examples separated by the vertical bar character (**|**). Unless also enclosed in brackets to make the group of choices optional, one of the group of choices must be used in the statement. In the prototype syntax example below, the SQL statement may include either the ASC or the DESC keyword, but not both. Because the list of choices (ASC and DESC) is enclosed in brackets, use of either keyword is optional.

```
ORDER BY column_reference [ASC | DESC]
```

Note Do not mistake the choice typographical symbol for the concatenation function, defined as two vertical bar characters together (**||**).

Conventions

See also

Certain rules apply during use of local SQL. These rules govern the naming of columns and tables, the format for DATE values, the use of boolean values, and other aspects of the local SQL language. Failure to abide by these rules when composing SQL statements will result in the SQL statements not working and errors incurred.

Table names

Column names

Date formats

Time formats

Boolean literals

Table correlation names

Column correlation names

Embedded comments

Table names

See also

ANSI-standard SQL confines each table name to a single word comprised of alphanumeric characters and the underscore symbol, "_". Local SQL, however, is enhanced to support multi-word table names.

```
SELECT *
FROM customer
```

Local SQL supports full file and path specifications in table references. Table references with path or filename extensions must be enclosed in single or double quotation marks. For example:

```
SELECT *
FROM 'parts.dbf'
```

```
SELECT *
FROM "c:\sample\parts.dbf"
```

Local SQL also supports BDE aliases in table references. For example:

```
SELECT *
FROM ":pdox:table1"
```

If you omit the file extension for a local table name, the table is assumed to be the table type specified in the BDE configuration. The default table type is specified either in the default driver setting or in the default driver type for the standard alias associated with the query.

Finally, local SQL permits table names to duplicate SQL keywords as long as those table names are enclosed in single or double quotation marks. For example:

```
SELECT passid
FROM "password"
```

Column names

See also

ANSI-standard SQL confines each column name to a single word comprised of alphanumeric characters and the underscore symbol, "_". Local SQL, however, is enhanced to support multi-word column names.

Local SQL supports Paradox multi-word column names and column names that duplicate SQL keywords as long as those column names are

- Enclosed in single or double quotation marks
- Prefaced with an SQL table name or table correlation name

For example, the following column name consists of two words:

```
SELECT E."Emp Id"
FROM employee E
```

In the next example, the column name is the same as the SQL keyword DATE:

```
SELECT datelog."date"
FROM datelog
```

Date formats

See also

Local SQL expects date literals to be in a U.S. date format, MM/DD/YY or MM/DD/YYYY. International date formats are not supported. To prevent date literals from being mistaken by the SQL parser for arithmetic calculations, enclose them in quotation marks. This keeps 1/23/1998 from being mistaken for 1 divided by 23 divided by 1998.

```
SELECT *
FROM orders
WHERE (saledate <= "1/23/1998")
```

Leading zeros for the month and day fields are optional.

If the century is not specified for the year, the BDE setting FOURDIGITYEAR controls the century. If FOURDIGITYEAR is set to FALSE and the year is specified with only two digits, years 49 and less will be prefiex with 20 and years 50 and higher with 19. If

For example, with FOURDIGITYEAR set to FALSE, the SQL statement below returns rows where the SaleDate column contains dates of “5/5/1980” or “5/5/2030”.

```
SELECT *
FROM orders
WHERE (saledate = "5/5/30") OR
  (saledate = "5/5/80")
```

To query using years outside these bounds, specify the century in the date literal.

```
SELECT *
FROM orders
WHERE (saledate = "5/5/1930") OR
  (saledate = "5/5/2080")
```

Time formats

See also

Local SQL expects time literals to be in the format hh:mm:ss AM/PM; where hh are the hours, mm the minutes, and ss the seconds. When inserting new data with a time value, the AM/PM designator is optional and is case-insensitive ("AM" is the same as "am"). The time literal must be enclosed in quotation marks.

```
INSERT INTO WorkOrder
(ID, StartTime)
VALUES ("B00120","10:30:00 PM")
```

Indicate which half of the day (morning or after noon) a time literal falls under in one of two ways. If an AM or PM marker is specified, that determines the half of the day. If no AM/PM designator is specified, the hour field is compared to 12. If the hour is less than twelve, the time is in the AM; if greater than 12, after noon. The hour field overrides an AM/PM designator. For example, the time literal "15:03:22 AM" is translated as "3:03:22 PM".

Boolean literals

The boolean literal values TRUE and FALSE may be represented with or without quotation marks.

```
SELECT *
FROM transfers
WHERE (paid = TRUE) AND NOT (incomplete = "FALSE")
```

Table correlation names

See also

Table correlation names are used to explicitly associate a column with the table from which it comes. This is especially useful when multiple columns of the same name appear in the same query, typically in multi-table queries. A table correlation name is defined by following the table reference in the FROM clause of a SELECT query with a unique identifier. This identifier, or table correlation name, can then be used to prefix a column name.

If the table name is not a quoted string, the table name is the default implicit correlation name. An explicit correlation name the same as the table name need not be specified in the FROM clause and the table name can prefix column names in other parts of the statement.

```
SELECT *
FROM customer
  LEFT OUTER JOIN orders
    ON (customer.custno = orders.custno)
```

If the table name is a quoted string, you need to do one of the following:

Prefix column names with the exact quoted string used for the table in the FROM clause.

```
SELECT *
FROM "customer.db"
  LEFT OUTER JOIN "orders.db"
    ON ("customer.db".custno = "orders.db".custno)
```

Use the full table name as a correlation name in the FROM clause (and prefix all column references with the same correlation name).

```
SELECT *
FROM "customer.db" CUSTOMER
  LEFT OUTER JOIN "orders.db" ORDERS
    ON (CUSTOMER.custno = ORDERS.custno)
```

Use a distinctive token as a correlation name in the FROM clause (and prefix all column references with the same correlation name).

```
SELECT *
FROM "customer.db" C
  LEFT OUTER JOIN "orders.db" O
    ON (C.custno = O.custno)
```

Column correlation names

See also

Use the AS keyword to assign a correlation name to a column, aggregated value, or literal. Column correlation names cannot be enclosed in quotation marks and so cannot contain embedded spaces. In the statement below, the tokens Sub and Word are column correlation names.

```
SELECT SUBSTRING(company FROM 1 FOR 1) AS sub, "Text" AS word
FROM customer
```

Embedded comments

Comments, or remarks, can be embedded in SQL statements to add clarity or explanation. Text is designated as a comment and not treated as SQL by enclosing it within the beginning **/*** and ending ***/** comment symbols. The symbols and comments need not be on the same line.

```
/*
  This is a comment
*/
SELECT SUBSTRING(company FROM 1 FOR 1) AS sub, "Text" AS word
FROM customer
```

Comments can also be embedded within an SQL statement. This is useful when debugging an SQL statement, such as removing one clause for testing.

```
SELECT company
FROM customer
/* WHERE (state = "TX") */
ORDER BY company
```

Reserved words

See also

Below is an alphabetical list of words reserved by local SQL. Avoid using these reserved words for the names of metadata objects (tables, columns, and indexes). An "Invalid use of keyword error" occurs when reserved words are used as names for metadata objects. If a metadata object must have a reserved word as it name, prevent the error by enclosing the name in quotation marks and prefixing the reference with the table name.

```
ACTIVE                  DO                      MANUAL                  ROLLBACK
ADD                     DOMAIN                  MAX                     SECOND
ALL                     DOUBLE                  MAXIMUM_SEGMENT         SEGMENT
AFTER                   DROP                    MERGE                   SELECT
ALTER                   ELSE                    MESSAGE                 SET
AND                     END                     MIN                     SHARED
ANY                     ENTRY_POINT             MINUTE                  SHADOW
AS                      ESCAPE                  MODULE_NAME             SCHEMA
ASC                     EXCEPTION               MONEY                   SINGULAR
ASCENDING               EXECUTE                 MONTH                   SIZE
AT                      EXISTS                  NAMES                   SMALLINT
AUTO                    EXIT                    NATIONAL                SNAPSHOT
AUTOINC                 EXTERNAL                NATURAL                 SOME
AVG                     EXTRACT                 NCHAR                   SORT
BASE_NAME               FILE                    NO                      SQLCODE
BEFORE                  FILTER                  NOT                     STABILITY
BEGIN                   FLOAT                   NULL                    STARTING
BETWEEN                 FOR                     NUM_LOG_BUFFERS         STARTS
BLOB                    FOREIGN                 NUMERIC                 STATISTICS
BOOLEAN                 FROM                    OF                      SUB_TYPE
BOTH                    FULL                    ON                      SUBSTRING
BY                      FUNCTION                ONLY                    SUM
BYTES                   GDSCODE                 OPTION                  SUSPEND
CACHE                   GENERATOR               OR                      TABLE
CAST                    GEN_ID                  ORDER                   THEN
CHAR                    GRANT                   OUTER                   TIME
CHARACTER               GROUP                   OUTPUT_TYPE             TIMESTAMP
CHECK                   GROUP_COMMIT_WAIT_T     OVERFLOW                TIMEZONE_HOUR
CHECK_POINT_LENGTH        IME                   PAGE_SIZE               TIMEZONE_MINUTE
COLLATE                 HAVING                  PAGE                    TO
COLUMN                  HOUR                    PAGES                   TRAILING
COMMIT                  IF                      PARAMETER               TRANSACTION
COMMITTED               IN                      PASSWORD                TRIGGER
COMPUTED                INT                     PLAN                    TRIM
CONDITIONAL             INACTIVE                POSITION                UNCOMMITTED
CONSTRAINT              INDEX                   POST_EVENT              UNION
CONTAINING              INNER                   PRECISION               UNIQUE
COUNT                   INPUT_TYPE              PROCEDURE               UPDATE
CREATE                  INSERT                  PROTECTED               UPPER
CSTRING                 INTEGER                 PRIMARY                 USER
CURRENT                 INTO                    PRIVILEGES              VALUE
CURSOR                  IS                      RAW_PARTITIONS          VALUES
DATABASE                ISOLATION               RDB$DB_KEY              VARCHAR
DATE                    JOIN                    READ                    VARIABLE
DAY                     KEY                     REAL                    VARYING
DEBUG                   LONG                    RECORD_VERSION          VIEW
DEC                     LENGTH                  REFERENCES              WAIT
DECIMAL                 LOGFILE                 RESERV                  WHEN
DECLARE                 LOWER                   RESERVING               WHERE
DEFAULT                 LEADING                 RETAIN                  WHILE
DELETE                  LEFT                    RETURNING_VALUES        WITH
DESC                    LEVEL                   RETURNS                 WORK
DESCENDING              LIKE                    REVOKE                  WRITE
DISTINCT                LOG_BUFFER_SIZE         RIGHT                   YEAR
```

The following are operators used in local SQL. Avoid using these characters in the names of metadata

objects.

```
||, -, *, /, <>, <, >, ,(comma), =, <=, >=, ~=, !=, ^=, (, )
```

Unsupported language

See also

The following SQL-92 language elements are not used in local SQL.

ALLOCATE CURSOR (Command)
ALLOCATE DESCRIPTOR (Command)
ALTER DOMAIN (Command)
CASE (Expression)
CHECK (Constraint)
CLOSE (Command)
COALESCE (Expression)
COMMIT (Command)
CONNECT (Command)
CONVERT (Function)
CORRESPONDING BY (Expression)
CREATE ASSERTION (Command)
CREATE CHARACTER SET (Command)
CREATE COLLATION (Command)
CREATE DOMAIN (Command)
CREATE SCHEMA (Command)
CREATE TRANSLATION (Command)
CREATE VIEW (Command)
CROSS JOIN (Relational operator)
CURRENT_DATE (Function)
CURRENT_TIME (Function)
CURRENT_TIMESTAMP (Function)
DEALLOCATE DESCRIPTOR (Command)
DEALLOCATE PREPARE (Command)
DECLARE CURSOR (Command)
DECLARE LOCAL TEMPORARY TABLE (Command)
DESCRIBE (Command)
DISCONNECT (Command)
DROP ASSERTION (Command)
DROP CHARACTER SET (Command)
DROP COLLATION (Command)
DROP DOMAIN (Command)
DROP SCHEMA (Command)
DROP TRANSLATION (Command)
DROP VIEW (Command)
EXCEPT (Relational operator)
EXECUTE (Command)
EXECUTE IMMEDIATE (Command)
FETCH (Command)
FOREIGN KEY (Constraint)
GET DESCRIPTOR (Command)
GET DIAGNOSTICS (Command)
GRANT (Command)
INTERSECT (Relational operator)
MATCH (Predicate)
NATURAL (Relational operator)
NULLIF (Expression)
OPEN (Command)
OVERLAPS (Predicate)
PREPARE (Command)
REFERENCES (Constraint)
REVOKE (Command)
ROLLBACK (Command)
Row value constructors
SET CATALOG (Command)
SET CONNECTION (Command)
SET CONSTRAINTS MODE (Command)
SET DESCRIPTOR (Command)
SET NAMES (Command)
SET SCHEMA (Command)
SET SESSION AUTHORIZATION (Command)
SET TIME ZONE (Command)
SET TRANSACTION (Command)
TRANSLATE (Function)
UNIQUE (Predicate)
USING (Relational operator)

Data manipulation overview

Category	Description
Statement list	Statements, or commands, that retrieve, modify, and delete data.
Clause list	Clauses of statements that affect how a statement operates.
Function list	Functions like UPPER that alter data and SUM that aggregate data.
Operator list	Arithmetic, comparison, logical, and string concatenation operators.
Predicate list	Keywords used in WHERE clauses to qualify rows returned by a statement.
Relational operators	Operators that allow joining multiple tables.
Updatable queries	Conditions under which queries are updateable or read-only.
Parameters in queries	Using parameters to make queries dynamic.

DML statement list

Local SQL supports the following data manipulation language (DML) statements:

DML Statements	Description
SELECT	Retrieves existing data from a table.
DELETE	Deletes existing data from a table.
INSERT	Adds new data to a table.
UPDATE	Modifies existing data in a table.

SELECT statement

See also

Retrieves data from tables.

```
SELECT [DISTINCT] * | column_list
FROM table_reference
[WHERE predicates]
[ORDER BY order_list]
[GROUP BY group_list]
[HAVING having_condition]
```

Description

Use the SELECT statement to

- Retrieve a single row, or part of a row, from a table, referred to as a singleton select.
- Retrieve multiple rows, or parts of rows, from a table.
- Retrieve related rows, or parts of rows, from a join of two or more tables.

The SELECT clause defines the list of items returned by the SELECT statement. The SELECT clause uses a comma-separated list composed of: table columns, literal values, and column or literal values modified by functions. Literal values in the columns list may be passed to the SELECT statement via parameters. You cannot use parameters to represent column names. Use an asterisk to retrieve values from all columns.

Columns in the column list for the SELECT clause may come from more than one table, but can only come from those tables listed in the FROM clause. See Relational Operators for more information on using the SELECT statement to retrieve data from multiple tables.

The FROM clause identifies the table(s) from which data is retrieved.

The following statement retrieves data for two columns in all rows of a table.

```
SELECT custno, company
FROM orders
```

Use DISTINCT to limit the retrieved data to only distinct rows. The distinctness of rows is based on the combination of the columns in the SELECT clause columns list. DISTINCT can only be used with simple column types like CHAR and INTEGER; it cannot be used with complex column types like BLOB and memo.

In lieu of a table, a SELECT statement may retrieve rows from a Paradox-style .QBE file. This is an approximation of an SQL view.

```
SELECT *
FROM "customers.qbe"
```

DELETE statement

See also

Deletes one or more rows from a table.

```
DELETE FROM table_reference
[WHERE predicates]
```

Description

Use DELETE to delete one or more rows from an existing table.

```
DELETE FROM "employee.db"
```

The optional WHERE clause restricts row deletions to a subset of rows in the table. If no WHERE clause is specified, all rows in the table are deleted.

```
DELETE FROM "employee.db"
WHERE (empno IN (SELECT empno FROM "old_employee.db"))
```

The table reference cannot be passed to the DELETE statement via a parameter.

INSERT statement

See also

Adds one or more new rows of data in a table

```
INSERT INTO table_reference
[(columns_list)]
VALUES (update_atoms)
```

Description

Use the INSERT statement to add new rows of data to a table.

Use a table reference in the INTO clause to specify the table to receive the incoming data.

The columns list is a comma-separated list, enclosed in parentheses, of columns in the table and is optional. The VALUES clause is a comma-separated list of update atoms, enclosed in parentheses. If no columns list is specified, incoming update values (update atoms) are stored in fields as they are defined sequentially in the table structure. Update atoms are applied to columns in the order the update atoms are listed in the VALUES clause. There must also be as many update atoms as there are columns in the table.

```
INSERT INTO "holdings.dbf"
VALUES (4094095, "BORL", 5000, 10.500, "1/2/1998")
```

If an explicit columns list is stated, incoming update atoms (in the order they appear in the VALUES clause) are stored in the listed columns (in the order they appear in the columns list). NULL values are stored in any columns that are not in a columns list.

```
INSERT INTO "customer.db"
(custno, company)
VALUES (9842, "Borland International, Inc.")
```

To add rows to one table from another, omit the VALUES keyword and use a subquery as the source for the new rows.

```
INSERT INTO "customer.db"
(custno, company)
SELECT custno, company
FROM "oldcustomer.db"
```

Update atom values may be passed to the INSERT statement via parameters. You cannot use parameters for the table reference and columns list.

Note Insertion of one or multiple rows from one table to another through a subquery is not supported.

UPDATE statement

Modifies one or more existing rows in a table.

```
UPDATE table_reference
SET column_ref = update_atom [, column_ref = update_atom...]
[WHERE predicates]
```

Description

Use the UPDATE statement to modify one or more column values in one or more existing rows in a table.

Use a table reference in the UPDATE clause to specify the table to receive the data changes.

The SET clause is a comma-separated list of update expressions. Each expression is composed of the name of a column, the assignment operator (=), and the update value (update atom) for that column. The update atoms in any one update expression may be literal values, singleton return values from a subquery, or update atoms modified by functions. Subqueries supplying an update atom for an update expression must return a singleton result set (one row) and return only a single column.

```
UPDATE salesinfo
SET taxrate = 0.0825
WHERE (state = "CA")
```

Update atom values may be passed to the UPDATE statement via parameters. You cannot use parameters for the table reference and columns list.

The optional WHERE clause restricts updates to a subset of rows in the table. If no WHERE clause is specified, all rows in the table are updated using the SET clause update expressions.

Clause list

Local SQL supports the following SQL statement clauses:

Clause	Description
FROM	Specifies the tables used for the statement.
WHERE	Specifies filter criteria to limit rows retrieved.
ORDER BY	Specifies the columns on which to sort the result set.
GROUP BY	Specifies the columns used to group rows.
HAVING	Specifies filter criteria using aggregated data.

FROM clause

See also

Specifies the tables from which a SELECT statement retrieves data.

```
FROM table_reference [, table_reference...]
```

Description

Use a FROM clause to specify the table or tables from which a SELECT statement retrieves data. The value for a FROM clause is a comma-separated list of table names. Specified table names must follow local SQL naming conventions for tables. For example, the SELECT statement below retrieves data from a single Paradox table.

```
SELECT *
FROM "customer.db"
```

See the section Relational Operators for more information on retrieving data from multiple tables in a single SELECT query.

The table reference cannot be passed to a FROM clause via a parameter.

Applicability

SELECT

WHERE clause

See also

Specifies filtering conditions for a SELECT or UPDATE statement.

```
WHERE predicates
```

Description

Use a WHERE clause to limit the effect of a SELECT or UPDATE statement to a subset of rows in the table. Use of a WHERE clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to TRUE or FALSE for each row in the table. Only those rows where the predicates evaluate to TRUE are retrieved by a SELECT statement or modified by an UPDATE statement. For example, the SELECT statement below retrieves all rows where the STATE column contains a value of "CA".

```
SELECT company, state
FROM customer
WHERE state = "CA"
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria. For example, the SELECT statement below retrieves all rows where the STATE column contains a value of "CA" and those with a value of "HI".

```
SELECT company, state
FROM customer
WHERE (state = "CA") OR (state = "HI")
```

The SELECT statement below retrieves all rows where the SHAPE column is "round" or "square", but only if the the COLOR column also contains "red". It would not retrieve rows where, for example, the SHAPE is "round" and the COLOR "blue".

```
SELECT shape, color, cost
FROM objects
WHERE ((shape = "round") OR (shape = "square")) AND
  (color = "red")
```

But without the parentheses to override the order of precedence of the logical operators, as in the statement that follows, the results are very different. This statement retrieves the rows where the SHAPE is "round", regardless of the value in the COLOR column. It also retrieves rows where the SHAPE column is "square", but only when the COLOR column contains "red". Unlike the preceding variation of this statement, this one would retrieve rows where the SHAPE is "round" and the COLOR "blue".

```
SELECT shape, color, cost
FROM objects
WHERE shape = "round" OR shape = "square" AND
  color = "red"
```

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or "parent" query.

Column references cannot be passed to a WHERE clause via parameters. Comparison values may be passed as parameters.

Note A WHERE clause filters data prior to the aggregation of a GROUP BY clause. For filtering based on aggregated values, use a HAVING clause.

Applicability

SELECT (with non-aggregated columns), UPDATE

ORDER BY clause

See also

Sorts the rows retrieved by a SELECT statement.

```
ORDER BY column_reference [, column_reference...] [ASC|DESC]
```

Description

Use an ORDER BY clause to sort the rows retrieved by a SELECT statement based on the values from one or more columns.

The value for the ORDER BY clause is a comma-separated list of column names. The columns in this list must also be in the SELECT clause of the query statement. Columns in the ORDER BY list can be from one or multiple tables. A number representing the relative position of a column in the SELECT clause may be used in place of a column name. Column correlation names can also be used in an ORDER BY clause columns list.

Use ASC (or ASCENDING) to force the sort to be in ascending order (smallest to largest), or DESC (or DESCENDING) for a descending sort order (largest to smallest). When not specified, ASC is the implied default.

The statement below sorts the result set ascending by the year extracted from the LASTINVOICEDATE column, then descending by the STATE column, and then ascending by the uppercase conversion of the COMPANY column.

```
SELECT EXTRACT(YEAR FROM lastinvoicedate) AS YY, state, UPPER(company)
FROM customer
ORDER BY YY DESC, state ASC, 3
```

See the section Relational Operators for more information on retrieving data from multiple tables in a single SELECT query.

Column references cannot be passed to an ORDER BY clause via parameters.

Applicability

SELECT

GROUP BY clause

See also

Combines rows with column values in common into single rows.

```
GROUP BY column_reference [, column reference...]
```

Description

Use a GROUP BY clause to combine rows with the same column values into a single row. The criteria for combining rows is based on the values in the columns specified in the GROUP BY clause. The purpose for using a GROUP BY clause is to combine one or more column values (aggregate) into a single value and provide one or more columns to uniquely identify the aggregated values. A GROUP BY clause can only be used when one or more columns have an aggregate function applied to them.

The value for the GROUP BY clause is a comma-separated list of columns. Each column in this list must meet the following criteria:

- Be in one of the tables specified in the FROM clause of the query.
- Be in the SELECT clause of the query.
- Cannot have an aggregate function applied to it.

When a GROUP BY clause is used, all table columns in the SELECT clause of the query must meet at least one of the following criteria, or it cannot be included in the SELECT clause:

- Be in the GROUP BY clause of the query.
- Be in the subject of an aggregate function.

Literal values in the SELECT clause are not subject to the preceding criteria.

The distinctness of rows is based on the columns in the column list specified. All rows with the same values in these columns are combined into a single row (or logical group). Columns that are the subject of an aggregate function have their values across all rows in the group combined. All columns not the subject of an aggregate function retain their value and serve to distinctly identify the group. For example, in the SELECT statement below, the values in the SALES column are aggregated (totaled) into groups based on distinct values in the COMPANY column. This produces total sales for each company.

```
SELECT company, SUM(sales) AS TOTALSALES
FROM sales1998
GROUP BY company
ORDER BY company
```

A column may be referenced in a GROUP BY clause by a column correlation name, instead of actual column names. The statement below forms groups using the first column, COMPANY, represented by the column correlation name Co.

```
SELECT company AS Co, SUM(sales) AS TOTALSALES
FROM sales1998
GROUP BY Co
ORDER BY 1
```

Note Derived values (calculated fields) cannot be used as the basis for a GROUP BY clause.

Column references cannot be passed to an GROUP BY clause via parameters.

Applicability

SELECT when aggregate functions used

HAVING clause

See also

Specifies filtering conditions for a SELECT statement.

```
HAVING predicates
```

Description

Use a HAVING clause to limit the rows retrieved by a SELECT statement to a subset of rows where aggregated column values meet the specified criteria. A HAVING clause can only be used in a SELECT statement when:

- The statement also has a GROUP BY clause.
- One or more columns are the subjects of aggregate functions.

The value for a HAVING clause is one or more logical expressions, or predicates, that evaluate to true or false for each aggregate row retrieved from the table. Only those rows where the predicates evaluate to true are retrieved by a SELECT statement. For example, the SELECT statement below retrieves all rows where the total sales for individual total sales exceed $1,000.

```
SELECT company, SUM(sales) AS TOTALSALES
FROM sales1998
GROUP BY company
HAVING (SUM(sales) >= 1000)
ORDER BY company
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

A SELECT statement can include both a WHERE clause and a HAVING clause. The WHERE clause filters the data to be aggregated, using columns not the subject of aggregate functions. The HAVING clause then further filters the data after the aggregation, using columns that are the subject of aggregate functions. The SELECT query below performs the same operation as that above, but data limited to those rows where the STATE column is "CA".

```
SELECT company, SUM(sales) AS TOTALSALES
FROM sales1998
WHERE (state = "CA")
GROUP BY company
HAVING (SUM(sales) >= 1000)
ORDER BY company
```

Subqueries are supported in the HAVING clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or "parent" query.

Note A HAVING clause filters data after the aggregation of a GROUP BY clause. For filtering based on row values prior to aggregation, use a WHERE clause.

Applicability

SELECT with GROUP BY

Function list

Local SQL supports the following data manipulation language functions:

String function	Description
Concatenation	Concatenates two string values.
LOWER	Forces a string to lowercase.
UPPER	Forces a string to uppercase.
SUBSTRING	Extracts a portion of a string value.
TRIM	Removes repetitions of a specified character from the left, right, or both sides of a string.

Aggregate function	Description
AVG	Averages all non-NULL numeric values in a column.
COUNT	Counts the number of rows in a result set.
MAX	Determines the maximum value in a column.
MIN	Determines the minimum value in a column.
SUM	Totals all numeric values in a column.

Data function	Description
CAST	Converts values from one data type to another.
EXTRACT	Extracts the year, month, or day field of a date.

Concatenation function

See also

Concatenates two character values.

```
value1 || value2
```

Description

Use the concatenation function to concatenate two character values. For example, the expression below returns the string "ABCdef".

```
"ABC" || "def"
```

The statement below uses the concatenation function to combine column values with character literals.

```
SELECT lastname || ", " || firstname
FROM names
```

Applicability

The concatenation function can only be used with character columns or literals. To use on values of other data types, the values must first be converted to CHAR using the CAST function.

Note: the concatenation function cannot be used with memo or BLOB columns.

LOWER function

See also

Converts all characters to lowercase.

```
LOWER(column_reference)
```

Description

Use LOWER to convert all of the characters in a table column or character literal to lowercase. For example, in the SELECT statement below the values in the NAME column appear all in lowercase.

```
SELECT LOWER(name)
FROM country
```

When applied to retrieved data of a SELECT statement, the effect is transient and does not affect stored data. When applied to the update atoms of an UPDATE statement, the effect is persistent and permanently converts the case of the stored values.

The LOWER function can be used in WHERE clause string comparisons to effect a case-insensitive comparison. Apply LOWER to the values on both sides of the comparison operator (if one of the comparison values is a literal, simply enter it all in lower case).

```
SELECT *
FROM names
WHERE LOWER(lastname) = "smith"
```

Applicability

LOWER can only be used with character columns or literals. To use on values of other data types, the values must first be converted to CHAR using the CAST function.

Note: the LOWER function cannot be used with memo or BLOB columns.

UPPER function

See also

Converts all characters to uppercase.

```
UPPER(column_reference)
```

Description

Use UPPER to convert all of the characters in a table column or character literal to uppercase. For example, in the SELECT statement below the values in the NAME column are treated as all in uppercase. Because the same conversion is applied to both the filter column and comparison value in the WHERE clause, the filtering is effectively case-insensitive.

```
SELECT name, capital, continent
FROM country
WHERE UPPER(name) LIKE UPPER("Pe%")
```

When applied to retrieved data of a SELECT statement, the effect is transient and does not affect stored data. When applied to the update atoms of an UPDATE statement, the effect is persistent and permanently converts the case of the stored values.

Applicability

UPPER can only be used with character columns or literals. To use on values of other data types, the values must first be converted to CHAR using the CAST function.

Note: the UPPER function cannot be used with memo or BLOB columns.

SUBSTRING function

See also

Extracts a substring from a string.

```
SUBSTRING(column_reference FROM start_index [FOR length])
```

Description

Use SUBSTRING to extract a substring from a table column or character literal, specified in the column reference.

FROM is the character position at which the extracted substring starts within the original string. The index for FROM is based on the first character in the source value being 1.

FOR is optional, and specifies the length of the extracted substring. If FOR is omitted, the substring goes from the position specified by FROM to the end of the string.

The example below, applied to the literal string "ABCDE" returns the value "BCD".

```
SELECT SUBSTRING("ABCDE" FROM 2 FOR 3) AS Sub
FROM country
```

In the SELECT statement below only the second and subsequent characters of the NAME column are retrieved.

```
SELECT SUBSTRING(name FROM 2)
FROM country
```

When applied to retrieved data of a SELECT statement, the effect is transient and does not affect stored data. When applied to the update atoms of an UPDATE statement, the effect is persistent and permanently converts the case of the stored values.

Applicability

SUBSTRING can only be used with character columns or literals. To use on values of other data types, the values must first be converted to CHAR using the CAST function.

Note: the SUBSTRING function cannot be used with memo or BLOB columns.

TRIM function

See also

Removes the trailing or leading character, or both, from a string.

```
TRIM([LEADING|TRAILING|BOTH] [trimmed_char] FROM column_reference)
```

Description

Use TRIM to delete the leading or trailing character, or both, from a table column or character literal. The TRIM function only deletes characters located in the specified position.

The first parameter indicates the position of the character to be deleted, and has one of the following values:

Value	Description
LEADING	Deletes the character at the left end of the string.
TRAILING	Deletes the character at the right end of the string.
BOTH	Deletes the character at both ends of the string.

The trimmed character parameter specifies the character to be deleted, if present. Case-sensitivity is applied for this parameter. To make TRIM case-insensitive, use the UPPER function.

FROM specifies the column or character literal from which to delete the character. The column reference for FROM can be a table column or a character literal.

Example variations:

TRIM syntax	Result
TRIM(LEADING "_" FROM "_ABC_")	"ABC_"
TRIM(TRAILING "_" FROM "_ABC_")	"_ABC"
TRIM(BOTH "_" FROM "_ABC_")	"ABC"
TRIM(BOTH "A" FROM "ABC")	"BC"

When applied to retrieved data of a SELECT statement, the effect is transient and does not affect stored data. When applied to the update atoms of an UPDATE statement, the effect is persistent and permanently converts the case of the stored values.

Applicability

TRIM can only be used with character columns or literals. To use on values of other data types, the values must first be converted to CHAR using the CAST function.

Note: the TRIM function cannot be used with memo or BLOB columns.

AVG function

See also

Returns the average of the values in a specified column or an expression.

```
AVG([ALL] column_reference | DISTINCT column_reference)
```

Description

Use AVG to calculate the average value for a numeric column. As an aggregate function, AVG performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the averaging, so values of 1, 2, 3, 0, 0, and 0 result in an average of 1. NULL column values are not counted in the calculation.

```
SELECT AVG(itemstotal)
FROM orders
```

ALL returns the average for all rows. When DISTINCT is not specified, ALL is the implied default.

DISTINCT ignores duplicate values when averaging values in the specified column.

AVG returns the average of values in a column or the average of a calculation using a column performed for each row (a calculated field).

```
SELECT AVG(itemstotal), AVG(itemstotal * 0.0825) AS AverageTax
FROM orders
```

When used with a GROUP BY clause, AVG calculates one value for each group. This value is the aggregation of the specified column for all rows in each group. The statement below aggregates the average value for the order totals column in the ORDERS table, producing a subtotal for each company in the COMPANY table.

```
SELECT C."company", AVG(O."itemstotal") AS Average,
  MAX(O."itemstotal") AS Biggest,
  MIN(O."itemstotal") AS Smallest
FROM "customer.db" C, "orders.db" O
WHERE (C."custno" = O."custno")
GROUP BY C."company"
ORDER BY C."company"
```

Applicability

AVG operates only on numeric values. To use AVG on non-numeric values, first use the CAST function to convert the column to a numeric type.

Note: the MAX function cannot be used with memo or BLOB columns.

COUNT function

See also

Returns the number of rows that satisfy a query's search condition.

```
COUNT(* | [ALL] column_reference | DISTINCT column_reference)
```

Description

Use COUNT to count the number of rows retrieved by a SELECT statement. The SELECT statement may be a single- or multi-table query. The value returned by COUNT reflects a reduced row count produced by a filtered dataset.

```
SELECT COUNT(amount)
FROM averaging
```

ALL returns the count for all rows. When DISTINCT is not specified, ALL is the implied default.

DISTINCT ignores duplicate values in the specified column when counting rows.

MAX function

See also

Returns the largest value in the specified column.

```
MAX([ALL] column_reference | DISTINCT column_reference)
```

Description

Use MAX to calculate the largest value for a numeric column. As an aggregate function, MAX performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, MAX returns a NULL value.

```
SELECT MAX(itemstotal)
FROM orders
```

ALL returns the largest value for all rows. When DISTINCT is not specified, ALL is the implied default.

DISTINCT ignores duplicate values when calculating the largest value in the specified column.

MAX returns the largest value in a column or a calculation using a column performed for each row (a calculated field).

```
SELECT MAX(itemstotal), MAX(itemstotal * 0.0825) AS HighestTax
FROM orders
```

When used with a GROUP BY clause, MAX returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The statement below aggregates the largest value for the order totals column in the ORDERS table, producing a subtotal for each company in the COMPANY table.

```
SELECT C."company", AVG(O."itemstotal") AS Average,
  MAX(O."itemstotal") AS Biggest,
  MIN(O."itemstotal") AS Smallest
FROM "customer.db" C, "orders.db" O
WHERE (C."custno" = O."custno")
GROUP BY C."company"
ORDER BY C."company"
```

Applicability

MAX can be used with all non-BLOB columns. When used with numeric columns, the return value is of the same type as the column (such as INTEGER or FLOAT). When used with a CHAR column, the largest value returned will depend on the Borland Database Engine (BDE) language driver used.

Note: the MAX function cannot be used with memo or BLOB columns.

MIN function

See also

Returns the smallest value in the specified column.

```
MIN([ALL] column_reference | DISTINCT column_reference)
```

Description

Use MIN to calculate the smallest value for a numeric column. As an aggregate function, MIN performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, MIN returns a NULL value.

```
SELECT MIN(itemstotal)
FROM orders
```

ALL returns the smallest value for all rows. When DISTINCT is not specified, ALL is the implied default.

DISTINCT ignores duplicate values when calculating the smallest value in the specified column.

MIN returns the smallest value in a column or a calculation using a column performed for each row (a calculated field).

```
SELECT MIN(itemstotal), MIN(itemstotal * 0.0825) AS LowestTax
FROM orders
```

When used with a GROUP BY clause, MIN returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The statement below aggregates the smallest value for the order totals column in the ORDERS table, producing a subtotal for each company in the COMPANY table.

```
SELECT C."company", AVG(O."itemstotal") AS Average,
  MAX(O."itemstotal") AS Biggest,
  MIN(O."itemstotal") AS Smallest
FROM "customer.db" C, "orders.db" O
WHERE (C."custno" = O."custno")
GROUP BY C."company"
ORDER BY C."company"
```

Applicability

MIN can be used with all non-BLOB columns. When used with numeric columns, the return value is of the same type as the column (such as INTEGER or FLOAT). When used with a CHAR column, the smallest value returned will depend on the Borland Database Engine (BDE) language driver used.

Note: the MIN function cannot be used with memo or BLOB columns.

SUM function

See also

Calculates the sum of values for a column.

```
SUM([ALL] column_reference | DISTINCT column_reference)
```

Description

Use SUM to sum all the values in the specified column. As an aggregate function, SUM performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, SUM returns a NULL value.

```
SELECT SUM(itemstotal)
FROM orders
```

ALL returns the smallest value for all rows. When DISTINCT is not specified, ALL is the implied default.

DISTINCT ignores duplicate values when calculating the smallest value in the specified column.

MIN returns the smallest value in a column or a calculation using a column performed for each row (a calculated field).

```
SELECT SUM(itemstotal), SUM(itemstotal * 0.0825) AS TotalTax
FROM orders
```

When used with a GROUP BY clause, SUM returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The statement below aggregates the total value for the order totals column in the ORDERS table, producing a subtotal for each company in the COMPANY table.

```
SELECT C."company", SUM(O."itemstotal") AS SubTotal
FROM "customer.db" C, "orders.db" O
WHERE (C."custno" = O."custno")
GROUP BY C."company"
ORDER BY C."company"
```

Applicability

SUM operates only on numeric values. To use SUM on non-numeric values, first use the CAST function to convert the column to a numeric type.

CAST function

See also

Converts specified value to the specified data type.

```
CAST(column_reference AS data_type)
```

Description

Use CAST to convert the value in the specified column to the data type specified. CAST can also be applied to literal and calculated values. CAST can be used in the columns list of a SELECT statement, in the predicate for a WHERE clause, or to modify the update atom of an UPDATE statement.

The Data_Type parameter may be one of most column data type applicable to the table type used: CHAR, INTEGER, NUMERIC, and so on. Certain column types cannot be used as the source or target data types: BLOB, MEMO, and BYTES.

The statement below converts a Paradox DATETIME column value to DATE.

```
SELECT CAST(saledate AS DATE)
FROM ORDERS
```

Converting a column value with CAST allows use of other functions or predicates on an otherwise incompatible data type, such as using the SUBSTRING function on a DATE column.

```
SELECT saledate,
  SUBSTRING(CAST(CAST(saledate AS DATE) AS CHAR(10)) FROM 1 FOR 1)
FROM orders
```

When applied to retrieved data of a SELECT statement, the effect is transient and does not affect stored data. When applied to the update atoms of an UPDATE statement, the effect is persistent and permanently converts the case of the stored values.

Note: the CAST function cannot be used with memo or BLOB columns.

EXTRACT function

See also

Returns one field from a date value.

```
EXTRACT(extract_field FROM column_reference)
```

Description

Use EXTRACT to return the year, month, or day field from a DATE or TIMESTAMP column. If the column used with the EXTRACT function contains a NULL value, the return value of EXTRACT will be NULL. If the value is not NULL, EXTRACT returns the value for the specified element in the date, expressed as a SMALLINT.

The Extract_Field parameter may contain any one of the specifiers: YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND. The specifiers YEAR, MONTH, and DAY can only be used with DATE and TIMESTAMP columns. The specifiers HOUR, MINUTE, and SECOND can only be used with TIMESTAMP and TIME columns.

```
SELECT saledate,
  EXTRACT(YEAR FROM saledate) AS YY,
  EXTRACT(MONTH FROM saledate) AS MM,
  EXTRACT(DAY FROM saledate) AS DD
FROM orders
```

The statement below uses a DOB column (containing birthdates) to filter to those rows where the date is in the month of May. The month field from the DOB column is retrieved using the EXTRACT function and compared to 5, May being the fifth month.

```
SELECT DOB, LastName, FirstName
FROM People
WHERE (EXTRACT(MONTH FROM DOB) = 5)
```

Applicability

EXTRACT operates only on DATE, TIME, and TIMESTAMP values. To use EXTRACT on non-date values, first use the CAST function to convert the column to a date type.

Note: while SQL-92 provides the EXTRACT function specifiers TIMEZONE_HOUR and TIMEZONE_MINUTE, these specifiers are not supported in local SQL.

Operators list

Local SQL supports the following operators:

Type	Operators
Arithmetic	`+ - * /`
Logical	`AND OR NOT`

Arithmetic operators

See also

Perform arithmetic operations.

```
numeric_value1 + numeric_value2
numeric_value1 - numeric_value2
numeric_value1 * numeric_value2
numeric_value1 / numeric_value2
```

Description

Use arithmetic operators to perform arithmetic calculations on data in SELECT queries. Calculations can be performed wherever non-aggregated data values are allowed, such as in a SELECT or WHERE clause. In the statement below, a column value is multiplied by a numeric literal.

```
SELECT (itemstotal * 0.0825) AS Tax
FROM orders
```

Arithmetic calculations are performed in the normal order of precedence: multiplication, division, addition, and then subtraction. To cause a calculation to be performed out of the normal order of precedence, use parentheses around the operation to be performed first. In the statement below the addition is performed before the multiplication.

```
SELECT (n.numbers * (n.multiple + 1)) AS Result
FROM numbertable n
```

Applicability

Arithmetic operators operate only on numeric values. To use arithmetic operators on non-numeric values, first use the CAST function to convert the column to a numeric type.

Logical operators

See also

Connect multiple predicates.

```
[NOT] predicate OR [NOT] predicate
[NOT] predicate AND [NOT] predicate
```

Description

Use the logical operators OR and AND to connect two predicates in a single WHERE clause. This allows the table to be filtered based on multiple conditions. Logical operators compare the boolean result of two predicate comparisons, each producing a boolean result. If OR is used, **either** of the two predicate comparisons can result on a TRUE value for the whole expression to evaluate to TRUE. If AND is used, **both** predicate comparisons must evaluate to TRUE for the whole expression to be TRUE; if **either** is FALSE, the whole is FALSE. In the statement below, if only one of the two predicate comparisons is TRUE (`reservdate < "1/31/1998"` or `paid = TRUE`), the row will be included in the query result set.

```
SELECT *
FROM reservations
WHERE ((reservdate < "1/31/1998") OR (paid = TRUE))
```

Logical operator comparisons are performed in the order of precedence: OR and then AND. To perform a comparison out of the normal order of precedence, use parentheses around the comparison to be performed first. The SELECT statement below retrieves all rows where the SHAPE column is "round" and the COLOR "blue". It also returns those rows where the COLOR column is "red", regardless of the value in the SHAPE column (such as "triangle"). It would not return rows where the SHAPE is "round" and the COLOR anything but "blue" or where the COLOR is "blue" and the SHAPE anything but "round".

```
SELECT shape, color, cost
FROM objects
WHERE ((shape = "round") AND (color = "blue")) OR
  (color = "red")
```

Without the parentheses, the default order of precedence is used and the logic changes. The statement below, a variation on the above statement, would return rows where the SHAPE is "square" and the COLOR is "blue". It would also return rows where the SHAPE is "square" and the COLOR is "red". But unlike the preceding statement, it would not return rows where the COLOR is "red" and the SHAPE "triangle".

```
SELECT shape, color, cost
FROM objects
WHERE shape = "round" AND color = "blue" OR
  color = "red"
```

Use the NOT operator to negate the boolean result of a comparison. In the statement below, only those rows where the PAID column contains a FALSE value are retrieved.

```
SELECT *
FROM reservations
WHERE (NOT (paid = "TRUE"))
```

Predicate list

Local SQL supports the following predicates:

Predicate	Description
Comparison	Compares two values.
BETWEEN	Compares a value to a range formed by two values.
EXISTS	Determines whether a value exists in a look-up table.
IN	Determines whether a value exists in a list of values or a table.
LIKE	Compares, in part or in whole, one value with another.
IS NULL	Compares a value with an empty, or NULL, value.
SOME/ANY/ALL	Performs quantified comparisons.

Comparison predicates

See also

Compare two values.

```
value1 < value2          less than
value1 > value2          greater than
value1 = value2          equal to
value1 <> value2         not equal to
value1 != value2         not equal to (alternate syntax)
value1 >= value2         greater than or equal to
value1 <= value2         less than or equal to
```

Description

Use comparison predicates to compare two like values. Values compared can be: column values, literals, or calculations. The result of the comparison is a boolean value that is used in contexts like a WHERE clause to determine on a row-by-row basis whether a row meets the filtering criteria.

```
SELECT *
FROM orders
WHERE (itemstotal >= 1000)
```

Comparisons must be between two values of the same or a compatible data type. If one value is of an incompatible data type, convert that value with the CAST function to a compatible data type.

The result of a comparison predicate can be modified by a logical operator, such as NOT.

```
SELECT *
FROM orders
WHERE NOT (itemstotal >= 1000)
```

Note: comparison predicates can only be used in a WHERE or HAVING clause, or in the ON clause of a join; they cannot be used in the SELECT clause.

BETWEEN predicate

See also

Determines whether a value falls inside a range.

```
value1 [NOT] BETWEEN value2 AND value3
```

Description

Use the BETWEEN comparison predicate to compare a value to a value range. If the value is greater than or equal to the low end of the range and less than or equal to the high end of the range, BETWEEN returns a TRUE value. If the value is less than the low end value or greater than the high end value, BETWEEN returns a FALSE value. For example, the expression below returns a FALSE value because 10 is not between 1 and 5.

```
10 BETWEEN 1 AND 5
```

Use NOT to return the converse of a BETWEEN comparison. For example, the expression below returns a TRUE value.

```
10 NOT BETWEEN 1 AND 5
```

BETWEEN can be used with all non-BLOB data types, but all values compared must be of the same or a compatible data type. If one value is of an incompatible data type, convert that value with the CAST function to a compatible data type. Values used in a BETWEEN comparison may be column, literal, or calculated values.

```
SELECT saledate
FROM orders
WHERE (saledate BETWEEN "1/1/1988" AND "12/31/1988")
```

Hint BETWEEN is useful when filtering to retrieve rows with contiguous values that fall within the specified range. For filtering to retrieve rows with noncontiguous values, use the IN predicate.

EXISTS predicate

See also

Indicates whether values exist in a subquery.

```
EXISTS subquery
```

Description

Use the EXISTS comparison predicate to filter a table based on the existence of column values from the table in a subquery. The subquery is filtered using a WHERE clause comparing one or more columns in the filtered table to corresponding columns in the subquery. EXISTS returns a true value if the subquery has at least one row in its result set, false if zero rows are retrieved. The subquery is executed once for each row in the filtered table and the existence of rows in the subquery is used to include or exclude the rows in the filtered table.

```
SELECT O.orderno, O.custno
FROM orders O
WHERE EXISTS
  (SELECT C.custno
   FROM customer C
   WHERE (C.custno = O.custno))
```

The subquery may be further filtered with other conditions. For example, the statement below returns the rows pertaining to all customers who have placed orders the totals for which exceed $1000.

```
SELECT C.company, C.custno
FROM customer C
WHERE EXISTS
  (SELECT O.custno
   FROM orders O
   WHERE (O.custno = C.custno) AND
     (O.itemstotal > 1000))
```

Use NOT to return the converse of an EXISTS comparison.

IN predicate

See also

Indicates whether a value exists in a set of values.

```
value [NOT] IN (value_set)
```

Description

Use the IN comparison predicate to filter a table based on the existence of a column value in a specified set of comparison values. The set of comparison values may be either static using a comma-separated list of literals or dynamic using the result set from a subquery.

The value to compare with the values set can be any or a combination of: a column value, a literal value, or a calculated value.

The comparison set can be a static comma-separated list of literal values.

```
SELECT C.company, C.state
FROM customer C
WHERE (C.state IN ("CA", "HI"))
```

The comparison set can also be the result set from a subquery. The subquery may return multiple rows, but must only return a single column for comparison.

```
SELECT C.company, C.state
FROM customer C
WHERE (C.state IN
  (SELECT R.state
  FROM regions R
  WHERE (R.region = "Pacific")))
```

Use NOT to return the converse of an IN comparison.

IN can be used with all non-BLOB data types, but all values compared must be of the same or a compatible data type. If one value is of an incompatible data type, convert that value with the CAST function to a compatible data type.

Hint IN is useful when filtering to retrieve rows with noncontiguous values. For filtering to retrieve rows with contiguous values that fall within a specified range, use the BETWEEN predicate.

LIKE predicate

See also

Indicates the similarity of one value as compared to another.

```
value [NOT] LIKE [substitution_char] comparison_value [substitution_char]
  [ESCAPE escape_char]
```

Description

Use the LIKE comparison predicate to filter a table based on the similarity of a column value to a comparison value. Use of substitution characters allows the comparison to be based on the whole column value or just a portion.

```
SELECT *
FROM customer
WHERE (company LIKE "Adventure Undersea")
```

The wildcard substitution character ("%") may be used in the comparison to represent an unknown number of characters. LIKE returns a TRUE when the portion of the column value matches that portion of the comparison value not corresponding to the position of the wildcard character. The wildcard character can appear at the beginning, middle, or end of the comparison value (or multiple combinations of these positions). For example, the statement below retrieves rows where the column value begins with "A" and is followed by any number of any characters. Matching values could include "Action Club" and "Adventure Undersea", but not "Blue Sports".

```
SELECT *
FROM customer
WHERE (company LIKE "A%")
```

The single-character substitution character ("_") may be used in the comparison to represent a single character. LIKE returns a TRUE when the portion of the column value matches that portion of the comparison value not corresponding to the position of the single-character substitution character. The single-character substitution character can appear at the beginning, middle, or end of the comparison value (or multiple combinations of these positions). Use one single-character substitution character for each character to be wild in the filter pattern For example, the statement below retrieves rows where the column value begins with "b" ends with "n", with one character of any value between. Matching values could include "bin" and "ban", but not "barn".

```
SELECT words
FROM dictionary
WHERE (words LIKE "b_n")
```

Use NOT to return the converse of a LIKE comparison.

Use ESCAPE when the wildcard character "%" or "_" appear as data in the column. The ESCAPE keyword designates an escape character. In the comparison value for the LIKE predicate, the character that follows the escape character is treated as a data character and not a wildcard character. Other wildcard characters in the comparison value are unaffected.

In the example below, the "^" character is designated as the escape character. In the comparison value for the LIKE predicate ("%10^%%"), the "%" that immediately follows the escape character is treated as data in the PercentValue. This allows filtering based on the string "10%".

```
SELECT *
FROM Sales
WHERE (PercentValue LIKE "%10^%%" ESCAPE "^")
```

LIKE can be used only with CHAR or compatible data types. If one value is of an incompatible data type, convert that value with the CAST function to a compatible data type. The comparison performed by the LIKE predicate is case-sensitive.

IS NULL predicate

See also

Indicates whether a column contains a NULL value.

```
column_reference IS [NOT] NULL
```

Description

Use the IS NULL comparison predicate to filter a table based on the specified column containing a NULL (empty) value.

```
SELECT *
FROM customer
WHERE (invoicedate IS NULL)
```

Use NOT to return the converse of a IS NULL comparison.

Note For a numeric column, a zero value is not the same as a NULL value.

SOME/ANY/ALL predicates

See also

Compares a column value to a column value in multiple rows in a subquery.

```
column_reference comparison_predicate SOME | ANY | ALL (subquery)
```

Description

Use the quantified comparison predicates SOME, ANY, and ALL to filter a table by comparing a column value with multiple comparison values. The quantified comparison predicates are used with comparison predicates to compare a column value to the multiple values in a column of a subquery.

The ANY predicate evaluates TRUE when the accompanying comparison predicate evaluates TRUE for **any** value from the subquery. The SOME predicate operates functionally the same as ANY. For example, using the statement below, for any row to be retrieved from the HOLDINGS table, the value in the PUR_PRICE column need only be greater than any **one** value returned in the subquery's PRICE column.

```
SELECT *
FROM "holdings.dbf" H
WHERE (H."pur_price" > ANY
  (SELECT O."price"
  FROM "old_sales.dbf"))
```

The ALL predicate evaluates TRUE when the accompanying comparison predicate evaluates TRUE for **all** values from the subquery. For example, using the statement below, for any row to be retrieved from the HOLDINGS table, the value in the PUR_PRICE column needs to be greater than **every** value returned in the subquery's PRICE column.

```
SELECT *
FROM "holdings.dbf" H
WHERE (H."pur_price" > ALL
  (SELECT O."price"
  FROM "old_sales.dbf"))
```

Note The subquery providing the comparison values for the quantified comparison predicates may retrieve multiple rows, but can only have one column.

Relational operators list

Local SQL supports the following join types:

Join operator	Description
Equi-join	Joins two tables, filtering out non-matching rows.
INNER	Joins two tables, filtering out non-matching rows.
OUTER	Joins two tables, retaining non-matching rows.
Cartesian	Joins two tables, matching each row of one table with each row from the other.
UNION	Concatenates the result set of one query with the result set of another query.
Heterogeneous	Joins two tables in different databases, including differing database types.

Equi-join

See also

Joins two tables based on column values common between the two, excluding non-matches.

```
SELECT column_list
FROM table_reference, table_reference [, table_reference...]
WHERE predicate [AND predicate...]
```

Description

Use equi-join to join two tables, a source and joining table, that have values from one or more columns in common. One or more columns from each table are compared in the WHERE clause for equal values. For rows in the source table that have a match in the joining table, the data for the source table rows and matching joining table rows are included in the result set. Rows in the source table without matches in the joining table are excluded from the joined result set. In the statement below, the CUSTOMER and ORDERS tables are joined based on values in the CUSTNO column, which each table contains.

```
SELECT *
FROM customer C, orders O
WHERE (C.custno = O.custno)
```

More that one table may be joined with an equi-join. One column comparison predicate in the WHERE clause is required for each column compared to join each two tables. The statement below joins the CUSTOMER table to ORDERS, and then ORDERS to ITEMS. In this case, the joining table ORDERS acts as a source table for the joining table ITEMS.

```
SELECT *
FROM customer C, orders O, items I
WHERE (C.custno = O.custno) AND
  (O.orderno = I.orderno)
```

Tables may also be joined using a concatenation of multiple column values to produce a single value for the join comparison predicate. Here, the ID1 and ID2 columns in JOINING are concatenated and compared with the values in the single column ID in SOURCE.

```
SELECT *
FROM source S, joining J
WHERE (S.ID = J.ID1 || J.ID2)
```

An ORDER BY clause in equi-join statements can use columns from any table specified in the FROM clause to sort the result set.

INNER join

See also

Joins two tables based on column values common between the two, excluding non-matches.

```
SELECT column_list
FROM table_reference
  [INNER] JOIN table_reference
    ON predicate
  [[INNER] JOIN table_reference
    ON predicate...]
```

Description

Use an INNER JOIN to join two tables, a source and joining table, that have values from one or more columns in common. One or more columns from each table are compared in the ON clause for equal values. For rows in the source table that have a match in the joining table, the data for the source table rows and matching joining table rows are included in the result set. Rows in the source table without matches in the joining table are excluded from the joined result set. In the statement below, the CUSTOMER and ORDERS tables are joined based on values in the CUSTNO column, which each table contains.

```
SELECT *
FROM customer C
  INNER JOIN orders O
    ON (C.custno = O.custno)
```

More than one table may be joined with an INNER JOIN. One use of the INNER JOIN operator and corresponding ON clause is required for each each set of two tables joined. One columns comparison predicate in an ON clause is required for each column compared to join each two tables. The statement below joins the CUSTOMER table to ORDERS, and then ORDERS to ITEMS. In this case, the joining table ORDERS acts as a source table for the joining table ITEMS. (The statement below appears without the optional INNER keyword.)

```
SELECT *
FROM customer C
  JOIN orders O
    ON (C.custno = O.custno)
  JOIN items I
    ON (O.orderno = I.orderno)
```

Tables may also be joined using a concatenation of multiple column values to produce a single value for the join comparison predicate. Here, the ID1 and ID2 columns in JOINING are concatenated and compared with the values in the single column ID in SOURCE.

```
SELECT *
FROM source S
  INNER JOIN joining J
    ON (S.ID = J.ID1 || J.ID2)
```

An ORDER BY clause in INNER JOIN statements can use columns from any table specified in the FROM clause to sort the result set.

OUTER join

See also

Joins two tables based on column values common between the two, including non-matches.

```
SELECT column_list
FROM table_reference
   LEFT | RIGHT | FULL [OUTER] JOIN table_reference
    ON predicate
  [LEFT | RIGHT | FULL [OUTER] JOIN table_reference
    ON predicate...]
```

Description

Use an OUTER JOIN to join two tables, a source and joining table, that have one or more columns in common. One or more columns from each table are compared in the ON clause for equal values. The primary difference between inner and outer joins is that, in outer joins rows from the source table that do not have a match in the joining table are **not** excluded from the result set. Columns from the joining table for rows in the source table without matches have NULL values.

In the statement below, the CUSTOMER and ORDERS tables are joined based on values in the CUSTNO column, which each table contains. For rows from CUSTOMER that do not have a matching value between CUSTOMER.CUSTNO and ORDERS.CUSTNO, the columns from ORDERS contain NULL values.

```
SELECT *
FROM customer C
  LEFT OUTER JOIN orders O
    ON (C.custno = O.custno)
```

The LEFT modifier causes all rows from the table on the left of the OUTER JOIN operator to be included in the result set, with or without matches in the table to the right. If there is no matching row from the table on the right, its columns contain NULL values. The RIGHT modifier causes all rows from the table on the right of the OUTER JOIN operator to be included in the result set, with or without matches. If there is no matching row from the table on the left, its columns contain NULL values. The FULL modifier causes all rows from the all tables specified in the FROM clause to be included in the result set, with or without matches. If there is no matching row from one of the tables, its columns contain NULL values.

More than one table may be joined with an INNER JOIN. One use of the INNER JOIN operator and corresponding ON clause is required for each each set of two tables joined. One column comparison predicate in an ON clause is required for each column compared to join each two tables. The statement below joins the CUSTOMER table to ORDERS, and then ORDERS to ITEMS. In this case, the joining table ORDERS acts as a source table for the joining table ITEMS.

```
SELECT *
FROM customer C
  FULL OUTER JOIN orders O
    ON (C.custno = O.custno)
  FULL OUTER JOIN items I
    ON (O.orderno = I.orderno)
```

Tables may also be joined using expressions to produce a single value for the join comparison predicate. Here, the ID1 and ID2 columns in JOINING are separately compared with two values produced by the SUBSTRING function using the single column ID in SOURCE.

```
SELECT *
FROM source S
  RIGHT OUTER JOIN joining J
    ON (SUBSTRING(S.ID FROM 1 FOR 2) = J.ID1) AND
    (SUBSTRING(S.ID FROM 3 FOR 1) = J.ID2)
```

An ORDER BY clause in OUTER JOIN statements can use columns from any table specified in the FROM clause to sort the result set.

Cartesian join

See also

Joins two tables in a non-relational manner.

```
SELECT *
FROM table_reference, table_reference [,table_reference...]
```

Description

Use the Cartesian join to join the column of two tables into one result set, but without correlation between the rows from the tables. Cartesian joins match **each** row of the source table with **each** row of the joining table. No column comparisons are used, just simple association. If the source table has 10 rows and the joining table has 10, the result set will contain 100 rows as each row from the source table is joined with each row from the joined table.

```
SELECT *
FROM "employee.dbf", "items.db"
```

UNION join

See also

Concatenates the rows of one table to the end of another table.

```
SELECT col_1 [, col_2, ... col_n]
FROM table_reference
UNION [ALL]
SELECT col_1 [, col_2, ... col_n]
FROM table_reference
```

Description

Use the UNION join to add the rows of one table to the end of another similarly structured SELECT query result sets. The SELECT statement for the source and joining tables must include the same number of columns for them to be UNION compatible. The table structures themselve need not be the same as long as those column included in the SELECT statements are.

```
SELECT custno, company
FROM customers
UNION
SELECT custno, company
FROM old_customers
```

Matching data types for a column is not always mandatory for data retrieved by the UNION across the multiple tables. If there is a data type difference between two tables for a given column, an error occurs if the same column from the second (or subsequent) table would lose data. For example, if the first table's column is of type DATE and the second table's of type TIMESTAMP, part of the TIMESTAMP value would be lost when put into a lesser DATE type column. A "Type mismatch in expression" error is generated for these situations. In general, when there are column differences between the tables, use the CAST function to convert the columns to a compatible type.

```
SELECT s.id, CAST(s.date_field AS TIMESTAMP)
FROM source s
UNION ALL
SELECT j.id, j.timestamp_field
FROM joiner j
```

Matching names is not mandatory for result set columns retrieved by the UNION across the multiple tables. Column name differences between the multiple source tables are automatically handled. If the first column of two tables has a different name, the first column in the UNION result set will use that from the first SELECT statement.

By default, non-distinct rows are aggregated into single rows in a UNION join. Use ALL to retain non-distinct rows.

To join two tables with UNION where one table does not have a column included by another, a compatible literal or expression may be used instead in the SELECT statement missing the column. For example, if there is no column in the JOINING table corresponding to the NAME column in SOURCE an expression is used to provide a value for a pseudo JOINING.NAME column. Assuming SOURCE.NAME is of type CHAR(10), the CAST function is used to convert an empty character string to CHAR(10).

```
SELECT s.id, s.name
FROM source s
UNION ALL
SELECT j.id, CAST("" AS CHAR(10))
FROM joiner j
```

Heterogeneous joins

See also

Joins two tables from different databases.

```
SELECT column_list
FROM ":database_reference:table_reference",
  ":database_reference:table_reference"
  [,":database_reference:table_reference"...]
WHERE predicate [AND predicate...]
```

Description

Use a heterogeneous join to join two tables that reside in different databases. The joined tables may be of different types (like dBASE to Paradox or Paradox to InterBase), but you can only join tables whose database types are accessible through the BDE (local, ODBC, or SQL Links). A hetergeneous join may be any of the joins supported by local SQL. The difference is in the syntax for the table reference: the database containing each table is specified in the table reference, surrounded by colons and the whole reference enclosed in quotation marks. The database specified as part of the table reference may be a drive and directory reference (for local tables) or a BDE alias.

```
SELECT *
FROM ":DBDEMOS:customer.db" C, ":BCDEMOS:orders.db" O
WHERE (C.custno = O.custno)
```

Updatable queries

See also

Definition...

Single-table queries

Queries that retrieve data from a single table are updatable provided that:

- There is no DISTINCT key word in the SELECT.
- Everything in the SELECT clause is a simple column reference or a calculated column, no aggregation is allowed. Calculated columns remain read-only.
- The table referenced in the FROM clause is an updatable base table.
- There is no GROUP BY or HAVING clause.
- There are no subqueries in the statement.
- There is no ORDER BY clause.

The read-only effect of an ORDER BY clause is negated and the query updatable if the ORDER BY clause uses a single column and there is a dBASE single-column primary or secondary index based on that same field. dBASE compound (expression) indexes will not negate the read-only effect of an ORDER BY clause. A Paradox single- or multi-field primary index will make the query updatable if the ORDER BY uses exactly the same columns (in the same order) as the index. Paradox secondary indexes will not negate the read-only effect of an ORDER BY clause.

Multi-table queries

All queries that join two or more tables will produce a read-only result set.

Calculated fields

For updateable queries with calculated fields, an additional field property identifies a result field as both read-only and calculated. Every call to the BDE function DbiPutField causes recalculation of any dependent fields.

Parameter substitutions in DML statements

Parameter markers can be used in DML statements in place of data values. Parameters are identified by a preceding colon (:). For example:

```
SELECT last_name, first_name
FROM "customer.db"
WHERE (last_name > :lname) AND (first_name < :fname)
```

Parameters allow the same SQL statement to be used with different data values to be used for comparisons. Parameters are placeholders for data values. At runtime, the front-end application fills the parameter with a value, before the query is executed. When the query is executed, the data values passed into the parameters are substituted for the parameter placeholder and the SQL statement is applied.

Parameters are used to pass data values to be used in WHERE clause comparison and as update atoms in updating statements. Parameters cannot be used to pass values for metadata object names (table and column names).

```
UPDATE orders
SET itemstotal = :TotalParam
WHERE (orderno = 1014)
```

Data values passed to SQL statements as parameters are enclosed in quotation marks (where applicable). Thus, when a front-end application supplies CHAR and DATE values for parameters, quotation marks need not be included when the parameter is populated with the values.

Data definition overview

Local SQL supports data definition language (DDL) for creating, altering, and deleting tables and indexes.

Local SQL does not permit the metadata object names to be represented by parameters in DDL statements.

Local SQL supports the following DDL statements:

DDL Statement	Description
CREATE TABLE	Creates a new table.
ALTER TABLE	Adds columns to and deletes columns from an existing table.
DROP TABLE	Deletes an existing table.
CREATE INDEX	Creates a new secondary index for an existing table.
DROP INDEX	Deletes an existing primary or secondary index.

CREATE TABLE statement

See also

Creates a table.

```
CREATE TABLE table_reference (column_definition [, column_definition,...] [,
  primary_key_constraint])
```

Description

Use the CREATE TABLE statement to create a dBASE or Paradox table, define its columns, and define a primary key constraint.

The table name reference for CREATE TABLE must comply with the rules described in the section on naming conventions. Table names with embedded spaces must be enclosed in quotation marks.

Column definitions consist of a comma-separated list of combinations of column name, data type, and (if applicable) dimensions. The list of column definitions must be enclosed in parentheses. The number and type of dimensions that must be specified varies with column type. See the section on defining column types for specific syntax of all supported column types.

Use the PRIMARY KEY (or CONSTRAINT) keyword to create a primary index for the new table.The following statement creates a Paradox table with a PRIMARY KEY constraint on the LAST_NAME and FIRST_NAME columns:

```
CREATE TABLE "employee.db"
(
  last_name CHAR(20),
  first_name CHAR(15),
  salary NUMERIC(10,2),
  dept_no SMALLINT,
  PRIMARY KEY (last_name, first_name)
)
```

An alternate syntax for creating a primary key constraint is using the CONSTRAINT keyword. While Paradox primary indexes do not have names, an arbitrary name needs to be provided to satisfy the CONSTRAINT keyword need for a token name.

```
CREATE TABLE "employee.db"
(
  last_name CHAR(20),
  first_name CHAR(15),
  salary NUMERIC(10,2),
  dept_no SMALLINT,
  CONSTRAINT z PRIMARY KEY (last_name, first_name)
)
```

Indicate whether the table is a Paradox or dBASE table by specifying the file extension when naming the table:

- ".DB" for Paradox tables
- ".DBF" for dBASE tables

If you omit the file extension for a local table name, the table created is the table type specified in the Default Driver setting in the System INIT page of the BDE Administrator utility. When specifying a file extension, the table name reference for CREATE TABLE must be enclosed in quotation marks.

Column definitions based on domains are not supported. Primary keys are the only form of constraint that can be defined with CREATE TABLE.

Note To create a table with columns that have non-alphanumeric characters or spaces in the column name, you must enclose the column name in quotation marks and prefix the quoted column name with the table name in quotes.

```
CREATE TABLE "abc.db" A
(
```

```
  ID CHAR(3),
  "abc.db"."funny name" CHAR(10)
)
```

Table column types

The following table lists SQL data types and how each is translated by the BDE to native Paradox and dBASE types.

The native column type names (and storage dimensions) in the table below are based on level 7 Paradox and dBASE tables. Column type names and availability vary across the various versions of Paradox and dBASE. For instance, the Binary column type was not available for dBASE IV (level 4) tables.

SQL Syntax	BDE Logical	Paradox	dBASE
SMALLINT	fldINT16	Short	Numeric(6,0)
INTEGER	fldINT32	Long	Long
DECIMAL	fldBCD	BCD(32,0)	Numeric(20,0)
DECIMAL(7)	fldBCD	BCD(32,0)	Numeric(7,0)
DECIMAL(7,2)	fldBCD	BCD(32,2)	Numeric(7,2)
NUMERIC	fldFLOAT	Number	Double
NUMERIC(7)	fldFLOAT	Number	Double
NUMERIC(7,2)	fldFLOAT	Number	Double
FLOAT	fldFLOAT	Number	Double
FLOAT(7)	fldFLOAT	Number	Double
FLOAT(7,2)	fldFLOAT	Number	Double
CHARACTER(10)	fldZSTRING	Alpha(10)	Character(10)
VARCHAR(10)	fldZSTRING	Alpha(10)	Character(10)
DATE	fldDATE	Date	Date
BOOLEAN	fldBOOL	Logical	Logical
BLOB(1,1)	fldstMEMO	Memo	Memo
BLOB(1,2)	fldstBINARY	Binary	Binary
BLOB(1,3)	fldstFMTMEMO	Formatted memo	Memo
BLOB(1,4)	fldstOLEOBJ	OLE	OLE
BLOB(1,5)	fldstGRAPHIC	Graphic	Binary
TIME	fldTIME	Time	Character(11)
TIMESTAMP	fldTIMESTAMP	Timestamp	Datetime
MONEY	fldFLOAT, fldstMONEY	Money	Double
AUTOINC	fldINT32, fldstAUTOINC	Autoincrement	Autoinc
BYTES	fldBYTES	Bytes(1)	N/A
BYTES(10)	fldBYTES	Bytes(10)	N/A

SQL data types

See also

The table below lists the SQL data types available in local SQL. Table columns, literals, parameter values, and calculation results will all be of one of these types. When defining columns in CREATE TABLE and ALTER TABLE statements, the SQL data types below are translated by the BDE into specific Paradox, dBASE, and FoxPro column types. These data types are also used with the CAST function when converting a value from one data type to another (except BLOB and memo types, on which CAST cannot operate).

While there are three SQL data types available that apply to floating point numbers (DECIMAL, NUMERIC, and FLOAT), each translates to a different native column type in local tables. Further, the native column type used varies depending on the particular local table type used (Paradox, dBASE, or FoxPro).

Column type	Definition syntax
`SMALLINT`	Small integer values. No scale or precision are specified.
`INTEGER`	Integer values. No scale or precision are specified.
`DECIMAL[(s[, p])]`	Floating point numbers. Scale and precision are each optional. If precision is specified, scale must also be.
`NUMERIC[(s[, p])]`	Floating point numbers. Scale and precision are each optional. If precision is specified, scale must also be.
`FLOAT(s, p)`	Floating point numbers. Scale and precision are each optional. If precision is specified, scale must also be.
`CHARACTER(length)`	Alpha-numeric type values. Specify length of column capacity, in bytes. Length must be between 1 and 254.
`VARCHAR(length)`	Alpha-numeric type values. Specify length of column capacity, in bytes. Length must be between 1 and 254. In local SQL, VARCHAR is functionally the same as CHAR.
`DATE`	Date values with no time portion. No scale or precision are specified.
`BOOLEAN`	Logical (TRUE/FALSE) values. No scale or precision are specified.
`BLOB(length, type)`	Streaming text or raw binary data. Specify length (column capacity), in bytes. Specify the type of BLOB column: Memo (1), Binary (2), Formatted Memo (3), OLE (4), Graphic/Binary (5). For Paradox BLOB columns, length must be between 0 and 240 (amount of data stored in .DB file); for dBASE tables between 0 and 32,767 (valid length has no practical effect on column created). Not all BLOB column types apply to all local table types or correspond to the same native column types in all table types.
`TIME`	Time values, with no date portion. No scale or precision specified.
`TIMESTAMP`	Date and time portions in same column. No scale or precision specified.
`MONEY`	Floating point number values. Scale and precision is automatic.
`AUTOINC`	Automatically incrementing column values. No scale or precision specified.
`BYTES(length)`	User-defined data types. Specify length (column capacity), in bytes.

ALTER TABLE statement

See also

Adds or deletes a column from a table.

```
ALTER TABLE table_reference DROP [COLUMN] column_reference | ADD [COLUMN]
  column_reference [,reference DROP [COLUMN] column_reference | ADD [COLUMN]
  column_reference...]
```

Description

Use the ALTER TABLE statement to add a column to or delete a column from an existing table. It is possible to delete one column and add another in the same ALTER TABLE statement.

The DROP keyword requires only the name of the column to be deleted. The ADD keyword requires the same combination of column name, type, and possibly dimension definition as CREATE TABLE when defining new columns. See the section on defining column types for the specific syntax of all supported column types.

The statement below deletes the column FULLNAME and adds the column LASTNAME.

```
ALTER TABLE "names.db"
DROP fullname, ADD lastname CHAR(25)
```

It is possible to delete and add a column of the same name in the same ALTER TABLE statement, however any data in the column is lost in the process. This allows quick redefinition of columns while still in the database design stages.

```
ALTER TABLE "names.db"
DROP lastname, ADD lastname CHAR(30)
```

If a column to be deleted is part of a primary key, the primary index is deleted. ALTER TABLE fails on an attempt to delete a column that is the target of a foreign key constraint (referential integrity).

To reference columns with non-alphanumeric characters or spaces embedded in the column name, you must enclose the column name in quotation marks and prefix the quoted column name with the table name in quotes.

```
ALTER TABLE "customer.db"
ADD "customer.db"."#ID" CHAR(3)
```

DROP TABLE statement

See also

Deletes a table.

```
DROP TABLE table_reference
```

Description

Use the DROP TABLE statement to delete an existing table. The statement below drops a Paradox table:

```
DROP TABLE "employee.db"
```

CREATE INDEX statement

See also

Creates a secondary index.

```
CREATE [UNIQUE] [ASC | DESC] INDEX index_reference ON table_reference
  (column_reference [,column_reference...])
```

Description

Use the CREATE INDEX statement to create a secondary index for an existing table. Index names may not have embedded spaces. Paradox indexes may be based on multiple columns. Due to the distinctive nature of dBASE expression indexes, only single-column indexes can be created with CREATE INDEX.

Use UNIQUE to create an index that raises an error if rows with duplicate column values are inserted. By default, indexes are not unique.

Use ASC (or ASCENDING) to create an index that orders data in an ascending direction (smallest to largest). DESC (or DESCENDING) creates a descending ordering (largest to smallest). When a direction imperative is not specified, ASC is the implied default.

The following statement creates a multi-column (compound) Paradox secondary index.

```
CREATE INDEX custdate ON "orders.db" (custno, saledate)
```

The following statement creates a unique dBASE secondary index.

```
CREATE UNIQUE INDEX namex ON "employee.dbf" (last_name)
```

The existence of indexes may affect the updatability of queries. See the section on updatable queries for more information.

DROP INDEX

See also

Deletes an index.

```
DROP INDEX table_reference.index_reference | PRIMARY
```

Description

Use the DROP INDEX statement to delete a primary or secondary index.

To delete a dBASE primary or secondary index or a Paradox secondary index, identify the index using the table name and index name separated by an identifier connector symbol (.).

```
DROP INDEX "employee.dbf".namex
```

To delete a Paradox primary index, identify the index with the keyword PRIMARY.

```
DROP INDEX orders.PRIMARY
```

The existence of indexes may affect the updatability of queries. See the section on updatable queries for more information.

Jan Kraski

Dedicated to the memory of

Jan Kraski

The monster got ‘im -- June 3, 1998

RIP