Local SQL

The Borland Database Engine (BDE) enables access to both local and remote database tables through local SQL (Structure Query Language). Local SQL (sometimes called "client-based SQL") is a subset of ANSI-92 SQL enhanced to support Paradox and dBASE (standard) naming conventions for tables and fields (called "columns" in SQL).

Local SQL lets you use SQL to query "local" standard database tables that do not reside on a database server (specifically Paradox or dBASE tables) as well as "remote" SQL servers. Local SQL is also essential to make multi-table queries across both local standard tables and those on remote SQL servers.

Naming conventions
This section describes naming conventions for tables and columns, syntax enhancements, and syntax limitations for local SQL.

The SQL statements are broken down into two different categories, DDL (Data Definition Language) and DML (Data Manipulation Language):

Data Manipulation
SQL statements in Data Manipulation Language (DML) used for selecting, inserting, updating, and deleting table data. Syntax and usage examples are included.

Data Definition
SQL statements in Data Definition Language (DDL) used for creating, altering, and dropping tables, and for creating and dropping indexes. The DDL transforms directly into BDE function calls. Syntax and usage examples are included.

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

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

Naming conventions

ANSI-standard SQL confines each table or column name to a single word comprised of alphanumeric characters and the underscore symbol (_). Local SQL, however, is enhanced to support more comprehensive names.

Tables

Local SQL supports full file and path specifications for table names. Table names with path or file-name 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 for table names. 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 the BDE Configuration Utility, either in the Default Driver setting in the System page or in the default drive type for the standard alias associated with the query or table.

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"
```

Columns

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 is two words:

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

In the next example, the column name duplicates the SQL DATE keyword:

```
SELECT DATELOG."DATE" FROM DATELOG
```

See also: Reserved words

Reserved words

Here is an alphabetical list of words reserved by Local SQL:

ACTIVE, ADD, ALL, AFTER, ALTER, AND, ANY, AS, ASC, ASCENDING, AT, AUTO, AUTOINC, AVG

BASE_NAME, BEFORE, BEGIN, BETWEEN, BLOB, BOOLEAN, BOTH, BY, BYTES

CACHE, CAST, CHAR, CHARACTER, CHECK, CHECK_POINT_LENGTH, COLLATE, COLUMN, COMMIT, COMMITTED, COMPUTED, CONDITIONAL, CONSTRAINT, CONTAINING, COUNT, CREATE, CSTRING, CURRENT, CURSOR

DATABASE, DATE, DAY, DEBUG, DEC, DECIMAL, DECLARE, DEFAULT, DELETE, DESC, DESCENDING, DISTINCT, DO, DOMAIN, DOUBLE, DROP

ELSE, END, ENTRY_POINT, ESCAPE, EXCEPTION, EXECUTE, EXISTS, EXIT, EXTERNAL, EXTRACT

FILE, FILTER, FLOAT, FOR, FOREIGN, FROM, FULL, FUNCTION

GDSCODE, GENERATOR, GEN_ID, GRANT, GROUP, GROUP_COMMIT_WAIT_TIME

HAVING, HOUR

IF, IN, INT, INACTIVE, INDEX, INNER, INPUT_TYPE, INSERT, INTEGER, INTO, IS, ISOLATION

JOIN

KEY

LONG, LENGTH, LOGFILE, LOWER, LEADING, LEFT, LEVEL, LIKE, LOG_BUFFER_SIZE

MANUAL, MAX, MAXIMUM_SEGMENT, MERGE, MESSAGE, MIN, MINUTE, MODULE_NAME, MONEY, MONTH

NAMES, NATIONAL, NATURAL, NCHAR, NO, NOT, NULL, NUM_LOG_BUFFERS, NUMERIC

OF, ON, ONLY, OPTION, OR, ORDER, OUTER, OUTPUT_TYPE, OVERFLOW

PAGE_SIZE, PAGE, PAGES, PARAMETER, PASSWORD, PLAN, POSITION, POST_EVENT, PRECISION, PROCEDURE, PROTECTED, PRIMARY, PRIVILEGES

RAW_PARTITIONS, RDB$DB_KEY, READ, REAL, RECORD_VERSION, REFERENCES, RESERV, RESERVING, RETAIN, RETURNING_VALUES, RETURNS, REVOKE, RIGHT, ROLLBACK

SECOND, SEGMENT, SELECT, SET, SHARED, SHADOW, SCHEMA, SINGULAR, SIZE, SMALLINT, SNAPSHOT, SOME, SORT, SQLCODE, STABILITY, STARTING, STARTS, STATISTICS, SUB_TYPE, SUBSTRING, SUM, SUSPEND

TABLE, THEN, TIME, TIMESTAMP, TIMEZONE_HOUR, TIMEZONE_MINUTE, TO, TRAILING, TRANSACTION, TRIGGER, TRIM

UNCOMMITTED, UNION, UNIQUE, UPDATE, UPPER, USER

VALUE, VALUES, VARCHAR, VARIABLE, VARYING, VIEW

WAIT, WHEN, WHERE, WHILE, WITH, WORK, WRITE

YEAR

OPERATORS:

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

Data manipulation

With some restrictions, local SQL supports the following statements for data manipulation:

- SELECT, for retrieving existing data
- INSERT, for adding new data to a table
- UPDATE, for modifying existing data
- DELETE, for removing existing data from a table

The following sections describe functions available to DML statements in local SQL.

- Parameter substitutions in DML statements
- Aggregate functions
- String functions
- Date function
- Operators
- Updateable queries

For additional illustrative examples, see:

- DML examples

SELECT

The SELECT statement is used to retrieve data from one or more tables. A SELECT that retrieves data from multiple tables is called a "join." Local SQL supports the following form of the SELECT statement:

```
SELECT [DISTINCT] column_list
FROM table_reference
[WHERE search_condition]
[ORDER BY order_list]
[GROUP BY group_list]
[HAVING having_condition]
[UNION select_expr]
```

Except as noted below, all clauses are handled as in ANSI-standard SQL. Clauses in square brackets are optional.

The column_list indicates the columns from which to retrieve data. For example, the following statement retrieves data from two columns:

```
SELECT PART_NO, PART_NAME
FROM PARTS
```

Choose one of the following topics for more information on using SELECT:

- FROM Clause
- WHERE Clause
- ORDER BY Clause
- GROUP BY Clause
- HAVING Clause
- UNION Clause
- Heterogeneous Joins

FROM clause

The FROM clause specifies the table or tables from which to retrieve data. Table_reference can be a single table, a comma-delimited list of tables, or can be an inner or outer join as specified in the SQL-92 standard. For example, the following statement specifies a single table:

```
SELECT PART_NO
FROM "PARTS.DBF"
```

The next statement specifies a left outer join for table_reference:

```
SELECT * FROM PARTS LEFT OUTER JOIN INVENTORY
ON PARTS.PART_NO = INVENTORY.PART_NO
```

WHERE clause

The optional WHERE clause reduces the number of rows returned by a query to those that match the criteria specified in search_condition. For example, the following statement retrieves only those rows with PART_NO greater than 543:

```
SELECT * FROM PARTS
WHERE PART_NO > 543
```

The WHERE clause can include the IN predicate, followed by a parenthesized list of values. For example, the next statement retrieves only those rows where a part number matches an item in the IN predicate list:

```
SELECT * FROM PARTS
WHERE PART_NO IN (543, 544, 546, 547)
```

In addition to scalar comparison operators (=, <, > ...) additional predicates using IN, ANY, ALL, EXISTS are supported.

ORDER BY clause

The ORDER BY clause specifies the order of retrieved rows. For example, the following query retrieves a list of all parts listed in alphabetical order by part name:

```
SELECT * FROM PARTS
ORDER BY PART_NAME ASC
```

The next query retrieves all part information ordered in descending numeric order by part number:

```
SELECT * FROM PARTS
ORDER BY PART_NO DESC
```

Calculated fields can be ordered by correlation name or ordinal position. For example, the following query orders rows by FULL_NAME, a calculated field:

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME, PHONE,
FROM CUSTOMER
ORDER BY FULL_NAME
```

Projection of all grouping or ordering columns is not required.

GROUP BY clause

The GROUP BY clause specifies how retrieved rows are grouped for aggregate functions.

HAVING clause

The HAVING clause specifies conditions records must meet to be included in the return from a query. It is a conditional expression used in conjunction with the GROUP BY clause. Groups that do not meet the expression in the HAVING clause are ommitted from the result set.

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. See WHERE Clause

In addition to scalar comparison operators (=, <, > ...) additional predicates using IN, ANY, ALL, EXISTS are supported.

UNION clause

The UNION clause combines the results of two or more SELECT statements to produce a single table.

Heterogeneous joins

Local SQL supports joins of tables in different database formats; such a join is called a "heterogeneous join."

When you perform a heterogeneous join, you may select a local alias. To select an alias, choose SQL| Select Alias. If you have not selected an alias, Local SQL will attempt to find the table in the current directory of the database which is being used. For example, the alias **:**WORK**:** might be the database handle passed into the function.

When you specify a table name after selecting a local alias:

- For local tables, specify either the alias or the path.
- For remote tables, specify the alias.

The following statement retrieves data from a Paradox table and a dBASE table:

```
SELECT DISTINCT C.CUST_NO, C.STATE, O.ORDER_NO
FROM "CUSTOMER.DB" C, "ORDER.DBF" O
WHERE C.CUST_NO = O.CUST_NO
```

You can also use BDE aliases in conjunction with table names.

INSERT

In local SQL, INSERT is of two forms:

```
  INSERT INTO CUSTOMER (FIRST_NAME, LAST_NAME, PHONE)
  VALUES(:fname, :lname, :phone_no)
```

Insertion from one table to another through a subquery is not allowed.

Examples

The following statement adds a row to a table, assigning values to two columns:

```
INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID) VALUES (52, "DGPII");
```

The next statement specifies values to insert into a table with a SELECT statement:

```
INSERT INTO PROJECTS
    SELECT * FROM NEW_PROJECTS
          WHERE NEW_PROJECTS.START_DATE > "6-JUN-1994";
```

UPDATE

There are no restrictions on or extensions to the ANSI-standard UPDATE statement.

DELETE

There are no restrictions on or extensions to the ANSI-standard DELETE statement.

Parameter substitutions in DML statements

Variables or parameter markers (?) can be used in DML statements in place of values. Variables must always be preceded by a colon (:). For example:

```
SELECT LAST_NAME, FIRST_NAME
FROM "CUSTOMER.DB"
WHERE LAST_NAME > :var1 AND FIRST_NAME < :var2
```

Aggregate functions

The following ANSI-standard SQL aggregate functions are available to local SQL for use with data retrieval:

- SUM(), for totaling all numeric values in a column
- AVG(), for averaging all non-NULL numeric values in a column
- MIN(), for determining the minimum value in a column
- MAX(), for determining the maximum value in a column
- COUNT(), for counting the number of values in a column that match specified criteria

Complex aggregate expressions are supported, such as:

```
SUM( Field * 10 )
SUM( Field ) * 10
SUM( Field1 + Field2 )
```

String functions

Local SQL supports the following ANSI-standard SQL string manipulation functions for retrieval, insertion, and updating:

- UPPER(), to force a string to uppercase
- LOWER(), to force a string to lowercase
- TRIM(), to remove repetitions of a specified character from the left, right, or both sides of a string
- SUBSTRING() to create a substring from a string

Substring

SUBSTRING() takes a string and creates a substring of that string.

```
SELECT SUBSTRING( CUSTNAME FROM 1 FOR 10 )  FROM CUSTOMER
```

This query return the first 10 characters of the CUSTNAME column.

Date function

Local SQL supports the EXTRACT() function for isolating a single numeric field from a date/time field on retrieval using the following syntax:

```
EXTRACT (extract_field FROM field_name)
```

For example, the following statement extracts the year value from a DATE field:

```
SELECT EXTRACT(YEAR FROM HIRE_DATE)
FROM EMPLOYEE
```

You can also extract MONTH, DAY, HOUR, MINUTE, and SECOND using this function.

Note: EXTRACT does not support the TIMEZONE_HOUR or TIMEZONE_MINUTE clauses.

Operators

Local SQL supports the following operators:

Type	Operator
Arithmetic	+ - * /
Comparison	< > = <> >= =< IS NULL IS NOTNULL
Logical	AND OR NOT
String concatenation	||

Updateable queries

SQL Links offers expanded support for both single table and multi-table updateable queries.

These restrictions apply to updates:

- Linking fields cannot be updated
- Index switching will cause an error

Restrictions on live queries

The semantics of live queries, for all data manipulation methods, return cursors that are functionally and semantically similar to cursors returned from the BDE function DbiOpenTable.

Single table queries or views are updateable provided that:

- There are no JOINs, UNIONs, INTERSECTs, or MINUS operations.
- There is no DISTINCT key word in the SELECT. (This restriction may be relaxed if all the fields of a unique index are projected.)
- Everything in the SELECT clause is a simple column reference or a calculated field, no aggregation is allowed.
- The table referenced in the FROM clause is either an updateable base table or an updateable view.
- There is no GROUP BY or HAVING clause.
- There are no subqueries that reference the table in the FROM clause and no correlated subqueries.
- Any ORDER BY clause can be satisfied with an index.

Additional restrictions may apply for fields or cursor methods. If an index is used to satisfy an order clause, the BDE function DbiSwitchToIndex returns an error.

Restrictions on live joins

Live joins depend upon composite cursors. Live joins may be used only if:

- All joins are left-to-right outer joins.
- All join are equi-joins.
- All join conditions are satisfied by indexes (for Paradox and dBASE)
- Output ordering is not defined.
- Each table in the join is a base table.
- The query contains no elements listed above that would prevent single-table updatability.

Constraints

You can constrain any updateable query by setting the query statement property stmtCONSTRAINED to TRUE before execution. A error will then be returned whenever a modify or insert would cause the new record to disappear from the 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.

BDE function calls on query results

If a query returns a cursor, that cursor will fully support the low-level query capabilities of a cursor returned from the BDE function DbiOpenTable. Thus filters and field maps may be applied to further refine the result set. Unlike cursors from open table, some operations such as DbiAddIndex or DbiSwitchToIndex are not be supported.

DML examples

The following clauses are supported:

```
SELECT FROM, WHERE, ORDER BY, GROUP BY, and HAVING
```

The following aggregates are supported:

```
SUM, AVG, MIN, MAX, COUNT
```

The following operators are supported:

```
, -, *, /, =, < >, IS NULL
```

UPDATE, INSERT, DELETE operations are fully supported.

The following examples show DML statements used with standard databases:

Example 1: UPDATE

```
update goods
  set city = 'Santa Cruz'
  where goods.city = 'Scotts Valley'
```

Example 2: INSERT

```
insert
  into goods ( part_no, city )
  values ( 'aa0094', 'San Jose' )
```

Example 3: DELETE

```
delete
  from goods
  where part_no = 'aa0093'
```

Example 4: SELECT used to join

The following example illustrates how the SELECT statement is supported as an equivalent to a JOIN:

```
select distinct p.part_no, p.quantity, g.city
  from parts p, goods g
  where p.part_no = g.part_no
  and p.quantity > 20
  order by p.quantity, g.city, p.part_no
```

A SELECT statement that contains a join must have a WHERE clause in which at least one field from each table is involved in an equality check.

Example 5: Sub-selects

Sub-select queries are supported. The following example illustrates this syntax:

```
select p.part_no
  from parts p
  where p.quantity in
   (select i.quantity
     from inventory i
     where  i.part_no = 'aa9393')
```

Example 6: GROUP BY

The following examples illustrate the GROUP BY clause:

```
select part_no, sum(quantity) as PQTY
  from parts
  group by part_no
```

Note: Aggregates in the SELECT clause must have GROUP BY clause if a projected field is used, as shown in the first example above.

Example 7: ORDER BY

The following example illustrates the ORDER BY with a DESCENDING clause:

```
select distinct customer_no
  from c:\data\customer
  order by customer_no descending
```

Data Definition

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

Views are supported.

Local SQL does not permit the substitution of variables for values in DDL statements.

The following DDL statements are supported:

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- CREATE INDEX
- DROP INDEX
- CREATE VIEW

For additional illustrative examples see:

- DDL Examples

CREATE TABLE

CREATE TABLE is supported with the following limitations:

- Column definitions based on domains are not supported.
- Constraints are limited to PRIMARY KEY for Paradox. Constraints are unsupported in dBASE.

For example, 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)
)
```

The same statement for a dBASE table should omit the PRIMARY KEY definition:

```
CREATE TABLE "employee.dbf"
(
LAST_NAME CHAR(20),
FIRST_NAME CHAR(15),
SALARY NUMERIC(10,2),
DEPT_NO SMALLINT
)
```

Creating Paradox and dBASE tables

You create a Paradox or dBASE table using local SQL 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 page of the BDE Configuration Utility.

Data type mappings for CREATE TABLE

The following table lists SQL syntax for data types used with CREATE TABLE, and describes how those types are mapped to Paradox and dBASE types by the BDE:

SQL Syntax	BDE Logical	Paradox	dBASE
SMALLINT	fldINT16	Short	Number (6,10)
INTEGER	fldINT32	Long Integer	Number (20,4)
DECIMAL(x,y)	fldBCD	BCD	N/A
NUMERIC(x,y)	fldFLOAT	Number	Number (x,y)
FLOAT(x,y)	fldFLOAT	Number	Float (x,y)
CHARACTER(n)	fldZSTRING	Alpha	Character
VARCHAR(n)	fldZSTRING	Alpha	Character
DATE	fldDATE	Date	Date
BOOLEAN	fldBOOL	Logical	Logical
BLOB(n,1)	fldstMEMO	Memo	Memo
BLOB(n,2)	fldstBINARY	Binary	Binary
BLOB(n,3)	fldstFMTMEMO	Formatted memo	N/A
BLOB(n,4)	fldstOLEOBJ	OLE	OLE

BLOB(n,5)	fldstGRAPHIC	Graphic	N/A
TIME	fldTIME	Time	N/A
TIMESTAMP	fldTIMESTAMP	Timestamp	N/A
MONEY	fldFLOAT, fldstMONEY	Money	Number (20,4)
AUTOINC	fldINT32, fldstAUTOINC	Autoincrement	N/A
BYTES(n)	fldBYTES(n)	Bytes	N/A

x = precision (default: specific to driver)
y = scale (default: 0)
n = length in bytes (default: 0)
1-5 = BLOB subtype (default: 1)

ALTER TABLE

Local SQL supports the following subset of the ANSI-standard ALTER TABLE statement. You can add new columns to an existing table using this ALTER TABLE syntax:

```
ALTER TABLE table ADD column_name data_type [, ADD column_name
data_type ...]
```

For example, the following statement adds a column to a dBASE table:

```
ALTER TABLE "employee.dbf" ADD BUILDING_NO SMALLINT
```

You can delete existing columns from a table using the following ALTER TABLE syntax:

```
ALTER TABLE table DROP column_name [, DROP column_name ...]
```

For example, the next statement drops two columns from a Paradox table:

```
ALTER TABLE "employee.db" DROP LAST_NAME, DROP FIRST_NAME
```

ADD and DROP operations can be combined in a single statement. For example, the following statement drops two columns and adds one:

```
ALTER TABLE "employee.dbf" DROP LAST_NAME, DROP FIRST_NAME, ADD FULL_NAME
CHAR[30]
```

DROP TABLE

DROP TABLE deletes a Paradox or dBASE table. For example, the following statement drops a Paradox table:

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

CREATE INDEX

CREATE INDEX enables users to create indexes on tables using the following syntax:

```
CREATE INDEX index_name ON table_name (column [, column ...])
```

Using CREATE INDEX is the only way to create indexes for dBASE tables. For example, the following statement creates an index on a dBASE table:

```
CREATE INDEX NAMEX ON "employee.dbf" (LAST_NAME)
```

Paradox users can create only secondary indexes with CREATE INDEX. Primary Paradox indexes can be created only by specifying a PRIMARY KEY constraint when creating a new table with CREATE TABLE.

DROP INDEX

Local SQL provides the following variation of the ANSI-standard DROP INDEX statement for deleting an index. It is modified to support dBASE and Paradox file names.

```
DROP INDEX table_name.index_name | PRIMARY
```

The PRIMARY keyword is used to delete a primary Paradox index. For example, the following statement drops the primary index on EMPLOYEE.DB:

```
DROP INDEX "employee.db".PRIMARY
```

To drop any dBASE index, or to drop secondary Paradox indexes, provide the index name. For example, the next statement drops a secondary index on a Paradox table:

```
DROP INDEX "employee.db".NAMEX
```

CREATE VIEW

A view creates a virtual table from a SELECT statement. You can look at just the data you need within this movable frame or window on the table, while the technical underpinnings are hidden. Instead of entering a complex qualified SELECT statement, the user simply selects a view.

CREATE VIEW describes a view of data based on one or more underlying tables in the database. The rows to return are defined by a SELECT statement that lists columns from the source tables. A view does not directly represent physically stored data. It is possible to perform select, project, join, and union operations on views as if they were tables.

CREATE VIEW enables users to create views on tables by using the following syntax:

```
CREATE VIEW view_name [ (column_name [, column_name]...)]
```

CREATE VIEW is supported in conjunction with the Client Data Repository (CDR). The CDR stores the SELECT statement that defines the view.

The "WITH CHECK OPTION" is supported to create a constrained view.

Views of Views are supported. However, the CASCADE/LOCAL view attribute is not supported, because all updateable views CASCADE the constraints.

DDL examples

The following examples show the use of DDL statements with standard databases.

Example 1a: DDL (DROP TABLE)

When the table name contains a period "." character, enclose the name in quotation marks:

```
drop table "c:\data\customer.db"
```

Example 1b: DDL (DROP TABLE)

No quotation marks are used if the table name does not contain the "." character:

```
drop table clients
```

Example 2: DDL (CREATE INDEX)

```
create index part on parts (part_no)
```

Paradox: Paradox primary indexes can be created only when creating the table. Secondary indexes are created as case insensitive and maintained, when possible.

dBASE: dBASE indexes are created as maintained. The Index name specified is the tag name.

For more information about different types of indexes, see DbiAddIndex in the *Borland Database Engine Online Reference*.

Example 3: DDL (DROP INDEX)

The syntax for drop index is tablename.indexname:

```
drop index parts.part_no
```

Paradox: For Paradox only, the syntax tablename.primary indicates the primary index:

```
drop index parts.primary
```