I

Part

SQL Reference

I.1 Introduction

This chapter describes the Structured Query Language (SQL) statements and functions supported by NexusDB using the TnxQuery component. This chapter does not teach you how to use SQL. If you are not familiar with SQL, please read one of the many other available books for this purpose.

I.2 Syntax Conventions

The SQL language is completely case insensitive, except within quoted string literals. However, for our purposes, the following conventions have been used in this manual.

Convention	Use
UPPERCASE	Denotes keywords used by NexusDB SQL engine. These words may not be used as table or column names etc.
italic	Denotes user-defined parameters such as table names etc.
<*italic*>	Signifies elements that may be broken into smaller components. eg. an aggregate function
[ ]	The syntax in the brackets is optional.
…	Denotes the previous clause may be repeated several times.
|	Indicates only one of the listed options may be used.
{ }	Denotes one of the listed clauses must be used

Table 17: SQL – Syntax Conventions

I.3 Naming Conventions

Table names must meet the following requirements:

- May contain characters allowed in any Windows filename.
- Must not contain path or file names.
- May not start with a digit.

Table names beginning with a # character signify a temporary table and are stored local to the current statement block.

Alias names may also be used within SQL statements. eg.

```
SELECT S.Name
FROM Students AS S
```

Column or field names may be of any length and include spaces and international characters. Punctuation characters are not permitted. To reference a column name containing a space or keyword, use double quotes. eg.

```
SELECT "Student Name"
FROM Students
```

I.4 Data Types

The following is a list of SQL data types supported by the NexusDB SQL engine and their corresponding NexusDB data types. To convert between types, see the CAST function.

SQL Language	NexusDB Data Type	SQL Language	NexusDB Data Type
BOOL	nxtBoolean	CHAR(*int*)	nxtShortString for int < 256 and int > 1
BOOLEAN	nxtBoolean	CHAR(*int*)	nxtNullString *for int >= 256*
TINYINT	nxtByte	CHARACTER	nxtChar
BYTE	nxtByte	CHARACTER(*int*)	nxtShortString for int < 256 and int > 1
BYTEARRAY	nxtByteArray	CHARACTER(*int*)	nxtNullString *for int >= 256*
WORD	nxtWord16	VARCHAR	nxtChar
DWORD	nxtWord32	VARCHAR(*int*)	nxtShortString for int < 256 and int > 1
INT	nxtInt32	VARCHAR(*int*)	nxtNullString *for int >= 256*
INTEGER	nxtInt32	SHORTSTRING	nxtShortString
SHORTINT	nxtInt8	STRING	nxtShortString
SMALLINT	nxtInt16	ASTRING	nxtShortString
LARGEINT	nxtInt64	NULLSTRING	nxtNullString
AUTOINC	nxtAutoInc	ACHAR	nxtNullString
FLOAT	nxtSingle	BLOB	nxtBlob
REAL	nxtDouble	TEXT	nxtBLOBMemo
EXTENDED	nxtExtended	IMAGE	nxtBLOBGraphic
MONEY	nxtCurrency	NCHAR	nxtWideChar
DATE	nxtDate	NCHAR(int)	nxtWideString *for int > 1*
TIME	nxtTime	NVARCHAR	nxtWideChar
DATETIME	nxtDateTime	NVARCHAR(int)	nxtWideString *for int > 1*
CHAR	nxtChar	RECREV	nxtRecRev

Table 18: SQL Data Types

I.5 Key Words

The following is a list of reserved words recognized by the NexusDB SQL engine.

#B
#I
#L
#S
#T
ABS
ACHAR
ADD
ALL
ALTER
AND
ANY
AS
ASC
ASSERT
ASTRING
AUTOINC
AVG
BETWEEN
BLOB
BLOCK
BLOCKSIZE
BOOL
BOOLEAN
BOTH
BY
BYTE
BYTEARRAY
CASCADE
CASE
CAST
CEILING
CHAR
CHARACTER
CHARACTER_LENGTH
CHAR_LENGTH
CHR
COALESCE
COLUMN
COUNT
CREATE
CROSS
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
DATE
DATETIME
DAY
DEFAULT
DELETE
DESC
DESCRIPTION
DISTINCT
DROP
DWORD
ELSE
EMPTY
ENCRYPTION
END
ESCAPE
EXISTS
EXP
EXTENDED
EXTRACT
FALSE
FLOAT
FLOOR
FOR
FROM
FULL
GROUP
GROW
GROWSIZE
HAVING
HOUR
IDENTITY
IGNORE
IMAGE
IN
INDEX
INITIAL
INITIALSIZE
INNER
INSERT
INT
INTEGER
INTERVAL
INTO
IS
JOIN
KANA
KEY
LARGEINT
LEADING
LEFT
LIKE
LOCALE
LOG
LOWER
MATCH
MAX
MIN
MINUTE
MONEY
MONTH
NATURAL
NCHAR
NONSPACE
NOT
NULL
NULLIF
NULLSTRING
NVARCHAR
ON
OR
ORDER
OUTER
PARTIAL
PERCENT
POSITION
POWER
PRIMARY
RAND
REAL
RECREV
RIGHT
ROUND
SECOND
SELECT
SESSION_USER
SET
SHORTINT
SHORTSTRING
SIZE
SMALLINT
SOME
SORT
STRING
SUBSTRING
SUM
SYMBOLS
TABLE
TEXT
THEN
TIME
TIMESTAMP
TINYINT
TO
TOP
TRAILING
TRIM
TRUE
TYPE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USE
USER
USING
VALUES
VARCHAR
WHEN
WHERE
WIDTH
WORD
YEAR

Table 19: SQL Keywords

I.6 SQL Syntax

DDL & DML Statements

ALTER
CREATE
DELETE
DROP
INSERT
SELECT
UPDATE

Joins

CROSS
NATURAL
UNION

Aggregates

AVG
COUNT
MAX
MIN
SUM

Scalar Functions

ABS
CASE
CAST
CEILING
CHARACTER_LENGTH
CHAR_LENGTH
CHR
COALESCE
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMEST
CURRENT_USER
EXP
EXTRACT
FLOOR
IDENTITY
LOG
LOWER
NULLIF
POSITION
POWER
RAND
ROUND
SESSION_USER
SUBSTRING
SYSTEM_ROW#
TRIM
UPPER
USER

Clauses

EXISTS
UNIQUE
MATCH
BETWEEN
LIKE
IN

Switches

#B+/-
#I+/-
#L+/-
#S+/-

Table 20: SQL Statements

I.6.1 DDL and DML Statements

I.6.1.1 ALTER

```
ALTER TABLE table_name
{ ADD [COLUMN] <table_elem>
 | DROP [COLUMN] column_name
 | ALTER [COLUMN] column_name { SET DEFAULT {literal | NULL | EMPTY | CURRENT_TIME
                                      | CURRENT_DATE | CURRENT_TIMESTAMP }
                                | DROP DEFAULT } }

<table_elem> = column_name <data_type>
            [ DEFAULT { <literal> | NULL | EMPTY | CURRENT_TIME | CURRENT_DATE
                        | CURRENT_TIMESTAMP  } ]
            [NOT NULL]
            [PRIMARY KEY | UNIQUE]
```

This statement can be used to alter columns within a table, whether it is to add a new column, change an existing one or remove a column.

The following example adds a new column to the Customers table:

```
ALTER TABLE Customers
ADD COLUMN NewCol char(5)
```

The following example removes the existing column, Postcode, from the Customers table:

```
ALTER TABLE Customers
DROP COLUMN Postcode
```

The following example defines a new default value for the City column on the Customers table:

```
ALTER TABLE Customers
ALTER COLUMN City
SET DEFAULT 'Brisbane'
```

The following example removes a previously defined default value from the Postcode column of the Customers table:

```
ALTER TABLE Customers
ALTER COLUMN Postcode
DROP DEFAULT
```

I.6.1.2 CREATE

```
CREATE TABLE table_name [{ BLOCKSIZE | BLOCK } integer]
                        [{ INITIALSIZE | INITIAL } integer]
                        [{ GROWSIZE | GROW } integer]
                        [ENCRYPTION 'RegisteredEncryptionEngine']
         ( <table_elem> | <table_constraints>
            [ , <table_elem> | <table_constraints> … ] )


CREATE [UNIQUE] INDEX index_name ON table_name ( <index> [ , <index> …] )

<table_elem> = column_name <data_type>
            [ DEFAULT { <literal> | NULL | EMPTY | CURRENT_DATE | CURRENT_TIME
                        | CURRENT_TIMESTAMP } ]
            [NOT NULL]
            [PRIMARY KEY | UNIQUE]

<table_constraints> = PRIMARY KEY | UNIQUE (column_name [ , column_name …]

<index> = column_name [ ASC | DESC ] [ LOCALE integer ]
      [ IGNORE { CASE | KANA TYPE | NON_SPACE | SYMBOLS | WIDTH } ]
      [ USE STRING SORT ]
```

These statements create new tables and Indices on tables respectively.

BLOCKSIZE defaults to 4kb if not specified but can take values 4, 8, 16, 32 or 64.

The following example creates a table and corresponding index:

```
CREATE TABLE Students BLOCKSIZE 8
(Student#      SMALLINT NOT NULL PRIMARY KEY,
 StudentName   CHAR(25),
 Address       CHAR(30),
 State            CHAR(3),
 PostCode      CHAR(4),
 Gender        CHAR)

CREATE INDEX StudentNameIndex
ON Students (StudentName ASC, State IGNORE CASE)
```

I.6.1.3 DELETE

```
DELETE FROM [ database_name . ] table_name [ [AS] <sqlName>]
 [WHERE <condition>]
```

This statement deletes all rows from the table that meet the condition.

The following example removes all orders more than 2 years old:

```
DELETE FROM Orders
WHERE OrderDate < CURRENT_DATE – INTERVAL ‘2’ YEAR
```

I.6.1.4 DROP

```
DROP TABLE table_name
DROP INDEX table_name . index_name
```

This statement can be used to delete entire tables or Indices.

The following example deletes the Orders table from the database:

```
DROP TABLE Orders
```

The following example deletes an index from the Orders table:

```
DROP INDEX Orders.TakenByIndex
```

I.6.1.5 INSERT

```
INSERT INTO table_name { DEFAULT VALUES
                        | (column_name [ , column_name …] ) <table_expr>
                        | <table_expr>
```

This statement inserts records into a table using either default or specified values.

The following examples both insert records into the Students table.

```
INSERT INTO Students
(Student#, StudentName ,Address, State, PostCode, Gender)
VALUES (1234567, ‘Joanne Smith’, ‘’,‘QLD’, 4567, ‘F’)
INSERT INTO Students
VALUES (1234678, ‘James Thomas’, ‘’, ‘NSW', 2345, ‘M’),
         (1345678, 'Ann Friar', ’25 Wilson Street’, ‘QLD’, 4054, ‘F’)
```

I.6.1.6 SELECT

```
SELECT [ALL | DISTINCT] [TOP n [PERCENT]]   <selection> [, <selection> …]
   FROM <table_ref> [, <table_ref> …]
   [WHERE <condition> [{AND | OR} <condition> …] ]
   [GROUP BY <column> [, <column> …] ]
   [HAVING <condition> [{AND | OR} <condition> …] ]
   [ORDER BY <order_item> [, <order_item> …] ]
```

The SELECT statement retrieves data from one or more tables within a single database. This statement returns a result set, which can be read-only or live.

The following list briefly describes the above commands:

SELECT - Specifies the columns or data that is to be retrieved. Simply stating * retrieves all columns. TOP *n* will only return the first n rows and TOP *n* PERCENT, respectively, the first n% rows that meet selection criteria.

FROM - Specifies the tables from which the data is to be collected. You may assign an alias to each table and use the alias in the other clauses of the SELECT statement.

WHERE - Specifies the criteria for record selection. Note that in the WHERE clause **a** AND **b** OR **c** is equivalent to (**a** AND **b**) OR **c**, such that AND takes precedence.

GROUP BY - Indicates how rows are to be grouped. Rows are grouped together based upon the values in one or more columns.

HAVING - Allows you to restrict the rows that are grouped based upon the values in the group. This is similar to a WHERE clause but applies to the fields in the GROUP BY clause.

ORDER BY - Indicates how the result set is to be ordered. You may order the result set using one or more columns in either ascending or descending order.

The following example lists all orders taken today which contain at least 5 of the selected item.

```
SELECT OrderNumber, Count(*) AS Num_Items
FROM OrderDetails
WHERE ItemNumber = :selectedItemNumber
GROUP BY OrderNumber
HAVING Num_Items >= 5
ORDER BY OrderNumber
```

The following is the syntax definitions for the above fields. Note that all fields containing *_name* are user defined names and must satisfy the previously mentioned naming conventions.

```
<selection> = * | <simple_expression> [AS column_name]
SELECT *, (Price * 1.1) AS "Price incl. tax"
FROM Items

<simple_expression> = <term> [{+ | - | "||" | * | / } <term> …]  where * and / take
precedence

<term> = [-] { ( <condition> )
       | ( <select_statement> )
       | <field>
       | <literal>
       | : parameter
       | <aggregate_function>
       | <scalar_function> }

<table_ref> =  <table_expression> [ [AS] sqlName] [(<insert_column_list>)]
         | [ database_name . ] table_name [ [AS] sqlName]
SELECT t1.*, t2.Name
FROM dbname.table1 AS t1, table2 t2
WHERE t1.id = t2.id

<condition> = [NOT] { <exists_clause>
                 | <unique_clause>
                 | <relational_clause>
                 | <match_clause>
                 | <between_clause>
                 | <like_clause>
                 | <in_clause>
                 | <test_clause> }

<literal> = { TRUE | FALSE
       | float
       | integer
       | string
       | BlobString
       | DATE string
       | TIME string
       | TIMESTAMP string
       | INTERVAL string { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
                  [ TO { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND } ] }
```

```
<table_expression> = <non_join_table_expression> [ UNION <non_join_table_expression> ]
               | <join_expression>

<non_join_table_expression> = <select_statement>
                        | (<table_expression>) }
                        | TABLE <table_ref>
                        | VALUES ( <value_list> [ , <value_list> …] )
```

```
SELECT Name, BestScore
FROM Students, (SELECT StudentID, Max(Grade) AS BestScore
                  FROM Results) AS B
WHERE Students.StudentID = B.StudentID
```

```
<value_list> = { DEFAULT | NULL | <simple_expression> }
         [ , { DEFAULT | NULL | <simple_expression> } …]

<order_item> = { <column> | integer } [ ASC | DESC ] [ LOCALE integer ]
                  [ IGNORE { CASE | KANA TYPE | NON_SPACE | SYMBOLS | WIDTH } ]
                  [ USE STRING SORT ]

<column> = [ table_name . ] field_name [ [ AS ] sql_name ]

<field> = [ table_name . ] field_name

<literal> = float | integer | string
   | DATE string
   | TIME string
   | TIMESTAMP string
   | INTERVAL string {YEAR | MONTH | DAY | HOUR | MINUTE | SECOND}
         [TO {YEAR | MONTH | DAY | HOUR | MINUTE | SECOND}]
   | 'TRUE' | 'FALSE'
```

I.6.1.7 UPDATE

```
UPDATE [ database_name . ] table_name [ [ AS ] sql_name ]
 SET <update_item> [ , <update_item>.]
 [WHERE <condition>]

<update_item> = column_name = DEFAULT | NULL | <simple_expression>
```

This statement updates items all rows from the table that meet the condition.

The following example gives all female employees a 5% raise:

```
UPDATE Employees
SET Salary = Salary * 1.05
WHERE Gender = 'F'
```

I.6.2 Joins

```
JOIN
<join_table> CROSS JOIN <table_ref>

<join_table> [NATURAL] [ INNER
                        | LEFT [ OUTER ]
                        | RIGHT
                        | FULL [ OUTER ] ]
      JOIN <join_table> [ { ON <condition>
```

```
                         | USING (field_name [, field_name ..] ) } ]

<join_table> = [ database_name . ] table_name [ [AS] sql_name ]
         | ( <table_expression> ) [ [AS] sql_name ]
```

Fundamentally, a join is a cross product of the rows (records) from all participating tables. That is, all possible combinations of all rows from the tables. In SQL, joins can be specified in two different ways: implicitly, or explicitly with the JOIN keyword. The following two statements are equivalent.

```
SELECT * FROM table1, table2                implicit join
SELECT * FROM table1 CROSS JOIN table2      explicit join
```

Generally, the ON clause of the JOIN looks and works just like a WHERE clause on a SELECT. However, an ON clause must specify one or more relations between the two tables being joined

The following example lists students and the subjects they are enrolled in.

```
SELECT student_name, subject_code
FROM students AS S JOIN enrolments
      ON S.Stud# = enrolments.Stud#
ORDER BY student_name
```

For joins where the names of the columns being joined on are the same, there's a more compact way to express the same join. The result of a JOIN with a USING clause differs slightly from one with an ON clause, with the joining columns only listed once each.

```
SELECT student_name, subject_code
FROM students JOIN enrolments
      USING (Stud#)
ORDER BY student_name
```

NATURAL joins are similar to USING joins with all columns of the same name paired.

```
SELECT *
FROM students as S NATURAL JOIN enrolments
```

The three previous join types discussed (NATURAL, JOIN ON, JOIN USING) are implicitly calculated as inner joins. This can also be explicitly specified by using the optional INNER keyword.

The difference between inner and outer joins has to do with NULL values. If no NULL values are involved in a join, then an outer join is equivalent to its inner form. If NULL values occur in any of the columns participating in the join expression, there will be no rows in the result table for those source rows. Generally in SQL, comparing a value of NULL to anything else (including another NULL) gives an undefined result.

Sometimes, however, it is necessary to get a row in the result that represents missing information in the source tables. An outer join allows you to specify that rows from either the left, right or both tables in the join, with no matching record in the 'opposite' table, should appear in the result.

The following example list all students and subjects they are enrolled in. Students not enrolled in any subjects will also be selected.

```
SELECT student_name, subject_code
FROM students LEFT OUTER JOIN enrolments USING (Stud#)
ORDER BY student_name
```

Note that all three cases (LEFT, RIGHT, and FULL) all imply outer joins, the OUTER keyword is actually obsolete and can be omitted.

I.6.3 Aggregate Functions

I.6.3.1 AVG

```
AVG([ALL | DISTINCT] <simple_expression>)
```

Calculates the average of numeric values in a column or expression.

This aggregate function can be used to calculate the average of values for a specific column or expression. All values in the column or expression must be numeric. Use the DISTINCT keyword to remove duplicate values before evaluating.

NULL values are excluded from all calculations. Note that if the return set has no rows, AVG will return NULL.

The following example calculates the average grade for each student:

```
SELECT StudenNum, AVG(Grade)
FROM Enrolments
GROUP BY Student
```

I.6.3.2 COUNT

```
COUNT([ALL | DISTINCT] <simple expression> )
```

Returns a count of the number of non-NULL values in the rows retrieved by a SELECT statement.

```
COUNT(*)
```

Returns the number of rows including NULL values.

COUNT(*) can be optimised to return very quickly if the SELECT retrieves from one table, no other columns are retrieved and there is no WHERE clause.

For example:

```
SELECT COUNT(*) FROM Students
```

The following example returns the number of female students:

```
SELECT COUNT(StudName)
FROM Students
WHERE Gender = "F"
```

If you specify the DISTINCT keyword, COUNT eliminates all duplicate rows. The following example returns the number of students enrolled in subjects:

```
SELECT COUNT(DISTINCT StudNum)
FROM Enrolments
```

I.6.3.3 MAX, MIN

```
MAX([ALL | DISTINCT] <simple_expression> )
```

Retrieves the largest value in a column or expression.

```
MIN([ALL | DISTINCT] <simple_expression> )
```

Retrieves the smallest value in a column or expression.

These functions can be used to find the largest and smallest values respectively in an expression. The expression can be made up of a constant, column or non-aggregate function.

These functions exclude rows having NULL values. If a query does not return any rows, the function returns NULL. The following example selects the highest and lowest salaries of employees:

```
SELECT MAX(Salary), MIN(Salary)
FROM Employees
```

I.6.3.4 SUM

```
SUM([ALL | DISTINCT] <simple_expression> )
```

Returns the sum of the numeric expression.

This aggregate function can be used to calculate the sum of values for a specific column or expression. All values in the column or expression must be numeric.

NULL values are excluded from all calculations. Note that if the return set has no rows, SUM will return NULL.

The following example calculates the total cost of each order made:

```
SELECT OrderNum, SUM(ItemPrice)
FROM Orders
GROUP BY OrderNum
```

I.6.4 Scalar Functions

I.6.4.1 ABS

```
ABS(<simple_expression>)
```

Calculates the absolute value of the numeric expression.

This mathematical function can be used to calculate the absolute value for a specific column or expression. All values in the column or expression must be numeric

For example:

```
SELECT ABS(5)     would return 5, and
SELECT ABS(-3)    would return 3.
```

I.6.4.2 CASE

```
CASE WHEN <case_expression_1> THEN <when_exp_1>
 [WHEN <case_expression_n> THEN <when_exp_n>…]
 [ELSE <else_exp>]
 END
```

The CASE function is similar to a nested IF function. The result is returned for the first condition, which is true. If there was no case_expression that returns true, then the expression after ELSE is returned. If there is no ELSE part then NULL is returned.

The following example displays the student name and gender as a string:

```
SELECT StudentName,
      CASE WHEN Gender = “F” THEN “Female”
            WHEN Gender ‘= “M” THEN “Male”
            ELSE “Unspecified” END
FROM Students
```

I.6.4.3 CAST

```
CAST(<simple_expression> AS data_type)
```

The CAST function converts a value in one internal storage format to another. The following example selects all records from the table Customers where the postcode column starts with '40'.

```
SELECT *
FROM Customers
WHERE CAST(PCode AS char(2)) = ‘40'
```

I.6.4.4 CEILING

```
CEILING(<simple_expression>)
```

Calculates the smallest integer value not less than the expression. This function can be used to round a number up to the nearest integer.

For example:

```
SELECT CEILING(5)      would return 5,
SELECT CEILING(5.2)       would return 6, and
SELECT CEILING(-3.5)      would return –3.
```

I.6.4.5 CHARACTER_LENGTH, CHAR_LENGTH

```
CHARACTER_LENGTH(<simple_expression>)
CHAR_LENGTH(<simple_expression>)
```

Returns the number of characters in a column or simple expression.

This statement will only be executed if expression is a string.

The following example finds all students who have a longer last name than first name:

```
SELECT FirstName, LastName
FROM Students
WHERE CHAR_LENGTH(LastName) > CHARACTER_LENGTH(FirstName)
```

I.6.4.6 CHR

```
CHR(<integer_expression>)
```

Returns a character corresponding to the value of the argument expression. The result is compatible with all NexusDB character types.

The following example returns all customer rows with a CR/LF pair in the Memo column.

```
SELECT *
```

```
FROM Customers
WHERE Memo LIKE '%' || CHR(13) || CHR(10) || '%'
```

I.6.4.7 COALESCE

```
COALESCE(<simple_expression> [, <simple_expression> …])
```

Returns the first non-NULL expression in the list. NULL is returned if all expressions are NULL. Each expression may be a column, constant, or function

The following example returns a single contact number for each student if available:

```
SELECT StudentName, COALESCE(HPhone, BBhone, Mobile)
FROM Students
```

I.6.4.8 CURRENT_DATE

```
CURRENT_DATE
```

Returns the current date of the NexusDB Server formatted based on the client settings.

The following example finds all future appointments:

```
SELECT *
FROM Appointments
WHERE ApptDate >= CURRENT_DATE
```

I.6.4.9 CURRENT_TIME

```
CURRENT_TIME
```

Returns the current time of the NexusDB Server formatted based on the client settings.

The following example finds all remaining appointments for today:

```
SELECT *
FROM Appointments
WHERE ApptDate = CURRENT_DATE AND ApptTime >= CURRENT_TIME
```

I.6.4.10 CURRENT_TIMESTAMP

```
CURRENT_TIMESTAMP
```

Returns the current date and time of the NexusDB server formatted based on the client settings.

The following example selects all new appointments made today:

```
SELECT *
FROM Appointments
WHERE BookingDate BETWEEN CURRENT_DATE AND CURRENT_TIMESTAMP
```

I.6.4.11 CURRENT_USER

```
CURRENT_USER
```

See User

I.6.4.12 EXP

```
EXP(<simple_expression>)
```

Returns the value of ***e*** (the base of natural logarithms) raised to the power of the numeric expression.

For example:

```
SELECT EXP(2)     would return 7.389056, and
SELECT EXP(-3)    would return 0.049787.
```

I.6.4.13 EXTRACT

```
EXTRACT({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND}
 FROM <simple_expression>))
```

Returns the specified portion of any field or expression of type Date, Time or TimeStamp.

The following example groups students by their year of birth:

```
SELECT StudentName, EXTRACT(YEAR FROM Dob) AS YearOfBirth
FROM Students
GROUP BY YearOfBirth
```

I.6.4.14 FLOOR

```
FLOOR(<simple_expression>)
```

Calculates the largest integer value not greater than the expression. This function can be used to round a number down to the nearest integer.

For example:

```
SELECT FLOOR (5)      would return 5,
SELECT FLOOR (5.2)    would return 5, and.
SELECT FLOOR (-3.5)      would return -4.
```

I.6.4.15 IDENTITY

```
IDENTITY
```

Returns the most recent autoinc value generated by the current SQL session. Assuming there's a column of type autoinc (not explicitly shown), the following example inserts two records in a table and links the second to the previous one via the Previous column:

```
INSERT INTO FamilyMembers (Name)
VALUES ('George Smith')
INSERT INTO Customers (Name, Father)
VALUES ('Bill Smith', IDENTITY)
```

I.6.4.16 LOG

```
LOG(<simple_expression>)
```

Returns the natural logarithm of the numeric expression.

For example:

```
SELECT LOG(2)     would return 0.693147
```

I.6.4.17 LOWER

```
LOWER(<simple_expression >)
```

Returns the expression string with all characters changed to lower case.

For example:

```
SELECT LOWER("Hello")          would return "hello", and
SELECT LOWER("QLD, 4000")      would return "qld, 4000".
```

I.6.4.18 NULLIF

```
NULLIF(<simple_expression >, <simple_expression>)
```

Returns NULL if expression 1 and 2 are equal.

This is equivalent to:

```
IF expr1 = expr2 THEN NULL ELSE expr1
```

The following example returns null for grades equal to 1:

```
SELECT Student#, NULLIF(grade,1)
FROM Students
```

I.6.4.19 POSITION

```
POSITION(<simple_expression > IN <simple_expression>)
```

Returns the position of the first occurrence of the first string expression in the second string expression. This function is case-sensitive. If the substring is not found within the string, 0 is returned.

The following example lists all subjects starting with "MAB":

```
SELECT *
FROM Subjects
WHERE POSITION('MAB' IN SubjectCode) = 1
```

I.6.4.20 POWER

```
POWER(<simple_expression >, <simple_expression>)
```

Returns the value of the first numeric expression raised to the power of the second numeric expression.

For example:

```
SELECT POWER(2, 3)     would return 8, and
SELECT POWER(3.5, 2.2)     would return 15.738.
```

I.6.4.21 RAND

```
RAND
```

Returns a random floating point number between 0 and 1.0.

For example:

```
SELECT RAND        could return 0.7482, and on repeat
SELECT RAND       could return 0.6445.
```

I.6.4.22 ROUND

```
ROUND(<simple_expression>)
```

Returns the expression rounded to the closest integer.

For example:

```
SELECT ROUND(7.6)    would return 8,
SELECT ROUND(2.3)    would return 2, and
SELECT ROUND(-1.3)   would return -1.
```

I.6.4.23 SESSION_USER

```
SESSION_USER
```

See User

I.6.4.24 SUBSTRING

```
SUBSTRING(<simple_expression> FROM <position> [ FOR <length>])
```

Returns a substring from a string expression.

For example:

```
SELECT SUBSTRING("Hello World" FROM 3 FOR 5)
would return "llo W"
```

The following example can be used to get last name of a customer.

```
SELECT SUBSTRING(Name FROM POSITION(' ' IN Name))
FROM Customer
```

I.6.4.25 SYSTEM_ROW#

```
SYSTEM_ROW#
```

When specified as a column in a query, the result of this function is the sequential (zero-based) row number in the result set. This may be used to generate a unique row-id for client-side controls that require it. The data type of SYSTEM_ROW# is nxtWord32. Note that, currently, SYSTEM_ROW# has no effect in grouped queries.

```
SELECT SYSTEM_ROW#, customer_name
FROM Customers
```

I.6.4.26 TRIM

```
TRIM( [LEADING | TRAILING | BOTH] [<simple_expression> FROM ]
<simple_expression>)
```

TRIM removes leading and/or trailing characters from a string.

This function is used to remove characters from the beginning and/or end of a string. If two expressions are specified, those characters of the first are trimmed from the second. Otherwise, the space character is removed. By default, the BOTH keyword is implemented if nothing is specified.

```
SELECT StudentName,
 TRIM(LEADING 'D' FROM StudentName),
 TRIM(TRAILING 'e' FROM StudentName),
 TRIM(BOTH 'A' FROM StudentName),
 TRIM('e' FROM StudentName),
 TRIM(BOTH FROM StudentName),
 TRIM(StudentName)
FROM Students
```

I.6.4.27 UPPER

```
UPPER(<simple_expression>)
```

Returns the expression string with all characters changed to upper case.

For example:

```
SELECT UPPER("Hello")      would return "HELLO"
```

The following example lists all states from which students live.

```
SELECT DISTINCT UPPER(State)
FROM Students
```

I.6.4.28 USER

```
USER
SESSION_USER
CURRENT_USER
```

Returns the current user name from the session object.

The following example finds all appointments made by the current user:

```
SELECT *
FROM Orders
WHERE TakenBy = CURRENT_USER
```

I.6.5 Clauses

I.6.5.1 EXISTS

```
EXISTS (<select_statement>)
```

The following example finds all students enrolled in more than 2 subjects.

```
SELECT student_name
FROM students S
WHERE EXISTS (SELECT COUNT(*)
                  FROM enrolls
                  WHERE student# = S.student#
                  HAVING COUNT(*) > 2)
```

I.6.5.2 UNIQUE

```
UNIQUE (<table_exp>)
```

The following example selects enrolled in more than 1 subject

```
SELECT student_name
FROM students AS S
WHERE NOT UNIQUE (SELECT student#
                         FROM enrolls
                         WHERE student# = s.student#)
```

I.6.5.3 MATCH

```
MATCH [ UNIQUE ] [ PARTIAL | FULL ] (<select_statement>)
```

```
SELECT *
FROM students
WHERE student# MATCH UNIQUE FULL (SELECT student#
                                          FROM enrolls)
```

I.6.5.4 BETWEEN

```
BETWEEN <simple_expression> [ ESCAPE <simple_expression> ] [ IGNORE CASE ]
```

The following example selects all employees that earn between $30,000 and $35,000.

```
SELECT teacher_name, salary
FROM teachers
WHERE salary BETWEEN 30000 AND 35000
```

I.6.5.5 LIKE

```
LIKE <simple_expression> [ ESCAPE <simple_expression> ] [ IGNORE CASE ]
```

The following example selects all students whose first name starts with "Jo"

```
SELECT student_name, gender
FROM students
WHERE student_name LIKE 'Jo%'
```

I.6.5.6 IN

```
IN ( <simple_expression> | <select_statement> )
```

This example selects students from NSW and QLD

```
SELECT student_name, course_name, state
FROM students, enrolls, courses
WHERE state in ('QLD', 'NSW') AND
 students.student# = enrolls.student# AND
 enrolls.course# = courses.course#
ORDER BY course_name, state
```

The next example finds students with the lowest GPA

```
SELECT student_name, gpa
FROM gpas
WHERE gpa IN (SELECT MIN(gpa) FROM gpas);
```

I.6.5.7 <relational_clause>

```
<simple_expression> {= | >= | > | < | <= | <> }
                [ ALL | ANY | SOME ] { <select statement> | <simple_expression> }
```

The following example finds students that have received a 3 in any subject.

```
SELECT student_name
FROM students
WHERE student# = ANY (SELECT student#
                           FROM enrolls WHERE grade = 3);
```

I.6.5.8 <test_clause>

```
<simple_expression> IS [NOT] {NULL |TRUE | FALSE | UNKNOWN}
```

The following example finds all students with information in both PCode and city fields.

```
SELECT Student_name
FROM students
WHERE COALESCE (PCode, city) IS NOT NULL
```

I.6.6 Switches

I.6.6.1 #B+/-

```
#B+ <select_statement>
#B- <select_statement>
```

This switch controls the behavior when copying columns that contain BLOBs. In the default #B- state, the query engine only copies a link to the original BLOB in the result set which can save on both memory and execution time. The exception to this is if the source table is a temporary table, in which case the query engine will copy the full BLOB from the temporary table to the result set. If switched on (#B+) the query engine will always copy the full BLOB to the result set. Since NexusDB allows for query results drawn on tables that are dropped within the same statement block, it is possible to write a SQL block so that BLOB links would refer to a table which had since been dropped, so understanding the implications of this setting is important.

```
SELECT * FROM BlobTable;
DROP BlobTable;
```

In this case, the SELECT must be prefixed with #B+ to force BLOBs to be copied from BlobTable into the query result.

I.6.6.2 #I+/-

```
#I+ <select_statement>
#I- <select_statement>
```

Indexing optimisation turns the use of Indices on and off for the following statement. This switch defaults to #I+ (on), and is usually only turned off when trying to troubleshoot the query optimizer.

I.6.6.3 #L+/-

```
#L+ <select_statement>
#L- <select_statement>
```

#L+ turns on logging for the statement block as a whole and should be placed before any other non-comment line. When logging is enabled, (default is off) the query engine collects information about simplifications, index use, and query metrics, and builds a text report which is returned to the client with the query result and status. The report can be accessed through the Log property of the nxQuery component and can be used to analyze the query for opportunities to rephrase statements for better performance.

I.6.6.4 #S+/-

```
#S+ <select_statement>
#S- <select_statement>
```

#S- turns query simplification off for the statement that follows the switch (default is on). Query simplification is a process where the structure of a parsed query is simplified. Eg. x BETWEEN y AND z would be simplified to x >= y AND x <= z .

As simplification occurs before the SQL statement is bound to the tables it works on, occasionally, simplification can convert a query into a less efficient alternative. For such cases, the #S- switch allows you turn simplification off. When logging is enabled (see #L+), the individual stages in the simplification process are logged.

I.7 General Information

I.7.1 Temporary Tables

NexusDB/SQL supports the notion of temporary tables, but the syntax for creating and accessing them is different from what is stipulated in the SQL-92 standard. In NexusDB/SQL, a table whose name starts with the pound symbol (#) is by definition a temporary table - visible only to the current SQL session. Most operations performed on temporary tables are significantly faster than the same operations on permanent tables.

I.7.2 Multi-statement blocks

NexusDB/SQL supports multi-statement blocks. Any number of SQL statements may be executed in a single ExecSQL or Open command. As you would expect, statements are executed in a sequential fashion. If an exception occurs in a statement, the following statements are not executed. For multi-statement blocks that generate more than one query result, only the last one is returned to the client. NexusDB/SQL fully supports query results drawn on tables that are dropped within the same statement block.

I.7.3 Blob literals

NexusDB/SQL uses a special syntax to allow the content of BLOB columns to be expressed as literals. A BLOB literal is a string of hexadecimal digits enclosed in square brackets. As each BLOB byte is encoded as two hex digits, the total number of hex digits in a BLOB literal must always be even.

Example:

```
INSERT INTO SomeTable (Key, BlobField1, BlobField2)
VALUES(1, [ABCDEF123456], [010101FF]);
```

I.7.4 Querying the data dictionary

NexusDB represents the most important data structures of its internal data dictionary as a set of virtual tables to the SQL engine. This makes it possible to run queries against the structure of the database using normal SQL syntax. The data dictionary is defined through the five tables, #TABLES, #FIELDS, #INDICES, #INDEXFIELDS, and #FILES. The following table defines the contents of each table:

Table	Content
#TABLES	Contains a row for each table in the dictionary. Does not include tables belonging to the virtual dictionary itself.
#FIELDS	Contains a row for every defined column in every table.
#INDICES	Contains a row for each defined index - both sequential and user-defined.
#INDEXFIELDS	Contains a row for each key segment in composite indices.
#FILES	Contains a row for each physical Nexus database file in the current database.

For example:

To get a list of all tables in the current database, you can do

```
SELECT TABLE_NAME FROM #TABLES;
```

To get a list of columns for a particular table, say 'Customers', you can do

```
SELECT FIELD_NAME FROM #FIELDS WHERE TABLE_NAME = 'Customers';
```