ThinkSQL RDBMS

User Guide

25th January 2004

Contents

1. Introduction

This manual introduces the SQL database query language as implemented by the ThinkSQL® Relational Database Management System (RDBMS). ThinkSQL uses ISO standard SQL, but existing knowledge of SQL from another database system would be useful. It's assumed that the reader is familiar with basic database concepts (e.g. tables and columns) and so database design theory (e.g. third normal form) won't be discussed here.

The example database

Most of the examples in this manual will be based on the restaurant example schema. This schema will hold information about a restaurant's menu items, sales and staff. It has some obvious limitations, but will serve its purpose as a context for the examples. It consists of the following base tables:

MENU_ITEM_GROUP	Holds details about each group of menu items
MENU_ITEM	Holds details about each menu item
SERVER	Holds details about each server
CHECK_HEADER	Holds details about each check
CHECK_DETAIL	Holds details about each check line

The schema definition and sample data can be found in appendix A.

2. An overview of SQL

SQL is an easy to learn but powerful language. It's the standard language for communicating with databases. Many of its commands can be used in a basic way and then made more complex by adding extra clauses or nesting statements within others.
Here's a summary of the most common SQL commands: tables are created (using `CREATE TABLE`); data is entered into the tables (using `INSERT`); data in tables can be modified or deleted (using `UPDATE` or `DELETE`); data can be queried and retrieved (using `SELECT`).

Syntax

A few of the more important rules of SQL syntax are:

- SQL keywords are not case sensitive. For our examples, we will use upper case for SQL keywords and user identifiers for clarity.
- Commands can be spread over multiple lines and so, for dumb terminal clients (e.g. Telnet) and batches of commands, a semicolon is required to signal the end of a command.
- SQL identifiers, e.g. names of tables and columns, must begin with a letter and can then contain letters, digits and the underscore character. They are not case sensitive, and they can't be the same as any of the SQL keywords, unless double quotes are used around the identifier (e.g. "SERVER").
- Single quotes are used to surround the non-keyword parts of commands that represent character string literals.

Terminology

Data management has been around for many years and the field is awash with terms, many of which are vaguely defined and overlap (especially 'database'!). We will use a few terms in this guide interchangeably, so we'll define the important ones here.

Server/database server
An instance of the ThinkSQL server application, or the computer it's running on.

Catalog/database
A named group of schemas that a server makes available to clients. Each ThinkSQL catalog is stored in a single operating system file, e.g. db1.dat.

Schema/database
A named group of objects, e.g. tables and constraints, that are owned by a user. We often use 'schema' to mean the schema design or definition, and 'database' to mean the schema complete with its data.

User/authorization identifier
A name within a catalog used to identify a client's connection to the server and its privileges on objects within the catalog. A user can own multiple schemas and has one default schema to which it initially connects.

3. The ThinkSQL Server

The server currently runs on Windows® or Linux®. Ensure the database server is running before trying to connect.

The server listens on TCP/IP with the service name of ‘thinksql’. If this service name does not exist then the default port of 9075 is used.

If the client is on the same computer as the server then use the host name of ‘localhost’, otherwise use the IP address or host name of the server.

Windows

Starting the server

Run the server monitor (ThinkSQLmonitor.exe), which provides a simple way to launch and shutdown the server process. A screen similar to the following will be displayed:

Clicking the 'start' button will attempt to start the server as a separate process with a primary (default) catalog of db1.

The monitor will poll the server every ten seconds to see if it’s still running or not. The frequency of these checks can be adjusted using the slider at the top of the monitor. The values go from 0 on the left (no polling) to 60 on the right (poll every sixty seconds).

Alternatively, you can start the server without using the monitor by running ThinkSQL.exe.

Note: the server executable, the db1.dat catalog file, and the licence file all need to be in the current working directory. Temporary files may need to be created in this directory while the server is running.

Stopping the server

Clicking the ‘stop’ button in the server monitor will disconnect any clients and shutdown the server process.
Note: Closing the monitor does not shutdown the server: the server runs as a separate process.

Alternatively, you can connect to the primary catalog on the server via Telnet or the ISQraw program as the ADMIN user and issue the shutdown command, e.g.

```
CONNECT TO '' USER 'ADMIN' PASSWORD 'admin';
SHUTDOWN
```

Linux

Starting the server

Start the server process using:

```
./ThinkSQL
```

To run the process in the background you can add an ampersand, e.g.

```
./ThinkSQL&
```

Note: the server executable, the db1.dat catalog file, and the licence file all need to be in the current working directory. Temporary files may need to be created in this directory while the server is running.

Stopping the server

Connect to the primary catalog on the server via Telnet or the ISQraw program as the ADMIN user and issue the shutdown command, e.g.

```
CONNECT TO '' USER 'ADMIN' PASSWORD 'admin';
SHUTDOWN
```

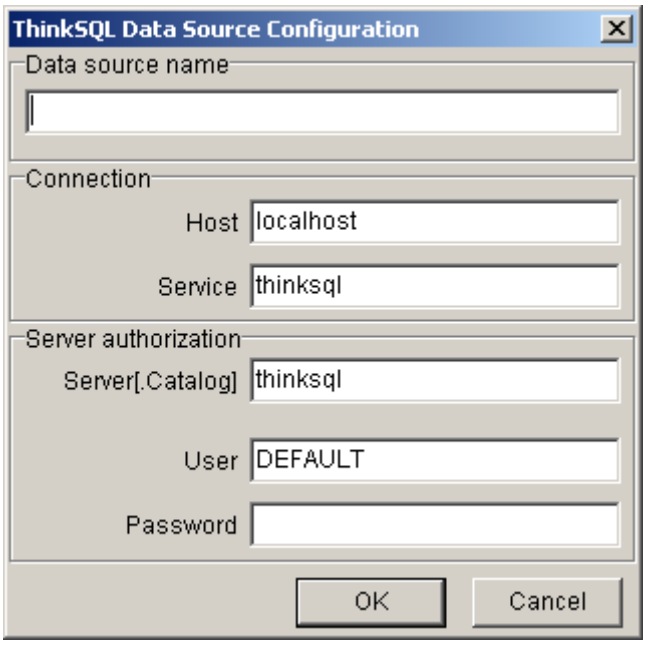
4. Clients

The database server uses a number of open interfaces to support many different types of client application. Each one is briefly described, together with the connection parameters available.

ODBC

This has been adopted as the standard SQL CLI. For details of the API, see the reference at http://www.microsoft.com/data/odbc. The ThinkSQL ODBC driver communicates directly with the server.

Adding a new ThinkSQL ODBC Data Source provides the following parameter configuration dialog:



JDBC™

For details on the JDBC API, see the reference at http://www.java.sun/jdbc. The ThinkSQL JDBC driver is written in pure Java™ (i.e. a type 4 driver), uses a subprotocol named thinksql, and communicates directly with the server.

The following details are required to configure a JDBC connection to ThinkSQL:

Driver class name: uk.co.thinksql.ThinkSQLDriver

URL for local default port: jdbc:thinksql://localhost:9075

The URL can also have an optional server[.catalog] appended after the port to connect to a particular catalog, e.g.

jdbc:thinksql://localhost:9075/.db2

The following can be passed as properties to the driver's connect method:

username
password

dbExpress™

This is a Borland® specification for use from Delphi™/Kylix™. The dbExpress components can be used to access the server. The ThinkSQL dbExpress driver is written in Delphi/Kylix and communicates directly with the server.

When developing applications with Delphi, the entire dbExpress driver can be statically linked into the client application to simplify deployment.

The following parameters can be specified to configure a dbExpress connection to ThinkSQL:

DriverName:	ThinkSQL
BlobSize:	-1
Database:	thinksql
HostName:	localhost
Password:	
User_Name:	DEFAULT

The Database parameter takes the form server[.catalog] to allow connection to a specific catalog.

Python DB-API

This specification is for use from the Python programming language. See the reference at http://www.python.org/peps/pep-0249.html for details. This module is written in pure Python and communicates directly with the server.

To use the module, `import ThinkSQL` and create a connection with the `connect()` function. This takes the following parameters:

dsn	
user	DEFAULT
password	
host	localhost
database	
port	9075

e.g. `con=connect(user='DEFAULT', host='serverMachine')`

Terminal-based (Direct SQL)

Commands are sent to the server and results are returned as formatted text for displaying on a terminal.

For the examples below, use a dumb-terminal program such as Telnet or the ISQLraw sample program that's included with ThinkSQL.

Starting the terminal client

When using the ISQLraw program, first select the Connection|Connect menu option to open a network connection. (If the server is not running on the same computer, first specify the host name in the Connection|Options dialog.). After typing a command, click the Execute button to send it to the server.

If using Telnet, set the 'local-echo' terminal preference option, and connect to the port 'thinksql' (or 9075 if you don't have a service-port mapping configured), e.g.

```
telnet localhost 9075
```

Press return to receive the welcome message from the server. Since commands can span multiple lines, you must add a semi-colon (;) and press Return to send it to the server for processing. When the server sees `CREATE PROCEDURE`, `CREATE FUNCTION` or `CREATE SCHEMA` it suspends treating the semi-colon as the command terminator, since a block of commands could be included each ending in a semi-colon. Instead, it waits for a full-stop (.) on a new line to mark the end of the command, e.g.

```
CREATE FUNCTION checksToday() RETURNS INTEGER
BEGIN
        DECLARE n INTEGER;

        SELECT COUNT(*) INTO n FROM restaurant.check_header
        WHERE CAST(start_time AS DATE)=CURRENT_DATE;

        RETURN n;
END;
.
```

When using the ISQLraw program, the full-stop is automatically added to the end of your command before it is sent to the server.

Connecting to the database server

The welcome message for terminal-based clients will be similar to:

```
Welcome to THINKSQL. 27 February 2003 10:32:27
ThinkSQL Relational Database Management System
Copyright © 2000-2003 ThinkSQL Ltd
Version 00.01.00

Licensed to any developer for up to 5 concurrent connections
```

To connect to the database server, use the `CONNECT` command. To connect to the example schema, type:

```
CONNECT TO 'THINKSQL' USER 'RESTAURANT'
```

'THINKSQL' is the server name (this can be left blank e.g. '').
'RESTAURANT' is an 'authorization identifier' in the example database and it is the owner of the restaurant schema (it needs no password).

This connect command will create a new connection and will allow commands to be sent to the database server and results to be sent back.

See Server connections for more details on the connect command.

5. Getting information

This section discusses how to extract information from an existing database. The examples given all use the RESTAURANT sample schema.

Basic retrieval

To retrieve information from a table, use the `SELECT` command. This command specifies which table to examine (using a `FROM` clause) and which columns to retrieve.

To list details of all menu items:

```
SELECT MENU_ITEM_NO, MENU_ITEM_NAME, MENU_ITEM_GROUP_NO, PRICE
FROM MENU_ITEM
```

This will display:

```
|MENU_ITEM_NO|                          MENU_ITEM_NAME|MENU_ITEM_GROUP_NO|  PRICE
================================================================================
|           1|                          Soup of the Day|                 1|   2.45
|           2|                                 Samosas|                 1|   1.95
|           3|                           Prawn Cocktail|                 1|   2.95
|         100|                                 Lasagne|                 2|   4.95
|         101|                     Spaghetti Bolognese|                 2|   1.95
|         102|                                  Paella|                 2|   6.95
|         103|                                  Borsch|                 2|   4.95
|         104|                              Irish stew|                 2|   3.95
|         105|                                Kedgeree|                 2|   4.99
|         106|                         Boeuf Bourgignone|                 2|   7.95
|         107|        Roast Beef and Yorkshire Pudding|                 2|   6.45
|         108|                            Chicken Madras|                 2|   5.45
|         109|                     Chicken Tikka Masala|                 2|   5.95
|         200|                              Boiled Rice|                 3|   1.95
|         201|                            Mashed Potato|                 3|   1.45
|         300|                                 Red Wine|                 7|   2.10
|         301|                                White Wine|                 7|   2.10
|         302|                                  Guinness|                 6|   2.99
|         303|                                     Lager|                 6|   1.85
|         304|                                  Lemonade|                 5|   1.20
|         305|                                      Cola|                 5|   1.20
|         400|                                 Ice Cream|                 4|   2.99
22 rows affected
```

In the above example, we specified each column (and so the order of the columns from left to right) that we wanted returned. In this case we specified all columns. A shorthand for ‘select all columns’ is to use an asterisk, as in:

```
SELECT *
FROM MENU_ITEM
```

Which will display exactly the same results as the previous command.

Filtering results

To select certain rows and ignore others, you can add a `WHERE` clause to this basic `SELECT` command. For example, to list details of menu items that have a price of more than five pounds:

```
SELECT *
FROM MENU_ITEM
WHERE PRICE>5.00
```

Will display just the following rows:

```
|MENU_ITEM_NO|                          MENU_ITEM_NAME|MENU_ITEM_GROUP_NO|  PRICE
================================================================================
|         102|                                 Paella|                 2|   6.95
|         106|                         Boeuf Bourgignone|                 2|   7.95
|         107|        Roast Beef and Yorkshire Pudding|                 2|   6.45
|         108|                           Chicken Madras|                 2|   5.45
|         109|                     Chicken Tikka Masala|                 2|   5.95
5 rows affected
```

The condition following the `WHERE` keyword can use the following comparison operators:

Comparison operator	Description
`=`	Equal to
`<`	Less than*
`>`	Greater than*
`<=`	Less than or equal to*
`>=`	Greater than or equal to*
`<>`	Not equal to
`BETWEEN x AND y`	Within a range x..y (inclusive and x<=y) *
`LIKE`	Character (or binary) comparison that can use the following wildcards: _ any single character (X'5F' for binary) % zero or more characters (X'25' for binary)
`IS NULL`	Test for missing data. Note: if missing data is compared using any other operator, the result is 'UNKNOWN' which means it won't generally match the filter condition. In some situations this can lead to unexpected results.
`IS NOT NULL`	Test for data

* not applicable to binary strings

Sorting results

To sort the results of a `SELECT` statement you can add an `ORDER BY` clause and specify the column or columns to sort by. For example, to sort the previous query's results by price:

```
SELECT *
FROM MENU_ITEM
WHERE PRICE>5.00
ORDER BY PRICE
```

Will display the following sorted rows:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                       |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|         108|Chicken Madras                       |                 2|   5.45
|         109|Chicken Tikka Masala                 |                 2|   5.95
|         107|Roast Beef and Yorkshire Pudding     |                 2|   6.45
|         102|Paella                               |                 2|   6.95
|         106|Boeuf Bourgignone                    |                 2|   7.95
5 rows affected
```

The default sort direction is in ascending order (`ASC`) but this can be reversed by specifying `DESC` after the column name in the `ORDER BY` clause. For example, to sort the previous query's results by descending price and then ascending menu item name within each price:

```
SELECT *
FROM MENU_ITEM
WHERE PRICE>5.00
ORDER BY PRICE DESC, MENU_ITEM_NAME
```

Will display the following sorted rows:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
==========================================================================
|         106|Boeuf Bourgignone                       |                 2|   7.95
|         102|Paella                                  |                 2|   6.95
|         107|Roast Beef and Yorkshire Pudding        |                 2|   6.45
|         109|Chicken Tikka Masala                    |                 2|   5.95
|         108|Chicken Madras                          |                 2|   5.45
5 rows affected
```

Advanced filtering

You can add many conditions after the `WHERE` clause and these will be combined to filter the results. Conditions can be combined using the logical separators, `AND` and `OR`; and conditions can be negated by prefixing them with `NOT`; plus parentheses can be used to group conditions together to override the default operator precedence. For example, to find every menu item that is a drink (i.e. has a menu item group number in the range 5 to 7) and also is priced over 2.00:

```
SELECT *
FROM MENU_ITEM
WHERE
MENU_ITEM_GROUP_NO>=5
AND MENU_ITEM_GROUP_NO<=7
AND PRICE>2.00
```

This has three conditions that are combined with the `AND` separator which means that they all must be satisfied (True) for a row to be selected. It would return the following:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
==========================================================================
|         300|Red Wine                                |                 7|   2.10
|         301|White Wine                              |                 7|   2.10
|         302|Guinness                                |                 6|   2.99
3 rows affected
```

To return menu items that are either in group number 1 or group number 2, we could use:

```
SELECT *
FROM MENU_ITEM
WHERE
MENU_ITEM_GROUP_NO=1
OR MENU_ITEM_GROUP_NO=2
```

Which returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
==========================================================================
|           1|Soup of the Day                         |                 1|   2.45
|           2|Samosas                                 |                 1|   1.95
|           3|Prawn Cocktail                          |                 1|   2.95
|         100|Lasagne                                 |                 2|   4.95
|         101|Spaghetti Bolognese                     |                 2|   1.95
|         102|Paella                                  |                 2|   6.95
|         103|Borsch                                  |                 2|   4.95
|         104|Irish stew                              |                 2|   3.95
|         105|Kedgeree                                |                 2|   4.99
|         106|Boeuf Bourgignone                       |                 2|   7.95
|         107|Roast Beef and Yorkshire Pudding        |                 2|   6.45
|         108|Chicken Madras                          |                 2|   5.45
|         109|Chicken Tikka Masala                    |                 2|   5.95
13 rows affected
```

And to remove the items from this result-set whose names begin with 'CHICKEN' we could use:

```
SELECT *
FROM MENU_ITEM
WHERE
(MENU_ITEM_GROUP_NO=1
OR MENU_ITEM_GROUP_NO=2)
AND MENU_ITEM_NAME NOT LIKE 'CHICKEN%'
```

Which returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
==========================================================================
|           1|Soup of the Day                         |                 1|   2.45
|           2|Samosas                                 |                 1|   1.95
|           3|Prawn Cocktail                          |                 1|   2.95
|         100|Lasagne                                 |                 2|   4.95
|         101|Spaghetti Bolognese                     |                 2|   1.95
|         102|Paella                                  |                 2|   6.95
|         103|Borsch                                  |                 2|   4.95
|         104|Irish stew                              |                 2|   3.95
|         105|Kedgeree                                |                 2|   4.99
|         106|Boeuf Bourgignone                       |                 2|   7.95
|         107|Roast Beef and Yorkshire Pudding        |                 2|   6.45
11 rows affected
```

Comparisons in SQL needn't be limited to single values, e.g. PRICE>5.00. The language is set-based and deals with rows of data at a time; so multiple columns or whole rows can be compared at once. For example, to find the check detail information for any sales of lemonade (item number 304) that were ordered in pairs (QTY=2):

```
SELECT *
FROM CHECK_DETAIL
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Would return:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY
====================================
|       5|       6|         304|    2
1 row affected
```

6. Joining tables

To find out details about the check returned in the previous query (check number 5) we need to refer to the CHECK_HEADER table to find the row that has a CHECK_NO value of 5. When the schema 'RESTAURANT' was designed, the CHECK_DETAIL table was related to the CHECK_HEADER table via a foreign key declaration on the CHECK_NO column so, rather than issue another simple query to find this, we can tell SQL to do the referencing for us using a table join. Joining tables is one of the most powerful features of SQL and is based on one of the fundamental concepts in relational theory.

Inner Joins

There are a number of ways to specify the join between the previous query and the CHECK_HEADER table. We will demonstrate them all, starting with the most readable ones where the join is specified in the `FROM` clause.

JOIN..USING

This specifies the common column(s) to be used to join the tables, e.g.

```
SELECT *
FROM CHECK_DETAIL JOIN CHECK_HEADER USING (CHECK_NO)
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Which returns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY  |SERVER_NO|START_TIME
=================================================================
|       5|       6|         304|    2|        1|2001-01-12 13:46:09
1 row affected
```

Notice the extra two result columns, SERVER_NO and START_TIME, that come from the appropriate row in the CHECK_HEADER table. We can use this readable syntax because CHECK_NO is the name of the joining column in both tables. This also means that the server can remove the second (redundant) occurrence of the CHECK_NO column that came from the CHECK_HEADER table, since it was already in the result set from the CHECK_DETAIL table.

NATURAL JOIN

Because CHECK_NO is the only column name shared by these two tables, another way of specifying the same table join would be to use the `NATURAL JOIN`:

```
SELECT *
FROM CHECK_DETAIL NATURAL JOIN CHECK_HEADER
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Which would return exactly the same results. This 'natural join' leaves the joining of the two tables up to the server, which uses all columns that have the same name and relies on the schema designer using special naming conventions. For this reason it cannot always be used.

JOIN..ON

If the columns that join the tables have different names then the `JOIN…ON` syntax can be used. This requires the joining columns to be made explicit, e.g.

```
SELECT *
FROM
CHECK_DETAIL JOIN CHECK_HEADER ON (CHECK_DETAIL.CHECK_NO=CHECK_HEADER.CHECK_NO)
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Which returns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY  |CHECK_NO|SERVER_NO|START_TIME
=============================================================================
|       5|       6|         304|    2|       5|        1|2001-01-12 13:46:09
1 row affected
```

Notice in this version that we get the same data returned but the duplicated CHECK_NO column from the CHECK_HEADER table is not automatically removed: all columns are returned and the joining columns are specified in the join condition following the `ON` keyword. When specifying column names in conditions or `SELECT` lists, it is important to prefix the column with its table name (separated by a full-stop) if more than one table is being referenced. This removes any ambiguity in case tables share the same column names.

Joins using the WHERE clause

Another, more common way, of specifying the same join which again requires the join details to be explicitly stated is to list the tables to be joined in the `FROM` clause and then specify the join details in the `WHERE` clause:

```
SELECT *
FROM CHECK_DETAIL, CHECK_HEADER
WHERE
CHECK_DETAIL.CHECK_NO=CHECK_HEADER.CHECK_NO
AND
(MENU_ITEM_NO,QTY)=(304,2)
```

Which returns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY  |CHECK_NO|SERVER_NO|START_TIME
=============================================================================
|       5|       6|         304|    2|       5|        1|2001-01-12 13:46:09
1 row affected
```

Again in this version we get the duplicated CHECK_NO column from the CHECK_HEADER table. Also, we had to state the way that the two tables were to be joined in the `WHERE` clause: this lets us join two tables using columns that have different names. The tables to be joined were simply listed in the `FROM` clause separated with a comma (with no `WHERE` clause, this would produce a Cartesian join). This is a less-flexible way of joining tables (especially when we need to use more elaborate join methods later) but is the one supported by other database management systems that haven't managed to implement the SQL standard.

A slightly more readable version of the table column-list version is to explicitly state that a Cartesian join (or Cross join) is being used, e.g.

```
SELECT *
FROM CHECK_DETAIL CROSS JOIN CHECK_HEADER
WHERE
CHECK_DETAIL.CHECK_NO=CHECK_HEADER.CHECK_NO
AND
(MENU_ITEM_NO,QTY)=(304,2)
```

If we now want to retrieve details about the server of this check we can use another table join to refer to the SERVER table. This has a declared primary key of SERVER_NO which is the column name used in the CHECK_HEADER table, so we can use the more readable syntax of:

```
SELECT *
FROM CHECK_DETAIL JOIN CHECK_HEADER USING (CHECK_NO) JOIN “SERVER” USING (SERVER_NO)
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Or

```
SELECT *
FROM CHECK_DETAIL NATURAL JOIN CHECK_HEADER NATURAL JOIN “SERVER”
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Which both return results without duplicate columns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY  |SERVER_NO|START_TIME         |SERVER_NAME
================================================================================
|       5|       6|         304|    2|        1|2001-01-12 13:46:09|John Smith
1 row affected
```

(Note that SERVER is a reserved keyword and so must be surrounded by double quotes to have it treated as an identifier.)
The same query using explicitly stated join conditions is:

```
SELECT *
FROM
CHECK_DETAIL
JOIN CHECK_HEADER ON (CHECK_DETAIL.CHECK_NO=CHECK_HEADER.CHECK_NO)
JOIN “SERVER” ON (CHECK_HEADER.SERVER_NO=“SERVER”.SERVER_NO)
WHERE (MENU_ITEM_NO,QTY)=(304,2)
```

Which returns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY  |CHECK_NO|SERVER_NO|START_TIME         |SERVER_NO|SERVER_NAME
================================================================================================
=========
|       5|       6|         304|    2|       5|        1|2001-01-12 13:46:09|        1|John Smith
1 row affected
```

And the same query using a more commonly supported but less readable syntax:

```
SELECT *
FROM
CHECK_DETAIL,
CHECK_HEADER,
“SERVER”
WHERE
CHECK_DETAIL.CHECK_NO=CHECK_HEADER.CHECK_NO
AND CHECK_HEADER.SERVER_NO=“SERVER”.SERVER_NO
AND (MENU_ITEM_NO,QTY)=(304,2)
```

Which returns the same results as the previous version:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY  |CHECK_NO|SERVER_NO|START_TIME         |SERVER_NO|SERVER_NAME
================================================================================================
|       5|       6|         304|    2|       5|        1|2001-01-12 13:46:09|        1|John Smith
1 row affected
```

Outer Joins

The joins discussed so far have all returned rows where a matching row exists on both sides of the join. These are known as *inner* joins. Where the `JOIN` keyword was used earlier, it could have been prefixed by the keyword `INNER` to make this more obvious, but this is the default type of join so `INNER` was not explicitly required.

There are occasions where you need to join two tables but would like to see all the selected rows from one of the tables even if there is no corresponding row in the other table. For example, if we want to join the MENU_ITEM table with the CHECK_DETAIL table on the MENU_ITEM_NO column (to retrieve all sales for each item) but also wanted a result row for the items that don’t have any sales. For these occasions we can use *outer* joins. An outer join can either be left, right or full, depending on which side of the join is to keep its unmatched rows. For our example we could use a left outer join:

```
SELECT MENU_ITEM_NO, MENU_ITEM_NAME, CHECK_DETAIL.*
FROM MENU_ITEM LEFT OUTER JOIN CHECK_DETAIL USING (MENU_ITEM_NO)
```

Which returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                        |CHECK_NO|LINE_NO |QTY
================================================================================
|           1|Soup of the Day                       |  <null>|  <null>|<null>
|           2|Samosas                               |       2|       1|    1
|           3|Prawn Cocktail                        |       1|       1|    1
|         100|Lasagne                               |       3|       1|    1
|         101|Spaghetti Bolognese                   |  <null>|  <null>|<null>
|         102|Paella                                |  <null>|  <null>|<null>
|         103|Borsch                                |  <null>|  <null>|<null>
|         104|Irish stew                            |       1|       2|    1
|         105|Kedgeree                              |  <null>|  <null>|<null>
|         106|Boeuf Bourgignone                     |       3|       2|    1
|         106|Boeuf Bourgignone                     |       5|       1|    1
|         107|Roast Beef and Yorkshire Pudding      |       4|       1|    2
|         107|Roast Beef and Yorkshire Pudding      |       5|       2|    1
|         108|Chicken Madras                        |       2|       2|    2
|         109|Chicken Tikka Masala                  |  <null>|  <null>|<null>
|         200|Boiled Rice                           |       2|       3|    2
|         200|Boiled Rice                           |       5|       4|    1
|         201|Mashed Potato                         |       3|       3|    1
|         201|Mashed Potato                         |       5|       3|    1
|         300|Red Wine                              |       1|       3|    1
|         300|Red Wine                              |       5|       5|    2
|         301|White Wine                            |       3|       4|    2
|         302|Guinness                              |       4|       2|    1
|         303|Lager                                 |       2|       4|    1
|         304|Lemonade                              |       2|       5|    1
|         304|Lemonade                              |       5|       6|    2
|         305|Cola                                  |  <null>|  <null>|<null>
|         400|Ice Cream                             |       4|       3|    1
|         400|Ice Cream                             |       5|       7|    2
29 rows affected
```

Here, each row in MENU_ITEM appears even if there is no matching CHECK_DETAIL row. Notice that each unmatched row appears once and the columns for the table on the right of the outer join are left NULL.

Swapping the order of the tables, we can demonstrate the right outer join (the `OUTER` keyword is optional):

```
SELECT MENU_ITEM_NO, MENU_ITEM_NAME, CHECK_DETAIL.*
FROM CHECK_DETAIL RIGHT OUTER JOIN MENU_ITEM USING (MENU_ITEM_NO)
```

Which returns exactly the same results as the left outer join because we reversed the table order:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                        |CHECK_NO|LINE_NO |QTY
================================================================================
|           1|Soup of the Day                       |  <null>|  <null>|<null>
|           2|Samosas                               |       2|       1|    1
|           3|Prawn Cocktail                        |       1|       1|    1
|         100|Lasagne                               |       3|       1|    1
|         101|Spaghetti Bolognese                   |  <null>|  <null>|<null>
|         102|Paella                                |  <null>|  <null>|<null>
|         103|Borsch                                |  <null>|  <null>|<null>
|         104|Irish stew                            |       1|       2|    1
|         105|Kedgeree                              |  <null>|  <null>|<null>
|         106|Boeuf Bourgignone                     |       3|       2|    1
|         106|Boeuf Bourgignone                     |       5|       1|    1
|         107|Roast Beef and Yorkshire Pudding      |       4|       1|    2
|         107|Roast Beef and Yorkshire Pudding      |       5|       2|    1
|         108|Chicken Madras                        |       2|       2|    2
|         109|Chicken Tikka Masala                  |  <null>|  <null>|<null>
|         200|Boiled Rice                           |       2|       3|    2
|         200|Boiled Rice                           |       5|       4|    1
|         201|Mashed Potato                         |       3|       3|    1
|         201|Mashed Potato                         |       5|       3|    1
|         300|Red Wine                              |       1|       3|    1
|         300|Red Wine                              |       5|       5|    2
|         301|White Wine                            |       3|       4|    2
|         302|Guinness                              |       4|       2|    1
|         303|Lager                                 |       2|       4|    1
|         304|Lemonade                              |       2|       5|    1
|         304|Lemonade                              |       5|       6|    2
|         305|Cola                                  |  <null>|  <null>|<null>
|         400|Ice Cream                             |       4|       3|    1
|         400|Ice Cream                             |       5|       7|    2
29 rows affected
```

As with inner joins, we can specify the outer join condition with `USING` or `ON`, or if we know the columns to be joined share the same name we can use the `NATURAL` keyword to leave the join condition to the server, e.g.

```
SELECT MENU_ITEM_NO, MENU_ITEM_NAME, CHECK_DETAIL.*
FROM MENU_ITEM NATURAL LEFT OUTER JOIN CHECK_DETAIL
```

Which again returns the same results as the first left outer join example because the shared column is the same as that specified in the first `USING` clause, MENU_ITEM_NO.

7. Advanced retrieval

Joining tables enables complex queries to be put to the server. This section will briefly list the built-in functions that allow expressions to be manipulated. It will describe how to query tables using set functions, how to use sub-queries, and then show how to use the set operators of SQL.

Built-in functions

SQL has a number of built-in functions that can be used to modify values.

UPPER/LOWER

These change the case of the parameter.

{ UPPER | LOWER } (<character value expression>)

```
SELECT MENU_ITEM_GROUP_NAME, LOWER(MENU_ITEM_GROUP_NAME), UPPER(MENU_ITEM_GROUP_NAME)
FROM MENU_ITEM_GROUP
```

Returns:

```
|MENU_ITEM_GROUP_NAME|2                   |3
============================================================
|Starter             |starter             |STARTER
|Main Course         |main course         |MAIN COURSE
|Side-dish           |side-dish           |SIDE-DISH
|Dessert             |dessert             |DESSERT
|Soft drink          |soft drink          |SOFT DRINK
|Beer                |beer                |BEER
|Wine                |wine                |WINE
7 rows affected
```

CAST

This converts one value to a specific data type.

CAST (<cast operand> AS <cast type>)

```
SELECT SERVER_NO, CAST(SERVER_NO AS CHAR(3))
FROM “SERVER”
```

Returns:

```
|SERVER_NO|2
==============
|        1|1
|        2|2
2 rows affected
```

Concatenation

This appends one character (or binary) string after another.

<character/blob value expression> || <character/blob factor>

```
SELECT CAST(SERVER_NO AS CHAR(3)) || SERVER_NAME
FROM “SERVER”
```

Returns:

```
|1
========================
|1  John Smith
|2  Mary Jones
2 rows affected
```

```
VALUES (X'ABCDEF'||X'123456')
```

Returns:

```
|1
=============
|ABCDEF123456
1 row affected
```

TRIM

This removes specified characters (or bytes) from the start and/or end of a character (or binary) string. The default character is a space (the default byte is X’00’) and the default is to trim from both ends. The following examples also show how to use VALUES to create a table with a specified name and column names with data in the FROM clause.

```
TRIM ( [ [ LEADING | TRAILING | BOTH]
         [ <character/blob value expression> ] FROM ]
       <character/blob value expression> )
```

```
SELECT
'['||SAMPLE||']',
'['||TRIM(TRAILING FROM SAMPLE)||']',
'['||TRIM(LEADING FROM SAMPLE)||']',
 '['||TRIM(BOTH FROM SAMPLE)||']'
FROM
(
       VALUES ('   SPACE IS DEFAULT  ')
) AS X (SAMPLE)
```

returns:

```
|1                      |2                      |3                      |4
=============================================================================================
|[   SPACE IS DEFAULT  ]|[   SPACE IS DEFAULT]  |[SPACE IS DEFAULT  ]   |[SPACE IS DEFAULT]
1 row affected
```

```
SELECT SAMPLE, TRIM(TRAILING '0' FROM SAMPLE), TRIM(LEADING '0' FROM SAMPLE),
TRIM(BOTH '0' FROM SAMPLE)
FROM
(
       VALUES ('00005.00')
) AS X (SAMPLE)
```

returns:

```
|SAMPLE  |2       |3       |4
===================================
|00005.00|00005.  |5.00    |5.
1 row affected
```

POSITION

Returns the position of a specified character (or binary) string within another string.

```
POSITION ( <string/blob value expression>
            IN <string/blob value expression> )
```

```
SELECT MENU_ITEM_GROUP_NAME, POSITION('r' IN MENU_ITEM_GROUP_NAME)
FROM MENU_ITEM_GROUP
```

Returns:

```
|MENU_ITEM_GROUP_NAME|2
======================
|Starter             |4
|Main Course         |9
|Side-dish           |0
|Dessert             |6
|Soft drink          |7
|Beer                |4
|Wine                |0
7 rows affected
```

```
VALUES (POSITION(X'EF' IN X'ABCDEF123456'))
```

Returns:

```
|1
=======
|     3
1 row affected
```

SUBSTRING

Returns a specified portion of a character (or binary) string.

```
SUBSTRING ( <character/blob value expression>
            FROM <numeric value expression>
            [ FOR <string/blob length> ] )
```

```
SELECT MENU_ITEM_GROUP_NAME, SUBSTRING(MENU_ITEM_GROUP_NAME FROM 3),
SUBSTRING(MENU_ITEM_GROUP_NAME FROM 3 FOR 4)
FROM MENU_ITEM_GROUP
```

Returns:

```
|MENU_ITEM_GROUP_NAME|2                   |3
=============================================================
|Starter             |arter               |arte
|Main Course         |in Course           |in C
|Side-dish           |de-dish             |de-d
|Dessert             |ssert               |sser
|Soft drink          |ft drink            |ft d
|Beer                |er                  |er
|Wine                |ne                  |ne
7 rows affected
```

CHARACTER_LENGTH/CHAR_LENGTH

Returns the number of characters in a character string.

```
{ CHAR_LENGTH | CHARACTER_LENGTH } ( <string value expression> )
```

```
SELECT MENU_ITEM_GROUP_NAME, CHARACTER_LENGTH(MENU_ITEM_GROUP_NAME)
FROM MENU_ITEM_GROUP
```

Returns:

```
|MENU_ITEM_GROUP_NAME|2
======================
|Starter             |7
|Main Course         |11
|Side-dish           |9
|Dessert             |7
|Soft drink          |10
|Beer                |4
|Wine                |4
7 rows affected
```

OCTET_LENGTH

Returns the number of octets (bytes) in a character (or binary) string.

OCTET_LENGTH (<string/blob value expression>)

```
SELECT MENU_ITEM_GROUP_NAME, OCTET_LENGTH(MENU_ITEM_GROUP_NAME)
FROM MENU_ITEM_GROUP
```

Returns:

```
|MENU_ITEM_GROUP_NAME|2
======================
|Starter             |7
|Main Course         |11
|Side-dish           |9
|Dessert             |7
|Soft drink          |10
|Beer                |4
|Wine                |4
7 rows affected
```

CURRENT_USER/SESSION_USER/SYSTEM_USER

Returns the current authorisation/user/operating-system user.

```
VALUES(CURRENT_USER)
```

Returns:

```
|1
====================
|RESTAURANT
1 row affected
```

CURRENT_DATE

Returns the current date.

```
VALUES(CURRENT_DATE)
```

Returns something similar to:

```
|1
==
|2001-04-15
1 row affected
```

CURRENT_TIME

Returns the current time.

```
VALUES(CURRENT_TIME)
```

Returns something similar to:

```
|1
==
|12:18:00
1 row affected
```

CURRENT_TIMESTAMP

Returns the current date and time.

```
VALUES(CURRENT_TIMESTAMP)
```

Returns something similar to:

```
|1
==
|2001-04-15 12:18:00
1 row affected
```

CASE

This returns one of a number of values according to some criteria.

```
CASE
      WHEN <search condition> THEN <result>…
      [ ELSE <result> ]
END
```

```
SELECT
  SERVER_NO,
  CASE
    WHEN COUNT(*)>20 THEN 'FAST'
    WHEN COUNT(*)>10 THEN 'MEDIUM'
  ELSE
    'SLOW'
  END AS SERVER_SPEED
FROM
  CHECK_DETAIL JOIN CHECK_HEADER USING (CHECK_NO)
GROUP BY SERVER_NO
```

Returns:

```
|SERVER_NO|SERVER_SPEED
======================
|        1|MEDIUM
|        2|SLOW
2 rows affected
```

If the criteria are all tests for equality against some value, a shorthand syntax can be used, e.g.

```
CASE <value expression>
      WHEN <value expression> THEN <result>…
      [ ELSE <result> ]
END
```

```
SELECT
  CHECK_NO,
  START_TIME,
  CASE SERVER_NO
    WHEN 1 THEN 'ONE'
    WHEN 2 THEN 'TWO'
  ELSE
    '?'
  END AS “SERVER”
FROM CHECK_HEADER
```

Returns:

```
|CHECK_NO|START_TIME         |SERVER
===================================
|       1|2001-01-12 11:55:12|ONE
|       2|2001-01-12 12:07:34|ONE
|       3|2001-01-12 13:24:39|ONE
|       4|2001-01-12 13:45:17|TWO
|       5|2001-01-12 13:46:09|ONE
5 rows affected
```

There are two other shorthand forms of the case statement:

```
NULLIF ( <value expression> , <value expression> )
```

e.g. `NULLIF(x,y)` is equivalent to `CASE WHEN x=y THEN NULL ELSE x END`

```
COALESCE ( <value expression> { , <value expression> }… )
```

e.g. `COALESCE(x,y)` is equivalent to `CASE WHEN x IS NOT NULL THEN x ELSE y END`

Aggregates and grouping

There are a number of SQL functions that work on columns from sets of rows or from whole tables. These are known as set functions and they produce aggregate results. As an example, to find the highest menu item price we can use the MAX function against the PRICE column of the entire MENU_ITEM table as follows:

```
SELECT MAX(PRICE)
FROM MENU_ITEM
```

Which would return:

```
|1
========
|   7.95
1 row affected
```

Because the selected column is calculated (rather than stored in the table) the server gives it a generic name based on its position in the result table – in this case 1. We can give the resulting column a specific name using a column alias as follows:

```
SELECT MAX(PRICE) AS HIGHEST_PRICE
FROM MENU_ITEM
```

The alias must have a valid identifier name, which means it can't contain spaces etc. The results would be:

```
|HIGHEST_PRICE
==============
|         7.95
1 row affected
```

Here are all the other set functions against the menu item table:

```
SELECT
MAX(PRICE) AS HIGHEST_PRICE,
MIN(PRICE) AS LOWEST_PRICE,
AVG(PRICE) AS MEAN_PRICE,
SUM(PRICE) AS TOTAL_PRICE
FROM MENU_ITEM
```

The resulting column aliases should indicate what the functions produce.

```
|HIGHEST_PRICE|LOWEST_PRICE|MEAN_PRICE|TOTAL_PRICE
=================================================
|         7.95|        1.20|      3.57|      78.72
1 row affected
```

There is one more set function, COUNT, which works in a similar way but not against any particular column. It counts the number of rows and so has a dummy column parameter of *, e.g.

```
SELECT COUNT(*) AS NUMBER_OF_ROWS
FROM MENU_ITEM
```

This returns:

```
|NUMBER_OF_ROWS
===============
|            22
1 row affected
```

Note that specifying any column without a set function in a SELECT clause that already has a set function does not make sense and would give a syntax error. See the section on grouping below for ways to make such non-aggregated columns meaningful in conjunction with aggregates.

These set functions can be used against filtered result-sets to give more useful information. For example, the following query counts the number of menu items in the price range 2.00 to 5.00, and the maximum and minimum prices within this range.

```
SELECT COUNT(*), MIN(PRICE), MAX(PRICE)
FROM MENU_ITEM
WHERE PRICE BETWEEN 2.00 AND 5.00
```

Which returns:

```
|1       |2      |3
========================
|      10|   2.10|   4.99
1 row affected
```

One other way of using these set functions is to apply them to groups of rows. The `GROUP BY` clause can be added to a query to enable these functions to be applied to groups within the selected rows rather than the whole set. For example, to list the average price for each menu item group we could use:

```
SELECT MENU_ITEM_GROUP_NO, AVG(PRICE) AS AVERAGE_PRICE
FROM MENU_ITEM
GROUP BY MENU_ITEM_GROUP_NO
```

Which would return:

```
|MENU_ITEM_GROUP_NO|AVERAGE_PRICE
==================================
|                 1|         2.45
|                 2|         5.35
|                 3|         1.70
|                 4|         2.99
|                 5|         1.20
|                 6|         2.42
|                 7|         2.10
7 rows affected
```

Note that the MENU_ITEM_GROUP_NO can be specified in the `SELECT` clause because it is one of the grouping columns and so has a single value per group.
To make the above results more meaningful we would join to the MENU_ITEM_GROUP table to get the group name:

```
SELECT MENU_ITEM_GROUP_NO, MENU_ITEM_GROUP_NAME, AVG(PRICE) AS AVERAGE_PRICE
FROM MENU_ITEM JOIN MENU_ITEM_GROUP USING (MENU_ITEM_GROUP_NO)
GROUP BY MENU_ITEM_GROUP_NO, MENU_ITEM_GROUP_NAME
```

Which would give:

```
|MENU_ITEM_GROUP_NO|MENU_ITEM_GROUP_NAME|AVERAGE_PRICE
=====================================================
|                 1|Starter             |         2.45
|                 2|Main Course         |         5.35
|                 3|Side-dish           |         1.70
|                 4|Dessert             |         2.99
|                 5|Soft drink          |         1.20
|                 6|Beer                |         2.42
|                 7|Wine                |         2.10
7 rows affected
```

Note that we now have to group by two columns to be able to `SELECT` them both without using set functions (even though we, and in this case the database server, can infer that there could only be one menu item group name per group number).

Sometimes we need to filter the results after the grouping has been done, for example to select only those menu item groups with an average price over 2.50. For this we can use the `HAVING` clause. This is similar to a `WHERE` clause, but is applied after a `GROUP BY` and has access to group-level aggregates:

```
SELECT MENU_ITEM_GROUP_NO, AVG(PRICE) AS AVERAGE_PRICE
FROM MENU_ITEM
GROUP BY MENU_ITEM_GROUP_NO
HAVING AVG(PRICE)>2.50
```

Which would return:

```
|MENU_ITEM_GROUP_NO|AVERAGE_PRICE
================================
|                 2|         5.35
|                 4|         2.99
```

The set functions can also be applied to non-numeric columns, e.g. dates and character strings, except AVG and SUM aren't then applicable (also MIN and MAX aren't applicable to large objects).

Single-row subqueries

Often it is useful to use the results from a query in another query. For example, to find all the menu items that have the lowest price we might first build a query to find that lowest price, e.g.

```
SELECT MIN(PRICE)
FROM MENU_ITEM
```

Which as we found above, returns:

```
|1
========
|   1.20
1 row affected
```

We can now use the result of this query in another query by surrounding it in parentheses as follows:

```
SELECT *
FROM MENU_ITEM
WHERE
PRICE=
(
        SELECT MIN(PRICE)
        FROM MENU_ITEM
)
```

Which returns the menu item(s) with the lowest price, i.e.:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
=============================================================================
|         304|Lemonade                                |                 5|   1.20
|         305|Cola                                    |                 5|   1.20
2 rows affected
```

The previous example uses a SELECT statement inside an outer query, and is known as a subselect. The above example uses a simple comparison operator (=) and so the subquery must return a maximum of one row. Such single-row subselects can also be used in other places such as the SELECT clause, e.g.

```
SELECT
MENU_ITEM_NAME,
PRICE,
(SELECT AVG(PRICE) FROM MENU_ITEM) - PRICE AS DEVIATION
FROM MENU_ITEM
ORDER BY DEVIATION
```

Which would return the menu items and their price deviation from the mean price:

```
|MENU_ITEM_NAME                          |PRICE  |DEVIATION
============================================================
|Boeuf Bourgignone                       |   7.95|    -4.38
|Paella                                  |   6.95|    -3.38
|Roast Beef and Yorkshire Pudding        |   6.45|    -2.88
|Chicken Tikka Masala                    |   5.95|    -2.38
|Chicken Madras                          |   5.45|    -1.88
|Kedgeree                                |   4.99|    -1.42
|Borsch                                  |   4.95|    -1.38
|Lasagne                                 |   4.95|    -1.38
|Irish stew                              |   3.95|    -0.38
|Guinness                                |   2.99|     0.58
|Ice Cream                               |   2.99|     0.58
|Prawn Cocktail                          |   2.95|     0.62
|Soup of the Day                         |   2.45|     1.12
|White Wine                              |   2.10|     1.47
|Red Wine                                |   2.10|     1.47
|Boiled Rice                             |   1.95|     1.62
|Samosas                                 |   1.95|     1.62
|Spaghetti Bolognese                     |   1.95|     1.62
|Lager                                   |   1.85|     1.72
|Mashed Potato                           |   1.45|     2.12
|Cola                                    |   1.20|     2.37
|Lemonade                                |   1.20|     2.37
22 rows affected
```

Multi-row subqueries

Subselects can return more than one row and in these cases more complex comparison operators are required.

Multi-row comparison operator	Description
Simple-comparison-operator **ANY***	At least one row satisfies the simple-comparison
Simple-comparison-operator **ALL**	All rows satisfy the simple-comparison
IN	At least one row equals
EXISTS	At least one row is returned by the subselect

*SOME is a synonym for ANY.

For example, the following query:

```
SELECT PRICE FROM MENU_ITEM WHERE MENU_ITEM_GROUP_NO=1
```

Returns the following three rows:

```
|PRICE
========
|   2.45
|   1.95
|   2.95
3 rows affected
```

So the following query:

```
SELECT *
FROM MENU_ITEM
WHERE
PRICE > ANY
(
        SELECT PRICE FROM MENU_ITEM WHERE MENU_ITEM_GROUP_NO=1
)
```

Returns all menu items priced more than any one of the three prices returned from the subquery (i.e. more than 1.95, the lowest priced starter):

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|           1|Soup of the Day                        |                 1|   2.45
|           3|Prawn Cocktail                         |                 1|   2.95
|         100|Lasagne                                |                 2|   4.95
|         102|Paella                                 |                 2|   6.95
|         103|Borsch                                 |                 2|   4.95
|         104|Irish stew                             |                 2|   3.95
|         105|Kedgeree                               |                 2|   4.99
|         106|Boeuf Bourgignone                      |                 2|   7.95
|         107|Roast Beef and Yorkshire Pudding       |                 2|   6.45
|         108|Chicken Madras                         |                 2|   5.45
|         109|Chicken Tikka Masala                   |                 2|   5.95
|         300|Red Wine                               |                 7|   2.10
|         301|White Wine                             |                 7|   2.10
|         302|Guinness                               |                 6|   2.99
|         400|Ice Cream                              |                 4|   2.99
15 rows affected
```

And the following query:

```
SELECT *
FROM MENU_ITEM
WHERE
PRICE > ALL
(
        SELECT PRICE FROM MENU_ITEM WHERE MENU_ITEM_GROUP_NO=1
)
```

Returns all menu items priced more than all of the three prices returned from the subquery (i.e. more than 2.95, the highest priced starter):

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|         100|Lasagne                                |                 2|   4.95
|         102|Paella                                 |                 2|   6.95
|         103|Borsch                                 |                 2|   4.95
|         104|Irish stew                             |                 2|   3.95
|         105|Kedgeree                               |                 2|   4.99
|         106|Boeuf Bourgignone                      |                 2|   7.95
|         107|Roast Beef and Yorkshire Pudding       |                 2|   6.45
|         108|Chicken Madras                         |                 2|   5.45
|         109|Chicken Tikka Masala                   |                 2|   5.95
|         302|Guinness                               |                 6|   2.99
|         400|Ice Cream                              |                 4|   2.99
11 rows affected
```

The following query:

```
SELECT *
FROM MENU_ITEM
WHERE
PRICE IN
(
        SELECT PRICE FROM MENU_ITEM WHERE MENU_ITEM_GROUP_NO=1
)
```

Returns all menu items that have the same price as one of the three prices returned from the subquery:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|           1|Soup of the Day                        |                 1|   2.45
|           2|Samosas                                |                 1|   1.95
|           3|Prawn Cocktail                         |                 1|   2.95
|         101|Spaghetti Bolognese                    |                 2|   1.95
|         200|Boiled Rice                            |                 3|   1.95
5 rows affected
```

Note that using the `IN` operator is very similar to using a join. In fact another place a subquery can be used is in the `FROM` clause. In this case, the table must be given an alias – any valid identifier will do. As an example, the above query can be rewritten as:

```
SELECT *
FROM MENU_ITEM
NATURAL JOIN
(
        SELECT PRICE
        FROM MENU_ITEM
        WHERE MENU_ITEM_GROUP_NO=1
) AS SUBSELECT_TABLE
```

And returns the same results (but with a different column order because we used `SELECT *`):

```
|PRICE  |MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO
================================================================================
|   2.45|           1|Soup of the Day                         |                 1
|   1.95|           2|Samosas                                 |                 1
|   1.95|         101|Spaghetti Bolognese                     |                 2
|   1.95|         200|Boiled Rice                             |                 3
|   2.95|           3|Prawn Cocktail                          |                 1
5 rows affected
```

The `EXISTS` operator is satisfied if the subquery returns at least one row.

Union, Except and Intersect

SQL can combine two result sets using the set-theory operators of union, difference and intersection. The two result sets must have the same number of columns and corresponding columns must be the same type. The following examples will be based on two results sets which select menu item sales with single and double quantities respectively (the sample values have been ordered to make it easier to see how they are combined):

```
SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1
        |MENU_ITEM_NO
        =============
        |           2
        |           3
        |         100
        |         104
        |         106
        |         106
        |         107
        |         200
        |         201
        |         201
        |         300
        |         302
        |         303
        |         304
        |         400
        15 rows affected

SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2
        |MENU_ITEM_NO
        =============
        |         107
        |         108
        |         200
        |         300
        |         301
        |         304
        |         400
        7 rows affected
```

`UNION` returns rows that belong to either of the two results. By default, columns are matched from left to right and duplicate rows are removed. For example:

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
UNION
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

Returns all sold items, i.e.:

```
|MENU_ITEM_NO
=============
|           2
|           3
|         100
|         104
|         106
|         107
|         108
|         200
|         201
|         300
|         301
|         302
|         303
|         304
|         400
15 rows affected
```

To preserve duplicate rows, add the ALL option, e.g.

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
UNION ALL
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

Returns:

```
|MENU_ITEM_NO
=============
|           2
|           3
|         100
|         104
|         106
|         106
|         107
|         107
|         108
|         200
|         200
|         201
|         201
|         300
|         300
|         301
|         302
|         303
|         304
|         304
|         400
|         400
22 rows affected
```

EXCEPT returns rows that belong to the first result set but not the second. By default, columns are matched from left to right and duplicate rows are removed. For example:

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
EXCEPT
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

Returns all items sold only as singles, i.e.:

```
|MENU_ITEM_NO
=============
|           2
|           3
|         100
|         104
|         106
|         201
|         302
|         303
8 rows affected
```

Again, to preserve duplicate rows add the ALL option, e.g.

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
EXCEPT ALL
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

Returns:

```
|MENU_ITEM_NO
=============
|           2
|           3
|         100
|         104
|         106
|         106
|         201
|         201
|         302
|         303
10 rows affected
```

EXCEPT, unlike UNION and INTERSECT, is not commutative so swapping the order of the operands in this example, e.g.

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
EXCEPT
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
```

Will return different information, i.e. items sold only as doubles:

```
|MENU_ITEM_NO
=============
|         108
|         301
2 rows affected
```

INTERSECT returns rows that belong to both the first result and the second. By default, columns are matched from left to right and duplicate rows are removed. For example:

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
INTERSECT
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

Returns items sold as both singles and doubles, i.e.:

```
|MENU_ITEM_NO
=============
|         107
|         200
|         300
|         304
|         400
5 rows affected
```

And to preserve duplicate rows add the ALL option, e.g.

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
INTERSECT ALL
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

Returns:

```
|MENU_ITEM_NO
=============
|         107
|         200
|         300
|         304
|         400
5 rows affected
```

In the above examples, columns were matched by their left to right order. To match by column names, add CORRESPONDING after the set operator (or after the ALL if it is specified). As with the left to right matching, the corresponding columns must be of the same type. For example,

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
INTERSECT ALL CORRESPONDING
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

(In this case, the results would be the same as above since the single column has the same name in both results sets.)

To match by specific column names, add `CORRESPONDING BY (<comma separated column list>)`, for example:

```
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=1)
INTERSECT ALL CORRESPONDING BY (MENU_ITEM_NO)
(SELECT MENU_ITEM_NO FROM CHECK_DETAIL WHERE QTY=2)
```

(Again, for this single column example, the results are as above.)

8. Schema Definition

A schema is a named group of objects, e.g. tables, views, constraints. This section explains how to create and maintain schemas and the objects that they contain. The examples given will be taken from the RESTAURANT sample schema.

To create a schema use the following syntax:

```
CREATE SCHEMA [ <schema name> ]
[ <schema character set specification> ]
[AUTHORIZATION <schema authorization identifier> ]
[ <schema element>… ]
```

e.g.

```
CREATE SCHEMA restaurant AUTHORIZATION restaurant
```

If no <schema name> is specified, then the authorization clause is required and the <schema name> defaults to the <schema authorization identifier>.
The authorization clause specifies the owner of the schema and all objects within the schema. If no authorization clause is given, then the schema is owned by the current user.

Note that the `CREATE SCHEMA` command is a single SQL statement, so semi-colons must not separate the schema elements.

In ThinkSQL, creating a schema sets it to be the default schema for the owning user. Future connections by that user will use the new schema as the current schema.
You can only create schemas owned by your user, unless you are the ADMIN user in which case you can create them and give ownership to other users.

The following sections explain how to create the various schema elements. Each of the schema elements can be created individually outside of a schema definition, in which case they belong to the current schema, and must be created by the schema owner.

Deleting schemas
To remove a schema, use the `DROP SCHEMA` command:

```
DROP SCHEMA <schema name> { RESTRICT | CASCADE }
```

Here, and in other DROP commands, specifying the `RESTRICT` keyword prevents the item from being removed if any dependent objects exist. In this case, schema objects such as tables or routines would cause the `DROP SCHEMA` command to fail if `RESTRICT` was specified.

You must be the owner of the schema, or the ADMIN user.

Domains

Domains specify the type of data that can be stored in particular columns within tables. They extend the built-in data types specified below. A domain can also specify defaults and constraints that apply to columns defined using that domain. To create a domain:

```
CREATE DOMAIN <domain name> [ AS ] <data type>
[ <default clause> ]
[ <domain constraint definition>… ]
```

The data type is one of the built-in types as specified in the base-table section below, e.g. DATE.

Each domain can specify a default value to be used when inserting new rows:

```
DEFAULT
      <literal>
      | NULL
      | CURRENT_DATE
      | CURRENT_TIME
      | CURRENT_TIMESTAMP
      | USER
      | CURRENT_USER
      | SESSION_USER
      | SYSTEM_USER
```

Domains can have constraints, such as preventing missing values. Constraints will be discussed in a later section.

An example `CREATE DOMAIN` statement that defines a domain that could have been used for primary key columns in the RESTAURANT sample schema:

```
CREATE DOMAIN id_type AS INTEGER NOT NULL
```

A domain in SQL is really just used as a shorthand for column definitions when defining base tables. Changes to a domain after it has been referenced have no effect on existing table definitions unless they are re-created, e.g. from a script file.

Base tables

Base tables store rows of data in predefined columns. To create a base table:

```
CREATE TABLE <table name>
( <table element list> )
```

The table must be given a unique name within the schema and the <table element list> can consist of constraint definitions (see the Constraints section below), or column definitions, i.e.

```
<column name>
{ <data type> | <domain name> }
[ <default clause> ]
[ <column constraint definition>… ]
```

Each column must be given a unique name within the table and can be either a user-defined domain or one of the following data types:

Data type	Description	Defaults	Example literal	Synonyms
`CHARACTER(`*n*`)`	A fixed-length string padded to exactly *n* characters.	If *n* is not specified, 1 is assumed.	`'ThinkSQL'`	`CHAR(`*n*`)`
`CHARACTER VARYING(`*n*`)`	A variable-length string with a maximum of *n* characters.	If *n* is not specified, 1 is assumed.		`CHAR VARYING(`*n*`), VARCHAR(`*n*`)`

BIT(*n*)*	A fixed-length bit-string of exactly *n* bits.	If *n* is not specified, 1 is assumed.	B'001101'	
BIT VARYING(*n*)*	A variable-length bit-string with a maximum of *n* bits.	If *n* is not specified, 1 is assumed.		
NUMERIC(*p*,*s*)	A decimal number containing *p* digits, *s* of which are after the decimal point.	If *s* is not specified, 0 is assumed.	123.45	
DECIMAL(*p*,*s*)	A decimal number containing at least *p* digits, *s* of which are after the decimal point.	If *s* is not specified, 0 is assumed.	123.45	DEC(*p*,*s*)
INTEGER	A signed 32-bit binary integer.		123456	
SMALLINT	A signed 16-bit binary integer.		1234	
FLOAT(*p*)	A floating point number stored as at least 2^p bits.	If *p* is not specified, 32 is assumed.	123.45	REAL, DOUBLE PRECISION
DATE	A date.		DATE '2001-09-16'	
TIME(*s*)	A time with fractions of seconds up to *s* places of decimal.	If *s* is not specified, 0 is assumed.	TIME '14:30:00'	
TIMESTAMP(*s*)	A date and time with fractions of seconds up to *s* places of decimal.	If *s* is not specified, 6 is assumed.	TIMESTAMP '2001-09-16 14:30:00.5'	
TIME(*s*) WITH TIME ZONE	A time with time zone information and with fractions of seconds up to *s* places of decimal.	If *s* is not specified, 0 is assumed.	TIME '14:30:00+01:00'	
TIMESTAMP(*s*) WITH TIME ZONE	A date and time with time zone information and with fractions of seconds up to *s* places of decimal.	If *s* is not specified, 6 is assumed.	TIMESTAMP '2001-09-16 14:30:00.5+01:00'	
BINARY LARGE OBJECT(*n*)	A variable-length binary string with a maximum of *n* bytes. If K suffixes *n* then it represents Kilobytes; M represents Megabytes and G represents Gigabytes.	If *n* is not specified, 1K is assumed, i.e. 1024 bytes.	X'AA3F02' (i.e. a string of hexadecimal codes)	BLOB(*n*)
CHARACTER LARGE OBJECT(*n*)	A variable-length string with a maximum of *n* characters. If K suffixes *n* then it represents Kilobytes; M represents Megabytes and G represents Gigabytes.	If *n* is not specified, 1K is assumed, i.e. 1024 characters.	'ThinkSQL'	CHAR LARGE OBJECT(*n*), CLOB(*n*)

* deprecated (in anticipation of SQL 2003)

Each column can specify a default value to be used when inserting new rows:

```
DEFAULT
        <literal>
        | NULL
```

```
        | CURRENT_DATE
        | CURRENT_TIME
        | CURRENT_TIMESTAMP
        | USER
        | CURRENT_USER
        | SESSION_USER
        | SYSTEM_USER
```

For example:

```
CREATE TABLE CHECK_DETAIL (
        CHECK_NO INTEGER NOT NULL REFERENCES CHECK_HEADER,
        LINE_NO INTEGER NOT NULL,
        MENU_ITEM_NO INTEGER REFERENCES MENU_ITEM,
        QTY DECIMAL(5) DEFAULT 1,
        PRIMARY KEY (CHECK_NO, LINE_NO) )
```

Would set the line quantity to 1 if a default were needed during an insert (see the Insert section below).

Columns can each have constraints, such as preventing missing values or foreign key references. These can either be specified alongside each column definition or as part of the table definition. Constraints will be discussed in a later section.

An example `CREATE TABLE` statement that defines a table with two columns is:

```
CREATE TABLE "server" (
        server_no INTEGER NOT NULL,
        server_name VARCHAR(20),
        PRIMARY KEY (server_no) )
```

Modifying tables

To modify a table, use the `ALTER TABLE` command:

```
ALTER TABLE <table name>
        ADD <table constraint definition>
        | DROP CONSTRAINT <constraint name> { RESTRICT | CASCADE}
```

This allows a constraint to be added (see the Constraints section below) or dropped.

Note: a foreign key constraint depends upon a candidate key, and so could prevent the candidate key (or its table) from being dropped if `RESTRICT` is specified.

You must be the owner of the table, or the ADMIN user.

Deleting tables

To remove a table, use the `DROP TABLE` command (note that either `RESTRICT` or `CASCADE` must be specified and `RESTRICT` is recommended since the behaviour of the `CASCADE` keyword is yet to be implemented):

```
DROP TABLE <table name> { RESTRICT | CASCADE }
```

Constraints

Constraints are placed on tables and columns at schema design time to preserve the integrity of the data held in them. The server also uses constraints to optimise query processing. Every constraint has a unique name that can be given by the user or supplied by the system. The following constraints can be applied to tables:

Candidate key constraints

A candidate key uniquely identifies each row within the table and can comprise one or more columns, i.e. there is only one row with that key value within the table. In relational theory, the foundation for SQL, a relation is defined as having no duplicate rows. SQL does permit duplicate rows but this can lead to anomalous results and problems formulating some queries, so it's strongly advised that you always specify at least one candidate key per table to prevent duplicate rows. There are two ways to specify candidate keys using SQL constraints: a primary key or a unique constraint.

There can be only one primary key per table, but many unique constraints. The columns that make up the primary key must have the additional constraint that they cannot be null. To specify a primary key or unique constraint as a member of a <table element list> when creating base tables, use the following:

```
[ CONSTRAINT <constraint name> ]
{ PRIMARY KEY | UNIQUE } ( <column name list> )
```

If no constraint name is given, a unique name is generated by the system. For example:

```
CREATE TABLE CHECK_DETAIL (
        CHECK_NO INTEGER NOT NULL REFERENCES CHECK_HEADER,
        LINE_NO INTEGER NOT NULL,
        MENU_ITEM_NO INTEGER REFERENCES MENU_ITEM,
        QTY DECIMAL(5),
        PRIMARY KEY (CHECK_NO, LINE_NO) )
```

Specifies a two-column primary key for the CHECK_DETAIL sample table: notice that the constraint definitions are usually written after the column definitions. Note that CHECK_NO and LINE_NO must both have the `NOT NULL` constraint.

If the key constraint applies to only one column, a shorthand for the above is to specify the constraint as part of the column definition rather than as an additional <table element> by omitting the column list. For example:

```
CREATE TABLE “SERVER” (
        SERVER_NO INTEGER NOT NULL PRIMARY KEY,
        SERVER_NAME VARCHAR(20),
        )
```

would specify that the SERVER_NO column be the primary key for the sample SERVER table.

Foreign key constraints

A foreign key comprises one or more columns within a table (the referencing table) whose values must match the values of a candidate key in another table (the referenced table). For example, the value in the MENU_ITEM_NO column in every row in the sample CHECK_DETAIL table must match an entry in the MENU_ITEM table (or be NULL). This prevents the RESTAURANT database from becoming inconsistent by ensuring two things:

1. No CHECK_DETAIL row can be inserted or updated with an invalid (i.e. non-existent) MENU_ITEM_NO
2. No MENU_ITEM row can be deleted (or have its primary key updated) if it is being referenced by any entry in the CHECK_DETAIL table

To specify a foreign key constraint as a member of a <table element list> when creating base tables, use the following:

```
[ CONSTRAINT <constraint name> ]
FOREIGN KEY ( <referencing column name list> )
REFERENCES <table name> [ ( <reference column list> ) ]
[ MATCH { FULL | PARTIAL } ]
[ <referential triggered action> ]
```

If no constraint name is given, a unique name is generated by the system. If a <reference column list> is specified it must refer to a candidate key of the referenced table. If the <reference column list> is not specified, the primary key of the referenced table is assumed. In either case, the <referencing column name list> must match the referenced columns in number and type. For example:

```
CREATE TABLE MENU_ITEM (
        MENU_ITEM_NO INTEGER NOT NULL,
        MENU_ITEM_NAME VARCHAR(40),
        MENU_ITEM_GROUP_NO INTEGER,
        PRICE DECIMAL(6,2),
        PRIMARY KEY (MENU_ITEM_NO),
        FOREIGN KEY (MENU_ITEM_GROUP_NO)
                REFERENCES MENU_GROUP (MENU_ITEM_GROUP_NO) )
```

Specifies that every value placed in the MENU_ITEM_GROUP_NO column in the MENU_ITEM table must match a corresponding MENU_ITEM_GROUP_NO value in the MENU_ITEM_GROUP table (or contain NULL, i.e. no value).

If the foreign key constraint applies to only one column, a shorthand for the above is to specify the constraint as part of the column definition rather than as an additional <table element> by omitting the FOREIGN KEY. For example:

```
CREATE TABLE CHECK_HEADER (
        CHECK_NO INTEGER NOT NULL,
        SERVER_NO INTEGER REFERENCES "SERVER",
        START_TIME TIMESTAMP(0),
        PRIMARY KEY (CHECK_NO) )
```

Specifies that every value in the SERVER_NO column in the CHECK_HEADER table must match a value in the primary key column of the SERVER table, i.e. the SERVER_NO column.

For multi-column foreign key definitions where one or more of the columns can contain NULLs (i.e. NOT NULL constraint is not specified) the MATCH clause can be added to the foreign key definition to determine what the behaviour should be when NULLs are present in some or all of the columns in the referencing table. The following behaviour is supported:

No MATCH clause	If any column is null then the foreign key is valid.
MATCH PARTIAL	If any column is null, it's not included in the comparison with the referenced table. This means that if all columns in the foreign key are null then it is valid.
MATCH FULL	If any column is null then the foreign key is invalid, unless all the columns are null.

Referential Actions

In some cases, if an attempt were made to delete (or update the referenced key of) a row in a table referenced by a foreign key, it would be preferable to allow the deletion (or key update) rather than reject it. ThinkSQL can help in this situation by automatically performing some actions on the referencing table that will preserve the integrity of the declared relationship while still allowing the deletion (or key update). For example, if an attempt is made to delete a CHECK_HEADER row, which has CHECK_DETAIL rows referencing it, the CHECK_DETAIL rows could be deleted as well as the CHECK_HEADER row. The following definition of the CHECK_DETAIL table would allow this:

```
CREATE TABLE CHECK_DETAIL (
        CHECK_NO INTEGER NOT NULL REFERENCES CHECK_HEADER ON DELETE CASCADE,
        LINE_NO INTEGER NOT NULL,
        MENU_ITEM_NO INTEGER REFERENCES MENU_ITEM,
        QTY DECIMAL(5),
        PRIMARY KEY (CHECK_NO, LINE_NO) )
```

When declaring a foreign key an ON DELETE and/or ON UPDATE clause can be added to specify which referential action to take should a referenced row be deleted or have its referenced key updated. The following actions can be taken:

NO ACTION	The delete (or key update) is rejected if any matching row in the referencing table exists. This is the default behaviour if no referential action is specified.
CASCADE	The delete or key update action is applied to the matching rows in the referencing table, i.e. a cascading deletion will remove all matching rows in the referencing table; a cascading update will update the foreign keys of all matching rows in the referencing table to be the same as the updated referenced key.
SET DEFAULT	The matching rows in the referencing table will have their foreign key columns set to their default values as specified in the table definition.
SET NULL	The matching rows in the referencing table will have their foreign key columns set to NULL.

Check constraints

A check constraint is used to validate the data inserted or updated in a table by evaluating the specified expression and reject any rows where the expression evaluates to false. To specify a check constraint as a member of a <table element list> when creating base tables, use the following:

```
[ CONSTRAINT <constraint name> ]
CHECK ( <search condition> )
```

If no constraint name is given, a unique name is generated by the system. The <search condition> must be deterministic, i.e. it cannot include functions that could return different values at different times, e.g. `CURRENT_DATE`, `CURRENT_USER` etc.
For example, we could add a check constraint to the MENU_ITEM table to ensure every item has a positive price:

```
CREATE TABLE MENU_ITEM (
        MENU_ITEM_NO INTEGER NOT NULL,
        MENU_ITEM_NAME VARCHAR(40),
        MENU_ITEM_GROUP_NO INTEGER,
        PRICE DECIMAL(6,2),
        PRIMARY KEY (MENU_ITEM_NO),
        FOREIGN KEY (MENU_ITEM_GROUP_NO)
                REFERENCES MENU_GROUP (MENU_ITEM_GROUP_NO),
        CHECK (PRICE>0) )
```

If the check constraint applies to only one column, a shorthand for the above is to specify the constraint as part of the column definition rather than as an additional <table element>. For example:

```
CREATE TABLE MENU_ITEM (
        MENU_ITEM_NO INTEGER NOT NULL,
        MENU_ITEM_NAME VARCHAR(40),
        MENU_ITEM_GROUP_NO INTEGER,
        PRICE DECIMAL(6,2) CHECK (PRICE>0),
        PRIMARY KEY (MENU_ITEM_NO),
        FOREIGN KEY (MENU_ITEM_GROUP_NO)
                REFERENCES MENU_GROUP (MENU_ITEM_GROUP_NO) )
```

If the check constraint is to prevent NULLs, i.e. `CHECK (`*`column`* `IS NOT NULL)`, then a commonly used shorthand for this is to add `NOT NULL` to the column definition rather than as an additional <table element>. For example:

```
CREATE TABLE MENU_ITEM_GROUP (
        MENU_ITEM_GROUP_NO INTEGER NOT NULL,
        MENU_ITEM_GROUP_NAME VARCHAR(20),
        PRIMARY KEY (MENU_ITEM_GROUP_NO) )
```

Deferred constraint checking

Constraint checks need not be applied immediately. In some cases it is necessary to delay the check until some other statement has been executed which will bring the database back to a consistent state. An example is where two tables both have foreign key references declared to each other: how do we insert the first row? One way is to defer the constraint checks until a row has been inserted into each table to satisfy the referential constraints. When the constraints are then applied, no violation is reported and the database is in a consistent state.
To support this, one of the following can be appended to any constraint definition:

```
INITIALLY DEFERRED | INITIALLY IMMEDIATE
```

This specifies the default constraint check time: immediate (the default) means check at the end of the statement execution; deferred means leave the checking until the transaction is committed (or constraint checks are requested by the user) – in the latter case, any constraint violation causes the whole transaction to be rolled back.

Also, the following can be added to the constraint definition:

```
[ NOT ] DEFERRABLE
```

which specifies whether the user can defer the constraint or not. The default is for ‘initially immediate’ constraints to be not deferrable; obviously ‘initially deferred’ constraints are deferrable.

To defer deferrable constraints for the current transaction, use:

```
SET CONSTRAINTS ALL DEFERRED
```

To apply deferred constraints immediately, use:

```
SET CONSTRAINTS ALL IMMEDIATE
```

Which will fail if any constraint fails, but will not automatically rollback the transaction and so could be used prior to issuing a `COMMIT`, which might.

Note: candidate keys referenced in a foreign key constraint cannot be deferrable.

Views

Views are predefined queries that return result sets which can be treated as tables. They don't actually store any data, but can have permissions applied to them. To create a view:

```
CREATE VIEW <table name> [( <view column list> )] AS
<query expression>
```

The view must be given a unique name within the schema and the query expression specifies the result set (ordering is not applicable to query expressions). The column names returned in the query expression can be overridden by specifying new names in parentheses after the view name.

Deleting views
To remove a view, use the `DROP VIEW` command:

```
DROP VIEW <table name> { RESTRICT | CASCADE }
```

Procedures and functions

Procedures and functions (SQL-invoked routines) are predefined batches of SQL commands that can receive and return parameters. The routines are stored and executed by the server and have the following advantages over client-side code:

Efficient
Having the server handle repeated execution of commands can be much faster and use fewer resources than repeatedly sending commands from the client.

Centralised
Storing the logic in a single location makes maintenance and application deployment much simpler.

Powerful
The control structures available for use in SQL procedures and functions allow a wide variety of program behaviour.

A procedure can have a number of in-coming, out-going and in-out parameters. A procedure can also return a result set via a cursor. A function is similar, but can only have in-coming parameters and returns a result. Function calls can be placed wherever a value expression would normally appear, whereas a procedure must be explicitly called. A function cannot contain any data modification statements.

Creating procedures and functions
To create a procedure:

```
CREATE PROCEDURE <procedure name>
( [ <SQL parameter declaration list> ] )
<routine body>
```

Where the parameter declaration list contains:

```
[ IN | OUT | INOUT ] <parameter name> <data type>
```

To create a function:

```
CREATE FUNCTION <function name>
( [ <SQL parameter declaration list> ] )
RETURNS <data type>
<routine body>
```

Where the parameter declaration list contains:

```
[ IN ] <parameter name> <data type>
```

And the RETURNS clause specifies the result type. To return the result from a function use:

```
RETURN <value expression>
```

In either case, the routine must be given a unique name within the schema. Notice that the parentheses surrounding the parameter list are mandatory, even if no parameters are declared, and parameters default to `IN` if no direction is specified, e.g.

```
CREATE FUNCTION calculateTax ( amount DECIMAL(8,2) ) RETURNS DECIMAL(8,2)
RETURN amount * 0.175;
```

The routine body is the command to be executed when the routine is invoked – this can be a compound block of statements (each of which must be terminated by a semi-colon) within the `BEGIN` and `END` keywords:

```
BEGIN
      <SQL statement list>
END;
```

e.g.

```
CREATE PROCEDURE setPrice (   item_no INTEGER,
                              new_price DECIMAL(6,2) )
BEGIN
       UPDATE menu_item SET price = new_price WHERE menu_item_no = item_no;
       INSERT INTO audit VALUES (CURRENT_TIMESTAMP, item_no, new_price);
END;
```

Deleting procedures and functions

To remove a routine, use the `DROP` command:

```
DROP { ROUTINE | PROCEDURE | FUNCTION }
<routine name> { RESTRICT | CASCADE }
```

Invocation

To call a procedure use:

```
CALL <procedure name> ( [ <argument list> ] )
```

Where the argument list contains the required in/in-out values and the required out/in-out variables, e.g.

```
CALL setPrice ( 303, 1.95 )
```

Would increase the price of Lager in the sample schema to 1.95.

To call a function, wherever a value expression can be used:

```
<function name> ( [ <argument list> ] )
```

Where the argument list contains the required in values and the result appears in an appropriate place, e.g.

```
SELECT
       CHECK_NO,
       SUM(QTY*PRICE) AS CHECK_TOTAL,
       calculateTax( SUM(QTY*PRICE) ) AS TAX
FROM
       MENU_ITEM JOIN CHECK_DETAIL USING (MENU_ITEM_NO)
GROUP BY CHECK_NO
```

Would retrieve all check totals and their calculated tax.

Variables

At the start of a compound statement or block, variables can be declared which are visible within the scope of the block and these can be referenced in expressions. To declare a variable use:

```
DECLARE <SQL variable name list>
      <data type> [ <default clause> ]
```

Where the default clause specifies an initial value and can be:

```
DEFAULT
      <literal>
      | NULL
      | CURRENT_DATE
      | CURRENT_TIME
      | CURRENT_TIMESTAMP
      | USER
      | CURRENT_USER
      | SESSION_USER
      | SYSTEM_USER
```

A variable (or an outgoing parameter) can be assigned a value using:

```
SET <variable> = <value expression>
```

Variables (and outgoing parameters) can also be assigned values from a single-row select statement:

```
SELECT <value expression list>
INTO <variable list>
FROM
<single row table expression>
```

An example function which declares and sets a number of variables:

```
CREATE FUNCTION priceRange() RETURNS DECIMAL(8,2)
BEGIN
       DECLARE lowest, highest, difference DECIMAL(8,2);

       SELECT MIN(price) INTO lowest FROM menu_item;

       SELECT MAX(price) INTO highest FROM menu_item;

       SET difference = highest – lowest;

       RETURN difference;
END;
```

Note that the semi-colon at the end of the function is required.

Control flow

Procedures and functions can make use of the following compound statements to control the flow of the routine:

IF

```
IF <search condition> THEN
      <SQL statement list>
      [ elseif clause list ]
[ ELSE <SQL statement list> ]
END IF
```

Where the elseif clause list can contain a number of:

```
ELSEIF <search condition> THEN
      <SQL statement list>
```

If the first search condition is true, the SQL statement list is processed and then control is passed to the statement after the `END IF`. Otherwise any elseif clause conditions are tested in sequence until one is found to be true, in which case its SQL statement list is processed and control passes to the statement following the `END IF`. Finally, if none of the conditions were true, and an `ELSE` clause was specified, then its SQL statement list is processed. E.g.

```
IF CURRENT_USER IN ('restaurant') THEN
       SELECT MAX (PRICE) INTO r FROM restaurant.menu_item;
ELSE
       SET r=NULL;
END IF;
```

WHILE

```
[ <start label>: ]
WHILE <search condition> DO
      <SQL statement list>
END WHILE [ <end label> ]
```

The SQL statement list is repeatedly processed while the search condition is true. The condition is tested before the list is processed, so the list won't be processed at all if the expression is initially false. The start/end labels must match if specified. E.g.

```
WHILE n>0 DO
       INSERT INTO testdata VALUES (n);
       SET n=n-1;
END WHILE
```

CASE

```
CASE
      <when clause list>
[ ELSE <SQL statement list> ]
END CASE
```

Where the when clause list can contain a number of:

```
WHEN <search condition> THEN
      <SQL statement list>
```

Each when clause condition is tested in sequence until one is found to be true, in which case its SQL statement list is processed and control passes to the statement following the `END CASE.` Finally, if none of the conditions were true, and an `ELSE` clause was specified, then its SQL statement list is processed.

REPEAT

```
[ <start label>: ]
REPEAT
      <SQL statement list>
UNTIL <search condition>
END REPEAT [ <end label> ]
```

The SQL statement list is repeatedly processed until the search condition is true. The condition is tested after the list is processed, so the list will be processed at least once. The start/end labels must match if specified.

LOOP

```
[ <start label>: ]
LOOP
      <SQL statement list>
END LOOP [ <end label> ]
```

The SQL statement list is repeatedly processed. The only way to stop the loop is to either `LEAVE` or `RETURN` (if the loop is in a function). The start/end labels must match if specified.

LEAVE

```
LEAVE <label>
```

The current compound statement (often a loop) is terminated and control is passed back to the block which has the specified label. This could be a block several levels of nesting away.

ITERATE

```
ITERATE <label>
```

The current iteration of the loop statement, which has the specified label, is stopped and control is passed back to the start of the loop. This could be a loop several levels of nesting away.

RETURN

```
RETURN <value expression>
```

The current function is terminated and the expression is passed back to caller.

Cursors

At the start of a compound statement or block, cursors can be declared. A cursor allows the routine to step through a result set one row at a time, with the 'current row' being pointed to by the current cursor position.

Declaring a cursor

To declare a cursor use:

```
DECLARE <cursor name> [ SENSITIVE | INSENSITIVE | ASENSITIVE ]
[ SCROLL ] CURSOR
[ WITH HOLD ]
[ WITH RETURN ]
FOR <query expression> [ <order by clause> ]
[ FOR { READ ONLY | UPDATE [ OF <column name list> ] } ]
```

The cursor name must be unique to the current session. The sensitivity of the cursor is set using the `SENSITIVE/INSENSITIVE` keywords and relates to whether changes to the data are visible to the cursor as it steps through the results. At the moment the sensitivity option is ignored because this is governed by the transaction's isolation level, e.g. 'serializable' means no changes are seen, whereas 'read committed' would mean committed changes are visible.

Scrollable cursors are created using the optional `SCROLL` keyword. These will be available in a future release, and allow random access navigation through the cursor.

If `WITH HOLD` is specified, the cursor remains open after the transaction has been committed. A rollback would always close the cursor. If this option is not specified then the cursor is always closed at the end of the transaction in which it was opened.

The `WITH RETURN` option will make the cursor available after the procedure has finished, if the cursor is open. The caller of the routine can then treat the result-set from this open cursor as any other query result-set.

Following the query expression, which defines the result-set for the cursor (with an optional ordering), FOR READ ONLY or FOR UPDATE can be specified. These determine whether the current row of the cursor can be updated, and are currently ignored: cursors are read-only in this release, but this might change in a future release.

Opening a cursor
To open a declared cursor:

```
OPEN <cursor name>
```

The cursor is positioned before the first row of the result-set.

Fetching a row from a cursor
To fetch a row from an open cursor:

```
FETCH [ [ <fetch orientation> FROM ] <cursor name>
INTO <fetch target list>

<fetch orientation> ::=
NEXT | PRIOR | FIRST | LAST
| ABSOLUTE <simple value> | RELATIVE <simple value>
```

The currently supported fetch orientation is NEXT (the default), which moves to the next row in the cursor. Other orientations will be supported in a future release.
To check when the cursor is moved after the last row, i.e. there are no more rows to be fetched, the SQLSTATE session variable can be used: a value starting with '02' (i.e. a class of '02', e.g. '02000') indicates no more data; a value starting with '00' indicates success, i.e. the fetch successfully moved to a valid row.
The use of SQLSTATE is currently limited to cursor operations, but its use will be extended when full error handling is available to user-defined routines.

Closing a cursor
To close an open cursor:

```
CLOSE <cursor name>
```

Examples

```
-- Sets variable 'a' to the highest menu_item_no
-- (not as efficient as SELECT MAX(menu_item_no) INTO a FROM restaurant.menu_item)

DECLARE test CURSOR
       FOR SELECT menu_item_no FROM restaurant.menu_item
       ORDER BY menu_item_no
       FOR READ ONLY;

DECLARE a INTEGER;

OPEN test;

WHILE SQLSTATE<>'02000' DO
       FETCH test INTO a;
END WHILE;

CLOSE test;
```

Sequences

Assigning unique values when inserting data is a common requirement and can be achieved using the MAX set function to find the current highest value in a column and adding 1 to it before inserting the new value. There are two drawbacks to this approach:

1. finding the current highest value is not a trivial operation in a multi-user environment, since the highest value depends on the current transaction's isolation level (e.g. whether uncommitted, or indeed committed, values are to be included or not).

2. during the time between the highest value being found and the incremented value being inserted, another user could use the same value leading to duplicates or one of the inserts being rejected if the column has a unique/primary-key constraint

An improvement on this, albeit an extension to the SQL standard at the moment, is to get the server to manage the unique values. ThinkSQL does this using sequences, which are also used internally for system catalogue entries.

A sequence is a counter that can be incremented by users to generate unique values. The counters are cached to reduce disk access (the default is to cache 20 values), and efficient concurrent access is achieved by queuing the requests for numbers. Values issued from a sequence constantly increase, so if a statement is rolled back the issued number is lost and not re-issued. This, along with multi-users sharing the same sequence can lead to gaps appearing in repeated calls to retrieve the next sequence value.

Creating a sequence
To create a sequence:

```
CREATE SEQUENCE <sequence name>
[ STARTING AT <simple value> ]
```

where the optional `STARTING AT` value specifies the first value to be issued. The default is 1.

Using a sequence
To get the next and latest values from a sequence use the following functions:

NEXT_SEQUENCE (<sequence name>)

This returns the next value in the sequence and has the side-effect of incrementing the counter.

LATEST_SEQUENCE (<sequence name>)

This returns the latest value taken from the sequence by this session (or null if none has been issued yet). It does not increment the counter.

Note that if more than one of these functions is used in a statement for the same sequence then the order of evaluation affects the outcome, e.g.

```
VALUES( NEXT_SEQUENCE(seq1), LATEST_SEQUENCE(seq1) )
```

is different from

```
VALUES( LATEST_SEQUENCE(seq1), NEXT_SEQUENCE(seq1) )
```

The two sequence functions can be used wherever a standard SQL function can be used, plus they can be used as column defaults in table definitions. Using `NEXT_SEQUENCE` as a column default is an ideal way of ensuring that new rows have a default unique value, while still allowing specific values to be inserted when needed, e.g.

```
CREATE SEQUENCE test_keys;
CREATE TABLE test
       (id INTEGER DEFAULT NEXT_SEQUENCE(test_keys) NOT NULL PRIMARY KEY,
        name VARCHAR(100) );
COMMIT;

INSERT INTO test (name) VALUES ('First'), ('Second'), ('Third');
INSERT INTO test (id, name) VALUES (0, 'Zero');

SELECT * FROM test;
```

Returns (assuming no other users started using the new sequence):

```
|id      |name
================================================================================
|       1|First
|       2|Second
|       3|Third
|       0|Zero
4 rows affected
Processing time: 00:00:00:010
```

Note: if no schema prefix is specified in a DEFAULT NEXT_SEQUENCE, the schema owning the table is assumed to be the sequence owner (not the current schema).

Deleting sequences
To remove a sequence, use the DROP command:

```
DROP SEQUENCE <sequence name> { RESTRICT | CASCADE }
```

Rights and privileges

The tables, views and other objects created within a schema can be fully accessed by the schema owner (e.g. as specified by the authorization clause in the schema's creation). Other users (authorization Ids) cannot access them unless they are given the privilege to do so via the GRANT statement. The following privileges can be granted:

Privilege	Description
SELECT	Privilege to access all columns in a specified table
SELECT(*c*)	Privilege to access a specific column in a specified table
DELETE	Privilege to delete rows from a specified table
INSERT	Privilege to insert into all columns in a specified table
INSERT(*c*)	Privilege to insert into a specific column in a specified table
UPDATE	Privilege to update all columns in a specified table
UPDATE(*c*)	Privilege to update a specific column in a specified table
REFERENCES	Privilege to reference all columns in a specified table from integrity constraints
REFERENCES(*c*)	Privilege to reference a specific column in a specified table from integrity constraints
USAGE	Privilege to use a specified domain
EXECUTE	Privilege to execute a specified function or procedure

Grant

The GRANT statement is as follows:

```
GRANT <object privileges> ON <object name>
TO <grantees>
[ WITH GRANT OPTION ]
```

Where <object_name> is either [TABLE] <table name> or DOMAIN <domain name> or PROCEDURE <procedure name> or FUNCTION <function name> or ROUTINE <routine name> and <grantees> is either PUBLIC (to mean all users, including future ones) or a comma separated list of <authorization identifier>s to indicate one or more specific users. The <object privileges> can be either ALL PRIVILEGES (to mean all privileges that the issuing user currently has) or a comma-separated list of the following actions which map onto the ones specified in the table above:

```
SELECT [ ( <privilege column list> ) ]
| DELETE
| INSERT [ ( <privilege column list> ) ]
| UPDATE [ ( <privilege column list> ) ]
| REFERENCES [ ( <privilege column list> ) ]
| USAGE
| EXECUTE
```

The optional WITH GRANT OPTION passes on the ability to grant these privileges to the specified user(s). Without this, the user(s) can make use of the privileges but can't pass them on to others.

An example GRANT statement which allows all users to read all columns in the CHECK_DETAIL table is:

```
GRANT SELECT ON CHECK_DETAIL TO PUBLIC
```

Revoke

A granted privilege can be removed using the REVOKE statement, which has the following syntax:

```
REVOKE [ GRANT OPTION FOR ] <object privileges>
ON <object name> FROM <grantees>
{ RESTRICT | CASCADE }
```

Where the <object privileges>, <object name> and <grantees> are the same as for the GRANT statement. If GRANT OPTION FOR is specified then the privilege itself is not revoked, just the ability to grant it to others.

An example REVOKE statement which removes a user's ability to execute a procedure is:

```
REVOKE EXECUTE ON setPrice FROM restaurant RESTRICT
```

9. Manipulating data

To add, change, or remove data from the tables in a schema the SQL commands described below are available. Each of these is subject to the constraints defined in the schema, and so could be rejected. See the section on constraints for more details.

Insert

This inserts new rows into a table.

```
INSERT INTO <table name>
[ ( <column name list> ) ] <query expression>
| DEFAULT VALUES
```

If the <column name list> is omitted all columns in the table are assumed in the left-right order that they were declared in the CREATE TABLE statement.
If DEFAULT VALUES is specified as the source of the new rows then a single row is inserted in which every column has its default value.
The <query expression> specifies the row or rows to be inserted. Each column from left to right is put into the corresponding column specified for the table from left to right. The <query expression> is typically either a SELECT statement or a VALUES table constructor. If VALUES is used, columns can be specified as DEFAULT or NULL.
An example:

```
INSERT INTO “server” VALUES (2,'Mary Jones')
```

Another example that inserts two rows:

```
INSERT INTO “server” VALUES
(1,'John Smith'),
(2,'Mary Jones')
```

Another example, assuming we've created a new table, check_header_copy, with the same definition as the check_header table:

```
INSERT INTO check_header_copy (check_no, server_no, start_time)
 SELECT (check_no, server_no, start_time) FROM check_header
```

Update

This updates existing rows in a table.

```
UPDATE <target table>
SET <set clause list>
[ WHERE <search condition> ]
```

Where the <set clause list> is a comma-separated list of one or more:

```
<column name> = <value expression>
```

If the WHERE clause is omitted, every row in the table is updated; otherwise only the rows that match the <search condition> are updated.
An example which increases every menu_item price by 10%:

```
UPDATE menu_item SET price=price*1.10
```

An example which reduces menu_item prices by 10% for items that are currently over 5.00:

```
UPDATE menu_item SET price=price*0.90 WHERE price>5
```

Delete

This deletes existing rows from a table.

```
DELETE FROM <target table>
[ WHERE <search condition> ]
```

If the WHERE clause is omitted, every row in the table is deleted; otherwise only the rows that match the <search condition> are deleted.

An example which deletes Paella from the menu:

```
DELETE FROM menu_item WHERE menu_item_no=102
```

10. Server connections

Connecting

To connect to the server, use one of the client APIs to submit a username and a password, and an optional server and catalog name. This will connect to the server and then connect to a particular catalog, or to the server's primary (default) catalog if none is specified.

To connect directly (e.g. via ISQLraw) to a catalog use:

```
CONNECT TO  { DEFAULT
            | <SQL-server-name>
                  [ AS <connection name> ]
                  [ USER <connection user name>
                        [ PASSWORD <password> ]
                  ]
            }
```

`CONNECT TO DEFAULT` will connect to the server's primary catalog as the DEFAULT user with the default schema of DEFAULT_SCHEMA. Specifying a user connects as that user.

The SQL-server-name can be left empty to connect to the server's primary catalog, or it can specify the server and catalog name separated by a full stop (.). The server name is reserved for future use and is currently ignored, so it can be omitted. The following SQL-server-names all refer to the same catalog (assuming db1 is the server's primary catalog):

‘’
‘thinksql.db1’
‘.db1’

This example connects as the sample schema owner on the server's primary catalog:

```
CONNECT TO ‘’ USER ‘RESTAURANT’
```

This example connects as BOB (with a password) to a catalog named cat2 (note that the cat2 catalog must already be open on the server):

```
CONNECT TO ‘.cat2’ USER ‘BOB’ PASSWORD ‘restrict123’
```

This example connects as the ADMIN user on the server's primary catalog (therefore receiving special privileges such as the ability to shutdown the server):

```
CONNECT TO ‘’ USER ‘ADMIN’ PASSWORD ‘admin’
```

Note that if a statement is issued before an explicit `CONNECT`, then the server will perform an implicit `CONNECT TO DEFAULT` before executing the statement.

Disconnecting

To disconnect from the server, use one of the client APIs to issue a disconnect call.

To disconnect directly (e.g. via ISQLraw) from a catalog use:

```
DISCONNECT  { DEFAULT
            | CURRENT
            | ALL
            | <connection name>
            }
```

`DEFAULT`, `CURRENT` and `ALL` refer to the current connection. If a connection name is specified, it must match the current connection's name for the disconnection to succeed.

11. Transaction processing

Every data manipulation or retrieval command issued against the database is in the context of a current transaction. Transactions are used to group statements that need to be executed together (often referred to as being *atomic*). The whole transaction (comprising many statements) can be confirmed or discarded.

The classic example is a transfer of money from bank account A to bank account B. The two steps, debiting A by £50 and crediting B by £50, must both be completed or neither of them must be completed: an all-or-nothing scenario. If there is a hardware failure between the two steps the database must remain in a consistent state. Many database statements need to be grouped in this way to ensure data integrity.

Note: unlike most other database systems, ThinkSQL treats every statement, including `CREATE` and `DROP` statements, as part of a transaction. These should be committed, and can be rolled back, like any other modification statement.

Starting a transaction

A transaction is implicitly started by the first data manipulation or retrieval command.

Commit

This command confirms the changes made by the current transaction. This ensures that all changes are saved.

```
COMMIT [ WORK ]
```

Rollback

This command discards all changes made by the current transaction. If the database server is stopped abruptly or a client disconnects before committing, any pending transaction is implicitly rolled-back.

```
ROLLBACK [ WORK ]
```

Transaction isolation

With multiple concurrent users, because the data modified during a transaction is not necessarily committed immediately, and because some transactions need to be executed in isolation, users need to decide whether their transaction can see changes made by others: this is known as the *isolation level*. To set the isolation level for the next transaction:

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE }
```

This setting must be performed before the transaction starts and lasts for the duration of the next transaction, i.e. until the next `COMMIT` or `ROLLBACK`.

The four isolation levels specified by SQL are:

Read uncommitted

This makes modifications by other transactions immediately visible, even if they haven't been committed yet (and so could still be rolled-back). This provides access to the very latest information at the risk of reading *dirty data*: data that may not be confirmed.

Read committed

This makes modifications by other transactions visible as soon as they have been committed. This avoids dirty reads, but can still give rise to *non-repeatable reads* and *phantom rows*. A non-repeatable read could arise if this transaction reads a row which is then modified by a committed transaction: re-reading the row will return different values. Phantom rows appear if this transaction reads rows that satisfy a given condition and another transaction inserts and commits new rows that also satisfy the condition: re-reading the rows will retrieve the previously unseen rows.

Repeatable read

This avoids the problems of dirty-reads and non-repeatable reads, but not necessarily phantom rows. In ThinkSQL it is implemented as serializable.

Serializable

This avoids dirty-reads, non-repeatable reads and phantom rows. Data modifications by other concurrent transactions are not made visible, *even after they have been committed*. Only data changes made by transactions that were committed when this transaction starts are visible, effectively isolating this transaction from any other and giving the impression that contemporary transactions are run in serial (hence the name).
This is the default isolation level in standard SQL.

Multiple version implementation

ThinkSQL uses multiple versions to support concurrent users and the isolation levels specified above. This gives greater access to data than more traditional locking schemes. In particular, a reader (even an isolated one) need never wait for access to rows, nor does it block other users from reading or updating or even deleting the same rows. Similarly, a writer never blocks other users from reading rows.

If a transaction attempts to update or delete a row which has already been modified by a later transaction (one that started after this one) then the attempt is rejected since it is too late in terms of serialisation of the transactions. The transaction should be rolled-back or committed as-is and restarted so that the update can be re-tried.

Because the default isolation is serializable, changes committed by subsequent transactions won't be visible so it may be necessary to issue a `COMMIT` before being able to see the latest changes. Alternatively, set the isolation level to read committed.

12. Managing the server

The following commands are not part of the standard grammar but have been added to allow manipulation of catalogs, users and the server.

Catalog maintenance

A catalog is a collection of schemas and users stored in a single operating system file, e.g. db1.dat. Each catalog has an ADMIN user who has special privileges within the catalog.

The ThinkSQL server can manage multiple open catalogs at the same time. When the server starts, the first catalog to be opened (db1 by default) becomes the primary catalog. The ADMIN user of this primary catalog has special privileges to do the following on the server:

- Shutdown the server
- Open and close catalogs on the server
- Create new catalogs
- Backup the primary catalog

Note that catalogs are self-contained to simplify deployment and maintenance, i.e. they contain their own user definition tables, transaction logs etc. and currently cross-catalog transactions (and so queries) are not supported.

It is recommended that you make regular offline and online backups of your catalog files.

Create

To create a new catalog:

```
CREATE CATALOG <catalog name>
```

The current transaction will be rolled back and the new catalog will remain open on the server as the default catalog for this connection.

For example:

```
CREATE CATALOG cat2
```

Would create and open a new catalog named cat2 (which would be stored permanently as an operating system disk file named cat2.dat).

The command fails if the catalog already exists. To drop a catalog, you need to use the operating system to delete or rename the catalog file (e.g. `rename cat2.dat cat2_old.dat`).

You must be the ADMIN user connected to the server's primary catalog. If no primary catalog is available, then this command can be issued without having to connect as ADMIN.

Open

To open a catalog for use on the server:

```
OPEN CATALOG <catalog name>
```

You must be the ADMIN user connected to the server's primary catalog.

Close

To close an open catalog on the server:

```
CLOSE CATALOG <catalog name>
```

Any users connected to the catalog will have their transactions automatically rolled back and will be disconnected.
The catalog cannot be the current catalog. This means the primary catalog cannot be closed using this command.

You must be the ADMIN user connected to the server's primary catalog.

Backup (online)

To create an on-line backup of the current catalog in a new catalog:

```
BACKUP CATALOG TO <backup catalog name>
```

The backup is taken as a snapshot based on the current transaction's view of the catalog. Other transactions can continue to access the catalog during the backup but any modifications by active or future transactions will not be included in the backup. The backup differs from an off-line copy of the catalog file because it performs the following tasks:

- Removes all old versions of rows
- Reorganises pages contiguously per table (i.e. defragments the catalog)
- Removes unallocated pages from the catalog file
- Re-indexes all tables
- Resets the transaction and statement status tables, which reduces each transaction's memory needs, initialisation time and table access time

These steps lead to improved server performance when accessing data in the catalog, and so such a backup should be done after periods of heavy updates to the catalog and the backup then used as the new live catalog (by simply renaming the catalog file using operating system commands, e.g. `rename db1.dat db1_old.dat` and then `rename db1_backup.dat db1.dat` (assuming the catalog backup was done to db1_backup and the live catalog is db1 and is off-line)).

The command fails if the backup catalog already exists. To drop a catalog, you need to use the operating system to delete or rename the catalog file (e.g. `rename db1_backup.dat db1_backup_old.dat`).

You must be the ADMIN user connected to the server's primary catalog.

Backup (offline)

To create an off-line backup of a catalog, ensure the catalog is not currently opened by an instance of the ThinkSQL server and then use the operating system file copy command to make a backup of the catalog file, e.g.

Windows

```
COPY /V db1.dat db1_backup.dat
```

Linux

```
cp db1.dat db1_backup.dat
```

Garbage collection

Each time the server starts, a garbage collection thread is automatically initiated which removes old versions of rows from the primary catalog. For other catalogs that have been opened on the server, a garbage collection thread can be started manually using:

```
GARBAGE COLLECT CATALOG <catalog name>
```

This should be done regularly for each active catalog, particularly after periods of heavy updates, to ensure unused table space is re-used.

You must be the ADMIN user connected to the server's primary catalog.

User maintenance

Each catalog has an ADMIN user who has special privileges to maintain users and schemas within that catalog. The default password for this user is 'admin' (lower case). The ADMIN user does not initially own any schemas. It does connect to a default schema of DEFAULT_SCHEMA, but has no privilege to create objects in it because it is owned by the DEFAULT user.

To see a list of the current catalog's users and their default schemas, use the INFORMATION_SCHEMA.USERS view.

Note: the following changes will only become visible to other transactions (and permanent) after they have been committed.

Creating new users

To create a new user, who will connect with a current schema of DEFAULT_SCHEMA:

```
CREATE USER <user name> [ PASSWORD <password> ]
```

The optional password clause expects a character string literal to be used to protect future connections by the new user. If no password clause is given, then the new user can specify any or no password to connect in future and the connection will always be allowed.

Example:

```
CREATE USER bob PASSWORD 'restrict123'
```

Often a new schema will be created for a new user. Creating a schema (which can be empty to start with) and specifying the new user in the authorization clause will automatically set the user's default schema to be the new schema, e.g:

```
CREATE SCHEMA bob_schema AUTHORIZATION bob;
```

will create a new schema, bob_schema, owned by bob and will set bob's default schema to bob_schema.

You must be the ADMIN user to create new users, and to create new schemas authorised by another user.

Modifying users

To modify an existing user's details:

```
ALTER USER <user name> SET
      { PASSWORD <password>
      | DEFAULT SCHEMA <schema name> }
```

SET PASSWORD can be used to change the user's password. Use SET PASSWORD NULL to indicate that no password is required (and any that is then provided will be ignored).
SET DEFAULT SCHEMA can be used to change the schema that the user initially connects to.

Examples:

```
ALTER USER bob SET PASSWORD 'new456';

ALTER USER bob SET DEFAULT SCHEMA restaurant
```

You must be the user, or the ADMIN user.

Deleting users

To delete an existing user:

```
DROP USER <user name> { RESTRICT | CASCADE }
```

You must be the ADMIN user.

Controlling transactions

To cancel the currently executing statement(s) of the specified transaction (within the current catalog):

```
CANCEL <transaction reference>
```

where the transaction reference can be found from the first part of the number specified against the connection when selecting 'Connections' from the ThinkSQL server monitor. E.g. 14 for the first entry in the following list (or INFORMATION_SCHEMA.ACTIVE_TRANSACTIONS."TRANSACTION"):

```
Current connections:
Catalog db1:
DEFAULT                         0000000014:0000000000 Serializable
MONITOR                         0000000000:0000000000 Serializable
```

You must be connected as the same user, or the ADMIN user.

To abort the currently executing statement(s) of the specified transaction (within the current catalog):

```
KILL <transaction reference>
```

where the transaction reference can be found as for the CANCEL command. The transaction is then rolled-back and its connection is closed.

You must be connected as the same user, or the ADMIN user.

Stopping the server

To stop the server and close any existing connections:

```
SHUTDOWN
```

If any transactions are still active they are automatically rolled-back.

You must be the ADMIN user connected to the server's primary catalog. If no primary catalog is available, then this command can be issued without having to connect as ADMIN.

13. Some more example queries

Example 1: list check line details sorted by check number

```
SELECT
        CHECK_NO, LINE_NO, MENU_ITEM_GROUP_NAME, MENU_ITEM_NAME, PRICE, QTY
FROM
        “SERVER” NATURAL JOIN
        CHECK_HEADER NATURAL JOIN
        MENU_ITEM_GROUP NATURAL JOIN
        MENU_ITEM NATURAL JOIN
        CHECK_DETAIL
ORDER BY
        CHECK_NO, LINE_NO
```

returns:

```
|CHECK_NO| LINE_NO|MENU_ITEM_GROUP_NAME|                          MENU_ITEM_NAME|  PRICE|  QTY
=============================================================================================
|       1|       1|             Starter|                          Prawn Cocktail|   2.95|    1
|       1|       2|         Main Course|                              Irish stew|   3.95|    1
|       1|       3|                Wine|                                Red Wine|   2.10|    1
|       2|       1|             Starter|                                 Samosas|   1.95|    1
|       2|       2|         Main Course|                           Chicken Madras|   5.45|    2
|       2|       3|           Side-dish|                              Boiled Rice|   1.95|    2
|       2|       4|                Beer|                                    Lager|   1.85|    1
|       2|       5|          Soft drink|                                Lemonade|   1.20|    1
|       3|       1|         Main Course|                                 Lasagne|   4.95|    1
|       3|       2|         Main Course|                         Boeuf Bourgignone|   7.95|    1
|       3|       3|           Side-dish|                             Mashed Potato|   1.45|    1
|       3|       4|                Wine|                                White Wine|   2.10|    2
|       4|       1|         Main Course|        Roast Beef and Yorkshire Pudding|   6.45|    2
|       4|       2|                Beer|                                  Guinness|   2.99|    1
|       4|       3|            Dessert|                                 Ice Cream|   2.99|    1
|       5|       1|         Main Course|                         Boeuf Bourgignone|   7.95|    1
|       5|       2|         Main Course|        Roast Beef and Yorkshire Pudding|   6.45|    1
|       5|       3|           Side-dish|                             Mashed Potato|   1.45|    1
|       5|       4|           Side-dish|                               Boiled Rice|   1.95|    1
|       5|       5|                Wine|                                Red Wine|   2.10|    2
|       5|       6|          Soft drink|                                Lemonade|   1.20|    2
|       5|       7|            Dessert|                                 Ice Cream|   2.99|    2
22 rows affected
```

Example 2: list all menu items and bracket their prices into maximum, minimum or in-between

```
SELECT
        MENU_ITEM_NAME,
        PRICE,
        CASE PRICE
                WHEN (SELECT MAX(PRICE) FROM MENU_ITEM) THEN 'MOST'
                WHEN (SELECT MIN(PRICE) FROM MENU_ITEM) THEN 'LEAST'
        ELSE
                'IN BETWEEN'
        END AS EXTREMITY
FROM
        MENU_ITEM
```

returns:

```
|MENU_ITEM_NAME                          |PRICE  |EXTREMITY
=========================================================
|Soup of the Day                         |   2.45|IN BETWEEN
|Samosas                                 |   1.95|IN BETWEEN
|Prawn Cocktail                          |   2.95|IN BETWEEN
|Lasagne                                 |   4.95|IN BETWEEN
|Spaghetti Bolognese                     |   1.95|IN BETWEEN
|Paella                                  |   6.95|IN BETWEEN
|Borsch                                  |   4.95|IN BETWEEN
|Irish stew                              |   3.95|IN BETWEEN
|Kedgeree                                |   4.99|IN BETWEEN
|Boeuf Bourgignone                       |   7.95|MOST
|Roast Beef and Yorkshire Pudding        |   6.45|IN BETWEEN
|Chicken Madras                          |   5.45|IN BETWEEN
|Chicken Tikka Masala                    |   5.95|IN BETWEEN
|Boiled Rice                             |   1.95|IN BETWEEN
|Mashed Potato                           |   1.45|IN BETWEEN
|Red Wine                                |   2.10|IN BETWEEN
|White Wine                              |   2.10|IN BETWEEN
|Guinness                                |   2.99|IN BETWEEN
|Lager                                   |   1.85|IN BETWEEN
|Lemonade                                |   1.20|LEAST
|Cola                                    |   1.20|LEAST
|Ice Cream                               |   2.99|IN BETWEEN
22 rows affected
```

Example 3: list all menu items and bracket their prices into 3 bands

```
SELECT
        MENU_ITEM_NAME,
        PRICE,
        CASE
                WHEN PRICE < 2.00 THEN 'LOW'
                WHEN PRICE > 6.00 THEN 'HIGH'
        ELSE
                'MEDIUM'
        END AS PRICE_BAND
FROM
        MENU_ITEM
```

returns:

```
|MENU_ITEM_NAME                          |PRICE  |PRICE_BAND
=========================================================
|Soup of the Day                         |   2.45|MEDIUM
|Samosas                                 |   1.95|LOW
|Prawn Cocktail                          |   2.95|MEDIUM
|Lasagne                                 |   4.95|MEDIUM
|Spaghetti Bolognese                     |   1.95|LOW
|Paella                                  |   6.95|HIGH
|Borsch                                  |   4.95|MEDIUM
|Irish stew                              |   3.95|MEDIUM
|Kedgeree                                |   4.99|MEDIUM
|Boeuf Bourgignone                       |   7.95|HIGH
|Roast Beef and Yorkshire Pudding        |   6.45|HIGH
|Chicken Madras                          |   5.45|MEDIUM
|Chicken Tikka Masala                    |   5.95|MEDIUM
|Boiled Rice                             |   1.95|LOW
|Mashed Potato                           |   1.45|LOW
|Red Wine                                |   2.10|MEDIUM
|White Wine                              |   2.10|MEDIUM
|Guinness                                |   2.99|MEDIUM
|Lager                                   |   1.85|LOW
|Lemonade                                |   1.20|LOW
|Cola                                    |   1.20|LOW
|Ice Cream                               |   2.99|MEDIUM
22 rows affected
```

Example 4: list all menu items that start with ‘chicken’

```
SELECT
  *
FROM
  MENU_ITEM
WHERE
  MENU_ITEM_NAME LIKE 'CHICKEN%'
```

returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                       |MENU_ITEM_GROUP_NO|PRICE
=============================================================================
|         108|Chicken Madras                       |                 2|   5.45
|         109|Chicken Tikka Masala                 |                 2|   5.95
2 rows affected
```

Example 5: list all menu items that have never been ordered (using a sub-select)

```
SELECT
  *
FROM
  MENU_ITEM
WHERE
  MENU_ITEM_NO NOT IN (SELECT DISTINCT MENU_ITEM_NO FROM CHECK_DETAIL)
```

returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|           1|Soup of the Day                        |                 1|   2.45
|         101|Spaghetti Bolognese                    |                 2|   1.95
|         102|Paella                                 |                 2|   6.95
|         103|Borsch                                 |                 2|   4.95
|         105|Kedgeree                               |                 2|   4.99
|         109|Chicken Tikka Masala                   |                 2|   5.95
|         305|Cola                                   |                 5|   1.20
7 rows affected
```

Example 6: list all check lines for menu item number 108 with a quantity of 2

```
SELECT *
FROM
  CHECK_DETAIL
WHERE
  (MENU_ITEM_NO, QTY) = (108, 2)
```

returns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NO|QTY
====================================
|       2|       2|         108|    2
1 row affected
```

Example 7: list all menu items priced 1.85, 1.95 or 2.10

```
SELECT *
FROM
       MENU_ITEM
WHERE
       PRICE IN (1.85, 1.95, 2.10)
ORDER BY PRICE
```

returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|         303|Lager                                  |                 6|   1.85
|         200|Boiled Rice                            |                 3|   1.95
|         101|Spaghetti Bolognese                    |                 2|   1.95
|           2|Samosas                                |                 1|   1.95
|         301|White Wine                             |                 7|   2.10
|         300|Red Wine                               |                 7|   2.10
6 rows affected
```

Example 8: list each check line’s details and its extended line total

```
SELECT
       CHECK_NO, LINE_NO, MENU_ITEM_NAME, QTY, PRICE, QTY*PRICE AS EXTENSION
FROM
       MENU_ITEM M CROSS JOIN CHECK_DETAIL D
WHERE
       D.MENU_ITEM_NO=M.MENU_ITEM_NO
```

returns:

```
|CHECK_NO|LINE_NO |MENU_ITEM_NAME                          |QTY  |PRICE  |EXTENSION
================================================================================
|       1|       1|Prawn Cocktail                          |    1|   2.95|        2.95
|       1|       2|Irish stew                              |    1|   3.95|        3.95
|       1|       3|Red Wine                                |    1|   2.10|        2.10
|       2|       1|Samosas                                 |    1|   1.95|        1.95
|       2|       2|Chicken Madras                          |    2|   5.45|       10.90
|       2|       3|Boiled Rice                             |    2|   1.95|        3.90
|       2|       4|Lager                                   |    1|   1.85|        1.85
|       2|       5|Lemonade                                |    1|   1.20|        1.20
|       3|       1|Lasagne                                 |    1|   4.95|        4.95
|       3|       2|Boeuf Bourgignone                       |    1|   7.95|        7.95
|       3|       3|Mashed Potato                           |    1|   1.45|        1.45
|       3|       4|White Wine                              |    2|   2.10|        4.20
|       4|       1|Roast Beef and Yorkshire Pudding        |    2|   6.45|       12.90
|       4|       2|Guinness                                |    1|   2.99|        2.99
|       4|       3|Ice Cream                               |    1|   2.99|        2.99
|       5|       1|Boeuf Bourgignone                       |    1|   7.95|        7.95
|       5|       2|Roast Beef and Yorkshire Pudding        |    1|   6.45|        6.45
|       5|       3|Mashed Potato                           |    1|   1.45|        1.45
|       5|       4|Boiled Rice                             |    1|   1.95|        1.95
|       5|       5|Red Wine                                |    2|   2.10|        4.20
|       5|       6|Lemonade                                |    2|   1.20|        2.40
|       5|       7|Ice Cream                               |    2|   2.99|        5.98
22 rows affected
```

Example 9: list all check numbers and their totals

```
SELECT
        CHECK_NO, SUM(QTY*PRICE) AS CHECK_TOTAL
FROM
        MENU_ITEM JOIN CHECK_DETAIL USING (MENU_ITEM_NO)
GROUP BY CHECK_NO
```

returns:

```
|CHECK_NO|CHECK_TOTAL
=====================
|       1|        9.00
|       2|       19.80
|       3|       18.55
|       4|       18.88
|       5|       30.38
5 rows affected
```

Example 10: list items priced 1.95 in group 3 or priced 1.20 in group 5

```
SELECT *
FROM
        MENU_ITEM
WHERE
(PRICE,MENU_ITEM_GROUP_NO) IN ( VALUES (1.95,3), (1.20,5) )
```

returns:

```
|MENU_ITEM_NO|MENU_ITEM_NAME                          |MENU_ITEM_GROUP_NO|PRICE
================================================================================
|         200|Boiled Rice                             |                 3|   1.95
|         304|Lemonade                                |                 5|   1.20
|         305|Cola                                    |                 5|   1.20
3 rows affected
```

Example 11: list menu items that have never been ordered (using `Except`)

```
SELECT *
FROM MENU_ITEM
  NATURAL JOIN
(
(SELECT * FROM MENU_ITEM)
EXCEPT CORRESPONDING
(SELECT * FROM CHECK_DETAIL)
) AS not_ordered
```

returns:

MENU_ITEM_NO	MENU_ITEM_NAME	MENU_ITEM_GROUP_NO	PRICE
1	Soup of the Day	1	2.45
101	Spaghetti Bolognese	2	1.95
102	Paella	2	6.95
103	Borsch	2	4.95
105	Kedgeree	2	4.99
109	Chicken Tikka Masala	2	5.95
305	Cola	5	1.20

7 rows affected

14. SQL Grammar

The SQL grammar currently accepted by ThinkSQL is listed here. Commands that are not part of the standard, e.g. user maintenance, are not listed here but are documented in the appropriate sections of this guide.

The following are terminals:

```
      <connection name>
      <connection user name>
      <character set specification>
      <schema name>
      <authorization identifier>
      <domain name>
      <data type>
      <digit> ::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Literal

```
…
<hexit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f

<binary string literal> ::=
      X <quote> [ { <hexit> <hexit> }… ] <quote>
      [ { <separator> <quote> [ { <hexit> <hexit> }… ] <quote> }… ]

<datetime literal> ::=
      <date literal>
      | <time literal>
      | <timestamp literal>

<date literal> ::=
      DATE <date string>

<time literal> ::=
      TIME <time string>

<timestamp literal> ::=
      TIMESTAMP <timestamp string>

<date string> ::=
      <quote> <unquoted date string> <quote>

<time string> ::=
      <quote> <unquoted time string> <quote>

<timestamp string> ::=
```

```
      <quote> <unquoted timestamp string> <quote>

<unquoted date string> ::=
      <date value>

<unquoted time string> ::=
      <time value> [ <time zone interval> ]

<unquoted timestamp string> ::=
      <unquoted date string> <space> <unquoted time string>

<date value> ::=
      <years value> <minus sign>
      <months value> <minus sign>
      <days value>

<time value> ::=
      <hours value> <colon>
      <minutes value> <colon>
      <seconds value>

<years value> ::= <datetime value>
<months value> ::= <datetime value>
<days value> ::= <datetime value>
<hours value> ::= <datetime value>
<minutes value> ::= <datetime value>
<seconds value> ::=
      <seconds integer value>
      [ <period> [ <seconds fraction> ] ]
<seconds integer value> ::= <unsigned integer>
<seconds fraction> ::= <unsigned integer>
<datetime value> ::= <unsigned integer>


<concatenation operator> ::= ||
<multiplier> ::= K | M | G
```

Data Definition

```
<schema definition> ::=
      CREATE SCHEMA <schema name clause>
      [ <schema character set or path> ]
      [ <schema element>… ]

<schema character set or path> ::=
      <schema character set specification>

<schema name clause> ::=
      <schema name>
      | AUTHORIZATION <schema authorization identifier>
      | <schema name> AUTHORIZATION <schema authorization identifier>

<schema authorization identifier> ::=
      <authorization identifier>

<schema character set specification> ::=
      DEFAULT CHARACTER SET <character set specification>

<schema element> ::=
      <table definition>
      | <view definition>
```

```
      | <domain definition>
      | <schema routine>
      | <grant statement>
      | <sequence definition>

<drop schema statement> ::=
      DROP SCHEMA <schema name> <drop behaviour>

<table definition> ::=
      CREATE [ <table scope> ] TABLE <table name>
      <table element list>
      [ ON COMMIT <table commit action> ROWS ]

<table scope> ::=
      <global or local> TEMPORARY

<global or local> ::=
      GLOBAL
      | LOCAL

<table commit action> ::=
      PRESERVE
      | DELETE

<table element list> :=
      <left paren> <table element> [ { <comma> <table element> }… ]
      <right parent>

<table element> ::=
      <column definition>
      | <table constraint definition>

<column definition> ::=
      <column name>
      { <data type> | <domain name> }
      [ <default clause> ]
      [ <column constraint definition>… ]

<column constraint definition> ::=
      [ <constraint name definition> ]
      <column constraint> [ <constraint characteristics> ]

<constraint name definition> ::=
      CONSTRAINT <constraint name>

<constraint characteristics> ::=
      <constraint check time> [ [ NOT ] DEFERRABLE ]
      | [ NOT ] DEFERRABLE [ <constraint check time> ]

<constraint check time> ::=
      INITIALLY DEFERRED | INITIALLY IMMEDIATE

<column constraint> :=
      NOT NULL
      | <unique specification>
      | <references specification>
      | <check constraint definition>

<default clause> ::=
      DEFAULT <default option>
```

```
<default option> ::=
      <literal>
      | <datetime value function>
      | USER
      | CURRENT_USER
      | SESSION_USER
      | SYSTEM_USER

<table constraint definition> ::=
      [ <constraint name definition> ]
      <table constraint> [ <constraint characteristics> ]

<table constraint> ::=
      <unique constraint definition>
      | <referential constraint definition>
      | <check constraint definition>

<unique constraint definition> ::=
      <unique specification> <left paren> <unique column list>
      <right paren>

<unique specification> ::=
      UNIQUE
      | PRIMARY KEY

<unique column list> ::=
      <column name list>

<referential constraint definition> ::=
      FOREIGN KEY <left paren> <referencing columns> <right paren>
      <referencing specification>

<references specification> ::=
      REFERENCES <referenced table and columns>
      [ MATCH <match type>]
      [ <referential triggered action> ]

<match type> ::=
      FULL
      | PARTIAL
      | SIMPLE

<referencing columns> ::=
      <reference column list>

<referenced table and columns> ::=
      <table name> [ <left paren> <reference column list>
      <right paren> ]

<reference column list> ::=
      <column name list>

<referential triggered action> ::=
      <update rule> [ <delete rule> ]
      | <delete rule> [ <update rule> ]

<update rule> ::=
      ON UPDATE <referential action>
```

```
<delete rule> ::=
      ON DELETE <referential action>

<referential action> ::=
      CASCADE
      | SET NULL
      | SET DEFAULT
      | NO ACTION

<check constraint definition> ::=
      CHECK <left paren> <search condition> <right paren>

<alter table statement> ::=
      ALTER TABLE <table name> <alter table action>

<alter table action> ::=
      <add column definition>
      | <alter column definition>
      | <drop column definition>
      | <add table constraint definition>
      | <drop table constraint definition>

<add table constraint definition> ::=
      ADD <table constraint definition>

<drop table constraint definition> ::=
      DROP CONSTRAINT <constraint name> <drop behavior>

<drop table statement> ::=
      DROP TABLE <table name> <drop behaviour>

<drop behaviour> ::=
      CASCADE
      | RESTRICT

<view definition> ::=
      CREATE VIEW <table name> <view specification> AS
      <query expression> [ WITH [ <levels clause> ] CHECK OPTION ]

<view specification> ::=
      [ <left paren> <view column list> <right paren> ]

<levels clause> ::=
      CASCADED
      | LOCAL

<view column list> ::=
      <column name list>

<drop view statement> ::=
      DROP VIEW <table name> <drop behaviour>

<domain definition> ::=
      CREATE DOMAIN <domain name> [ AS ] <data type>
      [ <default clause> ]
      [ <domain constraint>… ]

<domain constraint> ::=
      [ <constraint name definition> ]
      <check constraint definition> [ <constraint characteristics> ]
```

```
<schema routine> ::=
      <schema procedure>
      | <schema function>

<schema procedure> ::=
      CREATE <SQL-invoked procedure>

<schema function> ::=
      CREATE <SQL-invoked function>

<SQL-invoked procedure> ::=
      PROCEDURE <schema qualified routine name>
      <SQL parameter declaration list>
      <routine body>

<SQL-invoked function> ::=
      <function specification>
      <routine body>

<SQL parameter declaration list> ::=
      <left paren>
      [ <SQL parameter declaration>
            [ { <comma> <SQL parameter declaration> }… ] ]
      <right paren>

<SQL parameter declaration> ::=
      [ <parameter mode> ] [ <SQL parameter name> ]
      <paremeter type>

<parameter mode> ::=
      IN | OUT | INOUT

<parameter type> ::=
      <data type>

<function specification> ::=
      FUNCTION <schema qualified routine name>
      <SQL parameter declaration list>
      <returns clause>

<returns clause> ::=
      RETURNS <returns data type>

<returns data type> ::=
      <data type>

<routine body> ::=
      <SQL routine body>

<SQL routine body> ::=
      <SQL procedure statement>

<drop routine statement> ::=
      DROP <specific routine designator> <drop behaviour>

<specific routine designator> ::=
      <routine type> <member name>

<routine type>
      ROUTINE | FUNCTION | PROCEDURE
```

```
<member name> ::=
      <schema qualified routine name>

<SQL procedure statement> ::=
      <data definition statement>
      | <data manipulation statement>
      | <SQL control statement>

<SQL control statement> ::=
      <assignment statement>
      | <compound statement>
      | <case statement>
      | <if statement>
      | <iterate statement>
      | <leave statement>
      | <loop statement>
      | <while statement>
      | <repeat statement>
      | <for statement>

<assignment statement> ::=
      SET <assignment target> <equals operator> <assignment source>

<assignment target> ::=
      <target specification>

<assignment source> ::=
      <value expression>

<compound statement> ::=
      [ <beginning label> <colon> ]
      BEGIN [ [ NOT ] ATOMIC ]
      [ <local declaration list> ]
      [ <local cursor declaration list> ]
      [ <local handler declaration list> ]
      [ <SQL statement list> ]
      END [ <ending label> ]

<beginning label> ::= <statement label>

<ending label> ::= <statement label>

<statement label> ::= <identifier>

<local declaration list> ::= <terminated local declaration>…

<terminated local declaration> ::= <local declaration> <semicolon>

<local declaration> ::=
      <SQL variable declaration>
      | <condition declaration>

<cursor declaration> ::=
      DECLARE <cursor name> [ <cursor sensitivity> ]
      [ SCROLL ] CURSOR
      [ WITH HOLD ]
      [ WITH RETURN ]
      FOR <cursor specification>
```

```
<cursor sensitivity> ::= SENSITIVE | INSENSITIVE | ASENSITIVE

<cursor specification> ::=
      <query expression> [ <order by clause> ]
      [ FOR { READ ONLY | UPDATE [ OF <column name list> ] } ]

<open statement> ::=
      OPEN <cursor name>

<fetch statement> ::=
      FETCH [ [ <fetch orientation> ] FROM ]
      <cursor name> INTO <fetch target list>

<fetch orientation> ::=
      NEXT | PRIOR | FIRST | LAST
      | { ABSOLUTE | RELATIVE } <simple value specification>

<fetch target list> ::=
      <target specification> [ { <comma> <target specification> }… ]

<close statement> ::=
      CLOSE <cursor name>

<SQL statement list> ::= <terminated SQL statement>…

<terminated SQL statement> ::= <SQL procedure statement> <semicolon>

<SQL variable declaration> ::=
      DECLARE <SQL variable name list>
      <data type> [ <default clause> ]

<SQL variable name list> ::=
      <SQL variable name> [ { <comma> <SQL variable name> }… ]

<case statement> ::=
      <simple case statement>
      | <searched case statement>

<searched case statement> ::=
      CASE
            <searched case statement when clause>…
            [ <case statement else clause> ]
      END CASE

<searched case statement when clause> ::=
      WHEN <search condition> THEN <SQL statement list>

<case statement else clause> ::=
      ELSE <SQL statement list>

<if statement> ::=
      IF <search condition>
            <if statement then clause>
            [ <if statement elseif clause>… ]
            [ <if statement else clause> ]
      END IF

<if statement then clause> ::=
      THEN <SQL statement list>
```

```
<if statement elseif clause> ::=
      ELSEIF <search condition> THEN <SQL statement list>

<if statement else clause> ::=
      ELSE <SQL statement list>

<iterate statement> ::=
      ITERATE <statement label>

<leave statement> ::=
      LEAVE <statement label>

<loop statement> ::=
      [ <beginning label> <colon> ]
      LOOP
            <SQL statement list>
      END LOOP [ <ending label> ]

<while statement> ::=
      [ <beginning label> <colon> ]
      WHILE <search condition> DO
            <SQL statement list>
      END WHILE [ <ending label> ]

<repeat statement> ::=
      [ <beginning label> <colon> ]
      REPEAT
            <SQL statement list>
            UNTIL <search condition>
      END REPEAT [ <ending label> ]

<grant privilege statement> ::=
      GRANT <privileges>
      TO <grantee> [ { <comma> <grantee> }… ]
      [ WITH GRANT OPTION ]

<privileges> ::=
      <object privilege> ON <object name>

<object name> ::=
      [ TABLE ] <table name>
      | DOMAIN <domain name>

<object privileges> ::=
      ALL PRIVILEGES
      | <action> [ { <comma> <action> }… ]

<action> ::=
      SELECT [ <left paren> <privilege column list> <right paren> ]
      | DELETE
      | INSERT [ <left paren> <privilege column list> <right paren> ]
      | UPDATE [ <left paren> <privilege column list> <right paren> ]
      | REFERENCES [ <left paren>
            <privilege column list> <right paren> ]
      | USAGE
      | EXECUTE

<grantee> ::=
      PUBLIC
      | <authorization identifier>
```

```
<revoke privilege statement> ::=
      REVOKE [ GRANT OPTION FOR ] <privileges>
      FROM <grantee> [ { <comma> <grantee> }… ]
      <drop behaviour>

<sequence definition> ::=
      CREATE SEQUENCE <sequence name>
      [ STARTING AT <simple value> ]

<drop sequence statement> ::=
      DROP SEQUENCE < sequence name> <drop behaviour>
```

Data Manipulation

```
<select statement: single row> ::=
      SELECT [ <set quantifier> ] <select list>
      INTO <select target list>
      <table expression>

<select target list> ::=
      <target specification> [ { <comma> <target specification> }… ]

<target specification> ::=
      <column reference>
      | <SQL variable reference>


<delete statement: searched> ::=
      DELETE FROM <target table>
      [ WHERE <search condition> ]

<target table> ::=
      <table name>

<insert statement> ::=
      INSERT INTO <insertion target>
      <insert columns and source>

<insertion target> ::=
      <table name>

<insertion columns and source> ::=
      <from subquery>
      | <from default>

<from subquery> ::=
      [ <left paren> <insert column list> <right paren> ]
      <query expression>

<from default> ::=
      DEFAULT VALUES

<insert column list> ::=
      <column name list>

<update statement: searched> ::=
      UPDATE <target table>
      SET <set clause list>
      [ WHERE <search condition> ]
```

```
<set clause list> ::=
      <set clause> [ { <comma> <set clause> }… ]

<set clause> ::=
      <column name> <equals operator> <update source>

<update source> ::=
      <value expression>
```

Transaction Management

```
<set transaction statement> ::=
      SET TRANSACTION <transaction mode>
      [ { <comma> <transaction mode> }… ]

<transaction mode> ::=
      <isolation level>
      | <transaction access mode>
      | <diagnostics size>

<transaction access mode> ::=
      READ ONLY
      | READ WRITE

<isolation level> ::=
      ISOLATION LEVEL <level of isolation>

<level of isolation> ::=
      READ UNCOMMITTED
      | READ COMMITTED
      | REPEATABLE READ
      | SERIALIZABLE

<diagnostics size> ::=
      DIAGNOSTICS SIZE <number of conditions>

<number of conditions> ::= <simple value specification>

<commit statement> ::=
      COMMIT [ WORK ]

<rollback statement> ::=
      ROLLBACK [ WORK ]

<set constraints mode statement> ::=
      SET CONSTRAINTS <constraint name list> { DEFERRED | IMMEDIATE }

<constraint name list> ::=
      ALL
      | <constraint name> [ { <comma> <constraint name> }… ]
```

Connection Management

```
<connect statement> ::=
      CONNECT TO <connection target>

<connection target> ::=
      <SQL-server name>
      [ AS <connection name> ]
      [ USER <connection user name> [ PASSWORD <password> ] ]
      | DEFAULT
```

```
<connection object>
      DEFAULT
      | <connection name>


<disconnect statement> ::=
      DISCONNECT <disconnect object>

<disconnect object> ::=
      <connection object>
      | ALL
      | CURRENT
```

Scalar Expressions

```
<datatype> ::=
      <character string type>
      | <binary large object string type>
      | <bit string type>
      | <numeric type>
      | <datetime type>

<character string type> ::=
      CHARACTER [ <left paren> <length> <right paren> ]
      | CHAR [ <left paren> <length> <right paren> ]
      | CHARACTER VARYING <left paren> <length> <right paren>
      | CHAR VARYING <left paren> <length> <right paren>
      | VARCHAR <left paren> <length> <right paren>
      | CHARACTER LARGE OBJECT
      [ <left paren> <large object length> <right paren> ]
      | CHAR LARGE OBJECT
      [ <left paren> <large object length> <right paren> ]
      | CLOB [ <left paren> <large object length> <right paren> ]

<binary large object string type> ::=
      BINARY LARGE OBJECT [ <left paren>
            <large object length> <right paren> ]
      | BLOB [ <left paren> <large object length> <right paren> ]

<bit string type> ::=
      BIT [ <left paren> <length> <right paren> ]
      | BIT VARYING <left paren> <length> <right paren>

<numeric type> ::=
      <exact numeric type>
      | <approximate numeric type>

<exact numeric type> ::=
      NUMERIC [ <left paren>
            <precision> [ <comma> <scale> ] <right paren> ]
      | DECIMAL [ <left paren>
            <precision> [ <comma> <scale> ] <right paren> ]
      | DEC [ <left paren>
            <precision> [ <comma> <scale> ] <right paren> ]
      | INTEGER
      | INT
      | SMALLINT

<approximate numeric type> ::=
      FLOAT [ <left paren> <precision> <right paren> ]
```

```
      | REAL
      | DOUBLE PRECISION

<length> ::= <unsigned integer>

<large object length> ::= <unsigned integer> [ <multiplier> ]

<precision> ::= <unsigned integer>

<scale> ::= <unsigned integer>

<datetime type> ::=
      DATE
      | TIME [ <left paren>
            <time precision> <right paren> ]
            [ <with or without time zone> ]
      | TIMESTAMP [ <left paren>
            <timestamp precision> <right paren> ]
            [ <with or without time zone> ]

<with or without time zone> ::=
      WITH TIME ZONE
      | WITHOUT TIME ZONE

<time precision> ::= <time fractional seconds precision>

<timestamp precision> ::= <time fractional seconds precision>

<time fractional seconds precision> ::= <unsigned integer>


<set function specification> ::=
      COUNT <left paren> <asterisk> <right paren>
      | <general set function>

<general set function> ::=
      <set function type> <left paren> [ <set quantifier> ]
      <value expression> <right paren>

<set function type> ::=
      AVG | MAX | MIN | SUM | COUNT

<set quantifier> ::=
      DISTINCT
      | ALL

<numeric value function> ::=
      <position expression>
      | <length expression>
      | <sequence expression>

<position expression> ::=
      <string position expression>
      | <blob position expression>

<string position expression> ::=
      POSITION <left paren> <string value expression>
      IN <string value expression> <right paren>

<blob position expression> ::=
```

```
      POSITION <left paren> <blob value expression>
      IN <blob value expression> <right paren>

<length expression> ::=
      <char length expression>
      | <octet length expression>

<char length expression> ::=
      { CHAR_LENGTH | CHARACTER_LENGTH }
      <left paren> <string value expression> <right paren>

<octet length expression> ::=
      OCTET_LENGTH <left paren>
      <string value expression> <right paren>

<sequence expression> ::=
      { NEXT_SEQUENCE | LATEST_SEQUENCE }
      <left paren> <sequence name> <right paren>

<string value function> ::=
      <character value function>
      | <blob value function>

<character value function> ::=
      <character substring function>
      | <fold>
      | <trim function>

<character substring function> ::=
      SUBSTRING <left paren> <character value expression> FROM
      <start position> [ FOR <string length> ] <right paren>

<fold> ::=
      { UPPER | LOWER } <left paren> <character value expression>
      <right paren>

<trim function> ::=
      TRIM <left paren> <trim operands> <right paren>

<trim operands> ::=
      [ [ <trim specification> ] [ <trim character> ] FROM ]
      <trim source>

<trim source> ::=
      <character value expression>

<trim specification> ::=
      LEADING
      | TRAILING
      | BOTH

<trim character> ::=
      <character value expression>

<blob value function> ::=
      <blob substring function>
      | <blob trim function>

<blob substring function> ::=
      SUBSTRING <left paren> <blob value expression> FROM
```

```
      <start position> [ FOR <string length> ] <right paren>

<blob trim function> ::=
      TRIM <left paren> <blob trim operands> <right paren>

<blob trim operands> ::=
      [ [ <trim specification> ] [ <trim octet> ] FROM ]
      <blob trim source>

<blob trim source> ::= <blob value expression>

<trim octet> ::= <blob value expression>

<start position> ::= <numeric value expression>

<string length> ::= <numeric value expression>

<datetime value function> ::=
      <current date value function>
      | <current time value function>
      | <current timestamp value function>

<current date value function> ::=
      CURRENT_DATE

<current time value function> ::=
      CURRENT_TIME
      [ <left paren> <time precision> <right paren> ]

<current timestamp value function> ::=
      CURRENT_TIMESTAMP
      [ <left paren> <time precision> <right paren> ]

<cast specification> ::=
      CAST <left paren> <cast operand> AS <cast type> <right paren>

<cast operand> ::=
      <value expression> | <implicitly typed value expression>

<cast target> ::=
      <domain name>
      | <data type>

<value expression> :=
      <numeric value expression>
      | <string value expression>
      | <datetime value expression>
      | <collection value expression>

<collection value expression> ::=
      <value expression primary>

<value expression primary> ::=
      <parenthesized value expression>
      | <nonparenthesized value expression primary>

<parenthesized value expression> ::=
      <left paren> <value expression> <right paren>

<nonparenthesized value expression primary> ::=
```

```
      <unsigned value specification>
      | <column reference>
      | <set function specification>
      | <scalar subquery>
      | <case expression>
      | <cast expression>
      | <routine invocation>

<numeric value expression> ::=
      <term>
      | <numeric value expression> <plus sign> <term>
      | <numeric value expression> <minus sign> <term>

<term> ::=
      <factor>
      | <term> <asterisk> <factor>
      | <term> <solidus> <factor>

<factor> ::=
      [ <sign> ] <numeric primary>

<numeric primary> ::=
      <value expression primary>
      | <numeric value function>

<string value expression> ::=
      <character value expression>
      | <blob value expression>

<character value expression> ::=
      <concatenation>
      | <character factor>

<concatenation> ::=
      <character value expression> <concatenation operator>
      <character factor>

<character factor> ::=
      <character primary>

<character primary> ::=
      <value expression primary>
      | <string value function>

<blob value expression> ::=
      <blob concatenation>
      | <blob factor>

<blob factor> ::= <blob primary>

<blob primary> ::=
      <value expression primary>
      | <string value function>

<blob concatenation> ::=
      <blob value expression> <concatenation operator> <blob factor>

<datetime value expression> ::=
      <datetime term>
```

```
<datetime term> ::=
      <datetime factor>

<datetime factor> ::=
      <datetime primary>

<datetime primary> ::=
      <value expression primary>
      | <datetime value function>

<boolean value expression> ::=
      <boolean term>
      | <boolean value expression> OR <boolean term>

<boolean term> ::=
      <boolean factor>
      | <boolean term> AND <boolean factor>

<boolean factor> ::=
      [ NOT ] <boolean test>

<boolean test> ::=
      <boolean primary> [ IS [ NOT ] <truth value> ]

<truth value> ::=
      TRUE
      | FALSE
      | UNKNOWN

<boolean primary> ::=
      <predicate>
      | <parenthesized boolean value expression>
      | <nonparenthesized value expression primary>

<parenthesized boolean value expression> ::=
      <left paren> <boolean value expression> <right paren>

<case expression> ::=
      <case abbreviation>
      | <case specification>

<case abbreviation> ::=
      NULLIF <left paren> <value expression>
      <comma> <value expression> <right paren>
      | COALESCE <left paren> <value expression>
      { <comma> <value expression> }… <right paren>

<case specification> ::=
      <simple case>
      | <searched case>

<simple case> ::=
      CASE <case operand>
            <simple when clause>…
            [ <else clause> ]
      END

<searched case> ::=
      CASE
            <searched when clause>…
```

```
            [ <else clause> ]
      END

<simple when clause> ::=
      WHEN <when operand> THEN <result>

<searched when clause> ::=
      WHEN <search condition> THEN <result>

<else clause> ::= ELSE <result>

<case operand> ::= <value expression>

<when operand> ::= <value expression>

<result> ::=
      <result expression>
      | NULL

<result expression> ::= <value expression>
```

Query Expressions

```
<row value constructor> ::=
      <row value constructor element>
      | <left paren> <row value constructor element list>
        <right paren>
      | <row subquery>

<row value constructor element list> ::=
      <row value constructor element>
      [ { <comma> <row value constructor element> }… ]

<row value constructor element> ::=
      <value expression>

<table value constructor> ::=
      VALUES <row value expression list>

<row value expression list> ::=
      <row value expression> [ { <comma> <row value expression> }… ]

<table expression> ::=
      <from clause>
      [ <where clause> ]
      [ <group by clause> ]
      [ <having clause> ]

<from clause> ::=
      FROM <table reference list>

<table reference list> ::=
      <table reference> [ { <comma> <table reference> }… ]

<table reference> ::=
      <table primary>
      | <joined table>

<table primary> ::=
      <table or query name> [ [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ] ]
```

```
      | <derived table> [ [ AS ] <correlation name>
      [ <left paren> <derived column list> <right paren> ] ]
      | <left paren> <joined table> <right paren>

<derived table> ::=
      <table subquery>

<table or query name> ::=
      <table name>
      | <query name>

<derived column list> ::=
      <column name list>

<column name list> ::=
      <column name> [ { <comma> <column name> }… ]

<joined table> ::=
      <cross join>
      | <qualified join>
      | <natural join>

<cross join> ::=
      <table reference> CROSS JOIN <table primary>

<qualified join> ::=
      <table reference> [ <join type> ] JOIN <table reference>
      <join specification>

<natural join> ::=
      <table reference> NATURAL [ <join type> ] JOIN <table primary>

<join specification> ::=
      <join condition>
      | <named columns join>

<join condition> ::=
      ON <search condition>

<named columns join> ::=
      USING <left paren> <join column list> <right paren>

<join type> ::=
      INNER
      | <outer join type> [ OUTER ]

<outer join type> ::=
      LEFT
      | RIGHT
      | FULL

<join column list> ::= <column name list>

<where clause> ::=
      WHERE <search condition>

<group by clause> ::=
      GROUP BY <grouping specification>

<grouping specification> ::=
```

```
      <grouping column reference>
      | <concatenated grouping>

<grouping set list> ::=
      <grouping set> [ { <comma> <grouping set> }… ]

<concatenated grouping> ::=
      <grouping set> <comma> <grouping set list>

<grouping set> ::=
      <ordinary grouping set>

<ordinary grouping set> ::=
      <grouping column reference>

<grouping column reference> ::=
      <column reference>

<having clause> ::=
      HAVING <search condition>

<query specification> ::=
      SELECT [ <set quantifier> ] <select list>
      <table expression>

<select list> ::=
      <asterisk>
      | <select sublist> [ { <comma> <select sublist> }… ]

<select sublist> ::=
      <derived column>
      | <qualified asterisk>

<qualified asterisk> ::=
      <asterisked identifier chain> <period> <asterisk>
      | <all fields reference>

<asterisked identifier chain> ::=
      <asterisked identifier>
      [ { <period> <asterisked identifier> }… ]

<asterisked identifier> ::= <identifier>

<derived column> ::=
      <value expression> [ <as clause> ]

<as clause> ::= [ AS ] <column name>

<all fields reference> ::=
      <value expression primary> <period> <asterisk>

<query expression> ::=
      <query expression body>

<query expression body> ::=
      <non-join query expression>
      | <joined table>

<non-join query expression> ::=
      <non-join query term>
```

```
      | <query expression body> UNION [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>
      | <query expression body> EXCEPT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query term>

<query term> ::=
      <non-join query term>
      | <joined table>

<non-join query term> ::=
      <non-join query primary>
      | <query term> INTERSECT [ ALL | DISTINCT ]
      [ <corresponding spec> ] <query primary>

<query primary> ::=
      <non-join query primary>
      | <joined table>

<non-join query primary> ::=
      <simple table>
      | <left paren> <non-join query expression> <right paren>

<simple table> ::=
      <query specification>
      | <table value constructor>
      | <explicit table>

<explicit table> ::=
      TABLE <table name>

<corresponding spec> ::=
      CORRESPONDING
      [ BY <left paren> <corresponding column list> <right paren> ]

<corresponding column list> ::= <column name list>

<scalar subquery> ::=
      <subquery>

<row subquery> ::=
      <subquery>

<table subquery> ::=
      <subquery>

<subquery> ::=
      <left paren> <query expression> <right paren>
```

Predicates

```
<predicate> ::=
      <comparison predicate>
      | <between predicate>
      | <in predicate>
      | <like predicate>
      | <null predicate>
      | <quantified comparison predicate>
      | <exists predicate>
      | <unique predicate>
      | <match predicate>
```

```
<comparison predicate> ::=
      <row value expression> <comp op> <row value expression>

<comp op> ::=
      <equals operator>
      | <not equals operator>
      | <less than operator>
      | <greater than operator>
      | <less than or equals operator>
      | <greater than or equals operator>

<between predicate> ::=
      <row value expression> [ NOT ] BETWEEN
      <row value expression> AND <row value expression>

<in predicate> ::=
      <row value expression> [ NOT ] IN
      <in predicate value>

<in predicate value> ::=
      <table subquery>
      | <left paren> <in value list> <right paren>

<in value list> ::=
      <row value expression> { <comma> <row value expression> }…

<like predicate> ::=
      <character like predicate>
      | <octet like predicate>

<character like predicate> ::=
      <character match value> [ NOT ] LIKE
      <character pattern> [ ESCAPE <escape character> ]

<character match value> ::=
      <character value expression>

<character pattern> ::=
      <character value expression>

<escape character> ::=
      <character value expression>

<octet like predicate> ::=
      <octet match value> [ NOT ] LIKE
      <octet pattern> [ ESCAPE <escape octet> ]

<octet match value> ::= <blob value expression>

<octet pattern> ::= <blob value expression>

<escape octet> ::= <blob value expression>

<null predicate> ::=
      <row value expression> IS [ NOT ] NULL

<quantified comparison predicate> ::=
      <row value expression> <comp op> <quantifier>
      <table subquery>
```

```
<quantifier> ::= <all> | <some>

<all> ::= ALL

<some> ::= SOME | ANY

<exists predicate> ::=
      EXISTS <table subquery>

<unique predicate> ::=
      UNIQUE <table subquery>
Note: unique is for restricted/internal use

<match predicate> ::=
      <row value expression>
      MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ]
      <table subquery>

<search condition> ::=
      <boolean value expression>
```

Appendix A

```
CREATE SCHEMA RESTAURANT AUTHORIZATION RESTAURANT
       CREATE TABLE MENU_ITEM_GROUP (
              MENU_ITEM_GROUP_NO INTEGER NOT NULL,
              MENU_ITEM_GROUP_NAME VARCHAR(20),
              PRIMARY KEY (MENU_ITEM_GROUP_NO) )

       CREATE TABLE MENU_ITEM (
              MENU_ITEM_NO INTEGER NOT NULL,
              MENU_ITEM_NAME VARCHAR(40),
              MENU_ITEM_GROUP_NO INTEGER,
              PRICE DECIMAL(6,2),
              PRIMARY KEY (MENU_ITEM_NO),
              FOREIGN KEY (MENU_ITEM_GROUP_NO)
                     REFERENCES MENU_GROUP (MENU_ITEM_GROUP_NO) )

       CREATE TABLE "SERVER" (
              SERVER_NO INTEGER NOT NULL,
              SERVER_NAME VARCHAR(20),
              PRIMARY KEY (SERVER_NO) )

       CREATE TABLE CHECK_HEADER (
              CHECK_NO INTEGER NOT NULL,
              SERVER_NO INTEGER REFERENCES "SERVER",
              START_TIME TIMESTAMP(0),
              PRIMARY KEY (CHECK_NO) )

       CREATE TABLE CHECK_DETAIL (
              CHECK_NO INTEGER NOT NULL REFERENCES CHECK_HEADER,
              LINE_NO INTEGER NOT NULL,
              MENU_ITEM_NO INTEGER REFERENCES MENU_ITEM,
              QTY DECIMAL(5),
              PRIMARY KEY (CHECK_NO, LINE_NO) )

       CREATE VIEW FULL_CHECK_DETAIL AS
              SELECT *
              FROM RESTAURANT."SERVER"
              NATURAL JOIN RESTAURANT.CHECK_HEADER
              NATURAL JOIN RESTAURANT.MENU_ITEM_GROUP
              NATURAL JOIN RESTAURANT.MENU_ITEM
              NATURAL JOIN RESTAURANT.CHECK_DETAIL

       GRANT SELECT ON MENU_ITEM_GROUP TO PUBLIC
       GRANT SELECT ON MENU_ITEM TO PUBLIC
       GRANT SELECT ON "SERVER" TO PUBLIC
       GRANT SELECT ON CHECK_HEADER TO PUBLIC
       GRANT SELECT ON CHECK_DETAIL TO PUBLIC
       GRANT SELECT ON FULL_CHECK_DETAIL TO PUBLIC
;
```

In the sample database, these tables have the following rows in them:

MENU_ITEM_GROUP

MENU_ITEM_GROUP_NO	MENU_ITEM_GROUP_NAME

1	Starter
2	Main Course
3	Side-dish
4	Dessert
5	Soft drink
6	Beer
7	Wine

MENU_ITEM

MENU_ITEM_NO	MENU_ITEM_NAME	MENU_ITEM_GROUP_NO	PRICE
1	Soup of the Day	1	2.45
2	Samosas	1	1.95
3	Prawn Cocktail	1	2.95
100	Lasagne	2	4.95
101	Spaghetti Bolognese	2	1.95
102	Paella	2	6.95
103	Borsch	2	4.95
104	Irish stew	2	3.95
105	Kedgeree	2	4.99
106	Boeuf Bourgignone	2	7.95
107	Roast Beef and Yorkshire Pudding	2	6.45
108	Chicken Madras	2	5.45
109	Chicken Tikka Masala	2	5.95
200	Boiled Rice	3	1.95
201	Mashed Potato	3	1.45
300	Red Wine	7	2.1
301	White Wine	7	2.1
302	Guinness	6	2.99
303	Lager	6	1.85
304	Lemonade	5	1.2
305	Cola	5	1.2
401	Ice Cream	4	2.99

SERVER

SERVER_NO	SERVER_NAME
1	John Smith
2	Mary Jones

CHECK_HEADER

CHECK_NO	SERVER_NO	START_TIME
1	1	2001-01-12 11:55:12
2	1	2001-01-12 12:07:34
3	1	2001-01-12 13:24:39
4	2	2001-01-12 13:45:17
5	1	2001-01-12 13:46:09

CHECK_DETAIL

CHECK_NO	LINE_NO	MENU_ITEM_NO	QTY
1	1	3	1
1	2	104	1
1	3	300	1
2	1	2	1

2	2	108	2
2	3	200	2
2	4	303	1
2	5	304	1
3	1	100	1
3	2	106	1
3	3	201	1
3	4	301	2
4	1	107	2
4	2	302	1
4	3	401	1
5	1	106	1
5	2	107	1
5	3	201	1
5	4	200	1
5	5	300	2
5	6	304	2
5	7	401	2