

iWay

iWay SQL Reference
Version 5 Release 2.0

EDA, FIDEL, FOCUS, FOCUS Fusion, Information Builders, the Information Builders logo, TableTalk, and Web390 are registered trademarks and SiteAnalyzer, SmartMart, WebFOCUS, and WorldMART are trademarks of Information Builders, Inc.

Acrobat and Adobe are registered trademarks of Adobe Systems Incorporated.

Allaire and JRun are trademarks of Allaire Corporation.

UniVerse is a registered trademark of Ardent Software, Inc.

AvantGo is a trademark of AvantGo, Inc.

WebLogic is a registered trademark of BEA Systems, Inc.

SUPRA and TOTAL are registered trademarks of Cincom Systems, Inc.

Alpha, DEC, DECnet, and NonStop are registered trademarks and Tru64, OpenVMS, and VMS are trademarks of Compaq Computer Corporation.

CA-ACF2, CA-Datcom, CA-IDMS, CA-Top Secret, and Ingres are registered trademarks of Computer Associates International, Inc.

MODEL 204 and M204 are registered trademarks of Computer Corporation of America.

Paradox is a registered trademark of Corel Corporation.

StorHouse is a registered trademark of FileTek, Inc.

HP MPE/iX is a registered trademark of Hewlett Packard Corporation.

Informix is a registered trademark of Informix Software, Inc.

Intel is a registered trademark of Intel Corporation.

ACF/VTAM, AIX, AS/400, CICS, DB2, DRDA, Distributed Relational Database Architecture, IBM, MQSeries, MVS, OS/2, OS/390, OS/400, RACF, RS/6000, S/390, VM/ESA, and VTAM are registered trademarks and DB2/2, Hiperspace, IMS, MVS/ESA, QMF, SQL/DS, VM/XA and WebSphere are trademarks of International Business Machines Corporation.

INTERSOLVE and Q+E are registered trademarks of INTERSOLVE.

Orbit is a registered trademark of Iona Technologies Inc.

Approach and DataLens are registered trademarks of Lotus Development Corporation.

ObjectView is a trademark of Matesys Corporation.

ActiveX, FrontPage, Microsoft, MS-DOS, PowerPoint, Visual Basic, Visual C++, Visual FoxPro, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

Teradata is a registered trademark of NCR International, Inc.

Netscape, Netscape FastTrack Server, and Netscape Navigator are registered trademarks of Netscape Communications Corporation.

NetWare and Novell are registered trademarks of Novell, Inc.

Oracle is a registered trademark and Rdb is a trademark of Oracle Corporation.

Palm is a trademark and Palm OS is a registered trademark of Palm Inc.

INFOAccess is a trademark of Pioneer Systems, Inc.

Progress is a registered trademark of Progress Software Corporation.

Red Brick Warehouse is a trademark of Red Brick Systems.

SAP and SAP R/3 are registered trademarks and SAP Business Information Warehouse and SAP BW are trademarks of SAP AG.

Silverstream is a trademark of Silverstream Software.

CONNECT:Direct is a trademark of Sterling Commerce.

Java and all Java-based marks, NetDynamics, Solaris, SunOS, and iPlanet are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

PowerBuilder and Sybase are registered trademarks and SQL Server is a trademark of Sybase, Inc.

Unicode is a trademark of Unicode, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Due to the nature of this material, this document refers to numerous hardware and software products by their trade names. In most, if not all cases, these designations are claimed as trademarks or registered trademarks by their respective companies. It is not this publisher's intent to use any of these names generically. The reader is therefore cautioned to investigate all claimed trademark rights before using any of these names other than to refer to the product described.

Copyright © 2002, by Information Builders, Inc. All rights reserved. This manual, or parts thereof, may not be reproduced in any form without the written permission of Information Builders, Inc.

Printed in the U.S.A.

Preface

This manual provides information on the iWay Server Structured Query Language (SQL). It is primarily intended for those persons concerned with programming applications that use the iWay Server SQL.

iWay is a family of integrated middleware products which, when combined, insulate both application end-users and developers from the complexity and incompatibilities of proprietary computing environments.

How This Manual Is Organized

This manual includes the following chapters:

Chapter/Appendix		Contents
1	Introduction	Provides an overview of the Server SQL.
2	SQL Services	Explains how the server handles SQL requests, using either SQL Passthru or SQL Translation.
3	Language Elements	Describes the language elements of SQL, including characters, tokens, qualifiers and identifiers.
4	SQL Reference	Provides a summary of supported and unsupported SQL statements and expected results from queries.
5	Tables and Views	Describes how to build tables and views.
6	Preparing and Executing SQL Requests	Describes how to use PREPARE to submit parameterized SQL requests for subsequent execution using EXECUTE.
A	SQL Translate Keywords	Lists SQL keywords.
B	BNF Summary	Provides a summary of Backus Naur Form notation.

Documentation Conventions

The following conventions apply throughout this manual:

Convention	Description
THIS TYPEFACE or <i>this typeface</i>	Denotes syntax that you must enter exactly as shown.
<i>this typeface</i>	Represents a placeholder (or variable) in syntax for a value that you or the system must supply.
<u>underscore</u>	Indicates a default setting.
<i>this typeface</i>	Represents a placeholder (or variable) in a text paragraph, a cross-reference, or an important term.
this typeface	Highlights a file name or command in a text paragraph that must be lowercase.
<i>this typeface</i>	Indicates a button, menu item, or dialog box option you can click or select.
Key + Key	Indicates keys that you must press simultaneously.
{ }	Indicates two or three choices; type one of them, not the braces.
[]	Indicates a group of optional parameters. None are required, but you may select one of them. Type only the parameter in the brackets, not the brackets.
	Separates mutually exclusive choices in syntax. Type one of them, not the symbol.
...	Indicates that you can enter a parameter multiple times. Type only the parameter, not the ellipsis points (...).
.	Indicates that there are (or could be) intervening or additional commands.

Related Publications

Visit our World Wide Web site, <http://www.iwaysoftware.com>, to view a current listing of our publications and to place an order. You can also contact the Publications Order Department at (800) 969-4636.

Customer Support

Do you have questions about iWay Server Structured Query Language (SQL)?

Call Information Builders Customer Support Service (CSS) at (800) 736-6130 or (212) 736-6130. Customer Support Consultants are available Monday through Friday between 8:00 a.m. and 8:00 p.m. EST to address all your iWay Server Structured Query Language (SQL) questions. Information Builders consultants can also give you general guidance regarding product capabilities and documentation. Please be ready to provide your six-digit site code number (xxxx.xx) when you call.

You can also access support services electronically, 24 hours a day, with InfoResponse Online. InfoResponse Online is accessible through our World Wide Web site, <http://www.informationbuilders.com>. It connects you to the tracking system and known-problem database at the Information Builders support center. Registered users can open, update, and view the status of cases in the tracking system and read descriptions of reported software issues. New users can register immediately for this service. The technical support section of www.informationbuilders.com also provides usage techniques, diagnostic tips, and answers to frequently asked questions.

To learn about the full range of available support services, ask your Information Builders representative about InfoResponse Online, or call (800) 969-INFO.

Information You Should Have

To help our consultants answer your questions most effectively, be ready to provide the following information when you call:

- Your six-digit site code number (xxxx.xx).
- Your iWay Software configuration:
 - The iWay Software version and release.
 - The communications protocol (for example, TCP/IP or LU6.2), including vendor and release.
- The stored procedure (preferably with line numbers) or SQL statements being used in server access.
- The database server release level.

User Feedback

- The database name and release level.
- The Master File and Access File.
- The exact nature of the problem:
 - Are the results or the format incorrect? Are the text or calculations missing or misplaced?
 - The error message and return code, if applicable.
 - Is this related to any other problem?
- Has the procedure or query ever worked in its present form? Has it been changed recently? How often does the problem occur?
- What release of the operating system are you using? Has it, your security system, communications protocol, or front-end software changed?
- Is this problem reproducible? If so, how?
- Have you tried to reproduce your problem in the simplest form possible? For example, if you are having problems joining two data sources, have you tried executing a query containing just the code to access the data source?
- Do you have a trace file?
- How is the problem affecting your business? Is it halting development or production? Do you just have questions about functionality or documentation?

User Feedback

In an effort to produce effective documentation, the Documentation Services staff welcomes any opinion you can offer regarding this manual. Please use the Reader Comments form at the end of this manual to relay suggestions for improving the publication or to alert us to corrections. You can also use the Documentation Feedback form on our Web site, <http://www.iwaysoftware.com>.

Thank you, in advance, for your comments.

iWay Software Consulting and Training

Interested in training? Our Education Department offers a wide variety of training courses for iWay Software and other Information Builders products.

For information on course descriptions, locations, and dates, or to register for classes, visit our World Wide Web site (<http://www.iwaysoftware.com>) or call (800) 969-INFO to speak to an Education Representative.

Interested in technical assistance for your implementation? Our Professional Services department provides expert design, systems architecture, implementation, and project management services for all your business integration projects. For information, visit our World Wide Web site (<http://www.iwaysoftware.com>).

Contents

1. Introduction	1-1
SQL Translation Services	1-2
Language Dialect	1-2
2. SQL Services	2-1
SQL Passthru Services	2-2
Using SQL Passthru	2-2
Rules for SQL Passthru	2-4
SQL Passthru Processing	2-4
Enabling SQL Passthru	2-5
Specifying an RDBMS Using SET SQLENGINE	2-5
Setting the Database Engine Using a Client	2-5
Setting the Database Engine Using Stored Procedures	2-6
Setting the Database Engine Using the API	2-7
Catalog Requirements	2-8
Valid Database Engine Settings	2-8
SQL Translation Services	2-11
SQL Translation Operations	2-11
Setting Automatic Passthru	2-12
Dialect Translation	2-13
Rules for Dialect Translation	2-14
Dialect Translation Processing	2-15
Data Manipulation Language (DML) Generation	2-16
Join Optimization in DML Generation	2-17
Setting Join Strategy	2-17
Nested Loop Processing	2-18
Joining Columns of Unequal Length	2-20
Turning Join Optimization ON	2-21
WHERE Predicate Cloning	2-21
3. Language Elements	3-1
BNF Conventions	3-2
Characters	3-3
Letters	3-3
Digits	3-3
Special Characters	3-3

Tokens	3-5
Keywords	3-5
Literals	3-5
Identifiers	3-6
Identifier Naming Conventions	3-6
Qualifiers	3-6
4. SQL Reference	4-1
Supported and Unsupported SQL Statements	4-2
Supported SQL Statements and Features	4-2
Unsupported SQL Statements and Features	4-3
Expected Results From SQL Queries	4-3
SELECT Statement	4-4
FROM Clause	4-5
Outer Joins	4-5
Correlation Names	4-6
WHERE Clause	4-8
GROUP BY Clause	4-9
HAVING Clause	4-10
AS Clause	4-10
Aggregate Functions	4-11
Subqueries	4-12
Correlated Subqueries	4-13
Generating a Result Set	4-13
ORDER BY	4-14
Scalar Functions	4-14
Column Functions	4-15
Predicates	4-16
Comparison Predicates	4-16
Between Predicates	4-16
Like Predicates	4-17
Null Predicates	4-17
In Predicates	4-18
Quantified Predicates	4-18
Existence Predicates	4-18
Search Conditions	4-19
Data Manipulation Commands	4-20
INSERT	4-20
DELETE	4-21
UPDATE	4-21
Commit and Rollback	4-22
COMMIT	4-22
ROLLBACK	4-22

5. Tables and Views	5-1
Creating Tables	5-2
Data Types	5-2
Null Values	5-2
Column Names and Aliases	5-2
CREATE TABLE	5-3
Creating Views	5-4
CREATE VIEW	5-4
6. Preparing and Executing SQL Requests	6-1
Preparing SQL Requests	6-2
Executing SQL Requests	6-2
A. SQL Translate Keywords	A-1
Keywords	A-2
B. BNF Summary	B-1
Condensed SQL Language Syntax Definition	B-2

Contents

CHAPTER 1

Introduction

Topics:

- SQL Translation Services
- Language Dialect

SQL requests may be expressed in a universally applicable, ISO standard SQL dialect, or in a Database Management System (DBMS) specific dialect supported by any available native SQL engine.

When data is known to be homogeneous, when performance is a major concern, or when the unique features of a particular DBMS can be used advantageously, developers should submit queries native to the particular DBMS. In these instances, a service called SQL Passthru is used.

However, when direct usage of a DBMS engine is either impossible or undesirable, developers should invoke ISO standard SQL. In these instances, a service called SQL Translation is used. Every programmer should become familiar with ISO standard SQL because of its broad applicability and its ability to impose a relational view on non-relational data.

SQL Passthru Services is an optional component of a server. It provides SQL operations against proprietary Relational Database Management Systems (RDBMS). Using SQL Passthru, the server can pass SQL requests directly to the specified RDBMS for processing, without translating it in any way.

SQL Translation Services

SQL Translation Services is the core component of a server. Using a feature called Automatic Passthru, the server initiates one of two possible processes to translate SQL requests and generate internal SQL command streams:

- Dialect Translation converts inbound SQL statements to statements that can be processed by the target DBMS engine.
- Data Manipulation Language (DML) converts inbound SQL that is ineligible for Dialect Translation to the internal retrieval language, TABLE. A data adapter then converts this to the target DBMS engine's internal DML.

After one of these conversions is complete, the request is passed to a backend database access module for subsequent action. It is the responsibility of the backend module to interpret internal SQL text and to interact with individual DBMSs as required.

See Chapter 2, *SQL Services*, for more information on SQL Passthru and SQL Translation Services.

The purpose of this manual is to define ISO SQL in precise terms. It contains no information about native RDBMSs or their language support. Every mention of SQL within the body of this document should be considered a reference to ISO SQL—the dialect of SQL implemented by SQL Translation Services.

Language Dialect

The server is designed to accommodate the broadest possible range of SQL applications. Accordingly, the API encourages connector programs to generate SQL requests during run-time rather than rely on predefined, hard-wired commands. The major technical implication of this design philosophy is that servers bind and execute SQL requests dynamically.

In contrast, ISO SQL requires that SQL requests be hard-coded in the application programs that invoke them. Although the server, with its dynamic orientation, is not ISO-compliant in this sense, it adheres very closely to the standard. The syntax of every statement is patterned directly on the ISO SQL model and most statements are exactly the same.

SQL Translation Services has what might be called a “read-mostly” emphasis. It supports full ISO SELECT syntax and semantics. But certain features of SQL—CREATE TABLE, CREATE VIEW, INSERT, UPDATE, and DELETE—to be specific, are subject to restrictions. The lifetime of a table or view, for example, is limited to a single session, and a table or view manufactured by one user is not generally accessible to any other user. Finally, all database objects that come into being through the translator mechanism are managed by the server.

Such restrictions are in line with the read-mostly emphasis of SQL Translation Services. A table is regarded as a temporary repository for information extracted from a production database rather than an integral part of that database. SQL Translation Services invites the user to create one or more tables, populate them with data selected from production databases, and query them in the knowledge that they will disappear at the close of the session.

CHAPTER 2

SQL Services

Topics:

- SQL Passthru Services
- Using SQL Passthru
- Enabling SQL Passthru
- Catalog Requirements
- Valid Database Engine Settings
- SQL Translation Services
- SQL Translation Operations
- Dialect Translation
- Data Manipulation Language (DML) Generation
- Join Optimization in DML Generation

This chapter explains how the server handles SQL requests, using either SQL Passthru or SQL Translation.

SQL Passthru Services

SQL Passthru Services provides SQL operations against proprietary RDBMSs. The server can pass SQL requests directly to the specified RDBMS for processing, without intercepting the request or translating it in any way. This provides an efficient way to process requests against a specific RDBMS, while incurring minimal system overhead.

A data adapter for each type of supported relational data source needs to be installed. See the *Server Administration* manual for information about using data adapters.

Using SQL Passthru

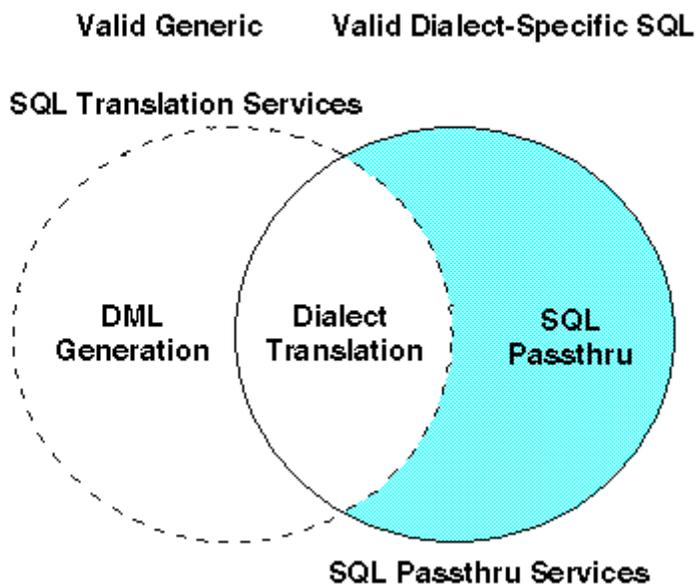
SQL Passthru Services can be used under certain conditions on a Hub or Full-Function Server.

A Hub or Full-Function Server can utilize SQL Passthru to operate in SQL Passthru mode by session or per request—the user is responsible for activating and deactivating SQL Passthru as needed. This requires the RDBMS engine to be set on a client, using either a stored procedure or the API.

In addition, a Hub or Full-Function Server using its dialect translation capability attempts to pass SQL requests to the target RDBMS using SQL Passthru.

The user must ensure that dialect-specific SQL is being passed to the source RDBMS when using SQL Passthru. A client application passing anything other than dialect-specific SQL receives an error message. SQL Passthru is shown in the solid-line circle in the figure below.

SQL Statements



The solid-line circle represents the supported dialect-specific SQL requests of SQL Passthru (some of which are also supported by Automatic Passthru).

Dialect translation and DML generation are discussed in *Dialect Translation* on page 2-13 and *Data Manipulation Language (DML) Generation* on page 2-16, respectively.

Rules for SQL Passthru

Because SQL Passthru sends SQL requests directly to the RDBMS, it only supports valid dialect-specific SQL. Valid dialect-specific SQL complies with the syntax required by the source RDBMS.

The user initiating the SQL request must ensure that the request:

- Uses the valid dialect-specific SQL of the RDBMS.
- Refers to fully-qualified table and column names of the RDBMS.



Note

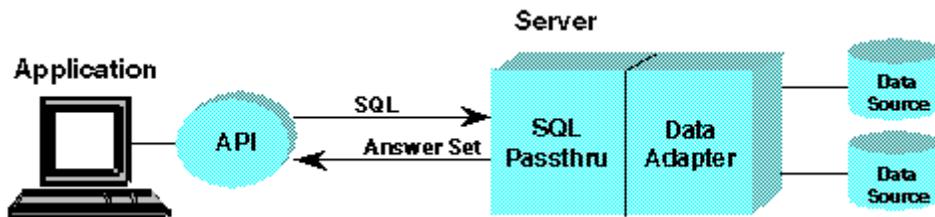
A three-part name can be used if the target RDBMS allows three qualifiers, such as LOCATION.CREATOR.TABLE.

If the RDBMS receives SQL requests that are not compatible with its dialect or table names, it generates an error message and returns it to the server. The server then returns the error message to the application.

SQL Passthru Processing

This section provides an example of SQL requests targeting a single RDBMS using SQL Passthru.

1. The client application issues a request in the dialect-specific SQL of the targeted RDBMS.
2. The server invokes an SQL Passthru agent, which passes the request to the applicable data adapter.
3. On receiving the request, the data adapter attaches to the RDBMS, using standard database attachment calls. The data adapter then passes the request to the RDBMS.
4. The RDBMS processes the request.
5. The results are returned to the client application.



For details on the SQL features supported for a specific RDBMS, see the documentation for the RDBMS and Chapter 4, *SQL Reference*.

Enabling SQL Passthru

There are two ways to enable SQL Passthru:

- Specifying an RDBMS using the SET SQLENGINE command.
- Setting the database engine using the client.

Both of these methods are discussed below.

Specifying an RDBMS Using SET SQLENGINE

If you configure a Hub or Full-Function Server, the server profile does not contain an engine setting by default. Instead, the Hub or Full-Function Server intercepts the request and then either passes it directly to the RDBMS (Automatic Passthru), or converts it into Data Manipulation Language before passing it to the RDBMS (SQL Translation).

Example Specifying a Particular RDBMS

```
SET SQLENGINE=SQLORA
```

In this example, SQLORA is the name used to specify Oracle. See *Valid Database Engine Settings* on page 2-8 for a complete list of database engine settings.

Setting the Database Engine Using a Client

A client connecting to a Hub or Full-Function Server can set SQL Passthru on demand. This effectively puts the server into SQL Passthru mode per session or per query.

When the client sets the engine in this manner, SQL Passthru behavior is altered for that client's session only; it does not affect any other client/server sessions.

The client application can set a database engine using two methods:

- Executing stored procedures that specify a database engine setting.
- Setting the database engine using the API.

Before using the client to change the database engine setting, you must configure the server environment. For more information, see the *Server Administration* manual.

Setting the Database Engine Using Stored Procedures

The client application can execute stored procedures to enable SQL Passthru on a per session or per request basis.

Example Putting the Client's Session in SQL Passthru Mode for All Requests

To put the client's session in SQL Passthru mode for all following requests, the client must execute a stored procedure that contains the SET SQLENGINE command, as shown in the following example.

```
SET SQLENGINE = SQLORA
  SQL SELECT COST FROM ORDERS;
  TABLE
  ON TABLE PCHOLD
  END
  SQL SELECT SALARY FROM EMPLOYEE;
  TABLE
  ON TABLE PCHOLD
  END
```

In this example, all SQL requests after the SET SQLENGINE command are passed directly to Oracle until the client disconnects. This is equivalent to setting the engine in the server profile, but affects only the client's session.

When the database engine is specified in a stored procedure, the server passes requests directly to the RDBMS for processing. This behavior persists as long as the database engine is set on the server. Therefore, RDBMS rules apply regarding SQL syntax and qualification of table and column names. If the request is not acceptable to the RDBMS, the server returns an error message to the client application.

In addition, when the database engine is set, ODBC stored procedure requests use RDBMS-specific ODBC programs.

Syntax How to Terminate SQL Passthru Mode

To terminate SQL Passthru mode, another stored procedure must be executed that contains the following command:

```
SET SQLENGINE = OFF
```

Example Putting the Client's Session in SQL Passthru Mode for a Single Request

To put the client's session in SQL Passthru mode for a single request, the client must execute a stored procedure that contains the SQL *engine* command. In this mode, the server processes any requests that include this command using SQL Passthru, while processing all other requests using SQL Translation. This is shown in the following example.

```
SQL SQLDS
SELECT COUNTRY FROM OWNER.CAR;
TABLE
ON TABLE PCHOLD
END

SQL SQLORA
SELECT * FROM OWNER.EMPLOYEE;
TABLE
ON TABLE PCHOLD
END
```

In this example, the server passes the first query directly to DB2, and the second to Oracle. The engine setting applies only for the duration of the SELECT statement. When the engine is specified in a stored procedure, RDBMS rules apply regarding SQL syntax and qualification of table and column names. If the request is not acceptable to the RDBMS, the server returns an error message to the client application.

For a list of database engine settings that can be specified in a stored procedure, see *Valid Database Engine Settings* on page 2-8.

Setting the Database Engine Using the API

An API application can issue a command to set the database engine to a specific RDBMS. All subsequent EDASQL and EDAPREPARE calls are directed to that RDBMS for the duration of the session. Remote procedure calls (executed by EDARPC), however, are not affected by the engine setting.

The engine setting can be changed as needed. In this way, applications can request data from multiple RDBMSs.

Syntax

How to Set the Database Engine Using the API

You can set the database engine in the API application by issuing the command

```
EDASET (0, eng)
```

where:

0

Is the code that indicates you are changing the database engine setting. Must be set to 0 (zero).

eng

Is a database engine setting for the API. When the database engine is set in the application, the server expects SQL in the dialect of the RDBMS. Otherwise, it returns an error message to the application. If no engine is specified, the server defaults to SQL Translation.

See *Valid Database Engine Settings* on page 2-8 for a list of permissible settings for RDBMSs.

You can also specify a database engine using the Connector for ODBC. For instructions, see the *Connector for ODBC* manual.

Catalog Requirements

The following considerations apply to cataloging metadata when using SQL Passthru:

- Applications, including ODBC applications, written to access RDBMS catalog tables can use the native RDBMS catalog.
- Before using the client to change the database engine setting, you must configure the server environment. For more information, see the *Server Administration* manual.

Valid Database Engine Settings

This section lists the valid database engine settings that you can:

- Include in the server profile.
- Issue in a stored procedure.
- Use in an API application.

Reference Engine Settings for a Server Profile or Stored Procedure

The following table lists valid database engine settings. If you are using a Hub or Full-Function Server, you may include one of the settings in a stored procedure.

SQLDBC	Teradata
SQLDS or DB2	DB2, DB2/6000, DB2/2 (for MVS, UNIX, and Windows NT)
SQLINF	Informix
SQLING	Ingres
SQLIPX	Unisys DMS/RDMS 1100/2200
SQLMAC	Microsoft Access
SQLMSS	Microsoft SQL Server
SQLNUC	Nucleus
SQLODBC	ODBC (Generic ODBC when DBMS is not supported)
SQLORA	Oracle
SQLPRO	Progress
SQLRDB	Rdb
SQLRED	Red Brick
SQLSYB	Sybase
SQLUV	Universe
EDA or SQLEDA	The SQL command or environment command is processed by the server, not by SQL Passthru. Note: This setting is not valid for CREATE SYNONYM.
OFF	Terminates SQL Passthru mode if previously activated in a stored procedure.

Reference Engine Settings for API Applications

The following table lists the valid database engine settings for use in API applications.

0	NOENGINE
1	DB2
2	ORACLE
3	TERADATA
4	SQL/DS
6	Microsoft SQL Server
8	INGRES
9	SYBASE
10	RDB
11	INFORMIX
13	Internal
15	SQL-400
16	NONSTOP-SQL
17	ALLBASE
18	PROGRESS
19	RED BRICK
20	UNIVERSE
23	IDMS/SQL
25	SQLODBC
26	SQLMSQ
27	SQLNUC
28	SQLMAC

SQL Translation Services

SQL Translation is used with Hub and Full-Function Servers. It facilitates SQL read operations against relational and non-relational data sources and write operations against relational data sources, IMS, and FOCUS data sources.

A Hub or Full-Function Server processes all incoming SQL requests using SQL Translation unless the application or profile setting sets SQL Passthru.

SQL Translation Operations

SQL Translation performs two distinct types of operations:

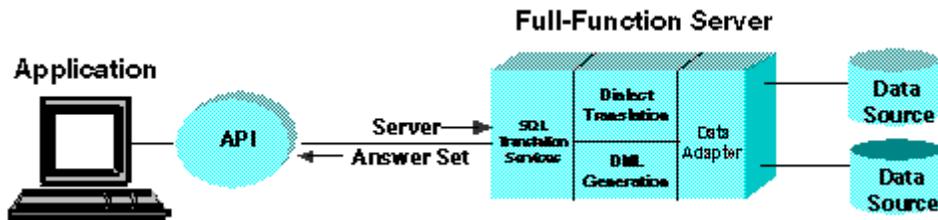
- Dialect Translation transforms the query into native SQL dialects.
- The query is reconstructed into Data Manipulation Language (DML), allowing for retrieval from the native RDBMS.

The mode of operation is determined by the Automatic Passthru (APT) variable setting. See *Setting Automatic Passthru* on page 2-12 for a description of this variable.

When an SQL request is processed, the following takes place:

1. The request is parsed to ensure that the syntax is valid. If it is not, an error is returned to the application. Parsing produces an abstract syntax tree data structure to be used in later analysis.
2. The Server Dynamic Catalog is used to determine the data source, information about the tables and columns, and the data types of the columns.
3. Semantic analysis is performed on the abstract syntax tree to ensure that the query conforms to the semantic rules. If not, an error is returned to the application.
4. If the Server Dynamic Catalog contains DBA or the query refers to a defined column in the catalog, DML Generation is required (see *Data Manipulation Language (DML) Generation* on page 2-16 for more information). Processing continues with Step 8.
5. If APT=ON or APT=ONLY and the data source is a single RDBMS engine, the feature flags for that RDBMS are obtained.
6. The statement is checked against the feature flags. If the construct has an analogous RDBMS construct as indicated by the feature flags, Dialect Translation is invoked to regenerate the request using the same or similar syntax of the RDBMS. (See *Dialect Translation* on page 2-13 for more information.) Processing continues with Step 10.
7. If APT=ONLY and the request is not translatable, the request is rejected and an error is returned to the application.

8. If APT=OFF, or if APT=ON and the data source is *not* from a single RDBMS or there is no feature flag for a given construct, DML Generation is invoked.
9. The appropriate data retrieval logic is triggered to process the translated dialect or the DML.
10. The results for SELECT operations from Dialect Translation or DML Generation are returned to the application as an answer set as shown below.



Setting Automatic Passthru

Automatic Passthru (APT) refers to the server profile option that enables SQL Translation Services to translate eligible incoming requests into native SQL dialects. This process, called Dialect Translation, passes the generated SQL to SQL Passthru. This processing is transparent to the application when APT is turned on.

You can use the API call EDASET to set the database engine. This engine, in turn, sets the Passthru mode. For more information on the EDASET API call, see the *API Reference* manual.

Syntax How to Set Automatic Passthru

To change the setting for APT, issue the following command on the server:

```
SQL
SET APT[=] set_apt ;
END
```

where:

set_apt

Allows you to set the Automatic Passthru mode. Possible values are:

[ON](#) turns APT on. This is the default setting in the server profile of a Hub or Full-Function Server. When this is the value of the APT setting, SQL Translation analyzes each inbound request to determine whether Dialect Translation can be invoked to pass the request to SQL Passthru.

The server determines whether the SQL request is valid for the targeted RDBMS:

- If the request passes all the rules of Dialect Translation, it is translated and passed to SQL Passthru for processing.
- If the server receives an SQL request that does not conform to the rules of Dialect Translation, it invokes DML Generation to translate the request into Data Manipulation Language to be handled by one or more data adapters. The data adapters then generate engine-specific DML and pass it to the DBMSs for processing.

See *Rules for Dialect Translation* on page 2-14 for the list of rules.

OFF turns APT off. All SQL requests are processed using DML Generation, which translates them into Data Manipulation Language to be processed by the appropriate data adapters.

ONLY states that all requests must be appropriate for Dialect Translation, that is, they must meet the requirements defined in *Rules for Dialect Translation* on page 2-14. No SQL requests go through DML Generation. If a request does not meet the requirements, the query is rejected, and an error is returned to the application.

This enables Data Administrators to configure access to servers to avoid any overhead that would be incurred when generating DML.

Dialect Translation

Dialect Translation is a feature of SQL Translation Services that enables a Hub or Full-Function Server to route inbound SQL requests to SQL-capable subservers and data adapters wherever possible. Dialect Translation avoids translation to Data Manipulation Language, while maintaining data location transparency.

Dialect Translation transforms an incompatible statement into one that can be processed by the destination SQL engine, while preserving the semantic meaning of the statement.

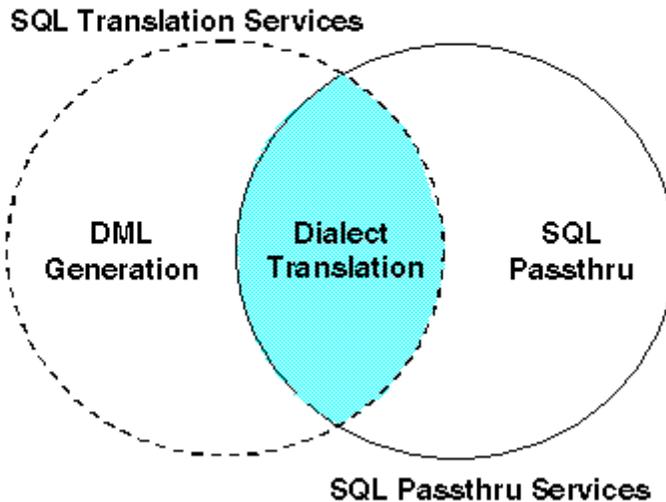
Dialect Translation is the default behavior of SQL Translation in a Hub or Full-Function Server. This behavior can be modified by changing the server profile or executing a stored procedure to modify the value of the APT variable.

To qualify for Dialect Translation, the request must meet the requirements outlined in *Rules for Dialect Translation* on page 2-14. If the syntax of a request does not conform to the syntax of the DBMS or is not translatable to the syntax of the DBMS, the request will be translated to DML.

Dialect Translation is shown in the shaded area of the following figure. Valid requests are represented by the broken-line circle as shown in the figure below.

SQL Statements

Valid Server SQL Valid Dialect-Specific SQL



Rules for Dialect Translation

If the incoming request meets the following criteria, Dialect Translation is invoked by SQL Translation Services:

- APT is set to ON or ONLY in the server profile.
- All data referenced by the request resides in an RDBMS.
- No defined columns in the table are referenced.
- The referenced tables contain no security restrictions as listed in the Server Dynamic Catalog.

- The syntax of the incoming SQL statement must correspond precisely to the syntax of the RDBMS, or must be translatable into the RDBMS's dialect of SQL based on the feature flags for that RDBMS. The server translates the following into the dialect of the target RDBMS:
 - The “not equal” symbol.
 - The substring scalar function, length.
 - The concatenation binary operator.
 - The PREPARE parameter marker.
 - Date literals, durations.
 - Cast-type scalar functions (those functions that apply to an expression or column name and change only the data type, not the value).
 - Outer joins (with limitations).
 - For fetch only.
 - `<> = NULL`.

If a request does not meet these requirements but is valid, the server translates the request into Data Manipulation Language using DML Generation. One or more data adapters then generate driver-specific DML and pass it to the DBMS for processing.

Dialect Translation Processing

The following steps illustrate how a request for relational data is processed using Dialect Translation on a Hub or Full-Function Server:

1. The abstract syntax tree data structure is traversed:
 - For each construct in the abstract syntax tree that corresponds to a feature flag, Dialect Translation substitutes a dialect-correct RDBMS construct.
 - Each table name is changed to its actual name in the targeted DBMS, as stored in the Server Dynamic Catalog.
 - Each column name is changed to its actual name, as stored in the Server Dynamic Catalog.
2. The SQL statement is rebuilt token by token from the abstract syntax tree.
3. The resulting generated SQL is passed to SQL Passthru for processing.

4. SQL Passthru sends the request to the RDBMS, which executes it and, for SELECT operations, returns an answer set.
5. The server returns any resulting answer set to the client.

Incoming SQL syntax is often compatible with the dialect-specific SQL of the target RDBMS. In that case, Dialect Translation substitutes the correct table and column names rather than translates the dialect.

Data Manipulation Language (DML) Generation

DML Generation transforms requests into a format that allows retrieval from native DBMS engines. Specifically, it translates the request into the Data Manipulation Language (DML) and invokes one or more data adapters. A data adapter generates the native DML of the DBMS and passes it to the DBMS for processing.

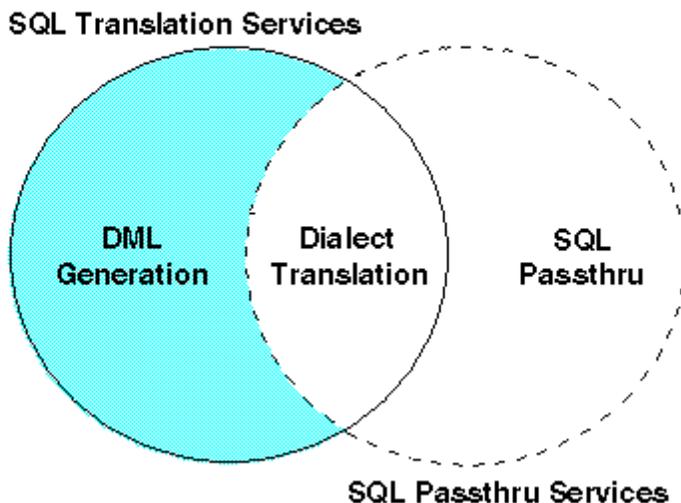
DML Generation is used when an application attempts to perform one of the following:

- Read from non-relational data sources.
- Address a join between heterogeneous data sources in a single SQL statement.
- Join two or more relational tables from different RDBMS engines.
- Join data sources that are distributed across servers.
- Send an SQL request using syntax for which there is no corresponding feature flag for the target RDBMS.
- Retrieve defined columns.
- Retrieve data from tables having security restrictions in the Server Dynamic Catalog.
- Use a Hub or Full-Function Server whose APT setting is OFF.

DML Generation is shown in the shaded portion of the solid-line circle as shown in the figure below.

SQL Statements

Valid Server SQL Valid Dialect-Specific SQL



Join Optimization in DML Generation

This section describes how joins can be optimized for DML Generation using SQL Translation and Hub Servers. This is particularly important when joining heterogeneous data sources across servers.

Setting Join Strategy

SQL Translation offers two types of join strategy in DML Generation: nested loop and sort/merge. To set a preferred join strategy, issue the following command in the Hub Server profile.

Syntax

How to Set Preferred Join Strategy

SQL EDA SET JOINTYPE {[NESTEDLOOP](#)|[SORTMERGE](#)}

where:

[NESTEDLOOP](#)

Selects all rows from the outer table that match screening conditions, then uses those rows to obtain qualified data from the inner tables. DML Generation uses this strategy only for an equijoin on joinable columns.

Nested Loop is very effective when the screening conditions limit the number of rows returned from a file. It is not recommended otherwise.

NESTEDLOOP is the default for JOINTYPE if this setting is deleted from the server profile.

[SORTMERGE](#)

Retrieves data from each table and then sorts, merges, and aggregates at the end of the process. When SORTMERGE is specified, DML Generation *always* uses a sort/merge strategy to retrieve data. By default, the Hub Server contains an entry in the server profile setting JOINTYPE to SORTMERGE.

Sort/merge is used automatically if a query fails nested loop conditions. A Hub Server uses sort/merge, even if nested loop is the set strategy, if the server determines that one or more join conditions are for non-joinable columns. A non-joinable column is one in a non-relational data source without a declared index or key.

Sort/merge works effectively with large volumes of data. A sort/merge always fetches all data from the outer table to disk, then joins them locally.

Nested Loop Processing

If JOINTYPE is not set or is set to NESTEDLOOP, DML Generation performs a set of optimization functions:

- Determines the order of the tables in the FROM clause.
- Determines which columns in the WHERE clause are joinable and at what cost.
- Determines whether the formats of the potential join columns are compatible.

DML Generation uses the following methodology:

1. Assigns a relative cost to each join candidate to find the least expensive join structure.
2. Examines each WHERE clause predicate.

If it finds an equijoin (that is, WHERE clauses of the form $fld1 = fld2$, where $fld1$ and $fld2$ are columns from two of the FROM clause tables), DML Generation determines if either $fld1$ or $fld2$ is joinable.

3. Examines the join columns to ensure that they have compatible formats.
4. Creates a matrix of joinability containing information about the costs of joining from table to table. Each join candidate is assigned a cost, as follows:

COST=1	An equijoin to a joinable column (the <i>to</i> column is joinable and the columns have compatible formats).
COST=16	An equijoin to a non-joinable column (the <i>to</i> column is not joinable or the columns do not have compatible formats).
COST=256	A non-equijoin or unrestricted Cartesian product (there is no such relationship present in the WHERE clause).

After it builds the matrix, DML Generation tries the following join strategies:

- **Single-path join** based on the FROM clause left-to-right order. The tables in the FROM clause are examined to determine if the first table is joinable to the second, the second to the third, and so forth. This is done by examining the matrix for each pair in FROM order. If each pair has a cost of 1, then this strategy is taken, resulting in the creation of a single-path nested loop structure.
- **Multi-path join** based on the FROM clause left-to-right order. DML Generation examines the matrix using the leftmost table as the outermost table. It then determines if it can join the FROM tables in such a way that each TO table is to the right of the FROM table and the cost of each is 1. This results in the creation of a multi-path nested loop structure.

- **Composite join structure.** DML Generation sorts the costs and chooses those with a cost of 1 over those with a higher cost. It builds a structure that either:
 - Is composed completely of equijoins based on joinable columns.
 - Requires the creation of intermediate hold tables to accomplish an equijoin on non-joinable columns.
 - Requires the creation of intermediate hold tables to accomplish a Cartesian product-based join.

DML Generation tries to respect the order of tables in the FROM clause. Therefore, optimal performance is achieved by making the most restricted table the outer table—that is, first in the FROM clause.

In any case, the least expensive join based on these rules is constructed and the rows returned are the same.

Joining Columns of Unequal Length

All data adapters allow joins of columns of unequal length. With this capability, DML Generation assigns a cost of 1 to alphabetic joins of unequal length and between short-packed (8-byte) and long-packed (16-byte) columns.

Reference Joinable Columns

DML Generation considers the following columns to be joinable:

Host Column	Target Column
A (any length)	A (any length)
short P	short P
Long P	long P
short P	long P
Long P	short P
I	I
D	D
F	F
new date	new date
old date (I)	old date (I)

Host Column	Target Column
new date	old date (I)
old date (I)	new date

**Note**

Old dates of alpha format are not supported. If you have such a column, it should be defined as alpha.

Some data adapters for legacy files allow a generic join, which violates relational join rules by including extra columns that do not strictly match the select list. For those data adapters, DML Generation reapplies the screening criteria after the records are retrieved to ensure strict relational compliance.

For all signed numeric columns, the handling of different values for the + and - is performed by the data adapter, *not* DML Generation. All values corresponding to + and - should be retrievable in a join.

Turning Join Optimization ON

Join optimization is controlled by the SET JOINOPT setting in the server profile. It is turned ON by default. For more information on SET JOINOPT, see the *Server Administration* manual.

WHERE Predicate Cloning

DML Generation also clones WHERE clause conditions, where appropriate, for optimal join performance. This optimization is performed for both nested loop and sort/merge joins when there is explicit value selection criteria on one of the columns.

DML Generation builds a table of equivalent columns, which have the same value for all rows in the answer set. The equivalencies are found by searching the WHERE clause for predicates that equate one database column to another (for example, $A = B$). Equijoins always fit this pattern. Next, all WHERE clause predicates on columns in the equivalence table are duplicated, and the column name in the predicate is replaced with its equivalent column.

Example Cloning the WHERE Clause Predicates Using Nested Loop

The following example illustrates an optimization procedure using a nested loop strategy. In this example:

- Table A is 100,000 rows, 100 of which refer to Smith (as a foreign key).
- Table B is 100 rows; one has Smith as its key.

For this SELECT statement

```
SELECT * FROM A,B WHERE B.key=A.key AND B.key= 'SMITH'
```

DML Generation clones the selection criteria for 'SMITH' by also applying it to A.key. Logically, the SELECT is transformed as follows:

```
SELECT * FROM A,B WHERE B.key=A.key AND B.key= 'SMITH' AND A.key= 'SMITH'
```

The subsequent processing involves the following operations:

1. Selecting rows from the outer table A by applying the generated WHERE clause criteria to retrieve the first 'SMITH' row.
2. Selecting rows from the inner table B by supplying the *from* column value of 'SMITH' and the WHERE clause criteria (also 'SMITH') to retrieve one row.
3. Repeating steps 1 and 2 a total of 99 more times.

The outer table SELECT returns 100 rows in one DBMS event. The inner SELECT is executed 100 times and returns one row for 100 DBMS events. The total is 200 *reads*, 101 DBMS events, and 100 rows returned for the query.

If DML Generation did not clone the WHERE predicate, the outer SELECT would return 100,000 rows, 99,900 of which would be rejected once the inner SELECT was applied using key values not equal to 'SMITH', which would return no rows. The total would have been 100,001 *reads*, 100,001 DBMS events, and 100 rows returned for the query.

Thus, with WHERE predicate cloning, maximum efficiency is reached: 200 *reads* instead of 100,001. WHERE clause cloning provides join optimization, and makes this operation almost as efficient, in this case, as the second example.

Example Cloning the WHERE Clause Predicates Using Sort/Merge

The following example illustrates an optimization procedure using a sort/merge strategy. In this example:

- Table A is 100,000 rows, 100 of which refer to Smith (as a foreign key).
- Table B is 100 rows; one has Smith as its key.

For this SELECT statement

```
SELECT * FROM A,B WHERE A.key=B.key AND B.key= 'SMITH'
```

DML Generation clones the selection criteria for 'SMITH' by applying it to A.key. Logically, the SELECT is transformed as follows:

```
SELECT * FROM A,B WHERE B.key=A.key AND B.key= 'SMITH' AND A.key= 'SMITH'
```

The subsequent processing involves the following operations:

1. Selecting rows from table A by applying the generated WHERE clause criteria to retrieve all 100 'SMITH' rows.

2. Selecting rows from table B by applying the original WHERE clause criteria to retrieve one row.
3. Merging the results from steps 1 and 2.

The first SELECT would return 100 rows for one DBMS event. The second would return one row for one DBMS event. The total is 101 *reads*, two DBMS events, and 100 rows returned for the query.

If DML Generation did not clone the WHERE predicate, there would have been 100,001 reads.

In each case, this transformation yields a query semantically equivalent to the original because A.key and B.key must be equal at all times to satisfy the preexisting equijoin criteria. In any retrieval scenario, the number of rows returned from table A will be reduced by the added predicate.

CHAPTER 3

Language Elements

Topics:

- BNF Conventions
- Characters
- Tokens
- Qualifiers

This chapter describes the Backus Naur Form (BNF) notation and the language elements of SQL.

BNF Conventions

This manual employs a variant of what is known as Backus Naur Form (BNF) notation to describe SQL syntax. In BNF, a language construct is defined as follows:

```
something ::= its definition
```

A language construct may be used on the right hand side of such an expression (sometimes called a production) to define something else. When a known construct appears on the right side of a production it is represented by its name enclosed within angled brackets (<...>).

```
a_new_thing ::= <any_old_thing>
```

The right hand side of a production may be composed of any sequence of bracketed names and printable characters. Curly brackets ({...}) are used to delimit choices separated by vertical bars and square brackets ([...]) are used to indicate optional phrases. When a bracketed expression is followed by an asterisk ([...]*) it represents zero, one or more of the enclosed language constituents. Printable characters employed outside of brackets in productions are meant to be used verbatim.

The ::= operator is the BNF equivalent of *is made up of* in English.

Example Using the ::= Operator

The following productions could be read as *computer books are made up of zero, one or more instances of information or examples and syntax*.

```
computer_books ::= [<technical_information>]* and syntax.  
technical_information ::= information | examples
```

Productions amount to formulas for creating a well-formed string in a given language. All of the following conform to the *computer_books* syntax:

```
information examples and syntax.  
and syntax.  
information information information and syntax
```

Reference **BNF Symbols**

The chart below summarizes the BNF symbols and their meanings.

Symbol	Meaning
{ }	Indicates a group of valid options. You must choose at least one of the enclosed options.
[]	Indicates a group of valid options. You can select one of the options or none.
::=	Indicates <i>is made up of</i> .
< >	Indicates the command or production qualifier or construct.
,	Indicates that you can select one or more or none of the options.
	Indicates that you can select one of the options or none.
[]*	Zero or more occurrences of item in brackets.

Characters

Characters are the fundamental building blocks of the SQL language. The user can compose names, phrases, expressions, literals, and other higher-level constructs with them. Upper and lower case letters, spaces, digits, and new line markers are all in the SQL character set, as are a variety of special symbols.

Letters

```
alpha ::= A..Z | a..z
```

A..Z means any character from A to Z.



Note

Digits

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0...9
```

```
digit ::= 0...9
```

Special Characters

```
special-character ::=
```

A special character is any character in the character set of a server other than a digit or an alpha.

<blank>	::=
<double quote>	::= "
<percent>	::= %
<ampersand>	::= &
<quote>	::= '
<left paren>	::= (
<right paren>	::=)
<asterisk>	::= *
<plus sign>	::= +
<minus sign>	::= -
<period>	::= .
<solidus>	::= /
<colon>	::= :
<semi-colon>	::= ;
<less than operator>	::= <
<equals operator>	::= =
<greater than>	::= >
<question mark>	::= ?
<left bracket>	::= [
<right bracket>	::=]
<underscore>	::= _
<vertical bar >	::=
<caret>	::= ^

Tokens

Tokens are sequences of characters having special lexical significance. There are three basic SQL token types: keywords, literals, and identifiers. A token can be followed by a delimiter or a separator.

```

delimiter ::= <alpha-literal> | , | ( | ) | < | > | . | : | =
           | * | + | - | / | <> | >= | <=
separator ::= <comment> | <space> | <newline>
comment   ::= --[<character>]*<newline>

```

Keywords

Keywords are tokens that have a predetermined meaning in the SQL language. For clarity, it is recommended that keywords not be used as identifiers. See Appendix A, *SQL Translate Keywords*, for a list of SQL keywords.

Literals

Literals are used in SQL statements to represent values that can occur in the database. Each type of literal represents a distinct kind of database value.

```

numeric-literal ::= [{+ | -}]<numeric-tail>
numeric-tail    ::= <integer>[.<integer>] | .<integer>
approx-literal  ::= <numeric-literal> E[{|+ | -}] <integer>
alpha-literal   ::= '[<lit-character>]*'
literal         ::= <alpha-literal> | <numeric-literal> |
                  <approx-literal>
integer         ::= <digit>[<digit>]*
lit-character   ::= <non-quote> | "

```



Note

A non-quote is any character in the character set of a server other than a single-quote symbol. Two single quotes can be used to represent an embedded quote (see the example below).

Example Specifying a Literal

```

'Down by the seaside'
'Ain't got nobody'
- 6.5

```

Identifiers

Identifiers are tokens used to name database objects. There are two types of identifiers: long and short. Short identifiers cannot exceed 8 characters in length; long identifiers can contain up to 48 characters. Both are formed according to the following lexical rules:

```

identifier      ::= <normal-identifier> | <delimited-identifier>
normal-identifier ::= <alpha>[<id-character>]*
delimited-identifier ::= "<any printable character string>"
id-character     ::= <alpha> | <digit> | _

```

Example Specifying an Identifier

```

DEPARTMENT
customer_id

```

Identifier Naming Conventions

Several terms are used in this document to represent identifiers. The purpose is to make it clear what kind of object is being manipulated by a given language construct. Syntactically, the terms have no particular significance—they all represent identifiers—but semantically, the difference is significant. Note that the names of certain classes of database objects must be short identifiers.

```
authorization-id ::= <short identifier>
location-id      ::= <long identifier>
column-name     ::= <long identifier>
table-name      ::= <short identifier>
view-name       ::= <long identifier>
statement-name  ::= <short identifier>
range-variable  ::= <long identifier>
procedure-name  ::= <identifier>
```

Qualifiers

Qualifiers are used to specify the precise information that is required.

```
table-ref        ::= [<table-qualifier>.]<table-name>
table-qualifier  ::= <loc-id>.<auth-id> | <auth-id>
column-ref       ::= [<column-qualifier>.]<column-name>
column-qualifier ::= <table-ref> | <range-variable>
```

CHAPTER 4

SQL Reference

Topics:

- Supported and Unsupported SQL Statements
- SELECT Statement
- Scalar Functions
- Column Functions
- Predicates
- Data Manipulation Commands
- Commit and Rollback

This chapter contains a summary of supported and unsupported SQL statements, clauses, variables, functions, predicates, and commands.

Supported and Unsupported SQL Statements

This section contains a summary of supported and unsupported SQL statements and the expected results from SQL queries.

Supported SQL Statements and Features

SQL Translation supports the following:

- SELECT, including SELECT ALL and SELECT DISTINCT.
- CREATE TABLE. The following data types are supported for CREATE TABLE: REAL, DOUBLE PRECISION, FLOAT, INTEGER, DECIMAL, CHARACTER, and SMALLINT.
- DROP TABLE for relational data sources.
- INSERT, UPDATE, and DELETE for relational data sources, IMS, Adabas, VSAM, and FOCUS data sources.
- Equijoins and non-equijoins.
- CREATE VIEW and DROP VIEW.
- PREPARE and EXECUTE.
- Delimited identifiers of table names and column names. Table and column names containing embedded blanks or other special characters in the SELECT list should be enclosed in double quotation marks.
- AS phrase, used in conjunction with SELECT statements.
- The UNION and UNION ALL operators.
- Non-correlated subqueries for all requests.
- Correlated subqueries for requests that are candidates for Dialect Translation to an RDBMS that supports this feature.
- Numeric constants, literals, and expressions of literals in the SELECT list.
- The following scalar functions for all queries: DECIMAL, FLOAT, INTEGER, and SUBSTR (or SUBSTRING).
- The following scalar functions for queries that are candidates for Dialect Translation if the RDBMS engine supports the scalar function type: CHAR, DATE, DAY, DAYS, DIGITS, HEX, HOUR, LENGTH, MICROSECOND, MINUTE, MONTH, SECOND, TIME, TIMESTAMP, VALUE, VARGRAPHIC, and YEAR.
- The concatenation operator, ||, or the syntax *alpha1* CONCAT *alpha2* used with literals or alphanumeric columns.
- The following column functions: COUNT, MIN, MAX, SUM, and AVG.

- Date literals of formats YYYY-MM-DD, YYYY/MM/DD, MM-DD-YYYY, and MM/DD/YYYY.
- All requests that contain ANY, SOME, and ALL that do not contain =ALL, <>ANY, and <>SOME.
- =ALL, <>ANY, and <>SOME for requests that are candidates for Dialect Translation if the RDBMS engine supports quantified subqueries.
- The special registers USER, CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP, CURRENT EDASQLVERSION, and CURRENT TIMEZONE.
- IS NULL and IS NOT NULL predicates.
- =NULL and <>NULL predicates.
- LIKE and NOT LIKE predicates.
- IN and NOT IN predicates.
- EXISTS and NOT EXISTS predicates.
- GROUP BY clause expressed using explicit column names.
- ORDER BY clause expressed using explicit column names or column numbers. Note that ORDER BY with UNION supports the column number syntax only.
- FOR FETCH ONLY feature to circumvent record locking.
- Continental Decimal Notation when the CDN variable is set in the server profile.
- National Language Support.
- Outer joins.
- Date arithmetic.

Unsupported SQL Statements and Features

SQL Translation does not support the following:

- More than 15 joins per SELECT.
- Correlated subqueries in DML Generation.
- =ALL, <>ANY, and <>SOME in DML Generation.

Expected Results From SQL Queries

All SQL queries produce Cartesian product style answer sets. In addition, all dates are converted to YYYYMMDD format on output regardless of their original format.

SELECT Statement

Among the most important features of SQL is its ability to query relational data sources and to provide an answer set based on defined criteria. A query can be a:

- SELECT statement that is used in conjunction with the FROM, WHERE, ORDER BY, GROUP BY, and HAVING clauses.
- Number of queries connected by the UNION keyword.
- Number of subqueries that are used within comparison predicates, in predicates, all-or-any predicates and existence tests.

The SELECT statement is the most powerful and complex SQL statement. The SELECT statement can be used along with INSERT, UPDATE, and DELETE to manipulate data and query the tables in a database. SELECT can be used with a number of clauses and qualifiers, such as FROM, WHERE, DISTINCT, or ALL to define how information will be presented and what information will be retrieved.

DISTINCT

Instructs the system to remove duplicates.

ALL

Instructs the system to ignore duplicates when and if they occur.

```
query-spec      ::= SELECT [<sel-qualifier>] <selection> <table-exp>
sel-qualifier   ::= ALL | DISTINCT
selection       ::= <select-list> | *
select-list     ::= <scalar-exp> [, <scalar-exp>]*
table-exp       ::= <from-clause> [<where-clause>] [<group-by-clause>]
                 [<having-clause>]
from-clause     ::= FROM <from-ref> [, <from-ref>]*
where-clause    ::= WHERE <search-condition>
group-by-clause ::= GROUP BY <column-name> [, <column-name>]*
having-clause   ::= HAVING <search-condition>
subquery        ::= (SELECT [ALL | DISTINCT] <result-spec>
<table-expression>)
result-spec     ::= <scalar-exp> | *
```

Example Selecting and Manipulating Values From the Employee Table

```
SELECT DEP_NBR, MAX(SALARY)
FROM EMPLOYEE
GROUP BY DEP_NBR
HAVING MAX(SALARY) > 100000
```

FROM Clause

A FROM clause establishes—conceptually—a candidate set, *S*, from which output tuples are eventually drawn. The FROM clause is defined as:

```

from-ref          ::= <table-definition> [<range-variable>]
table-definition ::= { <table-ref> | <joined-table> | ( <joined-table> ) }
joined-table      ::= <table-definition><join-spec><table-definition>
                  <on-condition>
join-spec         ::= { INNER JOIN | LEFT [OUTER] JOIN | RIGHT [OUTER]
JOIN }
on-condition      ::= ON <search-condition>

```

If the FROM clause consists of only table *T*, then the candidate set *S* is *T*. If the clause is

```
FROM T1, T2, ..., Tn
```

then *S* is defined to be the Cartesian product $T1 \times T2 \times \dots \times Tn$ of the tables. The Cartesian product of $T1 \dots Tn$ is the set of all possible tuples formed by concatenating, or pasting together, a row from *T1* with a row from *T2* with a row from *T3*, ..., and a row from *Tn*.

Outer Joins

A joined table specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

Inner joins can be thought of as the cross product of the tables (combine each row of the left table with every row of the right table), keeping only the rows where the join-condition is true. The result table may be missing rows from either or both of the joined tables. Outer joins include the inner join and preserve these missing rows. There are three types of outer joins:

- Left outer join includes rows from the left table that were missing from the inner join.
- Right outer join includes rows from the right table that were missing from the inner join.
- Full outer join includes rows from both the left and right tables that were missing from the inner join.

Example Defining a Candidate Set as the Join of Component Tables

The FROM clause may also define the candidate set as the “join” of component tables, as shown in the following example:

```
FROM CUSTOMER
FROM CUSTOMER, ORDER, INVENTORY
FROM T1 LEFT OUTER JOIN T2 ON T1.X=T2.Y
```

For this example, the candidate set S is formed as follows. First, the Cartesian product of the two tables (T1 and T2) is formed. Next, the on condition is used to remove those tuples or rows that do not satisfy the condition. If an INNER JOIN was requested, we would have S. Since, in actuality, LEFT OUTER JOIN was requested, we check if all rows of table T1 (the left table) display. For any rows that do not appear, we add a tuple consisting of a row from T1 with a row consisting of null values from T2. If this a RIGHT OUTER JOIN, we would add tuples to ensure that every row of T2 was represented.

If tables T1 and T2 contained the following data:

T1:	<u>column X</u>	T2:	<u>column Y</u>
	1		3
	2		4
	3		5

- T1 INNER JOIN T2 ON T1.X=T2.Y produces a table with only one row corresponding to the tuple <1,1>.
- T1 LEFT OUTER JOIN T2 ON T1.X=T2.Y contains the tuples <1,1>, <2,->, and <3,-> (the dash indicates a null value).
- T1 RIGHT OUTER JOIN T2 ON T1.X=T2.Y has the tuples <1,1>, <-,4>, and <-,5>.

Example Describing Join Table Results as a Component of Another Join

This example shows that the result of a joined table may in turn be a component of another join.

```
FROM (T1 INNER JOIN T2 ON T1.X=T2.Y)
LEFT OUTER JOIN T3 ON T1.X=T3.Z
```

Correlation Names

An identifier that is a correlation name is associated with a table within a particular scope. The scope of a correlation name is either a select statement: single row, subquery, query specification, or is a trigger definition. Scopes may be nested. In different scopes, the same correlation name may be associated with different tables or with the same table.

Example Defining Correlation Names

FROM clauses are also used to define correlation names. A correlation name, R, can assume the value of a row taken from a single specified table, T (R is said to range over T). Consider the following example:

```
FROM CUSTOMER C, PERSONNEL P
```

In this example, C ranges over CUSTOMER and P over PERSONNEL. Whenever a correlation name is introduced in the FROM clause of an SQL subselect, it must be used throughout the request. Thus, in the above example, C should be used in lieu of CUSTOMER within the confines of the select statement containing "FROM CUSTOMER C." Correlation names can be used as abbreviations for long table names. But correlation names offer more than notational convenience.

Example Defining Correlation Names by Joining the Employee Table to Itself

```
SELECT A.EMP_NBR, A.NAME
FROM EMPLOYEE A, EMPLOYEE B
WHERE A.SALARY > B.SALARY AND A.REPORTS_TO = B.EMP_NBR
```

In English, this query might read, "Get the names of all employees earning more than their managers." Here, producing the correct response entails joining the employee table to itself. The correlation names, A and B, are required to distinguish one conceptual copy of the employee data source from another.

There are a number of rules governing the use of correlation names in SQL statements:

- Correlation names must be unique within the subselect, S, in which they are introduced.
- Correlation names cannot duplicate the names of tables referenced in the FROM clause of S.
- Correlation names cannot duplicate the names of columns belonging to tables referenced in the FROM clause of S.
- When a table, T, is referenced more than once in a FROM clause, all but one instance of T must be accompanied by a correlation name.
- When a correlation name, R, is introduced for a table, T, in a subselect, R must qualify all qualified references to columns of T within that same subselect.

**Note**

Catalog and Schema identifiers are permitted, but are ignored by the server Universal Translator. That is, the system treats the following requests as if they were exactly the same because they each contain the same Table name (PERSON).

```
SELECT FIRST_NAME, LAST_NAME
FROM LONDON.SMITH.PERSON
WHERE SEX = 'F'
```

```
SELECT FIRST_NAME, LAST_NAME
FROM BLOGGS.PERSON
WHERE SEX = 'F'
```

```
SELECT FIRST_NAME, LAST_NAME
FROM PERSON
WHERE SEX = 'F'
```

WHERE Clause

The WHERE clause is part of the SELECT statement that specifies a search criteria. A WHERE clause is a search condition preceded by the keyword WHERE.

```
where-clause ::= WHERE <search-condition>
```

The WHERE clause can be qualified when combined with:

- Comparison predicates such as = equal to, < less than, or > greater than.
- Logical operators such as AND, OR, and NOT.
- Between predicates such as BETWEEN and NOT BETWEEN.
- Like predicates such as LIKE and NOT LIKE.
- Null predicates such as IS NULL and IS NOT NULL.
- Any or all predicates such as ALL, SOME, and ANY.

WHERE clauses are governed by two rules:

- Every column identified within a WHERE clause must either belong to S or be a correlation reference.
- Column functions cannot appear in WHERE clauses.

Syntax **How to Specify a Search Condition**

```
where-clause ::= WHERE <search-condition>
```

Example **Specifying a Search Condition**

```
SELECT DEP_NBR, MAX(SALARY)
FROM EMPLOYEE
GROUP BY DEP_NBR
WHERE MAX(SALARY) = 70000
```

GROUP BY Clause

The GROUP BY clause, which follows the WHERE clause in an SQL query, is used to group the elements of a table into sets. This clause is also used to produce a single row of results for each group of rows that have the same values.

Each column reference in the GROUP BY list must belong to a subset and must refer unambiguously to a column specified in the SELECT list. Columns within views whose values are derived by means of expressions, functions, or constants must never display in a GROUP BY list.

GROUP BY partitions a set into subsets, S1 .. Sn. Each subset corresponds to a fixed set of values assumed by the grouping columns. (Null values are considered to be equal for the purpose of forming groups.)

When GROUP BY is present, the elements in the select list must be single-valued by group. Select list elements can be aggregate functions, they can be columns referenced in the group by list (which by definition are single-valued by group), or they can be expressions. They must not assume multiple values for any group to which they belong.

The GROUP BY clause can be used with:

- Aggregates to provide summary values for each set.
- WHERE to locate the information specified and to group that information into sets.
- ORDER BY to sort the answer set into ascending (ASC) or descending (DSC) order:
 - ASC sorts the answer set in ascending order from the lowest to the highest.
 - DSC sorts the answer set in descending order from the highest to the lowest.
- HAVING to limit the number of rows returned within a group.

Syntax

How to Group Table Elements

```
group-by-clause ::= GROUP BY <column-name> [, <column-name>]*
```

where:

GROUP-BY

Groups the elements of a subset together to form a new intermediate result.

Example

Producing a Single Row for Each Department Number in the Employee Table

```
SELECT DEP_NBR, MAX(SALARY)
FROM EMPLOYEE
GROUP BY DEP_NBR
```

HAVING Clause

The HAVING clause is similar to the WHERE clause. WHERE eliminates rows, and the HAVING clause eliminates groups. The HAVING clause and WHERE clause have similar syntax. They both consist of a keyword followed by a search condition.

The HAVING clause applies qualifying conditions to groups after they have been formed. The HAVING clause complements the GROUP BY clause. The HAVING clause allows you to include column functions such as, COUNT, AVG, MAX, MIN, or SUM in the search condition. Each HAVING compares one column or column function expression of the group with another column function expression of the group or with a constant.

HAVING can be used with the logical operators ADD, OR, or NOT to combine conditions.

Expressions in the HAVING clause must be single-valued per group. Each column name in the search condition must reference a grouping column or be a correlated reference.

```
having-clause ::= HAVING <search-condition>
```

Example Specifying a Qualifying Condition

```
SELECT DEP_NBR, MAX(SALARY)
FROM EMPLOYEE
GROUP BY DEP_NBR
HAVING MAX(SALARY) > 100000
```

AS Clause

The AS clause makes it possible to rename existing column titles in your reports.

```
as-clause ::= AS <column-name>
```

Syntax How to Change Default Titles

```
SELECT field1 AS title1, field2 AS title2
```

where:

field

Can be a sort field, display field, column total, or row total.

title

Is the new column title lines.

Aggregate Functions

The SQL Translator provides five aggregate functions: COUNT, SUM, AVG, MAX, and MIN. These aggregate functions can be used with a:

- GROUP BY clause to organize the rows returned by a FROM clause.

- HAVING clause to eliminate sets that do not meet the selection criteria.

Each is represented by a name followed by a parenthetical expression containing a single argument. Arguments designate result set columns and can be preceded by the keyword DISTINCT to remove duplicates.

Aggregate functions must be coded as follows:

```
function-ref      ::= COUNT(*) | <distinct-fn-ref> | <all-fn-ref>
distinct-fn-ref  ::= [<fn-name> | COUNT] (DISTINCT <column-ref>)
all-fn-ref       ::= <fn-name> ( [ALL] <scalar-exp> )
fn-name          ::= AVG | MAX | MIN | SUM
```

Note that COUNT(*) is treated as a special case. COUNT(*) counts the number of rows in a table without eliminating duplicates.

The keyword DISTINCT can be used with any aggregate except COUNT(*).

Example Producing a Count From the Condition “CITY=Denver”

```
SELECT COUNT(*) FROM CLIENT WHERE CITY = 'Denver'
```

This would generate a single output element (a table consisting of one row and one column). The result would be produced by counting every row that satisfies the CITY = 'Denver' predicate. It is significant that each row participates in the count. Among the SQL functions, COUNT(*) is unique in that it does not reject an operand because it is null, or because it contains a null value.

Reference Function Names

The following functions operate on the set of non-null scalar values found within one column of a result table. All of these produce single-valued output. A list of function names and their meanings follow:

COUNT ([DISTINCT])	The number of (distinct) non-null elements in the column.
SUM ([DISTINCT])	The sum of the (distinct) elements in the column. This function can only be used with numeric columns.
AVG ([DISTINCT])	The average of the (distinct) elements in the column. This function can only be used with numeric columns.
COUNT	The number of non-null elements in the column.
SUM	The sum of the non-null elements in the column. This function can only be used with numeric columns.
AVG	The average of the non-null elements in the column. This function can only be used with numeric columns.

MAX	The maximum value in the column.
MIN	The minimum value in the column.

Subqueries

Subqueries, which access single-column tables from the database, are used within comparison predicates, in predicates, all-or-any predicates, and existence tests. A subquery is a SELECT statement that nests:

- Inside the WHERE, HAVING, or SELECT clause of another SELECT.
- Inside an UPDATE, DELETE, or INSERT statement.
- Inside another subquery.

```
subquery ::= (SELECT [ALL | DISTINCT] <result-spec>
<table-expression>)
result-spec ::= <scalar-exp> | *
```

Example Using Subqueries

```
SELECT ITEM FROM INVENTORY WHERE UNIT_COST <
    (SELECT AVG(UNIT_COST) FROM INVENTORY)
```

```
SELECT ITEM FROM INVENTORY WHERE UNIT_COST < ALL
    (SELECT UNIT_COST FROM INVENTORY
    WHERE ITEM_CODE LIKE 'X%')
```

```
SELECT ITEM, DESCRIPTION FROM ORDERS
WHERE ITEM_CODE NOT IN
    (SELECT ITEM_CODE FROM INVENTORY)
```

```
SELECT A.ITEM, A.DESCRPTION FROM ORDERS A
WHERE EXISTS
    (SELECT * FROM INVENTORY B
    WHERE A.ITEM_CODE = B.ITEM_CODE
    AND B.QTY_ON_HAND > 1000)
```

Subselects used within comparison predicates are required to return a single row of output. Subselects used within quantified predicates must return a single column of values, S. If S is empty or if the given comparison is true for all values of S, then $x \langle \text{relop} \rangle \text{ALL } S$ is true; it is false otherwise. If $x \langle \text{relop} \rangle S$ is true for at least one value of S, $x \langle \text{relop} \rangle \text{ANY } S$ is true.

Correlated Subqueries

A correlated subquery is a subquery expressed in terms of a value produced by an outer query. Consider, for example, the fourth SELECT statement in the preceding *Using Subqueries* example. The subselect in the EXISTS test contains the following comparison:

```
A.ITEM_CODE = B.ITEM_CODE
```

A.ITEM comes from the outer select; B.ITEM comes from the inner select. To develop an answer set for this query, the SQL Translator must—conceptually—evaluate the inner query once for every value of A.ITEM produced by the outer select.

Generating a Result Set

A join is a query that:

- Contains multiple table names, T1 .. Tn, in the FROM list.
- Specifies a link between T1 .. Tn in the WHERE clause.

Example Establishing a Join Condition of FACULTY and STUDENT

```
SELECT F.NAME, S.NAME
FROM FACULTY F, STUDENTS S
WHERE S.ADVISOR = F.EMP_NBR AND F.DEPT = 'Mathematics'
ORDER BY F.NAME
```

Conceptually, the SQL Translator evaluates this query in a series of steps:

- It forms a candidate set S1, the Cartesian product of FACULTY and STUDENT by pasting together all possible combinations of rows taken from the two tables. (If there were N students and M faculty members on file, S1 would consist of N*M rows. If there were X columns in FACULTY and Y columns in STUDENTS, S1 would contain X+Y columns.)
- The SQL Translator then identifies those rows of S1 which conform to the join condition S.ADVISOR = F.EMP_NBR. This would result in a (possibly) reduced candidate set, S2.
- The system further restricts S2 to those rows for which F.DEPT = MATHEMATICS. This produces a third candidate set, S3.
- Finally, the SQL Translator eliminates all columns not named in the SELECT list. The resulting set, known as a projection, would be returned to the user.

A production quality DBMS would never proceed in this manner because the overhead caused by the size of S1 would preclude that. Still, it is instructive to think of the query execution as taking place in this fashion.

The predicate, S.ADVISOR = F.EMP_NBR, is known as a join condition. Although, in practice, most join operations are based on equal comparisons (and are therefore called equijoins), the SQL Translator supports joins based on other valid SQL relational operators.

ORDER BY

The ORDER BY clause is used to sort the answer set by the values contained in one or more columns.

```
order-by-clause ::= ORDER BY <ord-spec> [, <ord-spec>]*
ord-spec       ::= {<integer> | <column-ref>} [{ASC | DSC}]
```

Example **Sorting a Result Set From Employee Table**

```
SELECT DEP_NBR, SALARY
FROM EMPLOYEE
ORDER BY DEP_NBR, SALARY
```

Scalar Functions

Scalar functions represent scalar values. The syntax is

```
scalar-exp ::= <term> | <scalar-exp> <addop> <term>
addop      ::= + | -
term       ::= <factor> | <term> <multop> <term>
multop     ::= * | /
factor     ::= <literal> | <column-ref> | <function-ref> |
              (<scalar-exp>)
```

Example **Specifying a Scalar Expression**

```
COUNT(DISTINCT EMP_NBR)
3 * (AMOUNT + BALANCE)
```

Column Functions

The SQL Translator provides column functions that can be used to summarize data: COUNT, SUM, AVG, MAX, and MIN. Each function is represented by a name followed by a parenthetical expression containing a single argument. Arguments designate result set columns and can be preceded by the keyword DISTINCT to remove duplicates. Column functions must be coded as follows:

```
function-ref ::= COUNT(*) | <distinct-fn-ref> | <all-fn-ref>
distinct-fn-ref ::= <fn-name> (DISTINCT <column-ref>)
all-fn-ref      ::= <fn-name> ( [ALL] <scalar-exp> )
fn-name        ::= AVG | MAX | MIN | SUM | COUNT
```

Note that COUNT(*) is treated as a special case in the definition of function-ref. COUNT(*) counts the number of rows in a table without eliminating duplicates.

The functions operate on the set of non-null scalar values found within one column of a table. All of them produce single-valued output.

Reference **Function Names**

The following table lists and defines function names.

COUNT	The number of non-null elements in the column.
SUM	The sum of the non-null elements in the column. SUM requires numeric input.
AVG	The average of the non-null elements in the column. This function requires numeric input.
MAX	The maximum value in the column.
MIN	The minimum value in the column.

Example **Counting the Number of Rows Including Duplicates**

This query would generate a single output row containing the count of every row that satisfies the CITY = 'Denver' predicate. It is significant that each row participates in the count. Among the SQL functions, COUNT(*) is unique in that it does not reject an operand because it is null or because it contains a null value.

```
SELECT COUNT(*) FROM CLIENT WHERE CITY = 'Denver'
```

Example **Counting the Number of Rows Excluding Duplicates**

This query, by ignoring duplicates, produces a true count of customers on file.

```
SELECT COUNT(DISTINCT CUST_NBR) FROM CUSTOMER
```

Example **Applying a Column Function to a Group of Values**

This query illustrates how column functions can be applied to groups of values (GROUP BY).

```
SELECT DEPARTMENT, SUM(SALARY), MAX(SALARY), MIN(SALARY)
FROM PERSONNEL
GROUP BY DEPARTMENT
```

Example **Using a Column Function Within a Subquery**

This query illustrates the use of a column function within a subquery.

```
SELECT MAKE, MODEL FROM CAR
WHERE DEALER_COST >
  (SELECT AVG(DEALER_COST) FROM CAR)
```

If a quantifier is not specified, then ALL is implicit. If SUM or AVG is specified, a character string, bit string, or datetime cannot be used.

Predicates

Predicates, which evaluate to true or false, can be applied to a row or a group of rows in an answer set, and display in data manipulation statements as well as in queries. There are seven types of predicates in SQL: comparison predicates, between predicates, like predicates, null predicates, in predicates, quantified predicates, and existence predicates.

Comparison Predicates

A comparison predicate compares two values—a scalar expression, and either some other scalar expression or a subquery result.

```
comparison-predicate ::= <scalar-exp> <relop> <comp-tail>
comp-tail             ::= <scalar-exp> | <subquery>
relop                 ::= = | <> | < | > | <= | >=
```

Example Comparing Values

```
SALARY > THRESHOLD
DEALER_COST < (SELECT AVG(DEALER_COST) FROM CAR)
DEPARTMENT = 100
```

Between Predicates

A between predicate is true if the value is greater than or equal to the first limit and less than or equal to the second limit. A between predicate specifies a range comparison.

```
between-predicate ::= <scalar-exp> [NOT] BETWEEN <limits>
limits             ::= <scalar-exp> AND <scalar-exp>
```

Example Specifying a Range

```
SPEED NOT BETWEEN LEGAL_LIMIT AND 70
PRICE BETWEEN GOING_RATE AND 1.1 * GOING_RATE
```

Like Predicates

A like predicate is used to isolate string values that conform to a given pattern.

```
like-predicate ::= <column-ref> [NOT] LIKE <literal>
```

A like predicate is true for a given row whenever the specified column (which must be of type string) *matches* a particular pattern. Patterns are expressed as literal strings composed of printable characters. Special symbols (“_” and “%”) are used to represent arbitrary characters and character strings. To be specific, an underscore represents a single character and the percent sign represents any sequence of zero or more characters. All other characters represent themselves.

Example Isolating String Values

```
LAST_NAME LIKE 'SM_TH'
CITY LIKE '%TON'
LAST_NAME NOT LIKE 'JONES%'
```

The first predicate would be true for such values as SMITH and SMYTH but not for JONES. The second would be true for TRENTON, PRINCETON, SCRANTON, and TON but not for PHILADELPHIA or AKRON. The third predicate excludes all rows that begin with JONES.

Since the SQL Translator permits the use of the backslash as an escape character, patterns containing control symbols can be expressed as string literals.

Example Full Support for ESCAPE Sequence

To search for rows containing isolated embedded escape characters, issue a query employing the following predicate:

```
SQL
SELECT * FROM ESC WHERE FLD01 LIKE '@%%' ESCAPE '@';
END
```

Null Predicates

A null predicate tests for a null value.

A null predicate is NULL if the referenced column of a given row contains a null value. A null predicate is NOT NULL if the referenced column of a given row contains a non-null value.

```
null-predicate ::= <column-ref> IS [NOT] NULL
```

Example Testing for a Null Value

```
LAST_NAME IS NOT NULL
ACCOUNT_ID IS NULL
```

In Predicates

An in predicate is true if the specified value, v, is a member of a set S. S can be defined by a static enumeration or by a subquery.

```
in-predicate ::= <scalar-exp> [NOT] IN <value-set>
value-set    ::= <subquery> | <enumeration>
enumeration  ::= ( <literal> [,<literal>]* )
```

Example Testing Whether a Value Is a Member of a Set

```
MAKE_OF_CAR IN ('CHEVROLET', 'FORD', 'PLYMOUTH')
ACCOUNT NOT IN (SELECT ACCT_NUMBER FROM CUSTOMER WHERE STATUS =
'DEADBEAT')
```

Quantified Predicates

Quantified predicates, which resemble the comparison predicates, produce a true result when all (any) of the values generated by a given subquery satisfy the stipulated comparison. Although three quantifiers—ALL, ANY, and SOME—are permitted, ANY and SOME are synonymous.

```
all-or-any-pred ::= <scalar-exp> <quant-op> <subquery>
quant-op       ::= <relop> <quantifier>
quantifier     ::= ALL | ANY | SOME
```

Example Generating Values That Satisfy a Specified Comparison

```
SOFTWARE > ALL (SELECT SOFTWARE FROM PRODUCTS WHERE PROD-NAME = 'FOCUS')
RATING > ANY (SELECT RATING FROM PRODUCTS WHERE COST > 5.00)
```

The first predicate searches for FOCUS, alone or in a longer string, such as FOCUS Six for Windows. The second is true for any RATING exceeding that of at least one \$5.00 product on file.

Existence Predicates

An existence predicate produces a true result if the given subquery has a significant (for example, non-null) answer set.

```
existence-predicate ::= [NOT] EXISTS <subquery>
```

Example Testing Whether a Subquery Has a Significant Answer Set

```
EXISTS (SELECT * FROM CANDIDATES WHERE EYE_COLOR = 'Blue')
```

Search Conditions

Predicates linked together by logical operators—AND, OR, and NOT—form search conditions.

```
search-condition ::= <boolean-term> |
                  <search-condition> OR <boolean-term>
boolean-term     ::= <boolean-factor> |
                  <boolean-term> AND <boolean-factor>
boolean-factor   ::= [NOT] <boolean-primary>
boolean-primary  ::= <predicate> | ( <search-condition> )
```

Search conditions are evaluated by systematically applying logical operators to the results generated by their constituent predicates. If no logical operators are present, a search condition is nothing more than a predicate and is evaluated accordingly. When logical operators are present, they are applied in a specific order: NOT before AND, AND before OR. Parenthesized search conditions are evaluated before logical external operators are taken into consideration.

Example Specifying a Search Condition

```
HEIGHT > 6 AND WEIGHT < 150 OR WEIGHT > 200
HEIGHT > 6 AND (WEIGHT < 150 OR WEIGHT > 200)
```

The first search condition produces a true result either for six footers weighing less than 150 pounds or for candidates weighing more than 200 pounds regardless of height. The second search condition would be true for any six footer weighing less than 150 pounds or more than 200 pounds. The parentheses in the second example reverse the implicit order in which the logical operators are applied.

Data Manipulation Commands

The server provides three basic commands for manipulating or modifying data:

- The INSERT statement introduces new rows into an existing table. Insertion can be accomplished one row at a time (insert-row) or in a block (insert-set).
- The DELETE statement removes a row or combination of rows from a table.
- The UPDATE statement permits the user to update a row or group of rows in a table.

When a WHERE clause is used in conjunction with an INSERT, DELETE, or UPDATE, it describes what is known as an SQL searched statement.

- In a searched update, every row that meets the search criteria is updated.
- In a searched delete, every row satisfying the condition specified in the WHERE clause is removed from the specified table.
- In a searched insert, a row is inserted in the column that satisfies the condition specified in the WHERE clause.

INSERT

Inserts rows or values into an existing table. The insertion of rows can be accomplished one row at a time (insert-row) or in a block (insert-set). You can insert rows or values into a table with a VALUE keyword or with a SELECT statement.

```
insert-statement ::= <insert-row> | <insert-set>
insert-set      ::= <insert-head> <query-spec>
insert-row      ::= <insert-head> [( <column-list> )] VALUES
                    (<val-list>)
insert-head     ::= INSERT INTO <table-name>
val-list        ::= <literal> [, <literal>]*
```

Syntax How to INSERT Values Into the Employee Table

```
INSERT INTO <table-name> VALUES (literal1 [, literal2]
[, literal3]....)
```

where:

table-name

Is the table name.

literal

Is a non-null value or constant: a specific unchangeable value that satisfies the constraints of its data type.

Example Inserting Values Into the Employee Table

```
INSERT INTO EMPLOYEE VALUES  
'12345678','Joseph','Bloggs','123 Main Street','New York',  
'NY','G','M','212 321-4561'
```

DELETE

Removes a row or a combination of rows, based on the search criteria specified.

```
delete-statement: searched ::= DELETE FROM <table-name> [Where  
<search condition>]
```

where:

table-name

Is the name of the table that contains the data you want to delete.

Where

Specifies the search criteria that must be met in order for the deletion to take place.

If a search condition is not specified, then all rows of the table are marked for deletion. If a search condition is specified, only those rows that meet the search criteria are marked for deletion.

Example Deleting Rows in the Employee Table

The following statement deletes all rows in the Employee table whose company name is Ragmount.

```
DELETE FROM EMPLOYEE WHERE company = 'Ragmount'
```

UPDATE

Changes the contents of a row or a group of rows in a table.

```
update-statement ::= UPDATE <table-name>  
                  SET <asg-list> [<where-clause>]  
asg-list          ::= <asg-element> [, <asg-element>]*  
asg-element      ::= <column-name> = {<scalar-expression> | NULL }
```

A WHERE clause is typically used with the UPDATE command to define which rows should be modified. Subqueries can also be included in the WHERE.

Example **Changing the Parts Row**

```
UPDATE PARTS
SET PRICE = 1.1 * PRICE
WHERE Partnum='R467Z'
```

Commit and Rollback

A commit terminates the current transaction normally. A rollback undoes all the changes to the data source initiated by the current transaction. Both commands remove all prepared SQL requests from the environment.

COMMIT

Terminates the current transaction normally.

```
commit-statement ::= COMMIT WORK
```

Example **Terminating the Current Transaction Normally**

```
COMMIT WORK
```

ROLLBACK

Undoes all the changes to the data source initiated by the current transaction.

```
rollback-statement ::= ROLLBACK WORK
```

Example **Undoing Changes**

```
ROLLBACK WORK
```

CHAPTER 5

Tables and Views

Topics:

- Creating Tables
- Creating Views

This topic describes how to build tables using the CREATE TABLE command. These tables consist of data types, null values, column names and aliases. In addition, this topic also describes how to create views using the CREATE VIEW command.

Creating Tables

A table is a rectangular array of data elements that can be viewed as either a collection of rows or a collection of columns. Every table is managed by a DBMS and accessed using the server.

The columns of a table have an inherent order; its rows do not. Every table must have at least one column but the number of rows can be zero. A table with no rows is said to be empty. If R is the number of rows in a table, and C is the number of columns, each column would have R values, and each row would consist of a sequence of C values arranged in column order.

CREATE is used to bring transient tables or views into existence, INSERT is used to insert a row or a block of rows into a table or view, and DROP is used to explicitly remove transient tables and views from the environment. Since the lifetime of a transient table is limited by the duration of a user session (that period of time during which the application maintains an active connection with the server), DROP is seldom required.

Data Types

Each column of a table has an associated data type. A data type determines the range of values (domain) that can be assigned to the elements of a column and the manner in which the elements can be manipulated and displayed.

Null Values

Columns can be created to contain null values. When a data element is null, we can infer that either a value has not been assigned to it, its value is unknown, or the column is nonapplicable. The existence of null values impacts system behavior in various ways. Comparisons involving null values, for example, are always false (except comparisons using the null predicate). Column functions, like SUM, operating on particular columns, ignore null values. But COUNT(*) counts every row, even if a row contains null values.

Column Names and Aliases

Every column of a table has a name. The name must begin with an alphabetic character (a-z). The rest of the name can contain numbers and the following special characters only:

\$
@
.
-

CREATE TABLE

CREATE TABLE creates a new database table

```

CREATE TABLE <table-name> (<column-definition> [,<column- definition>]*)
                                [<default-clause>] [<constraint>]
constraint                       ::= NOT NULL [UNIQUE]
default-clause                   ::= DEFAULT {<literal> | NULL}
default                           ::= <literal> | NULL
column-definition                ::= <column-name> <data-type>
data-type                        ::= <string-data-type> | <numeric-data-type>
numeric-data-type                ::= NUMERIC[<pr-sc>] | {DECIMAL | DEC} [<pr-sc>]
                                | INTEGER | INT | SMALLINT | FLOAT | REAL | DOUBLE PRECISION
pr-sc                             ::= (<precision>[,<scale>])

```

where:

table-name

Is the name of the table to be created. The length and format of the table-name must comply with standard SQL requirements.

column-name

Is the name of a column to be created. The length and format of the column-name must comply with standard SQL requirements. The maximum number of columns permitted in one CREATE TABLE is 254.

data-type

Is the data type of the column. Possible values are:

CHAR(<i>n</i>)	Fixed-length alphanumeric, where <i>n</i> is less than 254.
DATE	Date data types are used to store calendar dates.
INTEGER	Four-byte binary integer.
DECIMAL(<i>p, s</i>)	Packed decimal containing <i>p</i> digits with an implied number <i>s</i> of decimal points.
REAL	Four-byte, single-precision floating point.
FLOAT	Eight-byte, double-precision floating point.

The server creates a table when it receives a CREATE TABLE request. It requires that the table name be unique (for example, that it not conflict with the name of an existing table) and that it conform to the lexical constraints imposed on short identifiers (for example, that it consist of the number of characters supported, which typically is eight or less characters).

The number of possible columns in a table is 254. Any server data type can be assigned to a given column.

Example Creating a New Database Table

```
CREATE TABLE PERSON
(
  PID CHAR(8) NOT NULL,
  FIRST_NAME CHAR(20),
  LAST_NAME CHAR(20),
  ADDRESS CHAR(40),
  CITY CHAR(40),
  STATE CHAR(2),
  DOB,
  DATE,
  FLOOR,
  INT,
  SALARY,
  DECIMAL,
)
```

Creating Views

A view is a transient object that inherits most of the characteristics of a table. Like a table, a view is composed of rows and columns. The columns have data types and may, on occasion, contain null values. Views may be queried as if they were tables. But, unlike tables, views are dependent entities. Views are defined by selecting and pasting together elements from existing tables. When a view, V, is referenced in an SQL request, the server creates the illusion that V is a table by associating the appropriate data with it. You may not insert, update, or delete from an SQL view.

In the server, a view does not persist beyond the close of the session in which it is created. A session, for an individual connection, begins when an initial connection is established and continues until no remaining active connections exist.

The server allows you to create one or more views based on the information that is contained within a table, and to delete or drop the views created. The CREATE verb is used to create views and DROP is used to explicitly remove views from the environment.

CREATE VIEW

CREATE VIEW permits the creation of a view based on the fields contained in a table.

```
CREATE VIEW <view-name> [<column-list>] AS <query-specification>
column-list      ::= <column-name> [, <column-name>]
query-exp        ::= <query-exp-head> <order-by-clause>
query-exp-head   ::= <query-term> | <query-exp> UNION [ALL] <query-term>
order-by-clause ::= ORDER BY <ord-spec> [, <ord-spec>]*
ord-spec         ::= {<integer> | <column-ref>} [{ASC | DSC}]
query-term       ::= <query-spec> | ( <query-exp> )
query-spec       ::= SELECT [<sel-qualifier>] <selection> <table-exp>
```

Example **Creating a View**

```
CREATE VIEW person_view AS SELECT first_name, last_name FROM person
```

where:

person_view

Is the name of the view.

first_name

Is a column name.

last_name

Is a column name.

person

Is a table name.

CHAPTER 6

Preparing and Executing SQL Requests

Topics:

- Preparing SQL Requests
- Executing SQL Requests

These topics describe how to use PREPARE to submit parameterized SQL requests for subsequent execution using EXECUTE.

Preparing SQL Requests

The PREPARE statement permits you to submit parameterized SQL requests for subsequent execution using the EXECUTE verb. PREPARE contains a statement name and a statement prototype. A statement name must not conflict with any other user-defined object, and a statement prototype is an SQL request with possible parameter markers (question marks) coded in place of a literal.

The SQL Translator checks the prototype statement for correct syntax. If it finds no violations, it retains the statement for future use and returns an answer set description and a parameter count to the application (see the EDAPREPARE specification in the *API Reference* manual for a discussion of how to access this information). Prepared requests are not shareable, nor are they held permanently. Statement prototypes are removed from the environment at the close of an end user session or upon receipt of a COMMIT or ROLLBACK request. The server provides no other means for removing, retaining, or maintaining statement prototypes.

```
prepare-statement ::= PREPARE <stmt-name> FROM <prototype>
prototype         ::= <query-exp> | <delete-statement> |
                    <insert-statement> | <update-statement>
```

Example Preparing an SQL Request

```
PREPARE MY_QUERY FROM
  SELECT ENAME, DEPARTMENT
  FROM EMPLOYEE
  WHERE SALARY > ?
PREPARE NEW_CUST FROM
  INSERT INTO CUST(COMPANY, CNBR, SALESREP)
  VALUES (?, ?, ?)
```

Executing SQL Requests

The EXECUTE statement is used to invoke a prepared request by name.

When it receives an EXECUTE command, the SQL Translator locates the named statement, *S*, inserts parameter values (if any) into the text of *S*, reparses *S*, and, finally, executes *S*. EXECUTE results in a diagnostic message if:

- The SQL Translator is unable to locate a request with the given name.
- The number of parameters submitted differs from the number of parameter markers in the stored request.
- The parameter insertion process results in an illegal SQL statement.

```
execute-statement ::= EXECUTE <stmt-name> [USING <parmlist>]
parmlist          ::= <literal> [, <literal>]*
```

Example Invoking a Prepared Request

```
EXECUTE MY_QUERY USING 30000  
EXECUTE NEW_CUST USING "NEWCO", 10247, "BLOGGS"
```

While SQL imposes a limit of one parameter list per EXECUTE request, the API permits application programs to submit multiple parameter lists per EDAEXECUTE call (see the *API Reference* manual). This apparent contradiction is easily explained: the API recasts an EDAEXECUTE request with a batch of parameter lists into a batch of EXECUTE requests, each having a single parameter list.

APPENDIX A

SQL Translate Keywords

Topic:

- Keywords

This topic provides a list of the SQL Translator reserved words.

Keywords

Keywords have special meaning in SQL and may not be used as identifiers. Although some entries in the following list are not part of the server SQL language dialect, the server SQL Universal Translator treats them as keywords. This is for future expansion and to help diagnose certain kinds of syntax errors.

ALIAS	DECIMAL	JOIN	RAW
ALTER	DECLARE	KEY	REAL
ALL	DELETE	LEADING	REFERENCE
AND	DESC	LEFT	RIGHT
ANY	DISTINCT	LIKE	ROLLBACK
AS	DOUBLE	LOGICAL	SELECT
ASC	DROP	LONG	SERIAL
AVG	ELSEEND	MAX	SET
BETWEEN	ESCAPE	MIN	SMALLFLOAT
BEGIN	EXCEPT	MONEY	SMALLINT
BIND	EXECUTE	NOT	SOME
BIT	EXISTS	NULL	SQLID
BOTH	FETCH	NULLIF	SUM
BY	FILE	NUMBER	SYSNAME
CASE	FLOAT	NUMERIC	TABLE
CHAR	FOR	OF	TEXT
CHARACTER	FROM	OFF	TIME
CHECK	GRAPHIC	OPEN	TIMESTAMP
CLOSE	GROUP	ON	TIMEZONE
COALESCE	HAVING	ONLY	TO
COLFETCH	IMAGE	OPTION	USER
COMMIT	IN	OR	UNION
CONNECT	INNER	ORDER	UNIQUE
COUNT	INSERT	OUTER	USER_TYPE_NAME
CREATE	INT	PRECISION	USING
CURRENT	INTEGER	PRIMARY	UPDATE
DATABASE	INTERSECT INTO		
DATE	IS		
DATETIME			
DEC			

APPENDIX B

BNF Summary

Topic:

- Condensed SQL Language Syntax Definition

This topic provides the Backus Naur Form (BNF) notation to describe SQL syntax.

Condensed SQL Language Syntax Definition

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alpha ::= A..Z | a..z

special-character ::=
A special-character is any character other than a <digit> or an <alpha>.

delimiter ::= <alpha-literal> | , | ( | ) | < | > | . | : | = | * | + |
- | / | <> | >= | <= |

separator ::= <comment> | <space> | <newline>
comment ::= --[<character>]*<newline>

lit-character ::= <nonquote> | ''
nonquote ::=
A nonquote is any character other than the single quote "'". Two
consecutive quotes "" may be used to represent an embedded quote "'".

integer ::= <digit>[<digit>]*

numeric-literal ::= [{+ | -}]<numeric-tail>
numeric-tail ::= <integer>[.<integer>] | .<integer>
approx-literal ::= <numeric-literal> E[+ | -] <integer>
alpha-literal ::= '[<lit-character>]*'

literal ::= <alpha-literal> | <numeric-literal> | <approx-literal>

identifier ::= <normal-identifier> | <delimited-identifier>
normal-identifier ::= <alpha>[<id-character>]*
delimited-identifier ::= "<any printable character string>"

id-character ::= <alpha> | <digit> | _

authorization-id ::= <short identifier>

location-id ::= <long identifier>

column-name ::= <long identifier>

table-name ::= <short identifier>

view-name ::= <long identifier>

statement-name ::= <short identifier>

range-variable ::= <long identifier>

table-ref ::= [<table-qualifier>.]<table-name>
table-qualifier ::= <loc-id>.<auth-id> | <auth-id>

column-ref ::= [<column-qualifier>.]<column-name>
column-qualifier ::= <table-ref> | <range-variable>

string-data-type ::= {CHARACTER | CHAR}[(<length>)]
length ::= <integer>
```

```

numeric-data-type ::= NUMERIC[<pr-sc>] | {DECIMAL | DEC} [<pr-sc>] |
INTEGER | INT | SMALLINT | FLOAT | REAL | DOUBLE PRECISION
pr-sc ::= (<precision>[,<scale>])

date-data-type ::= DATE

scalar-exp ::= <term> | <scalar-exp> <addop> <term>
addop ::= + | -

term ::= <factor> | <term> <multop> <term>
multop ::= * | /
factor ::= <literal> | <column-ref> | <function-ref> | (<scalar-exp>)

comp-tail ::= <scalar-exp> | <subquery>

relop ::= = | <> | < | > | <= | >=

between-predicate ::= <scalar-exp> [NOT] BETWEEN <limits>
limits ::= <scalar-exp> AND <scalar-exp>

like-predicate ::= <column-ref> [NOT] LIKE <literal>

null-predicate ::= <column-ref> IS [NOT] NULL

in-predicate ::= <scalar-exp> [NOT] IN <value-set>
value-set ::= <subquery> | <enumeration>
enumeration ::= ( <scalar-exp> [,<scalar-exp>]* )

all-or-any-pred ::= <scalar-exp> <quant-op> <subquery>
quant-op ::= <relop> <quantifier>
quantifier ::= ALL | ANY | SOME

existence-predicate ::= EXISTS <subquery>

search-condition ::= <boolean-term> | <search-condition> OR
<boolean-term>
boolean-term ::= <boolean-factor> | <boolean-term> AND
<boolean-factor>
boolean-factor ::= [NOT] <boolean-primary>
boolean-primary ::= <predicate> | ( <search-condition> )

function-ref ::= COUNT(*) | <distinct-fn-ref> | <all-fn-ref>
distinct-fn-ref ::= [<fn-name> | COUNT] ( DISTINCT <column-ref> )
all-fn-ref ::= <fn-name> ( [ALL] <scalar-exp> )
fn-name ::= AVG | MAX | MIN | SUM

```

```

query-exp          ::= <query-exp-head> <order-by-clause>
query-exp-head    ::= <query-term> | <query-exp> UNION [ALL] <query-term>
order-by-clause   ::= ORDER BY <ord-spec> [, <ord-spec>]*
ord-spec          ::= {<integer> | <column-ref>} [{ASC | DSC}]
query-term        ::= <query-spec> | ( <query-exp> )
query-spec        ::= SELECT [<sel-qualifier>] <selection> <table-exp>
sel-qualifier     ::= ALL | DISTINCT
selection         ::= <select-list> | *
select-list       ::= <scalar-exp> [, <scalar-exp>]*

table-exp         ::= <from-clause> [<where-clause>] [<group-by-clause>]
                   [<having-clause>]
from-clause       ::= FROM <from-ref> [, <from-ref>]*
from-ref          ::= <table-definition> [<range-variable>]
table-definition  ::= { <table-ref> | <joined-table> | ( <joined-table> ) }
joined-table      ::=
<table-definition><join-spec><table-definition><on-condition>
join-spec         ::= { INNER JOIN | LEFT [OUTER] JOIN | RIGHT [OUTER] JOIN
}
on-condition      ::= ON <search-condition>
where-clause      ::= WHERE <search-condition>
group-by-clause  ::= GROUP BY <column-name> [, <column-name>]*
having-clause     ::= HAVING <search-condition>

subquery          ::= (SELECT [ALL | DISTINCT] <result-spec>
<table-expression>)
result-spec       ::= <scalar-exp> | *

prepare-statement ::= PREPARE <stmt-name> FROM <prototype>
prototype         ::= <query-exp> | <delete-statement> |
<insert-statement> |
                   <update-statement>

execute-statement ::= EXECUTE <stmt-name> [USING <parmlist>]
parmlist          ::= <literal> [, <literal>]*

table-definition  ::= CREATE TABLE <table-name>
                   (<column-definition> [,<column-definition>]*
                   [<default-clause>] [<constraint>])
constraint        ::= NOT NULL [UNIQUE]
default-clause    ::= DEFAULT <default>
default           ::= <literal> | NULL
column-definition ::= <column-name> <data-type>
data-type         ::= <string-data-type> | <numeric-data-type>

view-definition   ::= CREATE VIEW <view-name> [<column-list>] AS
                   <query-specification>
column-list       ::= <column-name> [, <column-name>]
drop-statement    ::= DROP {<table-name> | <view-name>}

```

```
insert-statement ::= <insert-row> | <insert-set>
insert-set      ::= <insert-head> <query-spec>
insert-row      ::= <insert-head> [<column-list>] VALUES <val-list>
insert-head     ::= INSERT INTO <table-name>
val-list        ::= <literal> [, <literal>]*

delete-statement ::= DELETE FROM <table-name> [<where-clause>]

update-statement ::= UPDATE <table-name> SET <asg-list> [<where-clause>]
asg-element      ::= <column-name> = {<scalar-expression> | NULL }
asg-list         ::= <asg-element>[, <asg-element>]*

commit-statement ::= COMMIT WORK

rollback-statement ::= ROLLBACK WORK
```

Index

A

aggregate functions 4-11
ALL predicate 4-18
alphabetic characters in SQL 3-3
answer sets in SQL 4-3
 generating 4-13
 sorting 4-14
ANY predicate 4-18
APT (Automatic Passthru) 2-11–2-12
 Dialect Translation and 2-13
AS clause 4-10
Automatic Passthru (APT) 2-11–2-12
 Dialect Translation and 2-13
AVG function 4-11, 4-15

B

Backus Naur Form (BNF) conventions 3-1–3-3, B-1–B-2
between predicates 4-16–4-17
BNF (Backus Naur Form) conventions 3-1–3-3, B-2

C

canceling transactions 4-22
cataloging metadata 2-8
character set in SQL 3-3
column functions 4-15
 applying to groups of values 4-16
 subqueries and 4-16
column titles 5-2
 renaming 4-10

 columns 4-11, 5-2
 joining 2-20
commit and rollback 4-22
COMMIT command 4-22
comparison predicates 4-16
correlated subqueries in SQL 4-13
correlation names in SQL 4-6–4-7
COUNT function 4-11, 4-15
COUNT(*) function 4-11, 4-15
counting rows 4-11, 4-15
CREATE TABLE command 5-1, 5-3–5-4
CREATE VIEW command 5-1, 5-4–5-5
creating tables 5-3–5-4
creating views 5-4–5-5

D

data adapters 2-16
 generating DML 2-16
data manipulation commands 4-20
Data Manipulation Language (DML) 1-2, 2-16
 generating 2-16–2-18, 2-20–2-23
 SQL Translation and 2-11
data types 5-2
database engine settings 2-8–2-10
database engines 2-5
 setting 2-5–2-8
dates in SQL 4-3
DELETE command 4-20–4-21
deleting rows 4-21

Index

Dialect Translation 1-2, 2-11, 2-13
 processing requests 2-15
 requirements 2-14

digits in SQL 3-3

DML (Data Manipulation Language) 1-2, 2-16
 generating 2-16–2-18, 2-20–2-23
 SQL Translation and 2-11

double colon = operator 3-2

E

elements of tables 4-9
 grouping 4-9–4-10

engine settings 2-8–2-10

EXECUTE command 6-2–6-3

existence predicates 4-18

F

FROM clause 4-5–4-7

G

GROUP BY clause in SQL 4-9–4-10

H

HAVING clause in SQL 4-10

I

identifiers in SQL 3-6, 4-6
 naming 3-6

in predicates 4-18

inner join structures 4-5–4-6

INSERT command 4-20–4-21

inserting rows 4-20–4-21

inserting values 4-21

ISO SQL model 1-2

J

join structures 2-17, 4-5–4-6
 optimizing 2-17–2-18, 2-20–2-23
 queries in SQL and 4-13

JOINOPT parameter 2-21

JOINTYPE parameter 2-18

K

keywords in SQL 3-5, A-2

L

language constructs 3-2

letters in SQL 3-3

like predicates 4-17

literals in SQL 3-5

M

MAX function 4-11, 4-15

MIN function 4-11, 4-15

N

nested loop join strategy 2-18, 2-22

null predicates 4-17–4-18

null values 5-2

O

optimizing join structures 2-18, 2-20–2-23

ORDER BY clause in SQL 4-9, 4-14

outer join structures 4-5–4-6

outer queries 4-13

P

predicates in SQL 4-16
 between 4-16–4-17
 comparison 4-16
 existence 4-18
 in 4-18
 like 4-17
 logical operators and 4-19
 null 4-17–4-18
 quantified 4-18

PREPARE statement 6-2

Q

qualifiers in SQL 3-6
 quantified predicates 4-18
 queries in SQL 4-3–4-4
 join structures and 4-13

R

RDBMS engine 2-2, 2-4
 setting 2-5
 reserved words A-1
 ROLLBACK command 4-22
 rows 5-2
 counting 4-11, 4-15
 deleting 4-21
 inserting 4-20–4-21
 updating 4-21
 running SQL requests 6-2–6-3

S

scalar functions 4-14
 search conditions 4-19
 search criteria 4-8
 specifying 4-8–4-10

SELECT statement 4-4, 4-20
 AS clause 4-10
 GROUP BY 4-9
 HAVING clause 4-10
 ORDER BY clause 4-14
 subqueries 4-12
 WHERE clause 4-8–4-9

setting a database engine 2-5, 2-7–2-8
 client applications and 2-5
 stored procedures and 2-6

SOME predicate 4-18

sort/merge join strategy 2-18, 2-23

special characters in SQL 3-3

specifying qualifying conditions to groups 4-10

SQL identifiers 3-6
 naming 3-6

SQL keywords 3-5

SQL language 3-3

SQL literals 3-5

SQL passthru 1-1, 2-2, 2-4–2-6, 2-8
 canceling 2-6
 requirements 2-4
 setting 2-7

SQL qualifiers 3-6

SQL queries 4-3–4-4

SQL requests 1-1–1-2, 2-1–2-2, 2-4–2-5, 2-11,
 6-1–6-2
 deleting 4-22
 running 6-2–6-3

SQL statements 4-2
 PREPARE 6-2
 SELECT 4-4
 unsupported 4-3

SQL syntax B-1–B-2

Index

SQL Translation Services 1-1–1-2, 2-11
 Data Manipulation Language and 2-11
 Dialect Translation 2-13
 generating DML 2-16

SQL Translator 4-13, 6-2, A-1

SQLENGINE command 2-5–2-6

subqueries in SQL 4-12–4-13

SUM function 4-11, 4-15

T

table columns 5-2

tables 5-2
 creating 5-1–5-4

tokens in SQL 3-5
 identifiers 3-6
 keywords 3-5
 literals 3-5

transactions 4-22
 canceling 4-22
 undoing changes 4-22

U

undoing changes to transactions 4-22

Universal SQL Translator A-2

UPDATE command 4-20–4-21

V

values 4-20
 inserting 4-20–4-21

views 5-4
 creating 5-4–5-5

W

WHERE clause in SQL 2-21, 2-23, 4-8–4-9, 4-20

WHERE clause predicates 2-22

Reader Comments

In an ongoing effort to produce effective documentation, the Documentation Services staff at Information Builders welcomes any opinion you can offer regarding this manual.

Please use this form to relay suggestions for improving this publication or to alert us to corrections. Identify specific pages where applicable. You can contact us through the following methods:

- Mail:** Documentation Services - Customer Support
Information Builders, Inc.
Two Penn Plaza
New York, NY 10121-2898
- Fax:** (212) 967-0460
- E-mail:** books_info@ibi.com
- Web form:** <http://www.informationbuilders.com/bookstore/derf.html>

Name: _____

Company: _____

Address: _____

Telephone: _____ Date: _____

E-mail: _____

Comments:

Reader Comments