



Ignite Technologies
Security and Compliance Solutions



Event Data Warehouse (EDW) Guide (DRAFT)

SenSage Standard 2016 (v. 6.1.1)

August 4, 2016

COPYRIGHT INFORMATION

No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopied, recorded, or otherwise, without the prior written consent of Ignite Technologies, Inc.

401 Congress Avenue Suite 2650

Austin, TX 78701

800-248-0027

info@ignitetechnology.com

Copyright © 2016 Ignite Technologies, Inc.

All Rights Reserved.

Published June 21, 2016

TABLE OF CONTENTS

PREFACE	15
Audience for this Book	15
SenSage AP SQL Reference Organization	15
Road Map to SenSage AP Documentation	17
Conventions Used in SenSage AP Documentation	19
Contacting Technical Support	20
CHAPTER 1: ABOUT THE EVENT DATA WAREHOUSE (EDW)	21
EDW and SenSage AP SQL	21
Business Logic and the SenSage AP SQL Engine	21
Other Configuration Support	21
OAE for Open Standards	21
VMWare to run virtual machine	22
CHAPTER 2: SENSAGEUSING THE SENSAGE EVENT DATA WAREHOUSE (EDW)	23
Querying and SenSage AP SQL	23
Writing Queries and SenSage AP SQL	23
EDW Loading and Data Storage	23
Common AP SQL Commands and Tasks	24
Defining Tables with SenSage AP SQL (AP SQL)	24
Syntax for Creating Tables	24
Renaming Tables	25
Syntax for Dropping Tables	25
Defining Column Filters with SenSage AP SQL	25
Syntax for Creating Column Filters	26
Syntax for Dropping Column Filters	30
Defining Views with SenSage AP SQL	30
Syntax for Creating Views	31
Declaring the Columns in a View	32
Selecting a Subset of Table Columns or Rows for a View	34
Specifying and Querying Against Time Ranges in a View	35
Using Processing Directives when Creating a View	35
Declaring a Union of Tables in a View	35
Syntax for Renaming Views	36
Syntax for Dropping Views	36
Syntax for Transferring Ownership of a View from One User to Another	37
Retrieving Information About a View	37
International Support in the EDW	38
Character Sets	38
Character Encoding Schemes	39
Encoding Schemes Used by the EDW	40
Character Encoding and Computer Fonts	40
EDW System Tables	41
CHAPTER 3: SENSAGE AP SQL	43

Overview of AP SQL SELECT Statements	43
Basic SELECT Syntax	43
Significant Terms	44
Keywords and Clauses Required in SELECT statements	44
About Tables and Namespaces	44
Target Clauses	45
Computed expressions in Target Clauses	45
Named Targets	45
Invisible targets	46
DISTINCT Modifier Keyword	46
FIRST and LAST Modifiers	46
FROM Clauses	47
Syntax of FROM Clauses	47
Table Specifications in FROM Clauses	47
Example FROM Clauses	48
Identifiers as the Table Name	48
String Expressions as the Table Name	48
Expression Macros as the Table Name	48
Including Rows from Bad Loads	49
DURING Clauses	49
Alternative Formats for DURING Clauses	50
Timestamp Precision in DURING Clauses	50
Subqueries and Views and the DURING Clause	51
Specifying DURING ALL in the View	51
WHERE Clauses	51
GROUP BY Clauses and Aggregation Queries	52
Aggregation Partitioning	52
Aggregation Functions	54
Multi-column GROUP BY	55
IN <i>n</i> PASSES	55
SLICE BY Clauses	56
HAVING Clauses	57
ORDER BY Clauses	58
UNION ALL Clauses	59
Restrictions on UNION ALL	60
Alternative to UNION ALL	60
Data Types	60
bool	60
float	61
int32	61
int64	61
timestamp	62
varchar	62
Converting varchar Values to timestamp Values	63
Varchar Representations of Durations	64
inet	64
inet Operators and Functions	65
Data Source Expressions	65
Column Expressions	65
Literal constants	66
Integer Literals	67
Floating-Point Literals	67
String Literals	67
Typecast Literals	68

Data Processing Expressions	68
Operators	69
Math Operators	69
The String Concatenation Operator	70
Comparison Operators	70
Logical Operators	72
Operator Precedence	73
Functions	73
Function Syntax	74
Asterisks as Column-Expression Arguments	74
The SORT BY Modifier Keyword	75
The DISTINCT Modifier Keyword	75
Conversion Expressions	75
CONVERT Expressions	76
Function-style Conversion Expressions	76
Typecast Literal Conversion Expressions	77
CASE Expressions	77
Processing Directives	78
Macros	78
About Macros	78
Expression Macros	79
Star Macros	79
Multiple Declarations of a Given Macro	81
Resolving Macro Identifiers	81
Overriding Multiple Macro Declarations	81
User-Defined Subroutines	82
Subqueries	84
Subqueries with DURING Clauses	84
Subqueries and UNION ALL Clauses	85
Subqueries and the WHERE clause	85
Subqueries and the ORDER BY Clause	86
Table-Name Substitutes	86
WHERE Clause Filters	86
Settings	87
The TIMEZONE Setting	88
The Scope of Processing Directives	88
Global State Modifiers	88
Local Definitions	89
Working with Lists	89
Multiple Values as Lists	89
Functions that Return Lists	90
List Acceptors	90
INTO Keyword	91
List Expressions and FROM Clauses	92
EXPLODE Keyword	92
EXPLODE BY Keyword Phrase	93
Some Helpful List Examples	93
 CHAPTER 4: SENSAGE AP SQL FUNCTIONS	 95
Conditional Evaluation Functions	95
_if()	95
Synopsis	95
Description	95
Arguments	96
Return Value	96

Exceptions	96
Example	96
_iftable()	96
Synopsis	96
Description	97
Arguments	97
Return values	97
Exceptions	97
List Functions	97
_list()	97
Synopsis	97
Description	98
Arguments	98
Return Value	98
Example	98
_nth()	98
Synopsis	98
Description	98
Arguments	99
Return Value	99
Exceptions	99
Example	99
Lookup Functions	99
_lookup()	99
Synopsis	100
Description	100
Arguments	100
Examples	104
Exceptions	107
Working with Lookup Files	107
_rev_dns()	109
Synopsis	110
Description	110
Arguments	110
Return Value	110
Exceptions	110
Examples	110
_tablematch()	110
Synopsis	110
Description	111
Arguments	112
Return Value	112
Example	112
Aggregation Functions	113
avg()	113
Synopsis	113
Description	114
Arguments	114
Return Value	114
Exceptions	114
Example	114
count()	114
Synopsis	114
Description	114
Arguments	114
Return Value	115
Examples	115

max()	115
Synopsis	115
Description	115
Arguments	115
Return Values	115
Exceptions	115
Example	116
min()	116
Synopsis	116
Description	116
Arguments	116
Return Values	116
Exceptions	116
Example	117
median()	117
Synopsis	117
Description	117
Arguments	117
Return Value	117
Exceptions	117
Example	117
sum()	118
Synopsis	118
Description	118
Arguments	118
Return Values	118
Exceptions	118
Examples	118
_first()	118
Synopsis	119
Description	119
Arguments	119
Return Value	119
_last()	119
Synopsis	119
Description	119
Arguments	119
Return Value	119
_strsum()	119
Synopsis	119
Description	120
Arguments	120
Return values	120
Statistical Aggregate Functions	120
var_pop()	120
Synopsis	120
Description	120
Arguments	121
Return Value	121
stddev_pop()	121
Synopsis	121
Description	121
Arguments	122
Return values	122
var_samp()	122
Synopsis	122
Description	122
Arguments	123

Return Value	123
stddev_samp()	123
Synopsis	123
Description	123
Arguments	124
Return values	124
variance()	124
stddev()	124
Logarithmic and Exponential Functions	124
_log()	125
Synopsis	125
Description	125
Arguments	125
Return Value	125
_log10()	125
Synopsis	125
Description	125
Arguments	125
Return Value	125
_pow()	126
Synopsis	126
Description	126
Arguments	126
Return Value	126
_exp()	126
Synopsis	126
Description	126
Arguments	126
Return Value	126
Numeric Rounding Functions	126
_abs()	127
Synopsis	127
Description	127
Arguments	127
Return Value	127
Exceptions	127
_ceil()	127
Synopsis	127
Description	127
Arguments	127
Return Value	128
_floor()	128
Synopsis	128
Description	128
Arguments	128
Return Value	128
Example	128
_round()	128
Synopsis	128
Description	128
Arguments	129
Return Value	129
Exceptions	129
String Functions	129
_strlowercase(), _lc()	130
Synopsis	130
Description	130

Arguments	130
Return Value	130
Examples	130
<code>_strupercase(), _uc()</code>	131
Synopsis	131
Description	131
Arguments	131
Return Value	131
<code>_strmd5(), _md5()</code>	131
Synopsis	131
Description	131
Arguments	131
Return Value	132
Exceptions	132
<code>_strmd5_64(), _md5_64()</code>	132
Synopsis	132
Description	132
Arguments	132
Return Value	132
Example	132
<code>_strlen()</code>	132
Synopsis	133
Description	133
Arguments	133
Return Value	133
Examples	133
<code>_strstr()</code>	133
Synopsis	133
Description	133
Arguments	133
Return Value	133
Examples	134
<code>_strmatch()</code>	134
Synopsis	134
Description	134
Arguments	134
Return Value	134
Exceptions	134
Examples	135
<code>_strmatchlist()</code>	135
Synopsis	135
Description	135
Arguments	135
Return Values	135
Example	136
<code>_strsplit()</code>	136
Synopsis	136
Description	136
Arguments	136
Return Values	136
<code>_strsplitxsv()</code>	136
Synopsis	136
Description	137
Return Values	137
<code>_strleft()</code>	137
Synopsis	137
Description	137
Arguments	137

Return Value	137
_strright()	137
Synopsis	137
Description	137
Arguments	138
Return Value	138
_strmiddle(), substr()	138
Synopsis	138
Description	138
Arguments	138
Return Value	138
_strrepeat()	138
Synopsis	138
Description	138
Arguments	139
Return Value	139
Exceptions	139
Examples	139
_strlpad()	140
Synopsis	140
Description	140
Arguments	140
Return Value	141
Exceptions	141
Examples	141
_strrpadd()	142
Synopsis	142
Description	142
Arguments	142
Return Value	143
Exceptions	143
Examples	143
_strtrim()	144
Synopsis	144
Description	144
Arguments	144
Return Value	145
Examples	145
_strlink()	145
Synopsis	145
Description	145
Arguments	146
Return Value	146
Examples	146
_strcat()	147
Synopsis	147
Description	147
Arguments	147
Return Value	147
Example	147
_strjoin()	147
Synopsis	147
Description	147
Arguments	147
Return Value	148
Example	148
_strformat(), _sprintf()	148
Synopsis	148

Description	148
Arguments	148
Format Specifiers	148
Return Value	150
Exceptions	150
Examples	150
Time Functions	150
_now(), now()	151
Synopsis	151
Description	151
Return Value	151
_time(), time()	151
Synopsis	151
Description	151
Arguments	151
Return Value	153
Examples	153
_timeadd()	153
Synopsis	154
Description	154
Arguments	154
Return Value	154
Examples	154
_timediff()	155
Synopsis	155
Description	155
Arguments	155
Return Value	155
Example	155
_timeformat(), _timef()	155
Synopsis	156
Description	156
Arguments	156
Return Value	158
Exceptions	158
Example	159
_timeparse(), _strptime(), _strftime()	159
Synopsis	159
Description	160
Arguments	160
Return Value	160
Exceptions	160
Examples	160
_timestart()	161
Synopsis	161
Description	161
Arguments	161
Return Value	161
Exceptions	161
Network Address Functions	162
_abbrev()	162
Synopsis	162
Description	162
Arguments	162
Return Value	163
Exceptions	163
Examples	163

<code>_broadcast()</code>	163
Synopsis	163
Description	163
Arguments	163
Return Value	163
Exceptions	163
Examples	164
<code>_family()</code>	164
Synopsis	164
Description	164
Arguments	164
Return Value	164
Exceptions	164
Examples	164
<code>_host()</code>	165
Synopsis	165
Description	165
Arguments	165
Return Value	165
Exceptions	165
Examples	165
<code>_hostmask()</code>	165
Synopsis	165
Description	165
Arguments	166
Return Value	166
Exceptions	166
Examples	166
<code>_masklen()</code>	166
Synopsis	166
Description	166
Arguments	166
Return Value	166
Exceptions	166
Examples	167
<code>_netmask()</code>	167
Synopsis	167
Description	167
Arguments	167
Return Value	167
Exceptions	167
Examples	167
<code>_network()</code>	167
Synopsis	167
Description	167
Arguments	168
Return Value	168
Exceptions	168
Examples	168
<code>_set_masklen()</code>	168
Synopsis	168
Description	168
Arguments	168
Return Value	168
Examples	169
<code>_mapto_ipv4(), _mapto_ipv6()</code>	169
Synopsis	169
Description	169

Arguments	169
Return Value	169
_inet_contains()	169
Synopsis	170
Description	170
Arguments	170
Return Value	170
_inet_contains_or_equal()	170
Synopsis	170
Description	170
Arguments	170
Return Value	171
_inet_plus(), _inet_minus()	171
Synopsis	171
Description	171
Arguments	171
Return Value	171
Examples	171
_inet_and(), _inet_not(), _inet_or(), _inet_xor()	172
Synopsis	172
Description	172
Arguments	172
Return Value	172
Examples	172
Miscellaneous Functions	172
_quantize()	173
Synopsis	173
Description	173
Arguments	173
Return Value	177
Exceptions	177
_fifo()	177
Synopsis	177
Description	177
Arguments	178
Return Value	178
Exceptions	178
_lms_taskid()	178
Synopsis	178
Description	178
Return Value	178
Example	178
_lms_buildinfo()	179
Synopsis	179
Description	179
Return Value	179
Example	179
_fromname()	180
Synopsis	180
Description	180
Return Value	180
Examples	180
_fromindex()	181
Synopsis	181
Description	181
Return Value	181
Examples	181

CHAPTER 5: CONFIGURING AND MANAGING OPEN ACCESS EXTENSION (OAE) 183

Overview	183
OAE Architecture and Process Flow	184
Obtaining Drivers and Client Software	185
Handling Failover and Replication	186
Mapping Tables and Views from the EDW to Postgres	186
Managing Users and Roles in Postgres	187
Using the Postgres Command-Line Tool to Log Into Postgres	188
Configuring OAE	188
Displaying version information	188
Displaying OAE Translations	188
Allowing for Correlated External Subqueries	189
Allowing only for Queries Sent to EDW	189
Returning EDW Query Error for a Call that Omits OAE "During" Clause	189
Limiting Row Maximum Returned from an EDW Query	190
Managing EDW External Tables in Postgres	190
Mapping Namespaces and Table Names Between the EDW and Postgres	191
Working with Schema Names	192
Working with Table, View, and Column Names	194
Synchronizing EDW Schema Changes to Postgres	194
OAE and EDW IP Address Support	194
Understanding OAE Security	194
Security Issues	195
Encryption and Single Sign-On	195

CHAPTER 6: USING OPEN ACCESS EXTENSION TO QUERY THE EDW . 197

Overview: Pass-Through Queries	197
Joins and Subqueries	198
Running Queries Against the EDW	199
Timestamp Restriction and the DURING Clause	200
Using oae.during()	200
Specifying DURING ALL	202
Specifying a Series of Time Periods	203
Specifying the Postgres Timestamp Function with Explicit Time Zone	203
Mapping Data Types	204
Additional Ways to Use EDW Extensions When Querying EDW Data ..	205
Writing a Pass-Through Query	205
Query 1: Example Pass-Through Query	206
Delimiting Quotation Marks	206
Query 2: Example Standard Query Against a Postgres Pass-Through View ..	207
Transaction Isolation Levels	207
Running DDL and DML Commands	208
Setting up OAE for EDW Multi-cluster Access	208
Group Tables and EDW Multi-Cluster Support	209
Creating and Using Group Tables	209
Using OAE over ODBC to Return Data to Excel: An Example	210
Verify psqLODBC is Installed on Your System or Download & Install It ..	211
Create a DSN and Point it to the Server that Runs OAE	211
Import Data Into Microsoft Office Excel from OAE EDW	213

CHAPTER 7: RUNNING AND USING SENSAGE AP ON VMWARE 217

Overview	217
Support for Additional Functionality of Virtualization Products	217
SenSage AP Virtualized Architecture Considerations	218
VMware Migration	218
VMware and Linux Versions	219
Hardware Configuration for ESX Server	219
ESX Server CPU	219
ESX Server RAM	219
Network Interface Card (NIC)	219
Storage	220
Virtual Machine Configuration	220
Deploying EDW and non-EDW components	220
Recommended Virtual Machine Configuration for SenSage AP Hosts	221
Reserving CPU Resources for the ESX Server to Improve Performance	221
Install VMware Tools	223
CHAPTER 8: PERL SUBROUTINES	225
About Perl Subroutines in SenSage AP SQL	225
How Perl Processing Works	225
Declaring Perl Functions	226
Declaring Perl Aggregates	227
Perl Execution Environment	229
Exiting from Perl Subroutines	229
List Support and Perl Functions	229
Using Macros in Perl Subroutines	230
Understanding Parallelism and Side Effects	231
Accessing External Modules	232
The use Directive	232
The @INC Variable	233
The Inline.pm Perl Module	233
Testing and Debugging Perl Subroutines	233
Running Perl Subroutines in Test Scripts	233
Printing Debugging Messages in Utility Logs	234
Printing Debugging Messages to External Files	235
Installing Perl Modules	235
CHAPTER 9: USING THE DBD DRIVER	237
Installation	237
Sample DBI Program	238
Explanation of the Sample DBI Program	238
DBI elements Supported by SenSage AP	240
SenSage AP DBD attributes	241
addamark_tableNamespace	241
addamark_rawOutputHandle	241
addamark_dbgprintRequest	242
addamark_oob_callback	242
APPENDIX A: MAPPING OPERATORS AND FUNCTIONS	243
Logical Operators	243

Comparison Operators	244
Mathematical Operators	244
Mathematical Functions	245
String Functions and Operators	246
Conversion Functions	247
Pattern Matching	248
Formatting Functions	248
Date/Time Operators	248
Inet Functions	249
Inet Operators	249
Conditional Expressions	249
Aggregate Functions	249
Aggregate Functions for Statistics	250
APPENDIX B: TIME ZONES	251
Time-Zone Conversion	251
Supported Time Zones	251
APPENDIX C: OPEN ACCESS EXTENSION CONSIDERATIONS	261
Translation Failure	261
Architecture Model Support	261
APPENDIX D: SENSAGE APSYSTEM TABLES	263
system.properties	263
system.cluster_properties	264
system.task_list	264
system.users	265
system.users2	265
system.userroles	265
system.userroles2	266
system.roles	266
system.roles2	266
system.permissions	267
system.rolepermissions	267
system.rolepermissions2	267
system.upload_info	268
system.raw_upload_info	268
<namespace>.storage.metadata	269
<namespace>.storage.metadata_history	270
<namespace>.storage.raw_metadata	270
<namespace>.storage.raw_metadata_with_ts	271
<namespace>.<tablename>.storage.metadata	272
INDEX	1

PREFACE

This book, the Event Data Warehouse Guide, describes the 6.2.0 version of the SenSage AP software. The SenSage *Event Data Warehouse Guide* includes Open Access Extension® (OAE)—a SenSage AP Open Interface that allows users to access EDW data with such open standards as ANSI SQL, ODBC, and JDBC.

NOTE: SenSage Analyzer also provides an Administration Mode interface that enables you to create users, roles, and assets, and to enable rules. For information on these and other administration tasks, see the *Administration Guide*.

This Preface contains the following sections:

- [“Audience for this Book”, next](#)
- [“SenSage AP SQL Reference Organization”, on page 15](#)
- [“Road Map to SenSage AP Documentation”, on page 17](#)
- [“Conventions Used in SenSage AP Documentation”, on page 19](#)
- [“Contacting Technical Support”, on page 20](#)

AUDIENCE FOR THIS BOOK

This book is directed to:

- **Developers**—who create SQL queries, views, and reports, who develop log adapter PTL files, and who use Open Access Extension to query event data stored in the SenSage Event Data Warehouse (EDW).
- **System Administrators**—who administer the SenSage AP Event Data Warehouse.

SENSAGE AP SQL REFERENCE ORGANIZATION

This book contains the following chapters:

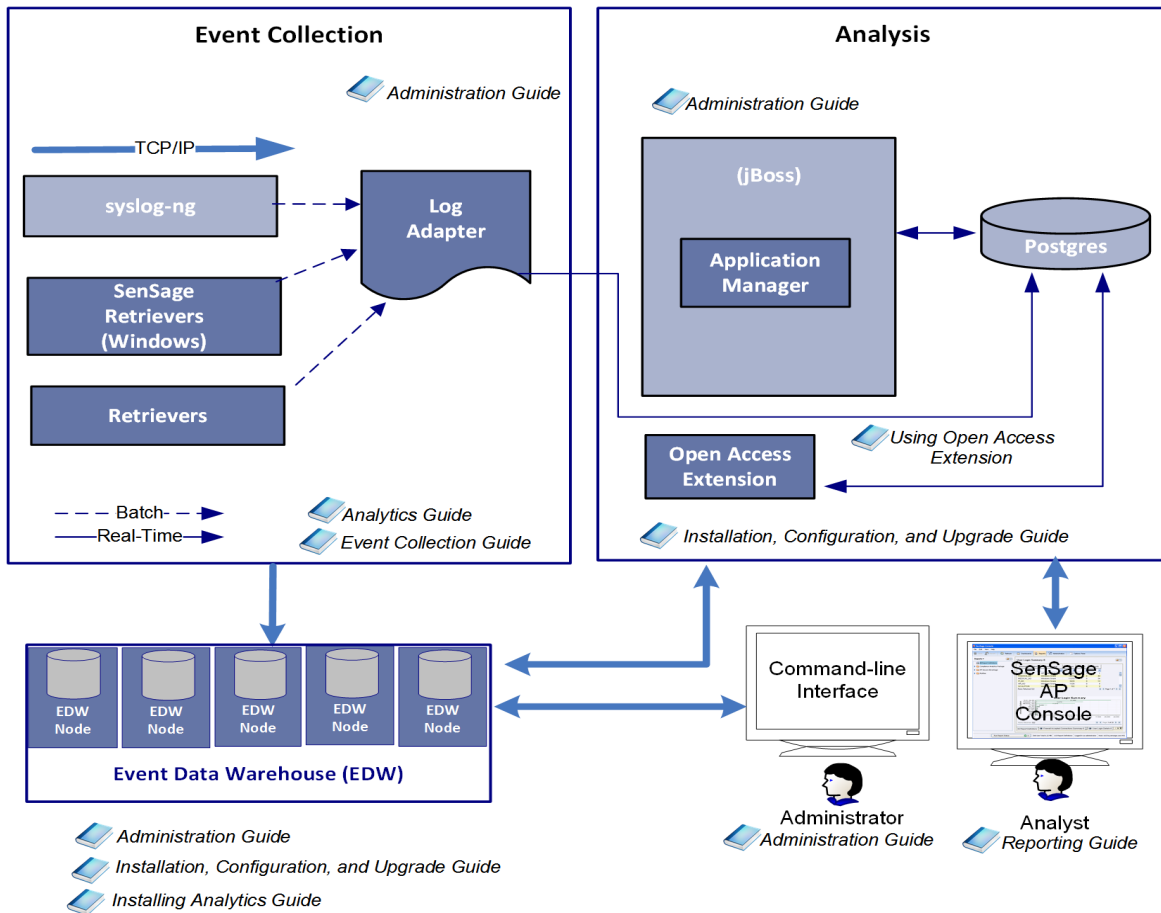
- [Chapter 1: About the Event Data Warehouse \(EDW\)](#)—Provides an overview of Event Data Warehouse (EDW) features.
- [Chapter 2: SenSageUsing the SenSage Event Data Warehouse \(EDW\)](#)—Provides information on using SenSage AP SQL (also known as AP SQL) to query the EDW and event-log entries, write queries, and perform common tasks with basic AP SQL commands.
- [Chapter 3: SenSage AP SQL](#)—Provides a reference to the a SenSage AP SQL language.
- [Chapter 4: SenSage AP SQL Functions](#)—Provides reference for the rich library of functions that SenSage AP SQL supports. Supported tasks include string matching, time handling, and mathematical functions, lookups, and conditional evaluations.
- [Chapter 5: Configuring and Managing Open Access Extension \(OAE\)](#)—Provides information on managing EDW external tables in Postgres and understanding OAE security.
- [Chapter 6: Using Open Access Extension to Query the EDW](#)—Provides information on how to use OAE to query the EDW with specific query types.

- [Chapter 7: Running and Using SenSage AP on VMware](#)—Provides recommended configurations for running a SenSage AP instance under VMware
- [Chapter 8: Perl Subroutines](#)—Provides a reference on how to create, debug, and use Perl routines from within SenSage AP SQL. These functions allow you to perform complex transforms on data.
- [Chapter 9: Using the DBD Driver](#)—Provides details on installing and using the DBD driver (DBD::Addamark), which implements the DBI (Database Independent) API over the Perl API of the EDW.
- [Appendix A: Mapping Operators and Functions](#)—Provides a reference on Postgres functions supported by the EDW.
- [Appendix B: Time Zones](#)—Lists every time zone supported by the EDW.
- [Appendix C: Open Access Extension Considerations](#)—Lists all known issues regarding Open Access Extension (OAE)
- [Appendix D: SenSage APSystem Tables](#)—Contains the schema of each EDW system table.

ROAD MAP TO SENSAGE AP DOCUMENTATION

This document, the *Event Data Warehouse Guide*, is part of the larger documentation set of your SenSage AP system. [Figure P-1](#) illustrates SenSage AP components and modules in the context of their function within the SenSage AP system.

Figure P-1: Road Map to SenSage AP Documentation



The table below describes all the manuals in the SenSage AP documentation set and the user roles to which they are directed.

Role	Tasks	Documentation
Analyst, Report Developer, System Administrator	<ul style="list-style-type: none"> Create and edit reports, charts, and models Create and edit dashboards Manage SenSage AP Users Manage SenSage AP Models Create and edit data models Write SQL code and queries 	<i>Analyzer Guide</i>

Role	Tasks	Documentation
Business Analyst or System Administrator	<ul style="list-style-type: none"> • Learn about Analytics • Use IntelliSchema views • Learn about the Foundation and Compliance Analytics packages • Learn about additional Analytics packages 	<i>Analytics Guide</i>
Business Analyst, Report Developer	<ul style="list-style-type: none"> • Start with two tutorials that introduce you to data modeling features • Create and edit advanced data models 	<i>Advanced Data Modeling Users Guide</i>
Developer, Report Developer	<ul style="list-style-type: none"> • Get an overview of advanced Analytics • View a listing of all sample models available in the Analytics package • Learn how to implement a Data Table in sample models 	<i>Advanced Analytics Developers Guide</i>
Developer, Report Developer	<ul style="list-style-type: none"> • Find details on each Advanced Analytics model component 	<i>Advanced Analytics Reference Guide</i>
Developer, Report Developer or Security Analyst	<ul style="list-style-type: none"> • Use SenSage AP SQL, SenSage AP SQL functions, and libraries to create reports or query the EDW • Access EDW data using open standards as ANSI SQL, ODBC, and JDBC • Create and use Perl code in SenSage AP SQL statements • Use the DBD Driver to query SenSage AP from other locations 	<i>Event Data Warehouse Guide</i>
Security System Administrator	<ul style="list-style-type: none"> • Configure retrievers, receivers, and collectors • Enable/disable log adapters • Configure SenSage Retriever • Create log adapter PTL files 	<i>Collector Guide</i>
System Administrator	<ul style="list-style-type: none"> • Install SenSage AP • Configure SenSage AP and its components • Configure Vmware 	<i>Installation, Configuration, and Upgrade Guide</i>
System Administrator	<ul style="list-style-type: none"> • Manage the SenSage Event Data Warehouse (EDW) • Manage the Collector • Manage users, groups, and permissions • Archive to nearline storage • Manage assets & monitor security alerts • Monitor log source health • Monitor system health • Troubleshoot • Error Messages 	<i>Administration Guide</i>

Role	Tasks	Documentation
Legal	Monitor third-party licenses	<i>Third-Party Open Source Licensing</i>

TIP: You can access the manuals listed above from:

- SenSage AP Welcome page

Click the **Documentation** hyperlink.

CONVENTIONS USED IN SENSAGE AP DOCUMENTATION

This convention...	Indicates...	Example
bold text	Names of user interface items, such as field names, buttons, menu choices, and keystrokes	Click Clear Filter .
<i>italic text</i>	Indicates a variable name or a new term the first time it appears	<code>http://<host>:<port>/index.mhtml</code> Use the <i>whammerjammer</i> to adjust the whamming frequency.
Courier text	Indicates a literal value, such as a command name, file name, information typed by the user, or information displayed by the system	<code>atquery localhost:8072 myquery.sql</code>
SMALL CAPS	Indicates a key on the computer keyboard	Press ENTER.
{ }	In a syntax line, curly braces surround a set of options from which you must choose one and only one. NOTE: Syntax specifications for SELECT statements include curly braces as part of the <code>{INCLUDE_BAD_LOADS}</code> keyword.	<code>{ start stop restart }</code>
[]	In a syntax line, square brackets surround an optional parameter	<code>atquery [options] <host>:<port> -</code>
	In a syntax line, a pipe within square brackets or curly braces separates a choice between mutually exclusive parameters NOTE: Syntax for defining a Nearline Storage Address (NSA) includes a pipe.	<code>{ start stop restart }</code> <code>[g m]</code>
...	In a syntax line, ellipses indicate a repetition of the previous parameter	The following example indicates you can enter multiple, comma-separated options: <code><option>[, <option>[...]]</code>

This convention...	Indicates...	Example
backslash (\)	A backslash in command-line syntax or in a command example behaves as the escape character on Unix. It removes any special meaning from the character immediately following it. In SenSage AP documentation, a backslash nullifies the special meaning of the newline character as a command terminator. Without the backslash, pressing ENTER at the end of the line causes the Unix system to execute the text preceding the ENTER. Without the backslash, you must allow long commands to wrap over multiple lines as a single line.	<pre>atquery --user=administrator \ --pass=pass:p@ss localhost:8072\ -e='SELECT * FROM system.users;'</pre>

CONTACTING TECHNICAL SUPPORT

For additional help, call +1 855-529-3929. Also you can log into the IgniteTech Technical Support web page under Services at <http://www.ignitetech.com> for SenSage AP documentation, product downloads, and additional information on contacting support to escalate help on issues that impact your production environment.

About the Event Data Warehouse (EDW)

The Event Data Warehouse (EDW) is a scalable database built for and dedicated to loading, storing, and analyzing event data. Event data from multiple sources is stored in a highly compressed format. Parallel processing enables clustered servers to execute as a single instance, allowing high-speed loading and querying on terabytes of data.

This architecture allows users to load and query massive data volumes in a single, logical database instance without partitioning. The EDW uses a proprietary data model that achieves high levels of compression, while still making all data fully available to query. For details on the EDW software, see the *Administration Guide*.

EDW AND SENSAGE AP SQL

SenSage AP SQL, also known as AP SQL, is a special implementation of standard SQL optimized for event-log analysis. Using command line utilities such as `atquery` and `atview`, you can execute AP SQL statements and functions to query and view EDW data. For details on command line utilities, see the *Administration Guide*. AP SQL statements and functions are described in the following sections of this guide:

- [“SenSage AP SQL”, on page 43](#)
- [“SenSage AP SQL Functions”, on page 95](#)

BUSINESS LOGIC AND THE SENSAGE AP SQL ENGINE

SenSage AP SQL can perform simple transformation and analysis of event-log data. The SenSage AP SQL engine also contains a Perl subsystem that provides the ability to perform complex transformations, also known as *business logic*. The Perl subsystem lets you declare and use Perl code in your SenSage AP SQL statements. For more details, see [“Perl Subroutines”, on page 225](#).

OTHER CONFIGURATION SUPPORT

OAE for Open Standards

Open Access Extension[®] (OAE) is a SenSage AP Open Interface that allows users to access EDW data with such open standards as ANSI SQL, ODBC, and JDBC. OAE uses the Postgres database query processing interface to provide access to SenSage AP EDW data. A user of a third-party tool such as SAS[®], MicroStrategy, or a data visualization tool can access and join data from EDW Tables and Postgres tables with Postgres-supported Open Application Program Interfaces such as ODBC and JDBC. OAE provides seamless and transparent access to Postgres tables and EDW tables, together in one relational repository.

For details, see [“Configuring and Managing Open Access Extension \(OAE\)”, on page 183](#) and [“Using Open Access Extension to Query the EDW”, on page 197](#).

VMWare to run virtual machine

SenSage AP supports its product in a virtualized environment, allowing you to deploy the product if you have virtualization mandates. For details, see [“Running and Using SenSage AP on VMware”, on page 217](#).

SenSageUsing the SenSage Event Data Warehouse (EDW)

The SenSage EDW provides its own version of SQL, SenSage AP SQL (also known as AP SQL) and both AP SQL and Perl functions to query event-log entries. A query is a SenSage AP SQL statement that extracts event-log data from the EDW data store. The EDW also provides utilities to load event-log entries and manage the operation of the EDW.

QUERYING AND SENSAGE AP SQL

You may need a report to perform the following operations:

- include data from external files
- manipulate lists of data
- concatenate and truncate string data
- nest queries so that the results of one query are used as the input to the next
- use libraries and Perl subroutines

To create a report that takes advantage of SenSage AP extensions to SenSage AP SQL, an advanced user must create a SQL report. To create a SQL report requires entering a SQL query directly into the report definition window. The user who creates such a report must understand the SQL query language and the SenSage AP extensions to the language.

WRITING QUERIES AND SENSAGE AP SQL

For those features of SenSage AP SQL required to write a query against stored data, see the following chapters in this guide:

- [Chapter 3: SenSage AP SQL](#)
- [Chapter 4: SenSage AP SQL Functions](#)
- [Chapter 8: Perl Subroutines](#)

EDW LOADING AND DATA STORAGE

For more information on those features of SenSage AP SQL required to load and store data, see the following chapters in the *Administration Guide*:

- "Loading, Querying the EDW"
- "Configuring and Managing SenSage AP"

COMMON AP SQL COMMANDS AND TASKS

The following sections provide common AP SQL commands and tasks.

Defining Tables with SenSage AP SQL (AP SQL)

Tables store log events that have been loaded into in an EDW instance.

Syntax for Creating Tables

```
CREATE TABLE [<namespace>.<table_name>
  ( ts timestamp[, <column_declaration>[, <column_declaration>[...]] )
;
```

SPECIFYING NAMES FOR TABLES AND COLUMNS

Follow these rules for the names of tables and columns:

- The first character must be an ASCII underscore (_) or an ASCII English A-Za-Z.
- Remaining characters can be any of the above, plus the ASCII numerals 0-9.
- The name can be from 1 through approximately 32 Kb characters in length.

DECLARING THE COLUMNS IN TABLES

Enclose the list of columns you declare for a table in parentheses. Separate column declarations with commas. There is virtually no limit to the number of columns that you can declare for a table.

Column declarations have the following form:

```
<column_name> <data_type>
```

The list of columns must declare a column named "ts", and its data type must be "timestamp". Other column declarations can have any name that follows the naming rules listed above, and its data type can be any of the data types supported by SenSage AP SQL.

For more information on data types supported by SenSage AP SQL, see [“Data Types”, on page 60](#).

Automatic Columns when Creating Tables

The EDW creates automatically a special column named _uploadid, with SQL data type varchar. The column holds the ID of the upload operation that inserted each row into the table.

For more information about the _uploadid column, see [“Tracking Uploads in the EDW”, on page 72](#) and [“Using the UPLOADS Command”, on page 112](#) in the *Administration Guide*.

CREATING TABLES IN SPECIFIC NAMESPACES

Tables in an EDW instance are located within groups called *namespaces*. If you do not specify a namespace when you create a table, it is located in the default namespace established for the EDW instance. If you specify the namespace at the time you create the table, the table is created within the specified namespace. If the specified namespace does not exist, it is created. For information on how to specify a namespace, see [“Specify a Namespace”, on page 96](#) in the *Administration Guide*.

IMPORTANT: If you create tables in different namespaces that store data from the same type of log source, We recommend that you name the tables identically.

When you create a table, you can specify the namespace explicitly in the `CREATE TABLE` statement, or you can specify it with the `--namespace` option of the `atquery` command. For example, assume you have a `.sql` file named `myTable.sql` that contains the following statement:

```
CREATE TABLE myNamespace.myTable
(
  ts          timestamp,
  HostName    varchar,
  ProgName    varchar,
  Message     varchar
)
;
```

Assume you run the following `atquery` command:

```
atquery --namespace=firewalls host02:8072 myTable.sql
```

A table named `myTable` will be created in the namespace `firewalls.myNamespace`.

PERMISSIONS FOR CREATING TABLES

To create a table, you must have `sls.create` permission in the EDW instance, and you must have `sls.namespace` permission on the namespace where you want the table to be created.

Renaming Tables

Use the `atmanage` command to change the names of tables.

Syntax for Dropping Tables

```
DROP TABLE [<namespace>.]<table_name>
;
```

When you drop a table, you must identify fully the namespace in which the table is located. You can specify the namespace explicitly in the `DROP TABLE` statement, or you can specify it with the `--namespace` option of the `atquery` command.

IMPORTANT: When you drop a table, views and reports that depend on the table become invalid. In addition, the log events loaded in the table are discarded from the EDW instance.

PERMISSIONS FOR DROPPING TABLES

To drop tables, you must have `sls.drop` permission in the EDW instance, and you must have `sls.namespace` permission on the namespace where the table you want to drop is located.

Defining Column Filters with SenSage AP SQL

Column filters let you influence the execution plans for sparse-row queries. In the EDW, create filters on columns instead of adding indexes. Column filters enhance performance for queries that use the:

- equals (= or ==) or not equals (<> or !=) and LIKE operators

- `_strleft()`, `_strright()`, `_strmiddle()`, and `_substr()` functions

NOTE: A column filter enhances performance only when the matched string is a constant.

Syntax for Creating Column Filters

The syntax below creates filters on column(s) in the specified table. If you specify column(s) in the command, a filter is created for each specified column.

```
CREATE FILTERS [<filter_name>] ON [<namespace>.<table_name>
  { (<column_name> [<partial_filter>])[, (<column_name> [<partial_filter>])
  [...]] | ALL }
;
```

NOTE:

- The table you specify with `<table_name>` and any columns you specify with `<column_name>` must exist. Specify ALL to create column filters on all the columns in the table, with the exception of the `ts` column. An error occurs if you specify `ts` as a `<column_name>`.
- Existing filters remain in place.

When you create column filters on a table, you must identify fully the namespace in which the table is located. You can specify the namespace explicitly in the `CREATE FILTERS ON` statement, or you can specify it with the `--namespace` option of the `atquery` command.

IMPORTANT:

- All filters have names. If you do *not* specify a filter name in the `create filters` statement, the filter is named identically to the column it filters. If you *do* specify a filter name in the `create filters` statement, you can filter on only one column. The rules for naming a filter are the same as those for naming tables and columns; for more information, see [“Specifying Names for Tables and Columns”](#), on page 24.
- Each filter definition on a table must be unique to that table.

SYNTAX FOR DEFINING PARTIAL FILTERS

Partial filters enable filtering on text in a specified part of a varchar column. These filters enhance performance when:

- a query uses any of the following SenSage AP string functions: `_strleft()`, `_strmiddle()`, `_strright()` or `_substr()`
- the matched string is a constant
- the length of the matched string is greater than or equal to the count or length specified in the definition of the column filter

NOTE: Additionally, the `LEADING`, `TRAILING`, and `ANY SUBSTRING` filters enhance performance when a query specifies equality on the named column.

The syntax for `<partial_filter>` is:

```
LEADING <count> | TRAILING <count> | SUBSTRING <offset>, <length> | ANY SUBSTRING
<length>
```

When you specify a partial filter with the forms `LEADING`, `TRAILING`, or `SUBSTRING`, the SQL engine applies the corresponding string function to the column value when it constructs the filter on a varchar column. A query applies the same function to the predicate string that is matched to use the filter.

NOTE: The examples in the next three sections assume a table named `cdr` in a namespace also named `cdr`.

Example of the LEADING Filter

For a leading filter, the filter is constructed on the first `<count>` characters in the column.

```
atquery --namespace=cdr localhost:8072 -e 'create filters caller_leading ON cdr
(calling_number LEADING 3);'
```

For this filter, assume a calling number of 1234567890. If a query run against this table includes either of the following predicates, the query takes advantage of this column filter:

```
where calling_number = "1234567890"
```

or

```
where _strleft(calling_number,3)="123"
```

Example of the TRAILING Filter

For a trailing filter, the filter is constructed on the last `<count>` characters in the column.

```
atquery --namespace=cdr localhost:8072 -e 'create filters callee_trailing ON cdr
(called_number TRAILING 4);'
```

For this filter, assume a called number of 9876543210. If a query run against this table includes either of the following predicates, the query takes advantage of this column filter:

```
where called_number = "9876543210"
```

or

```
where _strright(called_number,5)="43210"
```

Example of the SUBSTRING Filter

For a substring filter, the filter is constructed on the substring that starts at the specified `<offset>` and extends the specified `<length>`.

```
atquery --namespace=cdr localhost:8072 -e 'create filters callee_substr ON cdr
(called_number SUBSTRING 1,4);'
```

For this filter, again assume a called number of 9876543210. If a query run against this table includes either of the following predicates, the query takes advantage of this column filter:

```
where _substr(called_number,1,4)="8765"
```

or

```
where _strmiddle(called_number,1,4)="8765"
```

NOTE:

- When searching for a string on the left, the first character is in the zero (0) position.
- The syntax requires a comma to separate the offset and length values specified for the SUBSTRING filter. However, as illustrated below in [System Tables for Column Filters](#), the comma does not display in the filter definition.

Example of the ANY SUBSTRING Filter

For an ANY SUBSTRING filter, the filter is constructed on all substrings in a varchar column of the specified *<length>*.

```
atquery --namespace=cdr localhost:8072 -e 'create filters callee_anysubstr ON
cdr (called_number ANY SUBSTRING 8);'
```

For this filter, again assume a calling number of 1234567890. If a query run against this table includes any of the following predicates, the query takes advantage of this column filter:

```
where _strmiddle(calling_number,0,8)=="12345678"
```

or where the number of characters in the predicate is greater than or equal to the number defined for the filter:

```
_strmiddle(calling_number,0,9)=="123456789"
```

or where the predicate uses the `_substr()` function:

```
_substr(calling_number,0,9)=="123456789"
```

or where the predicate function includes only the first two arguments:

```
_substr(calling_number,1)=="234567890"
```

IMPORTANT: Specify a length filter cautiously because it tends to be much larger than the other filter types. You can estimate the relative size of the filter if you understand the data. For example, assume that you know that most values in the column are 10 characters long and you create a filter of length 7. The typical number of filter values will be 10 minus 7, or 3 filter values for each column value. In this case, the filter can be expected to be three times the size of the other filter types. On the other hand, if you know that most values in the column are 30 characters long and you create a filter of length 10, you can expect the filter to be twenty times the size of other filter types.

SYSTEM TABLES FOR COLUMN FILTERS

The `column_filter` column in `<namespace>.storage.metadata` system tables identifies all filters applied to all columns in a given table.

To view the column filters in the `cdr.cdr` table used in the examples in [Syntax for Defining Partial Filters](#) above, you can run the following command:

```
atview --namespace=cdr localhost:8072 columns cdr
```

Figure 2-1 and Figure 2-2 illustrate the output.

Figure 2-1: Left Side of Output

namespace	table_name	column_name	column_type
cdr	cdr	ts	timestamp
		_uploadid	varchar
		call_end	timestamp
		call_ref	int64
		called_imei	varchar
		called_imsi	varchar
		called_number	varchar
		calling_imei	varchar
		calling_imsi	varchar
		calling_number	varchar
		dialed_digits	varchar
			callee_substr substring 1 4, callee_
			callee_anysubstr any substring 8, ca

NOTE: The user created the second filter for `calling_number` without providing a filter name. The name defaulted to the name of the filtered column.

Figure 2-2: Right Side of Output

column_filter	no
callee_substr substring 1 4, callee_trailing trailing 4	
callee_anysubstr any substring 8, calling_number trailing 4, caller_leading leading 3	

NOTES ABOUT CREATING PARTIAL FILTERS

This section covers the following notes:

- “Multiple Filters on the Same Column”, next
- “Keywords”, on page 29

Multiple Filters on the Same Column

Several filters can be created on the same column, as illustrated above in [System Tables for Column Filters](#). When a query is run against a column with multiple filters, the SQL engine chooses the best filter that is fully contained by the string.

For example, assume that a column has three `LEADING` filters: `LEADING 8`, `LEADING 11`, and `LEADING 15`. In a query that is matching a 10-character string, the SQL engine uses the `LEADING 8` filter to evaluate the first 8 characters of the string.

Keywords

The following words have been added to the list of SQL keywords:

- ANY
- LEADING
- SUBSTRING
- TRAILING

None of these words can be used for table or column names.

If existing customers have tables or columns that use these words they will have to change them.

PERMISSIONS FOR CREATING COLUMN FILTERS

To create column filters on a table, you must have `sls.create` permission in the EDW instance, and you must have `sls.namespace` permission on the namespace where the table is located.

Syntax for Dropping Column Filters

The syntax for dropping column filters differs depending on whether you name the filter in the command.

DROPPING A NAMED FILTER

```
DROP FILTERS [<filter_name>] ON [<namespace>.<table_name>]
;
```

DROPPING FILTER(S) BY COLUMN NAME

```
DROP FILTERS ON [<namespace>.<table_name>]
{ (<column_name>)[, (<column_name>)[...]] | ALL }
;
```

NOTE:

- To drop a column filter, specify either the filter name or the column(s) that have filters you want to drop.
- When you list columns explicitly, only the columns you specify lose their filters; filters on other columns remain in place.
- Specify `ALL` to drop all column filters in the table.
- When you drop column filters from a table, you must identify fully the namespace in which the table is located. You can specify the namespace explicitly in the `DROP FILTERS ON` statement, or you can specify it with the `--namespace` option of the `atquery` command.

PERMISSIONS FOR DROPPING COLUMN FILTERS

To drop column filters from a table, you must have `sls.drop` permission in the EDW instance, and you must have `sls.namespace` permission on the namespace where the table is located.

Defining Views with SenSage AP SQL

Views let people see log events in tables in different ways. Views do not store log events themselves; they show selected columns and rows from other tables and views. You can use `SELECT` statements to query views in the same way you query tables.

This section describes these topics:

- [“Syntax for Creating Views”, next](#)
- [“Declaring the Columns in a View”, on page 32](#)
- [“Selecting a Subset of Table Columns or Rows for a View”, on page 34](#)
- [“Specifying and Querying Against Time Ranges in a View”, on page 35](#)
- [“Using Processing Directives when Creating a View”, on page 35](#)
- [“Declaring a Union of Tables in a View”, on page 35](#)
- [“Syntax for Renaming Views”, on page 36](#)
- [“Syntax for Dropping Views”, on page 36](#)
- [“Retrieving Information About a View”, on page 37](#)

NOTE: The term “table” means generally “tables and views” throughout SenSage AP documentation. Where the term applies to tables only, the restriction is mentioned specifically.

Syntax for Creating Views

```
CREATE [OR REPLACE] VIEW [<namespace>.<view_name> [ ( <column_list> ) ] AS
    <select_subclause>
;
```

The names of views follow the rules for the names of tables. For more information, see [“Specifying Names for Tables and Columns”, on page 24](#).

CREATING OR REPLACING A VIEW

If you create a view that already exists, an error occurs. Sometimes you want to create a view even if one by that name already exists. Use the `OR REPLACE` keyword phrase in conjunction with the `CREATE` keyword to cause the EDW to create a new view if one does not yet exist or to ignore a duplicate view error and replace the current view with a new definition. For an example, see [“Declaring Union Views with `_tablematch\(\)`”, on page 36](#).

CREATING VIEWS IN SPECIFIC NAMESPACES

Views in an EDW instance are located within EDW groups called *namespaces*. If you do not specify a namespace when you create a view, it is located in the default namespace established for the EDW instance. At the time you create the view, you can specify the namespace in which you want to create it; if the namespace does not exist, it is created. For information on how to specify a namespace, see [“Specify a Namespace”, on page 96](#) in the *Administration Guide*.

When you create a view, you can specify different namespaces for the view and the underlying table. In other words, you can create the view in one namespace based on a table in a different namespace. You can specify the namespace(s) explicitly in the `CREATE VIEW` statement or you can use the `--namespace` option of the `atquery` command or you can use a combination of these two options. If both table and view are in the same namespace, you can use the `--namespace` option of the `atquery` command to specify a shared namespace. If they are in different namespaces, you must explicitly specify the table’s namespace in the `CREATE VIEW` statement.

IMPORTANT: If you create views in different namespaces that represent data from the same type of log source, Sensage recommends that you name the views identically.

The following `CREATE VIEW` statement illustrates creation of a view whose namespace is different from the view's underlying table:

```
CREATE VIEW newNamespace.myView AS
  SELECT *
    FROM existingNamespace.myTable
  DURING ALL
;
```

Assume the `CREATE VIEW` statement is contained in a file called `myView.sql` and you run the following atquery command to create the view:

```
atquery --namespace=firewalls host02:8072 myView.sql
```

A view named `myView` will be created in the namespace `firewalls.newNamespace`, which selects data from a table named `myTable`, which is located in the `firewalls.existingNamespace` namespace. In this case, the view and its underlying table share the same parent namespace but are located in different subordinate namespaces below the parent. If `firewalls.newNamespace` did not exist before you ran the `CREATE VIEW` statement, it is created by the statement.

Assume next that you run the following query to retrieve data from the view:

```
atquery --namespace=firewalls.newNamespace host02:8072 -e "select count(*) from myView"
```

The query above selects and aggregates data from `firewalls.newNamespace.myView`. However, `firewalls.newNamespace.myView` pulls the data from `firewalls.existingNamespace.myTable`.

PERMISSIONS FOR CREATING VIEWS

To create views, you must have `sls.create` permission in the EDW instance, and you must have `sls.namespace` permission on the namespaces where you want the view to be created and where its underlying table is located.

Declaring the Columns in a View

The columns in a view are declared by the columns in the target clause of `<select_subclause>`. In the target clause you can:

- specify all columns from the table implicitly with an asterisk (*)
- specify columns from table explicitly with a column list

The data types of columns in the view are the same as the corresponding columns in the table.

SPECIFYING VIEW COLUMNS IMPLICITLY WITH AN ASTERISK

When you specify view columns with an asterisk (*) in the `SELECT` subclause, the view has the same set of columns as the underlying table. Their names and data types are identical to those in the table.

For example, assume a table named `myTable` has columns `ts`, `ClientDNS`, `Method`, and `URL`. The following `CREATE VIEW` statement creates a view named `myView` with the same set of columns.

```
CREATE VIEW myView AS
  SELECT *
```

```
FROM MyTable
DURING ALL
;
```

When the schema of the underlying table changes, the columns in a view that you declared with an asterisk in the target clause always conform to the underlying table.

SPECIFYING VIEW COLUMNS EXPLICITLY WITH A COLUMN LIST

When you specify view columns with an explicit column list, the view has only the listed columns from the underlying table; other columns in the table are excluded from the view. The data types of view columns are identical to the corresponding columns in the table, but you can rename the view columns with the `AS` modifier keyword.

IMPORTANT:

- You should include the `ts` column in *every* view definition. A view that does not include this column would return the entire table. Most likely, such a view would fill your disk with unwanted information.
- When the specified view columns are identical to the table columns, the `AS` keyword is optional. When a specified view column is an expression, the `AS` keyword is required. You can use any valid `SELECT` statement expression to create a view.

For example, assume a table named `myTable` that has columns `ts`, `ClientDNS`, `Method`, and `URL`. The following `CREATE VIEW` statement creates a view named `myView` with the same set of columns, but applies the `_strlink` function to the `URL` column. Because the view columns include an expression, that column must be renamed. Because the `ClientDNS` column is named identically in the view as in the table, the renaming of this column is optional.

```
CREATE VIEW myView AS
  SELECT ts, ClientDNS AS Client, Method, _strlink(URL, 'Click Here') AS
         anchor_tag
  FROM MyTable
  DURING ALL
;
```

NOTE:

- When you specify a column in the target clause as a computed expression, the data type of the column in the view is based on the data type of the computation.
- If additional columns are added to the underlying table, the additional table columns remain excluded from the view. If a column included in the view is removed from the underlying table, a query-time error states that the column cannot be found.
- When you query on a view in SenSage AP Console, always include the `DURING ALL` clause at the end of the main select. If you do not, the EDW will scan all data in the underlying tables.

RENAMING COLUMNS IN A VIEW

As illustrated above in [Specifying View Columns Explicitly with a Column List](#), you can give the columns in the view different names than they have in the underlying table from which the view selects its data. The new names follow the same rules that apply to table and columns names. For more information, see [“Specifying Names for Tables and Columns”, on page 24](#).

You can rename the view columns either with the `AS` keyword modifier or with a column list. In either case, you must explicitly list the view columns in the target clause of the `SELECT` subclause.

The example above uses the `AS` modifier keyword to rename some of the columns in the view. Alternately, you can specify a `<column_list>` with the column names you want in the view. The number and order of columns in `<column_list>` must match the number and order of columns in the target clause of the `SELECT` subclause. Enclose the list of column names that you declare for a view in parentheses. Separate column names with commas.

For example, the following statement creates a view named `myView` with the columns from `myTable`, and renames them in the view with a column list:

```
CREATE VIEW myView (Timestamp, Client, Method, URL) AS
  SELECT ts, ClientDNS, Method, Url
  FROM MyTable
  DURING ALL
;
```

IMPORTANT: You should not use a `<column_list>` to rename view columns if you use an asterisk (*) in the target clause of the `SELECT` subclause.

If you rename the same column in both the column list and with the `AS` modifier keyword in the target clause of the `SELECT` subclause, the name in the column list takes precedence.

Selecting a Subset of Table Columns or Rows for a View

Generally, you create views that expose only subsets of columns or rows from their underlying tables. You limit the columns in the target clause of the `SELECT` subclause. You limit the rows in the `WHERE` clause of the `SELECT` subclause.

For example, assume you have a table named `syslog` that stores log events from log sources that use the syslog protocol. The table contains log events from a variety of programs, including SSH and HTTP. This table has 7 columns: `ts`, `Hostname`, `Programe`, `ProcessID`, `Message`, `IP`, and `upload_id`. You want to create a view that exposes only three of these columns. Additionally, you want to limit the view to expose only those rows that show the syslog events from SSH.

The following statement creates a view from the `syslog` table that exposes only a subset of the columns and rows.

```
CREATE VIEW syslog_ssh (Timestamp, Host, Message) AS
  SELECT ts, Hostname, Message
  FROM syslog
  WHERE _strstr(_strlowercase(Programe), "ssh") > -1
  DURING ALL
;
```

The `WHERE` clause declares that the view shows only rows from the `syslog` table that contain "ssh" in the `Programe`. The view does not include the `Programe` column, because by its definition the view includes log events from a single program.

For another example, assume you have a table with all the web server log events for a particular region. You could create separate views that select rows only for one office in the region. If you create each office view in a separate namespace, you can give the security teams for each office access to their rows without exposing the rows from other offices.

Specifying and Querying Against Time Ranges in a View

Every view must conclude with the DURING clause. A SELECT statement that specifies a view in the FROM clause requires a DURING ALL clause if it is run from SenSage AP Console.

IMPORTANT: When you create a view, ensure that the DURING clause behaves as you expect. A badly formed DURING clause can cause the query to return incorrect or no results. For more information, see [“Subqueries and Views and the DURING Clause”](#), on [page 51](#).

Using Processing Directives when Creating a View

You can include processing directives, including the declarations for subqueries and Perl subroutines, anywhere within SELECT subclause of CREATE VIEW statements. For example:

```
CREATE VIEW myView AS
  WITH $table_name as "myNameSpace.myTable"
  SELECT *
    FROM $table_name
    DURING ALL
;
```

Subqueries and Perl subroutines are compiled by the EDW at the time you create the view. Subqueries and Perl subroutines execute at the time people query the view.

For more information, see [“Processing Directives”](#), on [page 78](#).

Declaring a Union of Tables in a View

You can declare a view that selects its columns and rows from a union of tables. As with all UNION queries, the SQL query engine places the following restrictions on the UNION ALL clause:

- All the subselects within the SELECT statement must return the same number of target columns.
- Each target column must have the same data type in each subselect, respectively.

DECLARING UNION VIEWS WITH THE UNION ALL CLAUSE

The following statement uses the UNION ALL clause to create a view of the union of the tables syslog_a, syslog_b, and syslog_c:

```
CREATE VIEW myUnionView AS
  SELECT ts, Hostname, Prognome, ProcessID, Message, IP
    FROM syslog_a
    DURING ALL

  UNION ALL

  SELECT ts, Hostname, Prognome, ProcessID, Message, IP
    FROM syslog_b
    DURING ALL

  UNION ALL

  SELECT ts, Hostname, Prognome, ProcessID, Message, IP
    FROM syslog_c
```

```
DURING ALL
;
```

For more information, see [“UNION ALL Clauses”, on page 59](#).

DECLARING UNION VIEWS WITH _TABLEMATCH()

Assume that the namespace `myNameSpace` contains the tables `syslog_a`, `syslog_b`, and `syslog_c`. The following statement uses the `_tablematch()` function to create or replace a view of the same union as the previous example:

```
CREATE OR REPLACE VIEW myUnionView AS
  SELECT ts, Hostname, Progame, ProcessID, Message, IP
     FROM @_tablematch( "syslog_.*", "myNameSpace" )
  DURING ALL
;
```

The tables included in the union are computed by `_tablematch()` when people run queries against the view, not when you create the view. If you add table `syslog_d` to `myNameSpace` after you create the view, the view exposes rows from all tables that begin with `"syslog_"`.

For more information, see [“_tablematch\(\)”, on page 110](#).

Syntax for Renaming Views

```
RENAME VIEW [<old_namespace>.]<old_view_name> TO
  [<new_namespace>.]<new_view_name>
;
```

IMPORTANT: When you rename a view, other views and reports that depend on it may no longer produce results. The query engine does not generate an error at the time of the change, but does generate an error when a query is run against the renamed view.

PERMISSIONS FOR RENAMING VIEWS

To rename views, you must have `sls.rename` permission in the EDW instance, and you must have `sls.namespace` permission on the namespace where the view you want to rename is located. If renaming the view results in moving it from one namespace to another, you must have `sls.namespace` permission on the target namespace, too.

Syntax for Dropping Views

```
DROP VIEW [<namespace>.]<view_name>
;
```

When you drop a view, you must identify fully the namespace in which the view is located. You can specify the namespace explicitly in the `DROP VIEW` statement, or you can specify it with the `-namespace` option of the `atquery` command.

IMPORTANT:

- When you drop a view, other views and reports that depend on it may no longer produce results. The query engine does not generate an error at the time of the change, but does generate an error when a query is run against the dropped view.
- Dropping a view does not discard any log events from the EDW instance.

PERMISSIONS FOR DROPPING VIEWS

To drop views, you must have `sls.drop` permission in the EDW instance, and you must have `sls.namespace` permission on the namespace where the view you want to drop is located.

Syntax for Transferring Ownership of a View from One User to Another

```
GRANT OWNER ON <view_name> TO <user_name>
```

When a user change roles or is deleted from the system -- for example, because the user moves to another department or leaves the company - all VIEWS owned by the user are disabled and unusable, unless you transfer ownership to another user. A VIEW gets its permissions from its owner. Thus, after ownership is transferred, `view_name` gets its permissions from new owner `user_name`. A VIEW can have only one user; when you transfer ownership to another user, the first user loses ownership. You must have `sls.admin` permissions to execute this command.

Retrieving Information About a View

You can query the EDW to get the following information about each view in your EDW instance:

- query definition
- namespace
- creator

To get this information, you must query the `storage.raw_metadata` system table. Querying system tables is documented in [“System Tables”](#), on page 127 of the *Administration Guide*.

The `storage.raw_metadata` system table contains the same columns as the `storage.metadata` system table, with the addition of the `data` column. The `data` column stores metadata only for those row types that require it. Each view will have at least three rows with values in this column: for the query, the namespace, and the creator. For a multi-host EDW instance, this table returns these three rows for each host in the instance. For information about the `storage.metadata` table, see [“<namespace>.storage.metadata”](#), on page 269.

When you query the `storage.raw_metadata` table for view information, the most relevant columns are `row_type`, `table_name`, and `data`. The following query, which was run from `atquery`, retrieves only these three columns and only those rows with text in the `data` column:

```
select row_type, table_name, data from storage.raw_metadata where data != ""
```

The example output below illustrates view information for a view named `PubsTest`.

```
| Results for SQL file >(standard input)< |
+-----+-----+-----+
| row_type | table_name | data |
|          |            |      |
| (varchar) | (varchar) | (varchar) |
|          |            |      |
+-----+-----+-----+
|VIEW_QUERY    | PubsTest | CREATE VIEW Pubs.PubsTest (Timestamp, Client, Method,
URL) AS\n SELECT ts, ClientDNS, Method, Url\n FROM  webserv DURING ALL\n |
|VIEW_NAMESPACE| PubsTest | myNamespace |
|
```

```
|VIEW_CREATOR|PubsTest|administrator  
|  
|+-----+-----+-----+-----+
```

NOTE: When you rename a view, you change only its name and not the statement that defines it. Therefore, if you query the `storage.raw_metadata` system table on a view that has been renamed, the `table_name` column in the query results displays the new name, but the view-creation statement in the `data` column still specifies the name used when the view was created. You can compare the name of the view in the `table_name` column with the name specified in the `CREATE VIEW` statement to determine whether a view has been renamed.

The example output below illustrates view information for a renamed view. The output displays information about the `PubsView` view. As shown in the `data` column, this view was originally named `PubsTest`.

```
| Results for SQL file >(standard input)< |
+-----+-----+-----+
| row_type | table_name | data
|
| (varchar) | (varchar) | (varchar)
|
+-----+-----+-----+
| VIEW_QUERY      | PubsView | CREATE VIEW Pubs.PubsTest (Timestamp, Client, Method,
URL) AS\n SELECT ts, ClientDNS, Method, Url\n FROM  webserv DURING ALL\n |
|
| VIEW_NAMESPACE | PubsView | myNamespace
|
| VIEW_CREATOR   | PubsView | administrator
|
| +-----+-----+-----+

```

INTERNATIONAL SUPPORT IN THE EDW

The EDW supports the storage of *international characters*, which are characters other than those in the U.S. English alphabet and some of its punctuation marks. For example, the Spanish word “España” contains the international character “ñ”. The EDW stores all text values with UTF-8 character encoding to ensure that international characters are stored and queried successfully.

This section describes these topics:

- “Character Sets”, next
- “Character Encoding Schemes”, on page 39
- “Character Encoding and Computer Fonts”, on page 40

Character Sets

A *character set* is a fixed set of characters and symbols stored in computer systems and transmitted across computer networks. The fixed set of characters is sometimes referred to as the *character repertoire* of a character set. For example, the character repertoire of the ASCII character set is the 26 upper- and lower-case letters of U.S. English, the numerals 0-9, some punctuation marks, and special symbols.

To facilitate storage and transmission, each member of a character repertoire is assigned a unique decimal number. This number is sometimes referred to as the *code position* or *code point* for a character. For example, the code point for the letter “A” in the ASCII character set is 65. The code point for “B” is 66, and so on.

Character sets are often represented in tables. Thus, a code point is the position of a character within a character-set table. The following table shows a portion of the ASCII character-set table.

Code Point	Character Name	Graphic Representation
62	Greater-than Symbol	
63	Question Mark	
64	Commercial At	
65	Upper-case “A”	
66	Upper-case “B”	
67	Upper-case “C”	
68	Upper-case “D”	
69	Upper-case “E”	

Character Encoding Schemes

A *character encoding scheme* is the method by which the characters in a particular character set are represented with bit patterns for storage in computers and transmission across computer networks. A single character set can be represented by different encoding schemes with different bit patterns.

Encoding schemes are often characterized by the way in which their bit patterns represent characters:

- **Single-byte encoding schemes**—each character in the character set is represented by a unique pattern of bits within a single byte. The character sets of single-byte encoding schemes are limited to 255 unique characters and symbols, excluding bytes with all bits set to zero. ISO 8859-1 is an example of a single-byte encoding scheme.
- **Double-byte encoding schemes**—each character in the character set is represented by a unique pattern of bits in a two-byte “word.” The character sets of double-byte encoding schemes are limited to roughly 65,000 unique characters and symbols. Unicode character encoding is an example of a double-byte encoding scheme. Unfortunately, as an encoding scheme, it cannot represent all the characters in the ever-expanding Unicode character set.
- **Variable-width encoding schemes**—each character in the character set is represented by a unique pattern of bits in a sequence of bytes. The character sets of variable-width encoding schemes are essentially unlimited in terms of the number of unique characters and symbols that can be represented. Some characters are represented by single-byte bit patterns, some by two-byte bit patterns, and so on. UTF-8 character encoding is an example of a variable-width encoding scheme that can represent all the characters in the Unicode character set and any additional characters that are added in the future.

Encoding Schemes Used by the EDW

There are too many encoding standards in use to enumerate them all. The ones that follow are relevant for understanding how the EDW stores, processes, and exports international character data.

Encoding Standard	Encoding Characteristics
ASCII / ISO 646	7-bit encoding, with 128 code points; a fixed-width encoding scheme with characters represented in computer systems by single, 8-bit bytes. ISO 646 encoding can represent the upper- and lower-case letters of the English alphabet, as well as a few English punctuation marks and special symbols. NOTE: In 8-bit byte streams, ASCII encoding generally sets high-order bits to 0. An ASCII byte stream is indistinguishable from an UTF-8 byte stream.
ISO 8859-1	8-bit encoding, with 255 code points; a fixed-width encoding scheme with characters represented by single, 8-bit bytes. ISO 8859-1 encoding can represent the Latin alphabet and many Western European variants. It cannot represent international characters from other languages, such as Cyrillic languages, Arabic, or Japanese. NOTE: When an ISO 8859-1 byte stream encodes only English text, the stream is indistinguishable from an ASCII or a UTF-8 byte stream.
Unicode / ISO 10646	A 16-bit encoding, with tens of thousands of code points; a fixed-width encoding scheme with characters represented by single, 16-bit bytes. ISO 10646 can represent characters from many European and non-European languages but has some limitations with Asian languages; although it has roughly 60,000 code points, the Unicode character set has 90,000 characters in its repertoire that need representation in computers.
UTF-8	Variable-width encoding scheme for Unicode, where the high-order bits of the first bytes of characters indicate how many of the bytes that follow are part of the same character. For example, if the high-order bit of the first byte in a character is 0, none of the following bytes are part of the character—it is a single-byte character. NOTE: When an UTF-8 byte stream encodes only English text, the stream is indistinguishable from an ASCII byte stream.

The EDW stores all character data with UTF-8 character encoding. For character data in the EDW that contains only English text, the character encoding is identical to ASCII encoding.

Character Encoding and Computer Fonts

Character encoding determines how computers store human-readable text internally and transmit it across computer networks. At some point, people want to read the text that computers store and transmit. *Computer fonts* are typefaces that determine how computer screens and printers display human-readable text from the machine-readable text inside computers.

For example, the first letter in the English alphabet is the letter “A”. Depending on the computer font used to display an “A”, it may appear as A.

Computer fonts are designed to display specific character sets or subsets of a character set. For example, most computer fonts can display the ASCII character set. In contrast, there are no computer fonts that can display all the characters in the Unicode character set. With Unicode, fonts are generally designed for a linguistic subset, such as Japanese.

Sometimes a block of text contains characters that a computer font is not designed to display. For example, there might be some Japanese characters in a block of English text. Generally, if a computer font cannot display a particular character, it displays a placeholder character instead, such as hollow rectangle. If you see these placeholder characters, you need to install or activate a different font. Consult the documentation for your computer or printer to learn how.

EDW SYSTEM TABLES

You can query the EDW system tables to get information about your SenSage AP system, which includes information such as the state of an EDW instance or all defined roles and which users have been assigned to them. When you query for this information, the EDW dynamically formats the result data as a standard table. For a listing of system tables, see [Appendix D: SenSage AP System Tables](#).

SenSage AP SQL

SenSage AP SQL, also known as AP SQL, is a special implementation of standard SQL that is optimized for event-log analysis. This chapter describes the SELECT statement.

IMPORTANT: Other statements found in standard SQL, such as UPDATE or DELETE, are not supported in AP SQL to avoid the risk of evidence tampering.

This chapter includes the following sections:

- “Overview of AP SQL SELECT Statements”, next
- “Target Clauses”, on page 45
- “FROM Clauses”, on page 47
- “DURING Clauses”, on page 49
- “WHERE Clauses”, on page 51
- “GROUP BY Clauses and Aggregation Queries”, on page 52
- “SLICE BY Clauses”, on page 56
- “HAVING Clauses”, on page 57
- “ORDER BY Clauses”, on page 58
- “UNION ALL Clauses”, on page 59
- “Data Types”, on page 60
- “Data Source Expressions”, on page 65
- “Data Processing Expressions”, on page 68
- “Processing Directives”, on page 78
- “Working with Lists”, on page 89

OVERVIEW OF AP SQL SELECT STATEMENTS

This section describes these topics:

- “Basic SELECT Syntax”, next
- “Keywords and Clauses Required in SELECT statements”, on page 44
- “About Tables and Namespaces”, on page 44

Basic SELECT Syntax

In AP SQL, a SELECT statement has the following, basic syntax:

```
SELECT [ALL|DISTINCT] [{FIRST|TOP|LAST|BOTTOM} n ]
      <expression>[AS <target>][, <expression>[AS <target>][...]]
FROM <table_specification>
[WHERE <conditional_expression>]
[GROUP BY [<expression>|<target>[, <expression>|<target>[...]]] [IN n PASSES]
[SLICE BY <conditional_expression>]
[HAVING <conditional_expression>]
[ORDER BY [<expression>|<target>[, <expression>|<target> [...]]]
[DURING {ALL|<start_time>, <end_time>[ OR <start_time>, <end_time>[ ...]]}]
;
```

Significant Terms

- Capitalized words along the left margin are called *keywords*. They have special meaning and introduce *clauses* within a SELECT statement.
- Clauses can include expressions, modifier keywords, and identifiers.
- An *identifier* is a reference to the name of such objects as a table, view, column, macro, or a target in a SELECT clause; typically the name is restricted to an alphanumeric character and an underscore.
- Every table column is defined with a data type, which include varchar, int32, and boolean. The term for a data type and its value is *literal*. Examples of literals are:
 - **string literal**—"This text is a string literal."
 - **numeric literals**—27 and 4.50 and -9
 - **boolean literals**—T, F, TRUE, FALSE, 1, 0

NOTE: Clauses must appear in the order shown above. SELECT statements end with semicolons (;).

Keywords and Clauses Required in SELECT statements

The following are required in SELECT statements:

- The **SELECT** keyword, which begins the statement.
- The target clause, which lists the columns in result sets returned by the statement.
- The **FROM** keyword, followed by a specification of the table to query.
- The **DURING** keyword, followed by the time frames to include in result sets returned by the statement.
- A terminating semi-colon (;).

The following example shows a SELECT statement with only the required parts:

```
SELECT *  
  FROM mytable  
  DURING ALL  
;
```

About Tables and Namespaces

The EDW organizes tables into a hierarchy of *namespaces*. The namespace hierarchy is similar to a hierarchical file system, with namespaces corresponding to directories or folders, and tables corresponding to files or documents. Like directories in file systems, one namespace can contain another. Two namespaces are separated by a period. For example, `ns1.ns2.ns3.mytable` references a different table than `ns1.ns2.mytable`.

The default namespace is `default`. To specify a table in a namespace within the default, you must explicitly specify the full namespace. The following example specifies a table whose namespace is within the default namespace:

```
default.ns1.ns2.ns3.mytable
```

In contrast, the following example specifies a table in a namespace that descends directly from the root of the hierarchy instead of from the default namespace.

```
ns1.ns2.ns3.mytable
```

Use namespaces to organize your tables in a meaningful way. For example you can organize by `company.division.department`, or by user, or application, or other structure.

TARGET CLAUSES

SELECT statements begin with a *target* clause, which lists the columns returned in the result sets of queries. Target clauses follow immediately after the SELECT keyword.

TIP: When using expressions, selecting a subset of columns generally improves performance.

This section describes these topics:

- [“Computed expressions in Target Clauses”, next](#)
- [“Named Targets”, on page 45](#)
- [“Invisible targets”, on page 46](#)
- [“DISTINCT Modifier Keyword”, on page 46](#)
- [“FIRST and LAST Modifiers”, on page 46](#)

Computed expressions in Target Clauses

Target clauses include expressions that may or may not mention columns in the table specified in the FROM clause. Target expressions can include math, string, comparison, and logic operators, as well as functions.

For example, the following SELECT statement returns a result set with a single column that contains values from the `ts` column formatted with the `_timef()` function.

```
SELECT _timef("%m/%d/%Y", ts)
FROM example_webserve_100
DURING ALL
;
```

Named Targets

The SQL query engine chooses default names for target columns in result sets, based on their expressions. You can specify the names for target columns with the AS modifier keyword.

For example, the following SELECT statement returns a result set with one column that has “Date” as its name.

```
SELECT _timef("%m/%d/%Y", ts) AS Date
FROM example_webserv_100
DURING ALL;
```

Invisible targets

Sometimes, it's helpful to include a target column for use in query processing that you do not want included as a column in the result set. Beyond convenience, this also saves client-server bandwidth for large result sets. Remove such target columns with the AS modifier keyword and specify a name with two underscores (__) as a prefix.

For example, the following SELECT returns a result set with one column of client-machine DNS names, sorted by timestamp, even though timestamp is not included in the result set.

```
SELECT ClientDNS, ts AS __hidden
FROM example_webserv_100
ORDER BY 2
DURING ALL;
```

DISTINCT Modifier Keyword

Sometimes, a query produces lots of identical records in the result set. If you would like to keep only the unique records, include the DISTINCT modifier keyword after the SELECT keyword that begins the statement or subquery.

For example, the following SELECT statement returns a result set with every unique DNS address seen in a given web log.

```
SELECT DISTINCT ClientDNS
FROM example_webserv_100
DURING ALL;
```

IMPORTANT: The DISTINCT keyword causes the SQL query engine to consume a lot of memory. Be careful about using DISTINCT in SELECT statements on tables with lots of rows. To enhance performance, Hexis Cyber Solutions recommends that you use GROUP BY instead of DISTINCT. For example, the following query returns the same results as the example above, but in less time:

```
SELECT ClientDNS
FROM example_webserv_100
GROUP BY 1
DURING ALL;
```

FIRST and LAST Modifiers

Often, only the first or last few rows of a query output are desired in the results set. To limit the output of a query to the rows at the top or bottom of the results, include FIRST or LAST and the number of rows, between the SELECT keyword and the target clause.

As examples:

```
SELECT FIRST 10 ClientDNS, RespSize
FROM example_webserv_100
DURING ALL;
```

```
SELECT LAST 10 ClientDNS, RespSize
FROM example_webserv_100
DURING ALL;
```

When you combine DISTINCT with FIRST or LAST, the DISTINCT modifier keyword is processed to derive an intermediate result set. The SQL query engine then applies FIRST or LAST to derive the final result set.

NOTE: The TOP and BOTTOM modifier keywords are synonyms for FIRST and LAST.

FROM CLAUSES

The FROM clause in a SELECT statement specifies the table to be queried. Unlike standard SQL, in which FROM clauses can include a list of tables, SQL SELECT statements generally only permit one table to be specified. Joins are not supported by SQL.

This section describes these topics:

- [“Syntax of FROM Clauses”, next](#)
- [“Table Specifications in FROM Clauses”, on page 47](#)
- [“Example FROM Clauses”, on page 48](#)
- [“Including Rows from Bad Loads”, on page 49](#)

Syntax of FROM Clauses

The FROM clause has the following syntax:

```
FROM <table_specification> [<alias_name>] [{INCLUDE_BAD_LOADS}]
```

The *<table_specification>* is an expression that evaluates to the name of a table in the EDW. For more information, see [“Table Specifications in FROM Clauses”, next](#).

As an option, the FROM clause lets you use *<alias_name>* to specify an abbreviated name for the table. Table names can be quite long when namespaces are included. Use the alias as a table identifier in later lines of the SELECT statement.

As an option, the `{INCLUDE_BAD_UPLOADS}` modifier keyword instructs the SQL query engine to include rows that were loaded by failed load operations. For more information, see [“Including Rows from Bad Loads”, on page 49](#).

NOTE: The curly braces ({}) are part of the `{INCLUDE_BAD_UPLOADS}` keyword.

Table Specifications in FROM Clauses

Table specifications can be identifiers, literals, or expressions that evaluate to the name of a table in the EDW. For example, the specification can have one of these forms:

- The name of a table, including the namespace if there is one:

```
FROM namespace1.mytable
```

- A string expression that evaluates to the name of a table:

```
WITH $namespace = "myNamespace"  
...  
FROM $namespace + "." + "example_webserv_100"  
...
```

TIP: To learn how to include data from several tables in a single result set, see [“UNION ALL Clauses”](#), on page 59 and [“Defining Views with SenSage AP SQL”](#), on page 30.

Example FROM Clauses

The examples that follow show different ways to specify the table FROM clauses:

- [Identifiers as the Table Name](#)
- [String Expressions as the Table Name](#)
- [Expression Macros as the Table Name](#)

The queries in the examples below all return the distinct IP addresses from the `example_webserv_100` table.

Identifiers as the Table Name

The simplest table specification in a FROM clause is the table name itself, used as an identifier without enclosing quotation marks.

```
SELECT distinct ClientIP  
FROM example_webserv_100           -- note the table name is an identifier  
DURING all;
```

String Expressions as the Table Name

The table specification in a FROM clause can be a string concatenation expression that resolves to the name of a table. The following example uses a concatenation expression.

```
SELECT distinct ClientIP  
FROM 'example' + '_webserv_100'  
DURING all
```

When the query engine executes the statement, it queries the `example_webserv_100` table. Any expression or function that returns a string value can be used as the table expression.

Expression Macros as the Table Name

Often there are many tables that share a common schema. Use an expression macro as the table expression so that you can write a common SELECT statement for all tables that share the

common schema. The expression macro provides a default table name, which can be overridden when the query engine executes the statement.

```
WITH $source as 'example_webserv_100'

SELECT distinct ClientIP
FROM $source
DURING all;
```

For more information, see [“Expression Macros”, on page 79](#).

Including Rows from Bad Loads

The EDW treats each attempt to load log-entries into a table as a single unit of work. If a load operation fails, any rows loaded into the table before the failure occurs are considered suspect. When you query a table, rows from failed loads are excluded by default. If you want rows from bad loads included in query processing, use the `{INCLUDE_BAD_UPLOADS}` modifier keyword at the end of the FROM clause.

For example:

```
SELECT distinct ClientIP
FROM example_webserv_100 {INCLUDE_BAD_UPLOADS}
DURING all;
```

NOTE:

- Include the curly braces (`{}`).
- A successful load does not necessarily insert every row from the source log file. Typically log files contain data that cannot be parsed and loaded; however, bad data does not prevent these files from loading successfully. A failed load is caused by a protocol error, such as not receiving all the expected data or a load cancellation.
- If a load and a query operation are running against the same table at the same time and the load completes before the query, the query may return none, some, or all of the data from the load (depending on when the query is executed).

For related information, see:

- [“Tracking Uploads in the EDW”, on page 72](#) of the *Administrator Guide*
- ["system.upload_info" and "system.raw_upload_info", on page 268](#)
- [“Checking Locking Status and IPC State Information”, on page 177](#) of the *Administrator Guide*

DURING CLAUSES

Most queries of event-log data involve a specific time frame, whether it's “yesterday”, “last year,” or “1 hour ago.” To facilitate these types of queries, SQL supports the proprietary DURING clause, which allows you to specify the time frame for the query.

The DURING clause works with the timestamp column named `ts`, which is mandatory in every event-log table. The SQL query engine matches the time frame you specify in the DURING clause

with values in the `ts` column of the table. Only rows that match the time frame are included in the results set.

Alternative Formats for DURING Clauses

The DURING clause has several forms:

- **DURING ALL**

The simplest form of the DURING clause uses the ALL modifier keyword to specify the time frame. It specifies the entire time frame represented in the table. In other words, the SQL query engine ignores values in the `ts` column when it determines which rows to include in the result set.

- **DURING** *<lower_bound>*, *<upper_bound>*

The DURING clause takes two expressions, separated by a comma, that evaluate to timestamps. Together, the two expressions restrict query results to a single time frame. Only rows with values in the `ts` column that fall between the bounding timestamps are included in the results set. Specify the *<upper_bound>* and *<lower_bound>* with expressions that evaluate to timestamps. For example, you can use functions like `_time()`, `_strptime()`, and so on.

- **DURING** [*<lower_bound>*, *<upper_bound>*] **OR DURING** [*<lower_bound>*, *<upper_bound>*] **OR DURING** ...

The DURING clause lets you specify a series of time frames. Enclose each time frame within square brackets, and introduce each additional time frame with OR DURING.

For example, the following statement specifies a single time frame. The SELECT statement returns the values of the `ClientDNS` and `RespSize` columns from rows in the `example_webserve_100` table where the `ts` column is later than or equal to midnight, February 1, 2002, and earlier than or equal to the last second of March 31, 2002.

```
SELECT ClientDNS, RespSize
FROM example_webserve_100
DURING time('Feb 01 00:00:00 2002'), time('Apr 01 00:00:00 2002');
```

NOTE: The end time above includes the first microsecond of April 1.

Timestamp Precision in DURING Clauses

The DURING clause compares specified timestamps with the timestamps in the `ts`. The precision of `ts` columns is microseconds, provided the source log entries carry timestamps with that precision.

You can query time frames with microsecond precision, depending on the format of the varchar value you provide to the `time()` function. The examples in this chapter use the Unix date format, which specifies timestamps with a precision of seconds. Use the ISO 8601 format to specify timestamps with a precision of microseconds.

For more information, see [Chapter 4: SenSage AP SQL Functions](#).

Subqueries and Views and the DURING Clause

A view or subquery whose FROM clause specifies table(s) instead of views or subqueries must include a DURING clause. A view or subquery whose FROM clause specifies another view or subquery instead of a table, does not require a DURING clause.

NOTE: Ensure that the DURING clause behaves as you expect. A badly formed DURING clause can cause the query to return incorrect results. The sections below describe considerations when defining the DURING clause.

Specifying DURING ALL in the View

If a view has specified `DURING ALL` as its time range and you query against the view, always include the `DURING ALL` clause at the end of the main select.

- If you include the `DURING ALL` clause in the main select, the run-time user can limit the time range through the Date Criteria field.
- If you do not include the `DURING ALL` clause in the main select, the EDW will scan all data in the tables from which the view selects its data.

SPECIFYING ACTUAL TIME RANGE IN THE VIEW—OVERLAPPING TIME RANGES

If the `DURING` clause in the view specifies an actual time range, and the statement that selects from the view specifies a different but overlapping time range, the query engine resolves the time period to the intersection of the two time ranges.

In other words, if the view specifies a time range between January 1 and June 30 of last year and the `SELECT` statement specifies a time range between April 1 and August 30 of the same year, the query engine resolves the time range to April 1 through June 30.

NOTE: The results are the same when the `SELECT` statement is run from SenSage AP Console. If the time range specified in the Date Criteria field overlaps the time range specified in the view, the query engine resolves the time range to the intersection of the two time ranges.

SPECIFYING ACTUAL TIME RANGE IN THE VIEW—NON-OVERLAPPING TIME RANGES

If the `DURING` clause in the view specifies an actual time range, and the statement that selects from the view specifies a different and *non*-overlapping time range so that the time ranges do *not* intersect, the query returns no rows. This behavior is identical to a query whose `WHERE` clause evaluates to false.

In other words, if the view specifies a time range between January 1 and June 30 of last year and the `SELECT` statement specifies a time range between July 1 and December 30 of the same year, the query returns no rows.

WHERE CLAUSES

Use the `WHERE` clause to describe the characteristics of rows to be included in the result set. The `WHERE` clause takes a *conditional expression* that specifies which rows to include. Conditional expressions use the Boolean logic operators AND, OR, and NOT, and comparison operators, such as BETWEEN, IN, LIKE, <, <=, =, >, and >=.

NOTE: Column filters enhance performance only for queries that use the equals (= or ==) or not equals (<> or !=), LIKE operators, and string functions. A column filter does not enhance performance with other operators and functions, such as less than or greater than (<, >, >=, <=), _min(), _max(), _upper(), _lower(), or IN. For more information, see [“Defining Column Filters with SenSage AP SQL”, on page 25](#).

For example, the following SELECT statement uses a WHERE clause to ensure that only rows with RespSize greater than zero are included in the result set.

```
SELECT ClientDNS, RespSize
FROM example_webserv_100
WHERE RespSize > 0
DURING ALL;
```

You can also use a subquery with the IN operator inside of a WHERE clause. See [“Using Subqueries with the IN Operator”, on page 71](#).

For example, the following SELECT statement uses a WHERE clause to ensure that only rows with RespSize greater than zero are included in the result set.

```
SELECT ClientDNS, RespSize
FROM example_webserv_100
WHERE RespSize > 0
DURING ALL;
```

GROUP BY CLAUSES AND AGGREGATION QUERIES

Aggregate queries use a GROUP BY clause, a HAVING clause, or contain an aggregate function. The semantics of aggregate queries differ from ordinary queries in an important way.

Generally, result sets contain one row for each row in the queried table that meets the selection criteria of the WHERE clause and which falls within the specified time frame of the DURING clause. In aggregation queries, rows are first collected into groups and the result will contain one row for each group of rows.

The SQL query engine performs aggregation before arranging the aggregated result rows in the sequence specified in the ORDER BY clause. When aggregation queries contains the FIRST or LAST keyword, only the specified first or last aggregation rows are returned in the final result set.

This section describes these topics:

- [“Aggregation Partitioning”, next](#)
- [“Aggregation Functions”, on page 54](#)
- [“Multi-column GROUP BY”, on page 55](#)
- [“IN n PASSES”, on page 55](#)
- [“SLICE BY Clauses”, on page 56](#)

Aggregation Partitioning

Usually, the GROUP BY option is used to specify how columns are partitioned. In the following example, the GROUP BY 1 option instructs the SQL query engine to collect all the rows from the

specified time frame, which are separated into groups, one for each unique value of the first expression in the target specification.

```
SELECT ClientDNS, max(RespSize)
  FROM example_webserv_100
  GROUP BY 1
  DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

The final result will contain `ColumnA` and the largest value of `RespSize` in each group

```
+-----+
| Results for SQL file >example-howto-sql-11.sql< |
+-----+-----+
| ClientDNS | max |
| (varchar) | (int32) |
+-----+-----*
output is post-sorted
+-----+-----*
|163.191.183.106| 7121|
|194.114.63.8 | 17252|
|199.166.228.8 | 7121|
|203.164.82.149 | 17252|
|208.147.25.154 | 37361|
|209.102.202.154| 37361|
|212.242.116.219| 0|
|212.9.190.79 | 17252|
|216.239.46.200 | 12203|
|216.239.46.58 | 0|
|216.239.46.79 | 0|
|65.184.59.157 | 216|
|65.194.51.154 | 37361|
|66.127.84.10 | 12203|
+-----+-----*
```

GROUP BY arguments can also be expressions. The above example could have been written as GROUP BY `ClientDNS`.

Occasionally, it is useful to compute an aggregate expression over a single group representing all the values, in which case no GROUP BY is needed. For example, the following query will return at most one row containing the total number of records in the time range in which `RespSize` is positive.

```
SELECT count(*)
  FROM example_webserv_100
  WHERE RespSize > 0
  DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

```
+-----+
| Results for SQL file >example-howto-sql-12.sql< |
+-----+-----+
| count |
| (int64) |
+-----*
output is post-sorted
+-----*
| 96 |
+-----*
```

Aggregation Functions

The previous examples demonstrate the use of aggregate functions. Unlike ordinary functions that return one value for each row, aggregate functions return one value for each group of rows. These are the aggregate functions you can use in target clauses.

Function	Returns
<code>min(x)</code>	Smallest value of <i>x</i> in the group
<code>max(x)</code>	Largest value of <i>x</i> in the group
<code>count(*)</code>	Number of rows in the group
<code>sum(x)</code>	Sum of the values of <i>x</i> in the group
<code>avg(x)</code>	Average of the values of <i>x</i> in the group
<code>median(x)</code>	Median of the values of <i>x</i> in the group
<code>_first(x)</code>	First of the values of <i>x</i> in the group seen by the system
<code>_last(x)</code>	Last of the values of <i>x</i> in the group seen by the system

NOTE: For more details about each of these aggregation functions, see [“Aggregation Functions”, on page 54](#).

The following restrictions apply to aggregate functions:

- They may not appear in the GROUP BY column expressions.
- They may not appear in the arguments to other aggregates (for example, no `min(sum(x))`).
- Not all aggregates support all data types (for example, no `sum(timestamp)`).
- When aggregates are used in a query, then all columns outside the GROUP BY expressions must be contained within aggregates.

The last restriction requires further explanation. Consider the invalid query:

```
-- example of an invalid query
SELECT count(*), ClientDNS
  FROM example_webserv_100
 WHERE RespSize > 0
    DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

Because `count(*)` appears in the expression and no GROUP BY is present, this query will return at most one row for the group consisting of all records. However the second expression in the select semantically represents the multiple values of `ClientDNS`. Note the SQL query engine has no meaningful way of returning two streams of columns with differing numbers of values in each.

Often when a query such as this is encountered, what is intended is usually something like:

```
SELECT count(*), _first( ClientDNS )
  FROM example_webserv_100
 WHERE RespSize > 0
    DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```



```

+-----+
| Results for SQL file >example-howto-sql-14.sql< |
+-----+
| count |      first      |
| (int64)| (varchar)      |
+-----+-----*
output is post-sorted
+-----+-----*
|      96|65.194.51.154|
+-----+-----*

```

Multi-column GROUP BY

The GROUP BY option also allows grouping on a combination of columns. For example, the following query returns the number of records in the specified time frame for each unique combination of the first and second column in the select (in this case `ClientDNS` and `RespSize`).

```

SELECT ClientDNS, RespSize, count(*)
  FROM example_webserv_100
  GROUP BY 1, 2
  DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')

```

```

+-----+
| Results for SQL file >example-howto-sql-15.sql< |
+-----+
| ClientDNS |RespSize| count |
| (varchar) | (int32) | (int64) |
+-----+-----*
output is post-sorted
+-----+-----*
|163.191.183.106|      204|      1|
|163.191.183.106|     7121|      1|
|194.114.63.8   |    17252|      1|
|199.166.228.8  |     7121|     10|
|203.164.82.149 |    17252|      1|
|208.147.25.154 |    37361|      6|
|209.102.202.154|    37361|      1|
|212.242.116.219|        0|      1|
|...
|212.9.190.79   |    12935|      1|
|212.9.190.79   |    17252|      1|
|216.239.46.200 |        0|      1|
|...
|66.127.84.10   |     7121|      2|
|66.127.84.10   |    12203|      1|
+-----+-----*

```

IN *n* PASSES

The GROUP BY clause can cause the EDW to consume a lot of memory on each host in the EDW instance. To reduce this memory burden, include the proprietary IN *n* PASSES subclause at the end of the GROUP BY clause, where *n* is an integer in the range of 2-10. Each pass then uses $1/n$ as much memory as a complete pass but requires *n* times as many scans through the data. For this reason, Hexis Cyber Solutions discourages the use of this feature unless absolutely

necessary for getting queries to complete -- for queries that don't consume lots of memory, it will dramatically reduce performance. Here's an example:

```
SELECT ClientDNS, RespSize, count(*)
  FROM example_webserv_100
 GROUP BY 1, 2 IN 3 PASSES
 DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

```
+-----+
| Results for SQL file >example-howto-sql-16.sql< |
+-----+-----+-----+
| ClientDNS | RespSize | count |
| (varchar) | (int32)  | (int64) |
+-----+-----+-----+
*
output is post-sorted
+-----+-----+-----+
|163.191.183.106|      204 |      1 |
|163.191.183.106|     7121 |      1 |
|194.114.63.8   |    17252 |      1 |
|199.166.228.8  |     7121 |     10 |
|203.164.82.149 |    17252 |      1 |
|208.147.25.154 |    37361 |      6 |
|209.102.202.154|    37361 |      1 |
|212.242.116.219|        0 |      1 |
...
|212.9.190.79   |    12935 |      1 |
|212.9.190.79   |    17252 |      1 |
|216.239.46.200 |        0 |      1 |
...
|66.127.84.10   |     7121 |      2 |
|66.127.84.10   |    12203 |      1 |
+-----+-----+-----+
*
```

SLICE BY CLAUSES

The SLICE BY clause is used with the GROUP BY clause to separate records that would normally be aggregated in a single group into multiple groups. For example, the following SELECT statement collects rows into groups according to the value of `ClientDNS` and creates a distinct group whenever a record with `Url = '/robots.txt'` is seen.

```
SELECT ClientDNS, count(*)
  FROM example_webserv_100
 GROUP BY 1
 SLICE BY Url = '/robots.txt'
 DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

```
+-----+
| Results for SQL file >example-howto-sql-17.sql< |
+-----+-----+-----+
| ClientDNS | count |
| (varchar) | (int64) |
+-----+-----+
*
output is post-sorted
+-----+-----+
|163.191.183.106|      2 |
|194.114.63.8   |      1 |
|199.166.228.8  |     10 |
+-----+-----+
*
```

```
|203.164.82.149 |      1|
|208.147.25.154 |      6|
|209.102.202.154|      1|
|212.242.116.219|      1|
|212.9.190.79   |     37|
|216.239.46.200 |      3|
|216.239.46.58  |      1|
|216.239.46.79  |      1|
|65.184.59.157  |      1|
|65.194.51.154  |      6|
|66.127.84.10   |     29|
+-----+-----*
```

The **SLICE BY** option is commonly used with the `_fifo()` function to split a group of records based on the previous value of an expression. The `_fifo()` function implements a simple first-in-first-out queue and for a given value of `<key>`, `_fifo(<key>, <a>,)` always returns `b` on the first call and on each subsequent call the previous value of `a` is returned. This means the following query will collect the rows into groups according to the value of `ClientDNS` but will create a distinct group whenever `_int32(ts)` increases by more than 10 (that is, more than 10 seconds have elapsed between records).

```
SELECT ClientDNS, count(*)
  FROM example_webserv_100
 GROUP BY 1
    SLICE BY _int32(ts) - _fifo(ClientDNS, _int32(ts), _int32(ts)) > 10
 DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002');
```

```
+-----+-----+
| Results for SQL file >example-howto-sql-18.sql< |
+-----+-----+
| ClientDNS | count |
| (varchar) | (int64)|
+-----+-----*
output is post-sorted
+-----+-----*
|163.191.183.106|      2|
|194.114.63.8   |      1|
|199.166.228.8  |      1|
...
|199.166.228.8  |      1|
|203.164.82.149 |      1|
|208.147.25.154 |      1|
|208.147.25.154 |      1|
...
|65.194.51.154  |      1|
|66.127.84.10   |      1|
|66.127.84.10   |      1|
|66.127.84.10   |     27|
+-----+-----*
```

HAVING CLAUSES

A **SELECT** statement uses the **HAVING** clause to eliminate groups from the result. For example, the following **SELECT** statement:

- collects rows into groups according to the value of the `ClientDNS` column

- counts the records in each group
- eliminates groups in which the sum of the values in the `RespSize` column is greater than 100

```
SELECT ClientDNS, count(*)
  FROM example_webserve_100
  GROUP BY 1
  HAVING sum(RespSize) > 100
  DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

```
+-----+
| Results for SQL file >example-howto-sql-19.sql< |
+-----+-----+
| ClientDNS | count |
| (varchar) | (int64)|
+-----+-----*
output is post-sorted
+-----+-----*
|163.191.183.106|      2|
|194.114.63.8   |      1|
|199.166.228.8  |     10|
|203.164.82.149 |      1|
|208.147.25.154 |      6|
|209.102.202.154|      1|
|212.9.190.79   |     37|
|216.239.46.200 |      3|
|65.184.59.157  |      1|
|65.194.51.154  |      6|
|66.127.84.10   |     29|
+-----+-----*
```

NOTE:

- The column referenced in the HAVING clause does not need to be included in the SELECT clause.
- Although the query must include at least one aggregated value, it does not require a GROUP BY clause.

The HAVING clause is similar to the WHERE clause. Both clauses restrict the values in the final result. The main difference is that the SQL query engine uses the WHERE clause to eliminate rows before it aggregates them, and it uses the HAVING clause to eliminate groups after it aggregates them.

ORDER BY CLAUSES

When executing a SELECT statement without an ORDER BY clause, the SQL query engine returns rows in whatever order they arrive from the storage system. If you include an ORDER BY clause, the engine sorts rows in an intermediate result set before producing the final result set. You provide a list of target columns as part of the ORDER BY clause to instruct the SQL query engine how to sort the results. Target columns can be specified by their ordinal number within the target clause.

For example, the following **SELECT** statement specifies that the result set should be sorted in ascending order of values in the target `ClientDNS` column, which has an ordinal number of 1.

```
SELECT ClientDNS
FROM example_webserv_100
ORDER BY 1
DURING ALL;
```

ORDER BY clauses can have multiple arguments, in which case it sorts the results by the first argument (the major sort field), using the second argument as the tie-breaker, and so on. The arguments in **ORDER BY** clauses can be expressions as well as ordinal numbers. For example:

```
SELECT ClientDNS, ts
FROM example_webserv_100
ORDER BY ClientDNS, 2
DURING ALL;
```

Control the sort order of each argument with the **ASC** or **DESC** modifier keywords. For example:

```
SELECT ClientDNS, RespSize
FROM example_webserv_100
ORDER BY _strlowercase(UserAgent) DESC, 1 ASC
DURING ALL;
```

ORDER BY clauses sort result rows in ascending order by default.

IMPORTANT: To enhance performance for a query that includes more than one subquery, Hexis recommends that you *do not* use **ORDER BY** in the first subquery.

UNION ALL CLAUSES

Generally, a single **SELECT** statement queries one table and generates a results set. Sometimes you want a results set with information queried from several tables, where the tables have the same schema but contain different sets of data.

For example, you may have several different tables that hold weblog entries from different web servers. The **UNION ALL** clause lets you combine queries of tables for each region into a **SELECT** statement that produces a unified result.

```
SELECT ClientDNS, RespSize
FROM example_webserv_100
DURING ALL
```

UNION ALL

```
SELECT ClientDNS, RespSize
FROM example_webserv2_100
DURING ALL
;
```

NOTE: SQL does not support **UNION** or **UNION DISTINCT**. You can eliminate duplicate rows in each subsidiary **SELECT** statement with the **DISTINCT** keyword, but you cannot eliminate rows that are duplicated across the subsidiary results. For information on an alternative solution, see [“Subqueries and UNION ALL Clauses”, on page 85](#).

IMPORTANT: To enhance performance, Hexis recommends that you use ORDER BY in a separate subquery after any UNION ALL queries.

Restrictions on UNION ALL

The SQL query engine places the following restrictions on the UNION ALL clause:

- All the subselects within the SELECT statement must return the same number of target columns.
- Each target column must have the same data type in each subselect, respectively.

Alternative to UNION ALL

AP SQL supports *list expressions* as an alternative way of producing a unified result set from two or more similar tables. For more information, see [“List Expressions and FROM Clauses”](#), on page 92.

DATA TYPES

This section describes the data types that SenSage AP SQL supports:

- “bool”, next
- “float”, on page 61
- “int32”, on page 61
- “int64”, on page 61
- “timestamp”, on page 62
- “varchar”, on page 62
- “inet”, on page 64

bool

The bool data type represents the logical Boolean values “true” and “false.”

The result of `CONVERT(<target_data_type>, <bool_value>)` is as follows:

Target Data Type	Conversion Result
int32	1 if the Boolean value is logically “true”, or 0 if the Boolean value is logically “false”
int64	1 if the Boolean value is logically “true”, or 0 if the Boolean value is logically “false”
float	1 if the Boolean value is logically “true”, or 0 if the Boolean value is logically “false”
timestamp	Not allowed; an exception is raised on attempts to convert bool values to timestamp values
varchar	“true” or “false”, depending on the logical Boolean value
inet	Not allowed; an exception is raised on attempt to convert bool values to timestamp values.

float

The float data type represents fractional quantities and very larger or small numbers. The internal representation is a 64-bit floating-point value in the range of -1.797693e+308 to 1.797693e+308.

The result of `CONVERT(<target_data_type>, <float_value>)` is as follows:

Target Data Type	Conversion Result
bool	Logical “true” if the value is non-zero; logical “false” if the value is 0
int32	The floating-point value converted to the closest integer, without rounding
int64	The floating-point value converted to the closest integer, without rounding
timestamp	The floating-point value, which represents the number of seconds since 1/1/1970 GMT, converted to the number of microseconds since 1/1/1970.
varchar	Formatted floating-point number. If the number of fractional digits exceeds 6, the number is formatted in scientific notation; otherwise, regular numeric formatting is used.
inet	Not allowed; an exception is raised on attempt to convert bool values to float values.

int32

The int32 data type represents signed 32-bit integer values in the range of -2,147,483,648 to +2,147,483,647. The int32 data type is the default for integer quantities. Use the int64 data type for larger integer values.

The result of `CONVERT(<target_data_type>, <int32_value>)` is as follows:

Target Data Type	Conversion Result
bool	Logical “true” if the value is non-zero; logical “false” if the value is 0
int64	The equivalent 64-bit integer value
float	The equivalent floating-point value
timestamp	Positive integer values, which represent the number of seconds since 1/1/1970 GMT, are converted to the number of microseconds since 1/1/1970. If the integer value is too large, the timestamp is set to the maximum value. For negative integer values, the timestamp is set to the minimum value.
varchar	Unformatted integer value
inet	Not allowed

int64

The int64 data type represents large integer quantities. The internal representation is a signed 64-bit integer value in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

The result of `CONVERT(<target_data_type>, <int64_value>)` is as follows:

Target Data Type	Conversion Result
bool	Logical “true” if the value is non-zero; logical “false” if the value is 0
int32	The equivalent 32-bit integer value
float	The equivalent floating-point value
timestamp	Positive integer values represent the number of microseconds since 1/1/1970 GMT and are stored in the timestamp value without conversion. For negative integer values, the timestamp is set to the minimum value.
varchar	Unformatted integer value
inet	Not allowed; an exception is raised on attempts to convert int54 values to inet values

timestamp

The timestamp data type represents date-and-time-of-day values, expressed as the number of microseconds since January 1, 1970, GMT. The latest timestamp that can be represented falls in the year 2038.

The result of `CONVERT(<target_data_type>, <timestamp_value>)` is as follows:

Target Data Type	Conversion Result
bool	Not allowed; an exception is raised on attempts to convert timestamp values to bool values
int32	The number of seconds since January 1, 1970, GMT, if the value is greater than 0; otherwise 0.
int64	The number of microseconds since January 1, 1970, GMT, if the value is greater than 0; otherwise 0.
float	The number of seconds since January 1, 1970, GMT, if the value is greater than 0; otherwise 0.
varchar	An ISO 8601 formatted timestamp. For more information, see “Converting varchar Values to timestamp Values” , on page 63.
inet	Not allowed; an exception is raised on attempts to convert timestamp values to inet values

varchar

The varchar data type represents character data stored with UTF-8 character encoding. This encoding enables the representation of international characters from the Unicode character set.

The maximum length of a varchar value is 2147483646 bytes, which is two bytes fewer than two gigabytes.

The result of `CONVERT(<target_data_type>, <varchar_value>)` is as follows:

Target Data Type	Conversion Result
bool	Logical “true” if the value is “true” or “max”; otherwise, logical “false”
int32	The equivalent 32-bit integer value, if the complete varchar value contains only leading spaces, followed by an optional plus or minus sign, followed by a sequence of decimal numerals with no thousands or decimal separators; otherwise, 0. If the varchar value contains only “min” the int32 value is -2,147,473,647; if the varchar value contains only “max” the int32 value is 2,147,473,647.
int64	The equivalent 64-bit integer value, if the complete varchar value contains only leading spaces, followed by an optional plus or minus sign, followed by a sequence of decimal numerals with no thousands or decimal separators; otherwise, 0. If the varchar value contains only “min” the int64 value is -9,223,372,036,854,775,807; if the varchar value contains only “max” the int64 value is 9,223,372,036,854,775,807.
float	The equivalent floating-point value, if the complete value contains: 1) only leading spaces, followed by an optional plus (+) or minus (-) sign, followed by a sequence of decimal numerals with no thousands and at most one decimal separator; 2) only leading spaces followed by a number expressed in valid scientific notation; otherwise, 0. If the varchar value contains only “min” the float value is -1.797693e+308; if the varchar value contains only “max” the float value is 1.797693e+308.
timestamp	A string containing the ISO 8601 (Time & Date) representation of the value. For example, the following varchar value represents a timestamp for October 10th, 1997, at 6:09 PM, UTC: '1997-10-10T17:18:09.000000Z' For more details, see “Converting varchar Values to timestamp Values”, next .
inet	IP host address that carries the 32- or 128-bit of IP address information in “network order (same as Big-Endian) and optionally, identifies the subnet where the host resides, all in a single field. If the subnet mask is 32 (IPv4) or 128 (IPv6), then the result is host only, If an empty string was converted to inet then the result is an empty string, and if a non-empty string that did not contain a valid IPv4/IPv6 address was converted to an inet, then the result is invalid.

Converting varchar Values to timestamp Values

The rules for converting a varchar value to a timestamp value are as follows:

- If the varchar value is 'min', the timestamp is assumed to be the smallest possible time supported by the system (the first microsecond of 1/1/1970).
- If the varchar value is 'max', the timestamp is assumed to be the greatest possible time supported by the system: 2,147,483,647,000,000 microseconds since 1/1/1970 (which is sometime in 2038-01-19).
- If the varchar value is a legal ISO 8601 timestamp (for example, 1997-10-10T17:18:09.000000Z), the timestamp is assumed to be the corresponding microsecond.
- If the varchar value has the form 'Mon Day Hour:Min:Sec Year' (for example, Oct 10 17:18:09 1997), the timestamp is assumed to be the first microsecond of the corresponding second.
- If the varchar value has the form "MM/DD/YY" (for example, 10/10/97), the timestamp is assumed to be the first microsecond of the corresponding day.

- If the varchar value has the form "MM/DD/YY HH:MM:SS" (for example, '10/10/97 17:18:09') then the timestamp is assumed to be the first microsecond of the corresponding second.
- If the varchar value is a relative time, the timestamp is assumed to be that number of seconds since 1/1/1970.

Varchar Representations of Durations

A duration is a value representing an amount of time. The internal representation is a structure with a string containing a number for the amount of time, followed by a string with one of the following codes that indicates the unit of time:

Unit Code	Unit of Time
usec	Microseconds. For example, 5usec.
msec	Milliseconds. For example, 5msec.
sec	Seconds. For example, 5sec.
min	Minutes. For example, 5min.
hr or hour	Hours. For example, 5hr or 5hour.
day	Days. For example, 5day.
mon	Months. For example, 5mon.
yr or year	Years. For example, 5yr or 5year.

inet

The inet data type is used to represent IPv4 and IPv6 addresses. In addition, when it comes to holding textual representations, the inet data type performs better than the varchar data type of the supporting inet functions.

The inet data type represents an IP host address and optionally identifies the subnet where the host resides, all in a single field. The textual representation of the inet includes the subnet sizes, for example 10.56.127.4.24 or 2002:a509:4482/64.

The subnet identity is represented by the netmask, which is the number of bits in the network part of the address. Note that if the netmask is 32 for IPv4 or 128 for IPv6, then the value indicates a host only and not a subnet. The technical representation of "host only" address can omit the subnet suffix, so that **10.56.127.4/32**, for example, can become **10.56.127.4**.

The following is a description of inet data type structure.:

inet Byte	Description
family	Indicates the type of inet where a value of 4 = IPv4 and 6 = IPv6. Note the meaning of the following other values: <ul style="list-style-type: none"> • 0 = None --an empty string was converted to an Inet • 1 = Invalid --a non-empty, string that did not contain a valid IPv4/IPv6 address was converted to an inet.
subnet	Optionally, indicates the number of bits in the subnet mask, identifying the subnet where the host resides, all in a single field. If the value is 32 for IPv4 or 128 for IPv6, then the value indicates a host-only, not a subnet.

inet Byte	Description
IP value	Carries the 32-or 128-bits of IP address information in "network order" (which is the same as Big-Endian).This means that an inet can be 2, 6 or 18 bytes in length, depending on the value of the 'family' byte.

inet Operators and Functions

The following sections show the operators and functions available using the inet data type.

INET OPERATORS

The following comparison operators in the table below are used to compare two INET values.

- The comparison operators work by first checking the family codes. The INET with a smaller family code value is smaller.
- If the family codes are equal, then compare common subnet prefix bits. The INET with the numerically smaller value is smaller.
- If the common subnet prefix bits are equal, then compare netmask bits of two INETs. The INET with the shorter subnet mask is 'smaller'.
- If the netmask bits are equal, then compare the host address of two INETs. The INET with numerically smaller host address is 'smaller'.

Comparator	Description	Examples
<	Less than	inet '193.169.1.5' < inet '193.169.1.6'
<=	Less than or equal	inet '193.169.1.6' <= inet '193.169.1.6'
==	Equal	inet 192.168.1.5' = inet '192.168.1.5
>=	Greater than or equal	inet '193.169.1.6' >= inet '193.169.1.5
>	Greater than	inet '193.169.1.6' > inet '193.169.1.5'

DATA SOURCE EXPRESSIONS

Data source expressions define the input data used in a SELECT statement.

This section describes the two basic sources of data:

- [“Column Expressions”, next](#)
- [“Literal constants”, on page 66](#)

Column Expressions

A *column expression* represents a column of data from a table or subquery. In the following SELECT statement for example, `ClientDNS` is a column expression representing the `ClientDNS` column in the `example_websrv_100` table.

```
SELECT ClientDNS
FROM example_websrv_100
DURING ALL;
```

Column expressions consist of a letter followed by a series of letters and digits. Generally the expression must match the name of a column in the table to be queried. Any expression that resolves to the name of a column can be used as a column expression.

Column expressions may include the table name as prefix, separated with a period (.) from the column name. For example, the previous statement is equivalent to the following:

```
SELECT example_webserv_100.ClientDNS
FROM example_webserv_100
DURING ALL;
```

The data type of a column expression must match the type of the underlying column in the table or subquery.

OPTIONAL COLUMN EXPRESSIONS

An *optional* column expression specifies a list of columns. The SQL query processor searches the list until it finds a column that is present in the table specified in the FROM clause. If none of the candidates are present, the value of the final (default) expression is returned. The data types of the columns in the list must be the same.

For example, based on the following SELECT statement, the query processor attempts to return values from the `HostName` if it exists in the table to be queried:

```
SELECT _optional_column(HostName, ClientDNS, '-')
FROM example_webserv_100
DURING ALL;
```

If the table does not contain a column named `HostName`, the query processor attempts to return values from `ClientDNS`. If neither column is found, the literal constant '-' is returned.

TIP: Optional columns are particularly helpful when used with list expressions in the FROM clause. For more information, see [“List Expressions and FROM Clauses”](#), on page 92.

Literal constants

A *literal constant* is simply a value. For example, the value `100` in the following SELECT statement is a literal constant:

```
SELECT ClientDNS
FROM example_webserv_100
WHERE RespSize > 100
DURING ALL;
```

There are four kinds of literals:

- [Integer Literals](#)
- [Floating-Point Literals](#)
- [String Literals](#)
- [Typecast Literals](#)

Integer Literals

An integer literal is a series of base 10 digits (i.e. '0', '1', '2', '3', '4', '5', '6', '7', '8' and '9'). In the query above, 100 is an integer literal. Integer literals have an implicit data type of `int32`.

Floating-Point Literals

A floating-point literal is a series of base 10 digits followed by a decimal point, some more digits, and an optional exponent. For example, the following values are floating-point literals:

```
1000.0 -- one thousand
3.1416 -- pi to 4 decimal places
1.0e6 -- one million
```

Floating-point literals have an implicit data type of `float`.

String Literals

A string literal is a sequence of characters enclosed between single or double quotes. For example, the values `'Feb 01 00:00:00 2002'` and `'Mar 31 23:59:59 2002'` are string literals. The following values are also string literals:

```
' ' -- two single quotes make an empty string
"" -- as do two double quotes
'http://www.acme.com' -- the literal value is everything between the quotes
```

Quotes themselves are not part of literal values; `'a'` is a string literal containing the single character `a`.

String literals may include new-line characters. For example, the following text is a single string literal that begins and ends with a single quote and contains newline codes and double quotes.

```
'sub Link {
  return "<a href=\"$_[0]\">$_[0]</a>";
}
```

A string literal may use the standard SQL convention to include embedded quotes by repeating the quote. The following two literals have the same value:

- `'adam's new shoes'` -- if you use two single quotes (`' '`) in a literal, it means include a single quote
- `"adam's new shoes"` -- or use double quotes around the value to include a single quote within it

String literals have an implicit data type of `varchar`.

SYNTAX FOR DEFINING LONG STRING LITERALS

It can be inconvenient to deal with quoting issues with apostrophes, backslashes, and so on -- particularly in complex and multi-line character strings. For example, you cannot simply cut-and-paste Perl code when declaring Perl functions, because the code may contain special characters and generally spans multiple lines.

To make it easier to use long varchar literals, the SQL parser supports *here document* syntax. Here document syntax alerts the SQL query engine that a long varchar value with special

characters and multiple lines has been defined. The syntax begins with double chevrons (`<<`) followed by a user-defined *limit string*. From that point forward, all characters, including new lines and other special characters, are treated as a single varchar value. The value ends with the next occurrence of *limit-string*, which must be at the beginning of its own line.

For example:

```
WITH $bigstring AS <<EOF
    This is a very long string with all sorts of funny characters
    '~!@#$%&*()_=-<>?,./[]\{}|;':"
    and it doesn't matter
EOF

SELECT top 1 $bigstring
FROM example_webserv_100
DURING ALL;
```

Note that the new-line character immediately after `<<EOF` and the new-line character before the final `EOF` are not part of the varchar value.

TIP: Use here document syntax when you declare user-defined functions and aggregates. For more information, see [“User-Defined Subroutines”, on page 82](#).

Typecast Literals

A typecast literal is a string literal preceded by a type name. For example, the following values are typecast literals:

- `int32 '10'` -- The integer 10 as a 32 bit integer
- `int64 '10'` -- The integer 10 as a 64 bit integer
- `bool '0'` -- The Boolean value 0 (false)
- `float '10.0'` -- The floating point value 10.0
- `timestamp 'Feb 05 00:00:00 2001'` -- The timestamp value for February 5th, 2001
- `varchar '10'` -- Same as '10'

Typecast literals specify explicitly the data type of a value. For example, if you specify the integer literal 10, it will be an `int32` value; specify `int64 '10'` instead if you need 10 to be an `int64` value.

DATA PROCESSING EXPRESSIONS

Data processing expressions do the computational work in a `SELECT` statement.

This section describes these topics:

- [“Operators”, next](#)
- [“Functions”, on page 73](#)
- [“Conversion Expressions”, on page 75](#)
- [“CASE Expressions”, on page 77](#)

Operators

An *operator* computes the result of a specific operation on two operands. In the following `SELECT` statement, the expression `ColumnB > 100` uses a comparison operator to include only rows where the value of `RespSize` is greater than 100. The greater-than symbol (`>`) is the operator; `RespSize` and 100 are the operands.

```
SELECT ClientDNS
FROM example_webserv_100
WHERE RespSize > 100
DURING ALL;
```

This section describes these topics:

- [“Math Operators”, next](#)
- [“The String Concatenation Operator”, on page 70](#)
- [“Comparison Operators”, on page 70](#)
- [“Logical Operators”, on page 72](#)
- [“Operator Precedence”, on page 73](#)

Math Operators

Math operators perform simple arithmetic computations. These are the math operators supported by AP SQL.

Symbol	Arithmetic Operation	Example
+	Addition	3 + 2 yields 5
-	Subtraction	3 - 2 yields 1
-	Unary minus	-3 yields the negative of 3; it operates as if the expression were 0 - 3. When you apply the unary minus operator to a negative value, the result is its absolute value.
*	Multiplication	3 * 2 yields 6
/	Division	3 / 2 yields 1.5
%	Modulo	3 % 2 yields 1

For example, the following `SELECT` statement uses the multiplication operator to compute the product of `RespSize` and 2.

```
SELECT ClientDNS
FROM example_webserv_100
WHERE RespSize * 2 > 100
DURING ALL;
```

Math operations can encounter results that require truncation:

- For numeric data types (`int32`, `int64`, `float`), the semantics of the underlying 'C' data type govern the result if an overflow or underflow occurs.
- If the result of a `timestamp` expression exceeds 2,147,483,648,000,000 microseconds, it is set to 2,147,483,648,000,000, which falls sometime in the year 2038.

The result of a math operation has a data type that depends on the data types of the operands. Generally, the result has the data type that is suitable for the result, which may differ from the data types of the operands.

The String Concatenation Operator

You can use the plus symbol (+) as a *string concatenation* operator when the operands are `varchar` values. For example, the following expression yields the `varchar` value “myTableSpace.myTable”.

```
'myTableSpace' + '.' + 'myTable'
```

If the result of a string concatenation operation exceeds 32,767 characters, the concatenated value is truncated.

The result of a string concatenation operation has `varchar` as its data type.

Comparison Operators

Comparison operators perform simple comparisons between one value and another. The two values and the comparison operator resolve to a `bool` value of “true” or “false”, depending on the outcome of the comparison. Use comparison operators to construct simple conditional expressions for WHERE, SLICE BY, and HAVING clauses.

BASIC COMPARISON OPERATORS

These are the basic comparison operators supported by AP SQL.

Symbol	Comparison Operation	Example Expression	Result
<	Less than	3 < 2	false
<=	Less than or equal to	3 <= 3	true
	Greater than	3 > 2	true
>=	Greater than or equal to	2 >= 2	true
=	Equal to	3 = 2	false
==		3 == 2	
<>	Not equal to	3 <> 2	true
!=		3 != 2	

For example, the following SELECT statement returns only `ClientDNS` values from rows in the `example_webserv_100` table where the corresponding value of `RespSize` is greater than 100.

```
SELECT ClientDNS
FROM example_webserv_100
WHERE RespSize > 100
DURING ALL;
```

The result of a basic comparison operation has `bool` as its data type.

ADVANCED COMPARISON OPERATORS

These are the advanced comparison operators supported by AP SQL.

Operator	Set Operation	Example
IN	Is the operand included in a set of enumerated values?	3 IN (2, 7, 5, 15, 3) yields true
BETWEEN	Is the operand included in the set of values defined by a beginning and ending range of consecutive values?	3 BETWEEN 2, 15 yields true
LIKE	Does the varchar operand match a particular pattern of characters?	'abc' LIKE 'a__' yields true

NOTE: SenSage AP SQL does not support the IS NULL and IS NOT NULL comparison operators, because the EDW does not support columns with NULL values.

The IN Operator

The IN comparison operator yields true if the value of an operand exists within a list of possible values. For example, the following SELECT statement returns `ClientDNS` values from rows where the value of `HttpVers` is 'HTTP/1.0' or 'HTTP/1.1'.

```
SELECT ClientDNS
FROM example_webserv_100
WHERE HttpVers IN ('HTTP/1.0', 'HTTP/1.1')
DURING ALL;
```

NOTE: EDW does not support columns with NULL values.

Use the corresponding NOT IN operator to determine if an operand is excluded from a list of values.

The result of an IN comparison has `bool` as its data type.

TIP: You can use a list variable as the right-hand operand of the IN operator. For more information on list variables, see [“Working with Lists”, on page 89](#).

Using Subqueries with the IN Operator

Instead of using the IN operator to specify an explicit list of values, you can use a subquery to derive the list of values. The subquery can reference the same table, view, or subquery as that used for the outer query, or the subquery can reference a different view, table, or subquery. The user must have the required permissions to access the view or table. Correlated subqueries that reference expressions from the outer query are not allowed. The subquery can be any general query and can include UNION clauses, WITH clauses, and subqueries. The subquery can have only one expression in the target list of its SELECT clause.

For example, the following SQL statement uses the WHERE clause to limit the results to those users in the engineering department:

```
SELECT username, hostname, result
FROM logins
WHERE username IN
(SELECT username FROM staff_list WHERE department="engineering")
DURING ALL;
```

The BETWEEN Operator

The BETWEEN comparison operator yields true if the value of an operand falls within a range of possible values. Use the corresponding NOT BETWEEN operator to determine if an operand is excluded from a range values.

The result of a BETWEEN comparison has `bool` as its data type.

The LIKE Operator

The LIKE comparison operator yields true if the value of a varchar operand matches a particular pattern of characters. The matching pattern can contain explicit characters and special wildcard characters. The percent sign (%) is a wildcard that matches zero or more characters of any kind; an underscore matches any single character in a particular position within the pattern.

For example, the following SELECT statement returns `ClientDNS` values from rows where the value of `HttpVers` begins with the characters 'HTTP/1'.

```
SELECT ClientDNS
FROM example_webserv_100
WHERE HttpVers LIKE 'HTTP/1%'
DURING ALL;
```

Because the matching pattern includes the percent wildcard (%), rows with 'HTTP/1.0' and 'HTTP/1.1' are returned in the result set.

NOTE: AP SQL supports neither asterisks (*) nor questions marks (?) as wildcard characters, nor does it support the optional ESCAPE subclause in LIKE comparisons.

IMPORTANT: To enhance performance, Hexis recommends that instead of the LIKE operator, you use the `_strstr()` function as in: `_strstr(<<VAL>%COL>,<VAL>)` instead of `<COL> like %<VAL>%`.

Use the corresponding NOT LIKE operator to determine if a varchar operand does not match a pattern of characters.

The result of a LIKE comparison has `bool` as its data type.

Logical Operators

Logical operators perform simple “true” or “false” comparisons between expressions that themselves resolve to `bool` values. Use logical operators to construct compound conditional expressions for WHERE, SLICE BY, and HAVING clauses.

These are the logical operators supported by AP SQL.

Operator	Logic Operation	Examples
AND	Logical AND; true only if both operands are true.	<ul style="list-style-type: none"> • <code>true AND true</code> yields true • <code>true AND false</code> yields false • <code>false AND false</code> yields false
OR	Logical OR; true if either operand is true.	<ul style="list-style-type: none"> • <code>true OR true</code> yields true • <code>true OR false</code> yields true • <code>false OR false</code> yields false
NOT	Logical negation	<ul style="list-style-type: none"> • <code>NOT true</code> yields false • <code>NOT false</code> yields true

In the following **SELECT** statement, the **WHERE** clause has a compound conditional expression that compares two simple conditional expressions with the **AND** logical operator.

```
SELECT ClientDNS
FROM example_webserv_100
WHERE (RespSize > 100) AND (HttpVers = 'HTTP/1.1')
DURING ALL;
```

The statement returns values in the `ClientDNS` column only for rows where both comparisons are true; the value of `RespSize` must be greater than 100, and the value of `HttpVers` must be the string `'HTTP/1.1'`.

The result of a logic operation has `bool` as its data type.

Operator Precedence

When several operators appear in a complex expression, they take precedence in the following order.

Symbol	Operation
-	Unary minus
* / %	Multiplication, division, and modulo
+ -	Addition, subtraction, and string concatenation
< <= > >= = <> IN BETWEEN LIKE	Comparison
NOT	Logical negation
AND	Logical AND
OR	Logical OR

Functions

Function expressions perform specialized calculations that return a single result. For example, the following **SELECT** statement uses two functions to calculate timestamp values for the **DURING** clause:

```
SELECT ClientDNS
FROM example_webserv_100
DURING time('Feb 01 00:00:00 2002'),
       _timeadd( time('Feb 01 00:00:00 2002'), 1, 'day')
;
```

The first function, `time()`, calculates the timestamp value that corresponds to the literal constant `'Feb 01 00:00:00 2002'`. The function takes one *argument*, a `varchar` that contains a date. It yields a timestamp value as its *return value*.

The second function, `_timeadd()`, also calculates a timestamp value. It takes three arguments instead of one, and it uses a different algorithm to compute its return value. The first argument is a timestamp, to which a specified amount and unit of time are added. This function interprets the second two arguments and adds them to the first. In the example above, the `_timeadd()` function returns a date that is one day later than the specified timestamp. In other words, this function

returns "Feb 02 00:00:00 2002". Notice that this function also contains the `time()` as an argument.

Function Syntax

In general, a function expression comprises the name of the function, followed by a comma-separated list of arguments enclosed in parentheses. For example, the following are function expressions:

- `_now()` -- a function that takes no arguments; it returns a timestamp for the current system time.
- `time('02/01/01')` -- a function that takes a varchar argument; in this case the argument is a literal constant, but any expression that yields a varchar representation of a date and time-of-day could be used.
- `_strcat(ColA, ColB, ColC)` -- a function that takes three varchar arguments and returns a varchar value that concatenates them; the arguments to this function are column expressions. The data types of the columns `ColA`, `ColB`, and `ColC` must be varchar.

The result of a function has the data type that the function declaration defines.

RELATED TOPICS

- To learn about built-in AP SQL functions, including their arguments and their return types, see [Chapter 4: SenSage AP SQL Functions](#).
- To learn how to declare user-defined functions, see [“User-Defined Subroutines”, on page 82](#), and [Chapter 8: Perl Subroutines](#).

Asterisks as Column-Expression Arguments

An asterisk (*) can be used as a special column-expression argument. The SQL query engine replaces the asterisk with a comma-separated list of the columns in the table identified in the FROM clause.

For example, the following SELECT statement:

```
SELECT _strcat(*)
FROM example_webserv_100
DURING ALL;
```

is equivalent to:

```
SELECT _strcat(ts, ClientIP, ClientDNS, Method, Url,
              HttpVers, RespCode, RespSize, Referrer,
              UserAgent, RespTime)
FROM example_webserv_100
DURING ALL;
```

COUNT(*)

When an asterisk is used as an argument, the `count()` aggregate function operates as a special case. Instead of replacing the asterisk with a comma-separated list of the columns in the table, the SQL query engine treats `count(*)` as if it were `count(1)`. The special nature of the `count(*)` expression is supported for compatibility with standard SQL.

The SORT BY Modifier Keyword

When an aggregate function is called, it is passed arrays of column values in the order by which the underlying column expression generates them. However, if a function expression that takes column expressions as arguments is followed by the SORT BY modifier keyword, the SQL query engine first sorts the column values on the specified function argument before invoking the function.

Consider the following SELECT statement:

```
SELECT ClientDNS,
       _strsum( Url, ts ) SORT BY 2
FROM example_webserv_100
GROUP BY 1
DURING ALL ;
```

The `_strsum()` aggregate function ignores the second argument as a target column. However, the `Url` values will be sorted according to the value of `ts` before `_strsum()` is invoked. The `_strsum()` function receives the `ClientDNS` values in the order of the `ts` values. This is often useful when using `_strsum()` to concatenate string values.

NOTE: Unless SORT BY is specified, the order of the records passed into an aggregate is non-deterministic. Two invocations can yield different sort orders. The EDW runs queries in parallel across a cluster of machines and cannot guarantee which machines will produce records the fastest.

The DISTINCT Modifier Keyword

The DISTINCT modifier keyword ensures that only the unique sets of values are passed into aggregate functions.

```
SELECT count( DISTINCT ClientDNS )
FROM example_webserv_100]
DURING ALL;
```

If an aggregate function takes multiple arguments, the DISTINCT modifier keyword makes the entire set unique. If any argument differs, the set is considered unique and the aggregate is called. The DISTINCT modifier keyword is useful with Perl aggregates.

NOTE: The DISTINCT and SORT BY modifiers can work together; both operations happen at once.

IMPORTANT: The SQL query engine can use a lot of memory to process function expressions that use the DISTINCT modifier keyword.

Conversion Expressions

A *conversion* expression performs a simple conversion of a data value from its current data type to another. Use conversion expressions to put data values into the form expected by subsequent operations.

There are three kinds of conversion expressions:

- [CONVERT Expressions](#)
- [Function-style Conversion Expressions](#)

- [Typecast Literal Conversion Expressions](#)
- [Typecast Literal Conversion Expressions](#)

CONVERT Expressions

A *CONVERT* expression has the form:

```
CONVERT (<target_data_type>, <value_to_convert>)
```

Use one of these allowed target data types.

Target Data Types
bool
int32
int64
float
timestamp
varchar
inet

For example, the following *SELECT* statement uses a *CONVERT* expression to convert values in the *ts* column from timestamp values to human-readable, character-based values:

```
SELECT CONVERT (varchar, ts)
  FROM example_webserv_100
 DURING ALL;
```

Function-style Conversion Expressions

A *function-style* conversion expression has one of the following forms, depending on the target data type:

Conversion Function	Conversion Result
<i>_bool</i> (x)	Equivalent to <i>CONVERT</i> (bool, x)
<i>_int32</i> (x)	Equivalent to <i>CONVERT</i> (int32, x)
<i>_int64</i> (x)	Equivalent to <i>CONVERT</i> (int64, x)
<i>_float</i> (x)	Equivalent to <i>CONVERT</i> (float, x)
<i>_timestamp</i> (x)	Equivalent to <i>CONVERT</i> (timestamp, x)
<i>_varchar</i> (x)	Equivalent to <i>CONVERT</i> (varchar, x)
<i>_inet</i> (x)	Equivalent to <i>CONVERT</i> (inet, x)

The example below illustrates how you can use the `_varchar()` function as an alternate to the example SELECT statement for CONVERT expressions:

```
SELECT _varchar(ts)
FROM example_webserv_100
DURING ALL;
```

Typecast Literal Conversion Expressions

A *typecast literal* conversion expression has the form:

```
<typecast_literal> '<literal_value>'
```

Use one of these typecast literals, depending on the target data type.

Typecast Literal
bool
int32
int64
float
timestamp
varchar
inet

This means the example SELECT statement for CONVERT expressions could also be written as:

```
SELECT varchar ts
FROM example_webserv_100
DURING ALL;
```

CASE Expressions

A *CASE* expression supports conditional target specifications. CASE expressions have the following syntax:

```
CASE WHEN <conditional-expression> THEN <value>
      WHEN <conditional-expression> THEN <value>
      ...
      ELSE <value>
END
```

For example, the following SELECT statement returns result rows with VARCHAR values of <1k, 1k, or >1k, depending on whether the value of `RespSize` is less than, equal to, or greater than 1024.

```
SELECT CASE WHEN RespSize < 1024 THEN "<1k"
            WHEN RespSize = 1024 THEN "1k"
            ELSE ">1k"
      END
FROM example_webserv_100
DURING ALL;
```

PROCESSING DIRECTIVES

The proprietary WITH keyword introduces clauses that direct how the SQL query engine processes SELECT statements. WITH clauses let you declare symbols, such as named constants, user-defined functions, and subqueries, that you can use within SELECT statements. WITH clauses also let you configure the execution environment in which queries run. The syntax of WITH clauses depends on the kind of directive the WITH clause declares.

This section describes these topics

- [“Macros”, next](#)
- [“User-Defined Subroutines”, on page 82](#)
- [“Subqueries”, on page 84](#)
- [“Table-Name Substitutes”, on page 86](#)
- [“WHERE Clause Filters”, on page 86](#)
- [“Settings”, on page 87](#)
- [“The Scope of Processing Directives”, on page 88](#)

Macros

This section describes these topics:

- [“About Macros”, next](#)
- [“Expression Macros”, on page 79](#)
- [“Star Macros”, on page 79](#)
- [“Multiple Declarations of a Given Macro”, on page 81](#)
- [“Resolving Macro Identifiers”, on page 81](#)
- [“Overriding Multiple Macro Declarations”, on page 81](#)

About Macros

Macros are processing directives that declare constant values that can be used within a SELECT statement. Macro definitions have the following syntax:

```
WITH $<id> AS <value> [OVERRIDE][, $<id> AS <value> [OVERRIDE] [...]]
```

For each occurrence of \$<id> after the declaration, the query engine replaces it with the constant value. The values of macros remain constant throughout the execution of SELECT statements. Macros are similar to literal constants, not programming variables.

Their benefits of macro directives over literal constants are:

- You declare the constant value in one location, and the value is used wherever the macro identifier is encountered in the remainder of the SELECT statement. To change the value of a literal constant repeated throughout a statement, you must find and change every occurrence.
- Macro identifiers can be much shorter than the values they define. Using the shorter, macro identifier in place of the longer literal expression makes your SELECT statements easier to read and comprehend.

- You can use macros in Perl subroutines so that your subroutines and your SELECT statement can use the same values during execution. For more information, see [“Using Macros in Perl Subroutines”, on page 230](#).

Expression Macros

An *expression* macro defines a named expression. The expression can be a column expression, a literal constant, or a complex expression that involves comparisons, functions, and logic operations.

For example, the following SELECT statement returns the number of log entries between 2/1/01 and 2/2/01:

```
WITH $start as time('Feb 01 00:00:00 2002'), $end as _timeadd($start, 1, 'day')

SELECT count(*)
  FROM example_webserv_100
  DURING $start, $end;
```

The preceding and following statements are equivalent, but the statement that uses expressions macros is easier to read.

```
SELECT count(*)
  FROM example_webserv_100
  DURING time('Feb 01 00:00:00 2002'),
         _timeadd( time('Feb 01 00:00:00 2002'), 1, 'day');
```

Expression macros are useful to break up complex expressions into more understandable forms and to enable parameterized queries.

For more information on how to use expression macros as query parameters, see: [Querying Data in Chapter 3, "Loading, Querying, and Managing the EDW" in the *Administration Guide*](#).

AUTOMATIC EXPRESSION MACROS

When you use the `atload` command to load source log entries into the EDW, you write a AP SQL SELECT statement that describes the columns in the target table that will store the source log data. The command declares some expression macros automatically for use in you SELECT statement.

For more information, see Automatic Expression Macros in Chapter 3, "Loading, Querying, and Managing the EDW" in the *Administration Guide*.

Star Macros

WITH can be used to declare macros that can be used as shortcuts for specifying multiple expressions in the same way that asterisks (*) can be used as a shortcut for specifying the all the columns in a table. For example, the following query returns the minimum and maximum values for the timestamp column:

```
WITH $exprs as '*'(min(ts) AS 'min time', max(ts) as 'max time')

SELECT $exprs, count(*)
  FROM example_webserv_100
  DURING ALL;
```

The above query is equivalent to:

```
SELECT min(ts) AS 'min time', max(ts) as 'max time', count(*)
FROM example_webserve_100
DURING ALL;
```

Star macros may be used in the target clause, the GROUP BY clause, and the ORDER BY clause. For example:

```
-- isolate the actual group and order criteria from the rest of the query
WITH $groupkey AS '*'(ClientDNS as "hostname" ASC, ClientIP as "ipaddress" DESC)

SELECT $groupkey, count(*)
FROM example_webserve_100
GROUP BY $groupkey
ORDER BY $groupkey
DURING ALL;
```

During the ordinary macro expansion process, the SQL compiler sees the `$groupkey` macro in the target list, finds the corresponding WITH statement, and replaces the `$groupkey` macro in the target list with the specified targets.

Here are some additional examples:

EXAMPLE 1

The following query:

```
WITH $columns AS '*'(ts AS 'Time', Host, ClientIP)

SELECT $columns
FROM test
DURING ALL;
```

returns the same result as:

```
SELECT ts AS 'Time', Host, ClientIP
FROM test
DURING ALL;
```

EXAMPLE 2

A star macro can contain a general list of expressions and can be used with functions in the obvious way:

```
WITH $args1 AS '*'("%m/%d/%Y %H:", ts)
WITH $args2 AS '*'("%02d", 30*((_int32(ts)%3600)/1800))

SELECT _timef($args1) + _sprintf($args2) AS 'time'
FROM test
DURING ALL;
```

EXAMPLE 3

A star macro may contain other star macros, as long as they are not recursive. [Example 1](#) above could be written as:

```
WITH $host AS Host
WITH $columns1 AS '*'($host, ClientIP)
WITH $columns AS '*'(ts AS 'TIME', $columns1)

SELECT $cols
  FROM test
 DURING ALL;
```

Multiple Declarations of a Given Macro

In general, only one declaration for any given macro should be present in a SELECT statement. Including two declarations of the same macro typically results in a SQL syntax error.

The exceptions to this general rule are as follows:

Exception	Description
Multiple Scopes	Local definitions of the same macro can be made as long as each such definition is in a different scope.
OVERRIDE	For any given macro, there can be up to one definition that includes the OVERRIDE keyword. This definition will override any subsequent definitions for that macro in the scope in which it is declared, and any nested scopes thereof. Note that it is legal to have more than one OVERRIDE definition of the same macro; however, the first OVERRIDE definition takes precedence.

Resolving Macro Identifiers

When the SQL query engine encounters a macro identifier in a SELECT statement, it searches back through the statement for the macro declaration. Conceptually, it is looking for the “closest” declaration of the macro.

Overriding Multiple Macro Declarations

The **OVERRIDE** keyword is helpful when a macro identifier has multiple declarations within the statement. When the SQL processor searches for the closest declaration, it continues searching if the closest one does not include the **OVERRIDE** keyword. It continues searching further back for a declaration with the **OVERRIDE** keyword. If it finds an overriding declaration, the processor uses it. If no overriding declaration is found further back, it uses the closest non-overriding definition.

For example:

```
SET $definition TO NULL
SET $frame TO $current-frame

WHILE $frame NOT NULL DO
  SET $this-def TO FIND($macro IN $frame)
  IF $this-def NOT NULL DO
    IF IS_OVERRIDE($this-def) DO
      SET $definition TO $this-def
      EXIT-WHILE
    ENDIF
    IF $definition NOT NULL DO
```

```
        SET $definition TO $this-def
    ENDIF
    SET $frame TO PARENT_OF($frame)
ENDWHILE

IF $definition IS NULL DO
    ERROR( "Undefined Symbol: %s", $macro)
ENDIF
```

By default, the SQL processor checks for redundant WITH statements of the same type (for example, two Perl functions with the same name) and throws an error. This behavior is especially important when building libraries of WITH clauses, to avoid accidentally using the same name twice.

However, it is also helpful to be able to override the default WITH declarations inside of these libraries, subqueries, and PTL files. Thus, any WITH statement may be appended with the word **OVERWRITE**, and it will override any future definitions with the same name (and type).

Putting it together:

```
include howto-sql-30.inc
WITH $count AS 52

SELECT $count
FROM example_webserv_100
DURING ALL;
```

If `howto-sql-30.inc` contained `WITH $count AS count(*)`, then you would get an error, but if it contained `WITH $count AS count(*) OVERWRITE`, the query would return the number of rows in the table.

User-Defined Subroutines

User-defined subroutines are processing directives that let you declare custom Perl functions or aggregates. Once declared, you can use these subroutines within SQL SELECT statements. The declarations of user-defined subroutines have the following syntax:

```
WITH <id> AS [BUILTIN] '<language>' [<return_type>] {FUNCTION|AGGREGATE} <<EOF
    <declaration>
EOF [OVERWRITE]
```

The optional **BUILTIN** keyword causes the declaration to be treated as if the function is built into the SQL query engine. This allows you to invoke the function directly by name. If you omit the **BUILTIN** keyword, you must invoke the function indirectly with the `_perl()` or `_perlagg()` SQL functions. For more information on these built-in SQL functions for calling Perl subroutines, see [Chapter 8: Perl Subroutines](#).

The value for `'<language>'` should always be `'perl15'`.

The value of `<return_type>` is any of the supported AP SQL data types: `bool`, `float`, `int32`, `int64`, and `varchar`. The data type of the return value is `varchar` by default.

The keywords **FUNCTION** and **AGGREGATE** indicate whether `<declaration>` is a Perl function or a Perl aggregate.

The value of `<declaration>` is a Perl script declaration. Generally, the `<declaration>` is enclosed within here document syntax, because declarations span multiple lines and use special characters, including semi-colons. For more information, see [“Syntax for Defining Long String Literals”](#), on page 67.

TIP: You can use macros in Perl subroutines. For more information, see [“Using Macros in Perl Subroutines”](#), on page 230.

DECLARING PERL FUNCTIONS

The following example declares a Perl function called `Link`, which returns an HTML link based on its argument.

```
WITH Link AS BUILTIN 'perl5' FUNCTION <<EOF
  sub Link {
    return "<a href=\"$_[0]\">$_[0]</a>";
  }
EOF
```

```
SELECT Link('http://' + ClientDNS)
FROM example_webserv_100
DURING ALL;
```

If the value of `ClientDNS` for a row in the table contains the text `'www.acme.com'`, the result set contains a corresponding row with the value:

```
'<a href="http://www.acme.com">www.acme.com</a>'
```

For a more detailed explanation, see [“Declaring Perl Functions”](#), on page 226.

DECLARING PERL AGGREGATES

The following example declares a Perl aggregate called `last()`. Because the declaration does not include the `BUILTIN` keyword, the `SELECT` statement invokes `last()` with the built-in `_peragg()` SQL function.

```
WITH last AS 'perl5' AGGREGATE <<EOF
  my %state;
  sub last { $state{$_[1]} = $_[2]; }
  sub last_final { return $state{$_[1]}; }
EOF

SELECT ClientDNS, _peragg('last', ClientDNS, RespSize)
FROM example_webserv_100
GROUP BY 1
DURING ALL;
```

Within the declaration of Perl aggregates, you define two Perl subroutines and any global variables that the two subroutines can access. The first subroutine has the same name as the Perl aggregate declaration. The second subroutine has the same name with a suffix of `_final`, and it must have `return` statement.

For a more detailed explanation, see [“Declaring Perl Aggregates”](#), on page 227.

Subqueries

Subqueries are processing directives that declare named SELECT phrases for use in your main SELECT statement. The result sets of subqueries can be used as if they were tables in the EDW. Declarations of subqueries have the following syntax:

```
WITH <id> AS (<subquery> [OVERRIDE][, <id> AS (<subquery>) [OVERRIDE][...]])
```

For example, the following SELECT statement queries the result set from the subquery named timestamps:

```
WITH timestamps AS (SELECT DISTINCT ts FROM example_webserv_100 DURING ALL)

SELECT *
  FROM timestamps;
```

You can nest subquery declarations, as the next example shows:

```
WITH subq1 AS (WITH subq2 AS (SELECT ts FROM foo DURING ALL) SELECT * FROM subq2)

SELECT *
  FROM subq1;
```

The main SELECT statement queries for its results from subq1. The results of subq1 are selected from another subquery, subq2, which is declared within the first subquery.

Subqueries are useful when multiple levels of aggregation are required. For example, the following query returns the number of distinct values of ClientDNS:

```
WITH subquery AS ( SELECT ClientDNS
                   FROM example_webserv_100
                   GROUP BY 1
                   DURING time('Feb 01 00:00:00 2002'),
                        time('Mar 31 23:59:59 2002')
                 )

SELECT count(*)
  FROM subquery;
```

IMPORTANT: For performance reasons, avoid subqueries that generate too many rows in their result sets; that is, 100,000 rows returned by a subquery is reasonable, but millions of rows may result in performance problems.

Subqueries with DURING Clauses

The DURING clause is required only in subqueries that select data directly from tables. Subqueries and SELECT statements that specify subqueries in the FROM clause instead of tables do not require DURING clauses unless they are run from SenSage AP Analyzer.

IMPORTANT: When you create a subquery that selects data directly from tables, ensure that the DURING clause behaves as you expect. A badly formed DURING clause can cause the query to return incorrect or no results. For more information, see [“Subqueries and Views and the DURING Clause”, on page 51](#).

Subqueries and UNION ALL Clauses

Subqueries can include UNION ALL clauses. For example:

```
WITH subq AS ( ... UNION ALL ... )

SELECT *
  FROM subq;
```

When the query engine executes the preceding SELECT statement, it executes the nested subqueries and produces a union of their result sets. The query engine then executes the main SELECT statement against the intermediate result set produced by the subquery `subq`.

IMPORTANT: To enhance performance, Hexis recommends after any UNION ALL subqueries, you use ORDER BY in a separate subquery.

CREATING UNIONS OF SUBQUERIES

You can create a union of subqueries in the main SELECT statement. For example:

```
WITH subq1 AS ( ... )
WITH subq2 AS ( ... )

SELECT *
  FROM subq1

UNION ALL

SELECT *
  FROM subq2;
```

Arbitrary nesting of subqueries and/or UNION ALLs is allowed.

ACHIEVING DISTINCT UNION RESULTS

With subqueries, you can overcome the limitation that UNION DISTINCT is not supported. Perform the UNION ALL operations in a subquery declaration. Then, use DISTINCT in the main SELECT statement. For example:

```
WITH subquery AS (SELECT ClientDNS, RespSize
                  FROM example_webserv_100
                  DURING ALL

                  UNION ALL

                  SELECT DISTINCT ClientDNS, RespSize
                  FROM example_webserv2_100
                  DURING ALL
                )

SELECT DISTINCT * FROM subquery;
```

Subqueries and the WHERE clause

You can use the `IN` or `NOT IN` operators inside of a `WHERE` clause to limit results to an explicit list of values. These operators can also use subqueries to create the list of results. See [“Using Subqueries with the IN Operator”, on page 71](#).

Subqueries and the ORDER BY Clause

To enhance performance for a query that includes more than one subquery, Hexis recommends that you do not use ORDER BY in the first subquery. Hexis also recommends that you use ORDER BY in a separate subquery after any UNION ALL queries or subqueries.

Table-Name Substitutes

Table-name substitutes are processing directives that declare table names which substitute for table identifiers in subqueries and in SELECT statements. Declarations of table-name substitutes have the following syntax:

```
WITH <id> AS TABLE <table> [OVERRIDE][, <id> AS TABLE <table> [OVERRIDE][...]]
```

The value of <id> is a table identifier that occurs elsewhere within subqueries or the SELECT statement. The value of <table> is the table name that should substitute for <id> when the query engine executes the compiled subqueries and SELECT statement.

For example:

```
WITH webserv AS TABLE example_webserv_100

SELECT count(*)
  FROM webserv
  DURING ALL;
```

NOTE: You can achieve the same effect with the `--tableswap` option of the `atload` command. For more information, see the topic *Querying Data* in Chapter 3, “Loading, Querying, and Managing the EDW” in the *Administration Guide*.

WHERE Clause Filters

WHERE clause filters are processing directives that declare a conditional expression to be added to WHERE clauses for a specified table. Declarations of WHERE clause filters have the following syntax:

```
WITH WHERE <table_id> AS (<conditional-expression>)
```

The value of <table_id> is a table identifier that occurs in FROM clauses within subqueries or the SELECT statement. The value of <conditional-expression> is a conditional expression that references columns in the identified table. When the query engine executes the SQL statement, it combines the WHERE clause filter and the explicit conditional expression of the WHERE clause. It uses the AND logical operator as the conjunction. If there is no explicit WHERE clause, the WHERE clause filter alone is applied.

For example, the following SELECT statement declares a WHERE clause filter to ensure that only rows from the `example_webserv_100` table that have 'HTTP/1.0' in their `HttpVers` columns are included in the results set, regardless of the conditions in the explicit WHERE clause.

```
WITH WHERE example_webserv_100 AS (HttpVers = 'HTTP/1.0')

SELECT count(*)
  FROM example_webserv_100
  WHERE ReapSize > 100
```



```
DURING ALL;
```

The preceding example returns the same results as if the WHERE clause had been written as follows:

```
WHERE (HttpVers = 'HTTP/1.0') AND (ReapSize > 100)
```

You can include multiple WITH WHERE declarations for the same table, and the effect is cumulative. Each *<conditional-expression>* is combined the others using the AND logical operator as the conjunction.

TIP: If the *<conditional-expression>* is long and needs to span multiple lines, use here document syntax. For more information, see [“Syntax for Defining Long String Literals”, on page 67](#).

Settings

Settings are processing directives that control the SQL query engine and configure the context in which SQL statements execute. Declarations of settings have the following syntax:

```
WITH <setting> <value> [OVERRIDE]
```

The following settings can be controlled through WITH clauses.

Setting	Value	Meaning
TIMEZONE	GMT	Sets a default time zone for manipulating timestamps
NOWTIMESTAMP	The current time	The timestamp returned by <code>_time('now')</code>
MINTIMESTAMP	1970-01-01T00:00:00	The timestamp returned by <code>_time(<time_specification>)</code>
MAXTIMESTAMP	2038-01-19T03:14:07	The timestamp returned by <code>_time(<time_specification>)</code>
GROUPCACHEPARTITIONS		Controls behavior of the SQL Aggregation engine
GROUPCACHEPARTITIONING		Controls behavior of the SQL Aggregation engine
AGGCACHEPARTITIONS		Controls behavior of the SQL Aggregation engine
AGGCACHEPARTITIONING		Controls behavior of the SQL Aggregation engine
FIFOCACHEPARTITIONS		Controls behavior of the SQL Aggregation engine
FIFOCACHEPARTITIONING		Controls behavior of the SQL Aggregation engine
DIVIDEBYZERO		Controls behavior when a divide-by-zero condition occurs
MODULOBYZERO		Controls behavior when a modulo-by-zero condition occurs
SUPRESSECEPTIONS		Controls whether error conditions stop query execution
ALLOWEXITPERL	false	Controls whether Perl subroutines can exit

Setting	Value	Meaning
DGBVARS		Supports debugging
NOPROGRESS	false	Suppresses the generation of progress indicators

You can declare the same setting multiple times. The earliest declaration takes effect and overrides any later declarations of the same setting.

The TIMEZONE Setting

One of the most common settings is `TIMEZONE`. It changes how the SQL query engine interprets timestamps when the time zone is unspecified. Many source log entries do not include time-zone indicators, and some formats for timestamps passed to the `time()` function cannot specify time zones. The setting has the following format:

```
WITH TIMEZONE '<time_zone>'
```

The allowed values for `<time_zone>` are listed in “Appendix B: Time Zones.

The following `SELECT` statement declares the `TIMEZONE` setting to be Pacific Time. The `varchar` arguments to the two `time()` expressions in the `DURING` clause do not specify their time zones. The `TIMEZONE` setting tells the query engine to interpret their time zones as Pacific Time. Without the setting, the query engine assumes their time zones are GMT.

```
WITH TIMEZONE "PST8PDT"
```

```
SELECT count(*)
  FROM example_webserve_100
  DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002')
```

Specifying time zones is important to ensure that queries return the intended results, especially when the source of the data generates times in a different time zone than yours.

The Scope of Processing Directives

Processing directives are classified by one of two types of scope:

- [Global State Modifiers](#)
- [Local Definitions](#)

Global State Modifiers

Directives that are *global state modifiers* affect the entire `SELECT` statement, including subqueries. You can declare them only in the top level of a `SELECT` statement. Syntax errors occur when you include global state modifiers as parts of subquery declarations.

These directives have the scope of global state modifiers.

Directive Syntax	For more information, see...
WITH <id> AS [BUILTIN] '<language>' [<return_type>] {FUNCTION AGGREGATE} <<EOF <declaration> EOF [OVERRIDE]	“User-Defined Subroutines”, on page 82
WITH <id> AS TABLE <table> [OVERRIDE]	“Table-Name Substitutes”, on page 86
WITH WHERE <table_id> AS (<conditional-expression>	“WHERE Clause Filters”, on page 86
WITH <setting> <value> [OVERRIDE]	“Settings”, on page 87

Local Definitions

Directives that are *local definitions* affect only the parts of SELECT statement in which they are declared. You can declare them anywhere within a SELECT statement, including subqueries.

These directives have the scope of local definitions.

Directive Syntax	For more information, see...
WITH \$<id> AS <value> [OVERRIDE]	“Macros”, on page 78
WITH <id> AS (<subquery>){OVERRIDE}	“Subqueries”, on page 84

WORKING WITH LISTS

Event-log data often has fields that contain lists of data. SenSage AP SQL provides facilities for working with lists.

This section describes these topics:

- [“Multiple Values as Lists”, next](#)
- [“Functions that Return Lists”, on page 90](#)
- [“List Acceptors”, on page 90](#)
- [“INTO Keyword”, on page 91](#)
- [“List Expressions and FROM Clauses”, on page 92](#)
- [“EXPLODE Keyword”, on page 92](#)
- [“EXPLODE BY Keyword Phrase”, on page 93](#)
- [“Some Helpful List Examples”, on page 93](#)

Multiple Values as Lists

Expressions and functions normally evaluate to a single value during query processing. However, there are times when you want them to evaluate to a list of values. When an expression or function evaluates to a list, you indicate whether you want to access the list or the single return value.

The syntax options for indicating your intent are:

- **<expression>**

Access the single return value and discard the list.

- **`<expression> INTO <varname>`**

Access the single return value and save the list in `<varname>` for later use

- **`@<expression>`**

Access the list and discard the single return value

- **`@<expression> INTO <varname>`**

Access the list and save the list in `<varname>` for later use. This form is rarely used, because `<varname>[0]` contains the normal single value; that is, it is not lost.

Functions that Return Lists

These functions can return lists instead of single return values when you precede them with the list symbol (`@`). Without the symbol, these functions return a single value.

- `_strmatchlist()`

If your regular expression contains matching sets of parentheses, the matching text between them becomes a return value. The normal return value is the number of elements matched.

- `_strsplit()` and `_strsplitxsv()`

When parsing strings into multiple elements, the results are returned as a list. The normal return value is the number of elements parsed.

- `_perl()`

Perl functions can call `Addamark::setInto()` to return multiple values. See the [“List Support and Perl Functions”](#), on page 229, for more information.

- `_into()`

The `_into()` function takes its arguments and stores them in a list. Specifically, `_into(*)` can be used to convert the list of table-columns (or subquery-targets) into a list.

- `_lookup()`

The multiple columns of each row are returned as a list. The first column is also copied into the normal result value, which makes `_lookup()` easier to use when there is only one result column.

List Acceptors

The following types of expressions can consume lists. The `@` symbol forces the SQL query engine to access the list, rather than the regular return value from the expression.

- **`<listgen> INTO <varname>`**

This clause can be appended after a list generator expression to give the list a name, such that the list elements can be used in other expressions. For more details, [“INTO Keyword”](#), next.

- **EXPLODE @<listgen>**

Transposes the list into multiple records. For more details, “[EXPLODE Keyword](#)”, on page 92, and “[EXPLODE BY Keyword Phrase](#)”, on page 93.

- **<expression> IN @<listgen>**

Matches the <expression> against the elements of <listgen> and returns Boolean “true” if there is a match.

- **_perl()**

Perl functions can be called with lists as their arguments, that is, `_perl("myfun", arg1, arg2, @<varname>, arg3)`. Perl programmers will recognize similarities with lists in Perl. For more information, see [Chapter 8: Perl Subroutines](#).

- **_strcat() and _strjoin()**

You can concatenate the values of a list together.

INTO Keyword

The INTO keyword accepts multiple return values and puts them in a named a list variable. Use of the INTO keyword has the following syntax:

```
<expression> INTO <listname>
```

The <listname> element of the syntax is called the *INTO* variable. You can use INTO variables elsewhere in the SQL statement. Use array index notation to access individual items in the list. Actual list items begin at <listname>[1]. For convenience, the item at <listname>[0] has a copy of the normal expression result.

The following example uses INTO to capture multiple matches from `_strmatchlist()`:

```
SELECT match[1], match[2]
   FROM example_webserve_100
  WHERE _strmatchlist(UserAgent, '([^(]*)\\((([^(]+)\\)\\)') INTO match
 DURING ALL;
```

In this query, the expressions `match[1]` and `match[2]` represent the additional values returned by the query.

Generally, INTO variables can be used anywhere in the SQL statement like any other expression or column identifier. For example, INTO variables can be passed into functions and used in mathematical or string expressions. However, causality must be preserved and loops are not allowed. For example:

```
-- this is illegal because it creates an INTO-loop
SELECT _strmatchlist(foo, 'x=(..),y=(..)') INTO bar,
       _strmatchlist(bar, 'x=(..),y=(..)') INTO foo,
   FROM Table
 DURING ALL;
```

The INTO keyword is optional. Use it only when you want to capture the list that an expression or function can return. You cannot use the INTO directive with an expression that produces only single return values. For example the following is invalid:

```
4+5 INTO sum -- incorrect use of INTO with a single-value expression
```

The expression 4+5 only evaluates to a single value, never to a list.

List Expressions and FROM Clauses

List expressions may be used in the FROM clause to specify an implicit union of the specified tables. For example, the following query returns the distinct ClientIP values from both the example_webserv_100 and example_webserv2_100 tables:

```
SELECT DISTINCT ClientIP
FROM @_list('example_webserv_100', 'example_webserv2_100')
DURING all;
```

Whenever the SQL query engine sees a FROM clause with a list expression, the engine verifies that all the tables in the list contain all the columns selected, and that all the types of all the columns are the same.

IMPORTANT: The DISTINCT keyword causes each query to produce unique intermediate results, but the final result set does not eliminate duplicates found in the separate tables.

EXPLODE Keyword

The EXPLODE keyword lets you transpose a list of values into a target column. For example, you may need to pass list elements into aggregates. The keyword has the following syntax:

```
EXPLODE <list_expression> [AS <target>]
```

The SQL query engine iterates through <list_expression>, replicating the input row in the output result set for each list element. The current list element is substituted in each iteration.

For example:

```
-- returns three records: "fred 12000", "mary 12000" and "bob 12000"
WITH subquery as (
  SELECT TOP 1 ts
  FROM example_webserv_100
  DURING ALL
)

SELECT EXPLODE @_strsplit(",", "fred,mary,bob") AS name, 12000 AS salary
FROM subquery -- only one record
;
```

The expression that follows the EXPLODE keyword must evaluate to a list variable.

IMPORTANT: You may use only one EXPLODE keyword per SELECT phrase. A statement can have two or more EXPLODE keyword only if each one exists in a separate subquery.

EXPLODE BY Keyword Phrase

The EXPLODE BY keyword phrase lets you transpose a list embedded in a `varchar` expression. The phrase has the following syntax:

```
EXPLODE <varchar_expression> BY '<separator>'
```

The value of `<varchar_expression>` must contain a list of items separated by the special character specified by `<separator>`. The expression to explode can be a single-byte or multi-byte string

For example, the expression in the statement below uses a comma (,) to separate the values in list:

```
WITH subquery as (
  SELECT TOP 1 ts
    FROM example_webserv_100
    DURING ALL
)

SELECT EXPLODE "fred,mary,bob" BY ',' ,12000 AS salary
  FROM subq -- only one record
;
```

A common use of EXPLODE BY is to compute histograms, as the next example shows:

```
WITH subquery as (
  SELECT EXPLODE Url BY '/' AS component
    FROM example_webserv_100
    DURING ALL
)

SELECT component, count(*)
  FROM subquery
 GROUP BY 1
 ORDER BY 2 DESC
;
```

Some Helpful List Examples

The following examples are often helpful in understanding lists.

```
_strjoin(",", @_strsplit(",", "a,b,c")) == "a,b,c"
_strjoin(",", @_strsplit(",", "a,b,c")) == @_strsplit(",", "a,b,c") == "3" (number
of elements parsed)
_if(_strsplit(",", "a,b,c") INTO v > 0, v[2], "") == "b"

-- should return 12, which is four records each exploded three times
WITH subq as (select top 4 ts from example_webserv_100 during all)
SELECT COUNT(EXPLODE @_strsplit(",", "a,b,c"))
  FROM subq
```


SenSage AP SQL Functions

SenSage AP SQL (also known as AP SQL) provides these functions for use in expressions in its Select statements:

- “Conditional Evaluation Functions”, next
- “List Functions”, on page 97
- “Lookup Functions”, on page 99
- “Aggregation Functions”, on page 113
- “Statistical Aggregate Functions”, on page 120
- “Logarithmic and Exponential Functions”, on page 124
- “Numeric Rounding Functions”, on page 126
- “String Functions”, on page 129
- “Time Functions”, on page 150
- “Network Address Functions”, on page 162
- “Miscellaneous Functions”, on page 172

NOTE: For type-conversion functions, see “Conversion Expressions”, on page 75.

CONDITIONAL EVALUATION FUNCTIONS

This section describes these functions that perform conditional evaluations.

Function	Purpose	Page
<code>_if()</code>	Evaluate a Boolean expression and return different values for true and false	page 95
<code>_iftable()</code>	Return different expressions depending on the table being queried	page 96

`_if()`

The `_if()` function evaluates a Boolean expression and returns different values for true or false.

Synopsis

```
_if( <condition>, <expr1>, <expr2> )
```

Description

If `<condition>` evaluates to true, the `_if()` function returns `<expr1>`. Otherwise, it returns `<expr2>`. Both expressions must evaluate to the same data type. You can use `_if()` recursively, as either `<expr1>` or `<expr2>`.

The following example uses `_if()` recursively as the second expression argument:

```
_if( <condition1>, <expr1>, _if( <condition2>, <expr2>, <expr3> ) )
```

The function evaluates `<condition1>` and returns `<expr1>` if it is `true`. Otherwise, the inner `_if()` function is invoked, and the outer `_if()` returns the result of evaluating `<condition2>`. If `<condition2>` is `true`, the outer `_if()` returns `<expr2>`; otherwise, it returns `<expr3>`. You can achieve further recursion if you specify `_if()` for `<expr2>` or `<expr3>`.

Expression arguments are evaluated only when their corresponding conditions are found to hold. For example, in simple, non-recursive uses of the function, `<expr1>` is evaluated only when `<condition>` is `true`. This is called *short-circuit boolean evaluation*. Short-circuit evaluation avoids unnecessary processing and the unwanted side effects of expression evaluation, such as evaluating Perl functions that modify global Perl variables.

Arguments

Argument	Description
<code><condition></code>	A <code>bool</code> , <code>int32</code> , or <code>float</code> expression
<code><expr1></code>	The value returned when <code><condition></code> evaluates to false or 0.
<code><expr2></code>	The value returned when <code><condition></code> evaluates to true or a non-zero value.

Return Value

The return data type of the `_if()` function is the data type of the last argument.

Exceptions

The `_if()` function raises SQL processing exceptions under any of these conditions:

- The condition arguments do not evaluate to a data type of `bool`, `int32`, or `float`.
- The data types of expression arguments do not match the data type of the last argument.

Example

The following query returns the IP addresses for clients which visited the `'/robots.txt'` URL more than three times.

```
SELECT ClientDNS
FROM example_webserve_100
GROUP BY 1
HAVING sum( _if( Url='/robots.txt',1,0 ) ) > 3
DURING ALL;
```

`_iftable()`

The `_iftable()` functions returns different expressions depending on the table being queried.

Synopsis

```
_iftable( <type>, <name1>, <expr1>, [<name2>, <expr2>[...], <else_expr> ] )
```

Description

If the table being scanned is `<name1>`, the `_iftable()` function returns `<expr1>`; otherwise, it tries each successive name. When a match is found, the function returns the corresponding expression. If no match is found, the function returns `<else_expr>`.

Expression arguments to `_iftable()` are evaluated only when the corresponding condition is true. This is called short-circuit Boolean evaluation. Short-circuit evaluation avoids unnecessary work and provides correct behavior for expressions that have side effects, such as Perl functions that modify global Perl variables.

Arguments

Argument	Description
<code><type></code>	The data type of the return value
<code><name></code>	Expressions that evaluate to table names
<code><expr></code>	Expressions returned when the table being scanned matches a specified name
<code><else_expr></code>	The expression returned when the table being scanned matches none of the names

Return values

The return type of the `_iftable()` expression is the data type specified by the first argument.

Exceptions

The `_iftable()` raises an SQL processing exception under any of these conditions:

- The data types of `<name>` arguments are not varchar.
- The data types of `<expr>` arguments do not match the data type specified in the `<type>` argument.

LIST FUNCTIONS

This section describes functions that perform operations on lists of expressions or values.

Function	Purpose	Page
<code>_list()</code>	Create a list variable from a list of values	page 97
<code>_nth()</code>	Return a value from a list variable	page 98

`_list()`

The `_list()` function creates a list variable from a list of values.

Synopsis

```
[@]_list( <expression>[, <expression>[...]] )
```

Description

Evaluates each *<expression>* and makes it available as the corresponding element of the list variable. The list variable returned by the function is used with the `EXPLODE` keyword and the `IN` comparison operator.

For more information on the use of the list variables the `_list()` function returns, see [“EXPLODE Keyword”, on page 92](#) and [“Advanced Comparison Operators”, on page 70](#).

Arguments

Argument	Description
<i><expression></i>	Expressions that evaluate to scalar values

Return Value

`_list()` returns the first element in the list.

`@_list()` returns the list variable.

List variables are sometimes called INTO variables, because the INTO keyword also creates list variables.

For more information, [“INTO Keyword”, on page 91](#).

Example

The following query filters on a comma-separated list of items.

```
SELECT ClientDNS
FROM example_webserve_100
WHERE _strmatch(UserAgent, '([ ^ ]+)', '') IN
      @_list("Mozilla/4.0", "Mozilla/4.08", "Mozilla/4.5")
DURING ALL;
```

`_nth()`

The `_nth()` function returns a value from a list variable.

Synopsis

```
_nth( <type>, <index>, <list_expr> [, <expr>[...]] )
```

Description

Returns the value of the specified item in a list of expressions or values, converting it to the specified *<type>*, if necessary.

For related information, see:

- [Appendix B: Time Zones](#)
- [“_list\(\)”, on page 97](#)
- [“_strmatch\(\)”, on page 134](#)

- “[_strsplit\(\)](#)”, on page 136
- “[_strcat\(\)](#)”, on page 147

Arguments

Argument	Description
<code><type></code>	The data type of the return value
<code><index></code>	An <code>int32</code> expression that evaluates to an index within <code><list_expr></code>
<code><list_expr></code>	A list expression

Return Value

The `_nth()` function returns the value from `<list_expr>` that corresponds to the specified `<index>`. If necessary, the value is converted to the data type specified by `<type>`. The first value in `<list_expr>` has 0 as its index. If the value of `<index>` is negative, it is considered to be 0. If the value of `<index>` exceeds the number of values in the list, the last expression is returned. If any `@` list expressions are present in the list of expressions, their list values will be expanded prior to selection.

Exceptions

The `_nth()` function raises a SQL processing exception if the value selected by the index cannot be converted to the specified data type.

Example

The following query splits the values into a list and returns the value of the fifth item in the list. If `_nth()` has fewer than five items defined as values, the last item will be used.

```
SELECT _nth(int32, 4, @_strsplit("/", Referrer))
FROM example_webserv_100
DURING ALL;
```

LOOKUP FUNCTIONS

This section describes these functions that look up and return values based on values passed as arguments.

Function	Purpose	Page
<code>_lookup()</code>	Look up values stored in external data sources	page 99
<code>_rev_dns()</code>	Looks up the DNS host name for a specified IP address	page 109
<code>_tablematch()</code>	Returns a list of table and view names that match a pattern	page 110

`_lookup()`

The `_lookup()` function uses values stored in an EDW table to retrieve and return values stored in flat files. This function enables you to provide consistent and meaningful values for data that is stored inconsistently. You can also use this function to correlate data between tables.

NOTE: The character encoding of the lookup file must be UTF-8.

Synopsis

```
_lookup( <filename>, <key>, <default_key_value_pair>[, <option_string> ]
```

Description

The `<key>` argument identifies a column in an EDW table or view. The `_lookup()` function uses the value of `<key>` in each table row to search for a corresponding value in the lookup file identified by `<filename>`. If the function finds the `<key>`, it returns the corresponding value from the field in the lookup file. If it finds no match, the function returns `<default_key_value_pair>`, which provides a default value for a specific key.

Use `<option_string>` to identify the formatting of the lookup file. Also use `<option_string>` to specify which field(s) in the lookup file contain values to return to your query. These arguments are optional only because the function provides default values if you do not specify a value. For information about the default values, see [“Basic Lookup Options”, on page 103](#).

You can also use `<option_string>` to tailor the operation of the `_lookup()` function. For example, you can specify how to parse the lookup value associated with `<key>` to extract values that are returned in a list instead of as a single value. For an example of such a query, see [“Example #3: Accepting Multiple Return Values”, on page 106](#).

The data in the lookup file is stored in character-separated fields. The number of fields in a row can vary from row to row.

Arguments

Argument	Description
<code><filename></code>	The fully qualified filename of the lookup file. NOTE: <ul style="list-style-type: none"> If the value of <code><filename></code> begins without a slash (/), the lookup file and its path are relative to the <code>dsroot</code> directory. The lookup file must be located in the same path on all EDW hosts. For more information, see “Working with Lookup Files”, on page 107 .
<code><key></code>	The column in the EDW table whose value is used to look up a value in the lookup file.
<code><default_key_value_pair></code>	If the lookup fails, the function uses the default key and textual equivalent to form the return value. This argument must: <ul style="list-style-type: none"> Evaluate to a constant; that is, it cannot vary by row. Provide a default key value and a corresponding text value for each field represented in the option string. For example, if the option string returns values for columns 2 and 4, <code><default_key_value_pair></code> must provide a default value the key as well as for four fields. The function parses the four fields to return the default value for columns 2 and 4. For an example, see “Example #3: Accepting Multiple Return Values”, on page 106 .

Argument	Description
<code><option_string></code>	<p>Describes the format of the lookup file, identifies the field in the lookup file to use as the key, and specifies which field(s) to return.</p> <p>NOTE:</p> <ul style="list-style-type: none"> The <code><option_string></code> argument must evaluate to a constant; that is, it cannot vary by row. This argument is optional because default values are provided for its two critical options. For more information, “Basic Lookup Options”, on page 103. <p>For more information, see “Specifying Options in the Option-String Argument”, next.</p>

SPECIFYING OPTIONS IN THE OPTION-STRING ARGUMENT

The value of the `<option_string>` is a list of options separated by semi-colons (;). The general syntax is:

```
<option_name>=<setting>[;<option_name>=<setting>[...]]
```

The options are classified as follows:

- [“Basic Parsing Options”, next](#)
- [“Basic Lookup Options”, on page 103](#)

TIP: Some `_lookup()` function options are required or very useful for basic queries. Others are needed only to perform advanced lookups. To facilitate usage of this function, this section provides tips that separate the basic from the advanced options.

- Basic options include:
 - parsing options—the field-separator character in the lookup file. If the file uses whitespace as the separator, you do not need to specify this basic parsing option.
 - lookup options—the field in the lookup file that serves as the key and its data type, and the fields that contain the values to return and their data types
- Advanced options include:
 - parsing options—the quotation-mark character in the lookup file; the function ignores separator characters within the quoted string and strips the quotation marks. You can also specify a non-default escape character that keeps the function from escaping special characters not enclosed within the specified quoting character.
 - lookup options—the character in the lookup file that identifies a comment, which causes the function to ignore text that follows the comment character.

Basic Parsing Options

The basic options to control parsing of the strings in the lookup file are `fieldsep` and `escape`.

- **fieldsep**

```
fieldsep=<style>[,<style>[...]]
```

For the style, specify a comma-separated list of field separators used in the lookup file. The default style is `"whitespace"`, which allows multiple whitespace characters between fields. You can specify `<style>` in any of the following ways:

- `{comma, csv, tab, tsv, pipe, psv, colon, semi, semicolon}`—interprets the separator as any single instance of any of the included characters; this style is the equivalent of `str:<char>`.
- `str:<string>`—sets a literal string as the separator; for example, `fieldsep=str:foo` uses `foo` as the separator.
- `char:<ascii_num>`—specifies the ASCII number of the character to use as the separator; for example, `fieldsep=char:13` specifies a new-line character as the separator.
- `{ws|whitespace}`—sets any number of contiguous whitespace characters as the separator; in other words, allows multiple whitespace characters between fields.

- **escape**

`escape={backslash|none}`

- `backslash`—treats `\<char>` as `<char>`

This setting specifies treatment of the backslash character and field separators as well as ordinary characters. For example, if your escape character is a backslash and your data includes two consecutive backslashes (`\\`), a backslash is treated as part of the data. The escape character defaults to `backslash`.

TIP: Basic usage of the `lookup()` function does not require changing the default value for the `escape` option. If you change the value to `none`, you must also set the `quoting` option.

- `none`—special characters cannot be escaped to be included in data values unless the data values are enclosed with the quoting character.

Advanced Parsing Option

The function also provides the `quoting` parsing options, which allow you to specify the style of the quotation mark character. Its syntax is:

`quoting=<style>[,<style>[...]]`

Comma-separated list of quoting options allowed, defaults to “`singleordouble`”.

- `{single|single-quote}`—if single-quotes are found around a string field, then ignore separator characters in that string and also strip the quotes.
- `{double|double-quote}`—same as single-quote, but for double-quotes
- `char:<ascii_num>`—specifies the ASCII number of the character to use as the quotation symbol; for example, `quoting=char:37` sets the percent (%) character as the quoting character

Advanced Usage: Rule of Thumb When Specifying Parsing

If the data in your lookup file contains field separators, use quoting options to isolate the data in specific fields. If the data in your lookup file contains quotation marks, use escape characters to isolate the data in specific fields.

For example, assume your escape character is `backslash` and your field separator is a comma. Assume also that the lookup data associated with a `<key>` has the following fields, one of which uses a comma (,) to separate an apartment number (#416) into a different field from street address (9617 Delaware St.):

```
234, Vijay Kumar, 9617 Delaware St., #416, Berkeley, CA, 94705
567, Jose Sanchez, 123 Main St., Oakland, CA, 94612
```


To force the apartment number into the same field as street address, you can use `escape=backslash` and format the data row as follows:

```
234, Vijay Kumar, 9617 Delaware St.\, #416, Berkeley, CA, 94705
```

Alternately, you could use `quoting=single` and format the data row as follows:

```
234, Vijay Kumar, '9617 Delaware St., #416', Berkeley, CA, 94705
```

NOTE: If commas, single quotes, and double quotes are part of your data, you can use `fieldsep=char:<n>`, where `<n>` is the ASCII number of a character that is not in your data. For example, use `fieldsep=char:037` to specify a percent symbol (%) as the separator and format the data row as follows:

```
234% Vijay Kumar% 9617 Delaware St., #416% Berkeley% CA% 94705
```

NOTE: The `_strsplitxsv()` function uses these same parsing options. To learn more about the function, see “[_strsplitxsv\(\)](#)”, on page 136.

Basic Lookup Options

Use the following options to control how to perform the lookup.

- **keycol**

```
keycol=<n>[:<type>]
```

This option identifies the field in the lookup file to use as the key when performing the lookup. Indexing begins at 1. You can append the field number with an optional type name, which forces the EDW engine to parse this column as the specified data type. The following example specifies that the key is the third field in the lookup file and the value of this field should be treated as a 32-bit integer:

```
keycol=3:int32
```

Allowable types are `int32`, `int64`, `float`, `timestamp`, and `varchar`. The default for the `keycol` option is `1:varchar`. The default for `<type>` is `varchar`.

- **valcols**

```
valcols=<column_spec>[, <column_spec>[...]]
```

This option specifies which fields to return when performing lookups. The format for `<column_spec>` is:

```
<n>[:<type>]
```

Indexing begins at 1. You can append each column number `<n>` with an optional specification of its data type. The default for the `valcols` option is `2:varchar`. The default for `<type>` is `varchar`.

Advanced Lookup Option

The function also provides the `commentprefix` lookup option, which allows you to configure the function to ignore comments in the lookup file. Its syntax is:

```
commentprefix=<style>
```

Typically, the value of `<style>` is the string that begins a comment line. Examples include `#` (shell-style or perl-style), `--` (SQL-style), or `//` (C++-style).

If the value of `<style>` starts with the string `inline:`, the function locates comments on lines with actual values. For example, if you specify `inline:#`, a lookup file can contain lines like the following:

```
# normal comment
# another normal comment -- whitespace is allowed.
123,col2val,col3val    # this is an inline comment (including the whitespace)
124,col2val # this is a comment, col3val #<-- oops, col3val is ignored
```

NOTE:

- In the example above, the value in the third column (**col3val**) is ignored because it is included in the inline comment.
- The default for `<style>` is `""`; that is, no comments are allowed in the lookup file.

Examples

This section provides three examples of `_lookup()` function usage:

- [“Example #1: Providing Meaningful and Consistent Values”, next](#)
- [“Example #2: Correlating Data Between Tables”, on page 106](#)
- [“Example #3: Accepting Multiple Return Values”, on page 106](#)

EXAMPLE #1: PROVIDING MEANINGFUL AND CONSISTENT VALUES

A common usage for the `_lookup()` function is to provide consistent and meaningful values for data that is stored inconsistently. For example, data collected from multiple sources often represents "success" and "failure" in different ways. Typically these values are stored as integers to reduce disk space. You can create a lookup file that stores corresponding text values, and use `_lookup()` to convert integer values stored in the EDW to text values that are meaningful to your users.

SenSage AP IntelliSchema presents examples of such lookup-file usage. IntelliSchema is a pre-defined data structure that uses SQL views to present unified access to data from disparate systems. Analytics reports generally query IntelliSchema views to display normalized event data for common event types from multiple information systems.

For example, one set of IntelliSchema views returns user-login information. Each view creates a **result** column to store integer values that represent the success or failure of the login. Each view uses the `userLogin.lookup` file to provide meaningful textual equivalents to the stored integer values. The lookup file contains all possible login values and their textual equivalent. Three lines are:

```
-1,Unknown
0,Failure
1,Success
```

To return meaningful values about the success of the login, the user-login IntelliSchema views include the following statement in the target list of the SELECT statement:

```
_lookup("<path>/userLogin.lookup", result, "-1,Unknown", "fieldsep=comma;keycol=1:int32;valcols=2:varchar") AS result,
```

Given that the general syntax is:

```
_lookup( <filename>, <key>, <default_key_value_pair>[, <option_string> )
```

The statement above uses the `_lookup()` function to:

- Locate the lookup file: "`<path>/userLogin.lookup`"
- Specify which column in the table represents the key: `result`
- Specify the default key/value pair: "`-1,Unknown`"
- Specify which field in the lookup file contains a value that should match a value in the key column: `keycol=1:int32`
- Specify which field in the lookup file to return as the value of the key column: `valcols=2:varchar`

When this statement is run against an EDW table, the EDW engine steps through the table and retrieves the value of the **result** column for each row. It compares the value to one of the values in the first field of the lookup file. If it finds a match, it displays the value of the second field as the output of the **result** column.

Because the default value is "`-1,Unknown`", if no value in the **result** column matches a value in the first field of the lookup file or the value of the **result** column evaluates to `-1`, the query output displays `Unknown` as the value of the **result** column.

NOTE: The default value is specified as two fields to match the two fields specified in the option string:

- the value of the key column as stored in the EDW table
- the corresponding lookup-file value that should be returned to represent the value of the key column.

The IntelliSchema login views contain the **privilege** column as well as the **result** column. The **privilege** column stores integer values that represent the login type: either standard or privileged. Because the values stored in this column are different from those in the **result** column, the `userLogin.lookup` file contains two additional lines. The full lookup file contains:

```
-1,Unknown
0,Failure
1,Success
20,Privileged
21,Standard
```

As illustrated above, the same lookup file enables lookup for two different columns. The user-login IntelliSchema views include both of the following statements in the target list of the SELECT statement:

```
_lookup("<path>/userLogin.lookup",result,"-
1,Unknown","fieldsep=comma;keycol=1:int32;valcols=2:vchar") AS result,
_lookup("<path>/userLogin.lookup",privileged,"-
1,Unknown","fieldsep=comma;keycol=1:int32;valcols=2:vchar") AS privileged
```

The second statement behaves like the first but, when this statement is run against an EDW table, the EDW engine steps through each row of the table and retrieves the value of the **privilege** column.

EXAMPLE #2: CORRELATING DATA BETWEEN TABLES

Data often derives from disparate sources and applications. You can use key-field information to link the data across EDW tables and lookup files.

For example, assume your EDW instance uses separate tables to store data about sales staff and customers. The **sales** table stores the sales person's name, territory, and phone number. The **customer** tables stores company name, street address, city, state, and country. Because territories change frequently, assume also that your site uses an application to dump territory information regularly into a flat file. The flat file contains each territory and the states they contain.

To report all customers assigned to each salesperson, you would create a query that:

- Retrieves each customer's state from the **customer** table
- Matches each state to a territory by running `_lookup()` against the territory file
- For each territory, locates the salesperson from the **sales** table

EXAMPLE #3: ACCEPTING MULTIPLE RETURN VALUES

Whereas [Example #1: Providing Meaningful and Consistent Values](#) above illustrates usage of the `_lookup()` function in the SELECT statement, the example below illustrates usage of the function in the WHERE clause. It also illustrates use of the INTO keyword to accept multiple return values and to put them in a named list variable.

The lookup data for this example is similar to that used in [“Advanced Usage: Rule of Thumb When Specifying Parsing”](#), on page 102:

```
234, Vijay Kumar, 9617 Delaware St.\, #416, Berkeley, CA, 94705
567, Jose Sanchez, 123 Main St., Oakland, CA, 94612, 415-644-9753
```

NOTE:

- The first row above uses the backslash (\) character to escape the comma (,) . If the comma were not escaped, it would separate an apartment number (#416) into a different field from street address.

To keep the apartment number in the same field as the street address, you must specify the backslash as the escape character in your `_lookup()` call.

- The second row above contains a phone number in addition to the standard information.

The `_lookup()` function does not require every row in the lookup file to contain the same number of fields. However the fields you return must be in the same position in every row of the file.

The query below specifies settings for the `fieldsep`, `escape`, `keycol`, and `valcols` options:

```
SELECT
    UserId,
    userinfo[1] as UserName,
    userinfo[2] as State
FROM
    sometable
WHERE
    _lookup("/tmp/users", UserId, "-1,anonymous,-,-,-",
    "fieldsep=comma;keycol=1:int32;valcols=2,5;escape=backslash")
    INTO userinfo != -1
DURING ALL;
```

The query above uses the value of the **UserId** column to locate each user's name and state from the lookup file and loads the values into the `userinfo` list. This query differs from the basic example, which used a value in the lookup file (`Unknown`) as its default value. If no value in the **UserId** column matches a value in the first field of the lookup file or the value of **UserId** evaluates to `-1`, the query output displays `anonymous` as the value of the **UserName** column. This query also specifies a hyphen (`-`) as the default value for the remaining columns. In other words, if no value in the **UserId** column matches a value in the first field of the lookup file or it evaluates to `-1`, the query output displays a hyphen as the default value of column 5 (`State`) as well as the default value for columns 3 and 4.

For information on the `INTO` keyword, see [“INTO Keyword”, on page 91](#).

Exceptions

The `_lookup()` function raises SQL processing exceptions under any of the following conditions:

- The `<key>` or the fields in `<default_key_value_pair>` cannot be parsed as the data type specified in `<options_string>`.
- The file specified by `<filename>` cannot be opened or read.
- Memory to hold the cached lookup file is exhausted.

Working with Lookup Files

Typically, the lookup file is either a static file that rarely changes, or the result of a “dump” from an application or corporate database, such as the list of active users.

IMPORTANT: To use the `_lookup()` function, the lookup file has to be propagated to every host in the EDW instance so that the EDW can read it. Specifically, this file must have the same path on every host and be readable by the EDW server running on each system (for example, readable by the “lms” user). One implementation, for example, is to mount the directory containing the lookup file onto each system. For an alternative configuration, see [“Centralizing the Lookup File”, on page 109](#).

DEBUGGING AND SCALE-UP

NOTE: The `_lookup()` function works by loading the entire file into memory. If a lookup file is too big, the EDW starts paging to disk, and eventually ends your query. For this reason, Hexis Cyber Solutions recommends testing on subsets of large lookup files first, then increasing the subset until the whole file has been loaded.

Caching the Lookup Data: Requirements and Limitations

To enhance performance, the lookup file is cached in memory. The first time it is called, the `_lookup()` function does the following:

- 1 Loads a lookup table from the lookup file into main memory.
- 2 Performs subsequent calls to this cached main memory representation.

IMPORTANT: Because the lookup data is cached:

- The administrator must manually distribute the lookup file to every host in the EDW instance and ensure that each copy is identical. For more information, see [“Working with Lookup Files”, on page 107](#). For more information on distributing the files, see [.Copying Files and Directories to Each Host \(clsync\)](#) in Chapter 2, “Managing HawkEye AP” of the Administration Guide. The lookup data must fit into memory. For more information, see [“Calling the _lookup\(\) Function While Loading or Querying”, next](#).

NOTE: The following is also true of the lookup data:

- The administrator must manually extract the data into the lookup file.

For example, `_lookup()` cannot automatically download a webpage (URL) that contains lookup data.

- The lookup key values must be unique.

The `_lookup()` function assumes that each key in the lookup file matches at most one record. This function cannot perform searches that scan or return multiple records.

- There is no support for XML-based lookup files.

Binary data is supported, but it cannot contain NULL characters (ASCII 0) or new-line characters (ASCII 13).

- There is no support for network-based or dynamic lookup data.

CALLING THE _LOOKUP() FUNCTION WHILE LOADING OR QUERYING

As with other EDW features, lookups can be performed during loads, during queries, or both. To use the `lookup()` function during loading, include it in the SELECT statement in the PTL.

Generally, IgniteTech recommends performing lookups during loads rather than during queries:

- Lookups into giant lookup files require lots of memory. More memory is available during loads.
- The amount of memory required is more consistent during loads. The amount of memory required during queries depends on the details of the query, such as the number of groups that the `GROUP BY` and `SLICE BY` clauses specify.
- If multiple queries use the results of the lookups, they can share the results if the results have been loaded to disk, instead of each performing the lookup.

- Updates to lookup files occur intermittently, not continuously. It can be easier to determine the currency of the lookup data if the lookup occurs during loads. When each query performs its own lookups, it is more difficult to determine which version of the lookup data was used.
- Performing lookups while loading makes the SELECT statement used for queries easier to read.

However, lookups during queries makes sense under these conditions:

- If you want to save storage space, perform lookups at query time to avoid creating additional columns.
- Query-time lookups can improve load performance.

IMPORTANT: IgniteTech recommends that you use your own data to test the advantages of load-time and query-time lookups to determine which approach works best in your situation.

The best reason to perform query-time lookups is the need for “fresher” results. For example, assume you use this function in a query that returns the top 10 users from California. If you use load-time lookups, the precise definition of this query is “top 10 users whose traffic had come from California at the time of the request”. If you use query-time lookups, the precise definition is “top 10 users who currently live in California, regardless of where they lived at the time of the request”.

In some cases, security analysts will want to run both types of queries, and you will have to perform both load-time and query-time lookups.

CENTRALIZING THE LOOKUP FILE

It is possible to store the lookup file on a host that uses a network file system (such as Samba or NFS) to access each host in the EDW instance. This configuration simplifies the updating process because you no longer have to copy the lookup file to every host in the instance (for example, using `clsync`). Generally, centralizing the lookup file makes sense only if the lookups are performed rarely; otherwise, concurrent queries can clog the network or file server.

To estimate the performance impact of centralizing the lookup file, assume that each host in the instance needs to read the file completely for each lookup call. If you have 10 EDW hosts, and each mounts the same file server, and both the EDW hosts and file server are using 100mbps network connections, each EDW host achieves at most 10mbps as they clog the server with requests. This performance concern diminishes if:

- the lookup file is small—because the EDW hosts can cache the whole file
- the file server can keep up with the EDW instance, for example, lots of disks reading in parallel plus faster networking like gigabit ethernet (1gbps) or 10gigE (10gbps).

`_rev_dns()`

The `_rev_dns()` function performs a kind of reverse DNS look up. Instead of starting with a DNS host name and returning the IP address of the host, the function takes an IP address and returns the DNS host name.

When this function is used, it loads the cache file, `dns_cache.txt`, and performs a DNS host name look-up on the file; if the information is not available, it queries the DNS server and stores

the result in `dns_cache.txt`. This cache file is stored in the AppSrvr directory, `/<prefix>/var/log/sls/<instance>/runtime/AppSrvr`.

Synopsis

```
_rev_dns( <IP_address> )
```

Description

Returns a string containing the host name for the server with the specified IP address.

Arguments

Argument	Description
<code><IP_address></code>	<p>The IP address to look up:</p> <ul style="list-style-type: none"> • If <code><IP_address></code> is an <code>int32</code> value, the IP address must be an <code>inet4</code> address in network byte order. • If <code><IP_address></code> is a <code>varchar</code> value, the IP address must be a presentation form of either a IPv4 or IPv6 address. • If <code><IP_address></code> is <code>inet</code>, the IP address must be an <code>inet</code> IPv4 or IPv6 address..

Return Value

The `_rev_dns()` function returns a `varchar` value containing the host name.

Exceptions

The `_rev_dns()` function raises a SQL processing exception when the data type of `<IP_address>` is neither `int32`, `varchar`, or `inet`.

Examples

If the argument held the `inet32` value of 167772597 (`inet4` address, in network byte order) for IP address 10.0.1.181, then the `_rev_dns()` function returns the DNS host name `qal.qaad..com`.

If the argument held the VARCHAR presentation of the IPv4 address "10.0.1.181" then the `_rev_dns()` function returns the DNS host name `qal.qaad..com`.

If the argument held an INET IPv4 address of (`_inet("10.0.1.181")`) then the `_rev_dns()` function returns the DNS host name `qal.qaad..com`.

_tablematch()

If preceded by the `@` expression, the `_tablematch()` function returns a list of table and view names that match a specified pattern and that are located in the default or specified namespace. If not preceded by the `@` expression, this function returns an integer that represents the number of matching tables and views found.

Synopsis

```
[@]_tablematch( <pattern>[, <namespace>[, <option_string>]]
```


Description

The `_tablematch()` function matches the specified regular expression to return a list of names that represent log tables and views of log tables in the default or specified namespace.

NOTE: The `_tablematch()` function does not return the names of system tables.

The value of `<pattern>` must:

- be a regular expression

For example, to represent a SQL string expression that contains a backslash (`\`), you must escape the backslash with another backslash, as `"\\\"`.

- evaluate to a constant string expression

The value must represent the actual names of tables. It cannot be a variable expression or other text that requires further processing to return the names of the tables. The names must be represented as strings within quotation marks.

The value for `<namespace>` defaults to an empty string (`""`). This value is relative to the default namespace, which is determined by the `--namespace` option of the `atquery` command. The function looks in the default namespace for table names that match `<pattern>`; specify a value for `<namespace>` to restrict the match to a namespace within the default.

If a value is specified for the namespace option of the `_tablematch()` function, the specified namespace is concatenated to the default namespace. The table below presents examples of how the query engine evaluates the namespace:

<code>atquery --namespace</code>	<code>_tablematch(<pattern> <namespace>)</code>	Namespace Used	Example
none specified	none specified	default as configured for EDW	default
specified as <code>IT</code>	none specified	value specified by <code>atquery --namespace</code>	<code>IT</code>
none specified	specified as <code>firewalls</code>	default as configured for EDW precedes value specified by <code><namespace></code>	<code>default.firewalls</code>
specified as <code>IT</code>	specified as <code>firewalls</code>	value specified by <code>atquery --namespace</code> precedes value specified by <code><namespace></code>	<code>IT.firewalls</code>

NOTE:

- When you run the query from SenSage AP Console, the namespace is the one specified in the report definition or changed at runtime by the user. For more information, see the Options in Chapter 3, "Loading, Querying, and Managing the EDW in the *Administration Guide* for the `atquery` utility.
- The value for namespace must be a constant string expression.

The value for `<option_string>` also defaults to an empty string (`""`). Specify `"r"` as `<option_string>` to search recursively through all subordinate namespaces, beginning with the namespace stem that concatenates the default namespace and the optional `<namespace>` argument. Like the other arguments, the value for the option string must be a constant string expression.

IMPORTANT: If you specify the third option, you must also specify a value for the namespace option, even to use the default namespace. In other words, to search recursively through all subordinate namespaces within the default namespace, you would specify:

```
FROM @_tablematch("some_pattern", "", "r" )
```

NOTE: All tables and views with names that match the specified pattern must share a common schema. The requirements are the same as those for running a subquery: column names and data types must match.

Arguments

Argument	Description
<code><pattern></code>	A regular expression to match the table names.
<code><namespace></code>	<p>(Optional.) The namespace in which the tables and views to be matched are located. The namespace is relative to the default namespace.</p> <p>IMPORTANT: This option defaults to an empty string (<code>""</code>) . You must specify a value for this option if:</p> <ul style="list-style-type: none"> You want to limit the match to a branch within the default namespace. In this case, specify the full subordinate namespace. You specify a value for <code><option_string></code>. In this case, you must specify a value for the namespace argument. If you do not want to specify a branch within the default namespace, specify an empty string (<code>""</code>) .
<code><option_string></code>	(Optional.) Specify <code>"r"</code> to indicate that you want a recursive match, in which the function looks in <code><namespace></code> and all of its subordinate namespaces.

Return Value

If not preceded by the `@` expression, the `_tablematch()` function returns an integer that represents the number of tables and views that match the regular expression. If preceded by the `@` expression, this function returns a list of the matching table and view names. The returned names are fully qualified. You can access the list in the same way that you access the lists returned by list functions.

For more information, see [“Working with Lists”, on page 89](#).

Example

The following `SELECT` statement uses the `_tablematch()` function to declare an implicit union of the tables located in the default namespace and its subordinate namespaces.

```
SELECT *
  FROM @_tablematch("syslog_.*", "", "r")
 DURING ALL
;
```

The match above represents all tables that begin with `syslog_`, such as `syslog_ab`, `syslog_bc`, and `syslog_cd`. This pattern follows standard regular expression syntax. If it had been written without the dot (`.`) character, it would have returned only names that terminated in the underscore (`_`) character.

TIP: Specify `".*" as <pattern>` to match all table and view names.

The following queries uses the `_tablematch()` function to indicate how many tables and views match the search expression.

```
-- The SELECT below returns a count of values for some_field found in each table
WITH subq1 as (SELECT _fromname() AS src, some_field, count(*) as cnt from
               @_tablematch("syslog_.*", "", "r") GROUP BY 1,2 DURING ALL)

-- The SELECT below returns the # of tables that contain a value for 'some_field'
WITH subq2 as (SELECT some_field, count(*) as cnt from subq1 GROUP BY 1)

-- The SELECT below returns the # of times a value for 'some_field'
-- was found out of 'total_tables'
SELECT some_field, cnt, @_tablematch("syslog_.*", "", "r") as total_tables
from subq2;
```

AGGREGATION FUNCTIONS

This section describes functions that perform aggregation on groups of rows.

Function	Purpose	Page
avg()	Computes the average of the values in the group	page 113
count()	Returns the number of rows in the group	page 114
max()	Returns the maximum value in a group	page 115
min()	Returns the minimum value in a group	page 116
median()	Computes the medium of the values in a group	page 117
sum()	Returns the sum of the values in a group	page 118
_first()	Returns the first value in a group	page 118
_last()	Returns the last value in a group	page 119
_strsum()	Returns the concatenation of the string values in a group	page 119

For more information on the use of aggregation functions, see [“GROUP BY Clauses and Aggregation Queries”, on page 52](#).

avg()

The `avg()` aggregate function computes the average of the values in the group.

Synopsis

```
avg( [DISTINCT] <column_expression> )
```

Description

Returns the average of the values in `<column_expression>` for all rows in a group.

Arguments

Argument	Description
<code><column_expression></code>	The target column with values to average. The allowed data types of the expression are <code>int32</code> , <code>int64</code> , <code>float</code> , and <code>timestamp</code> .

Return Value

The return value is the average. The data type of the return value generally matches the data type of `<column_expression>`.

Exceptions

The `avg()` function raises an SQL processing exception under these conditions:

- No column expressions are provided.
- A column expression has a data type that is not allowed.

Example

The `SELECT` statement returns the average number of bytes for all the records in the specified table:

```
SELECT avg(RespSize)
FROM example_webserve_100
DURING ALL;
```

count()

The `count()` aggregate function returns a count of the rows in the group.

Synopsis

```
count (*)
```

```
count ( DISTINCT <column_expression> )
```

Description

Returns the number of items in a group.

Arguments

Argument	Description
<code><column_expression></code>	The target column with values to count. The argument is ignored unless the <code>DISTINCT</code> modifier keyword is specified

Return Value

The `count()` function returns an `int64` value representing the number of items in the group.

Examples

The following `SELECT` statement returns the number of records in the specified table:

```
SELECT count(*)
FROM example_webserv_100
DURING all;
```

max()

The `_max()` aggregate function returns the maximum value in a group.

Synopsis

```
max( <column_expression> )
```

Description

The `_max()` aggregate function returns the maximum value of the given expression for all the items in a group.

Arguments

Argument	Description
<code><column_expression></code>	The target column with values in which to find the maximum value.

Return Values

Generally, the `max()` aggregate function returns the maximum of all values in the group. The data type of the return value is the same as the data type of `<column_expression>`.

The return value of the `max()` aggregate function has special meaning for these data types: [“Aggregation Functions”, on page 113](#) [“Aggregation Functions”, on page 113](#) [“Lookup Functions”, on page 99](#)

Data type of <code><column_expression></code>	Meaning of the Return Value
<code>bool</code>	Returns a <code>bool</code> representing the logical OR of all the values
<code>timestamp</code>	Returns a <code>timestamp</code> representing the date and time-of-day furthest in the future from midnight, January 1, 1970
<code>varchar</code>	Returns a <code>varchar</code> containing the maximum

Exceptions

The `max()` aggregate function raises a SQL processing exception if passed less than one expression.

Example

The following query returns the latest record for all the records in the `example_webserv_100` table:

```
SELECT max(ts)
FROM example_webserv_100
DURING all;
```

min()

The `_min()` aggregate function returns the minimum value in a group.

Synopsis

```
min( <column_expression> )
```

Description

The `min()` aggregate function returns the minimum value of the given expression for all the items in a group.

Arguments

Argument	Description
<code><column_expression></code>	The target column with values in which to find the minimum value; data type is <code>bool</code> or <code>varchar</code> .

Return Values

Generally, the `min()` aggregate function returns the minimum value of all values in the group. The data type of the return value is the same as the data type of `<column_expression>`.

The return value of the `max()` aggregate function has special meaning for these data types:

Data type of <code><column_expression></code>	Meaning of the Return Value
<code>timestamp</code>	Returns a <code>timestamp</code> representing the date and time-of-day closest to midnight, January 1, 1970.

Exceptions

The `min()` aggregate function raises a SQL processing exception under any of these conditions:

- No expressions are passed as arguments.
- The data type of `<column_expression>` is `bool` or `varchar`.

Example

The following query returns the earliest record for all the records in the `example_webserv_100` table:

```
SELECT min(ts)
FROM example_webserv_100
DURING all;
```

median()

The `_median()` aggregate function returns the median value in a group.

Synopsis

```
median( <column_expression> )
```

Description

The `median()` aggregate function returns the median value of the given expression for all the items in a group.

Arguments

Argument	Description
<code><column_expression></code>	The target column with values to calculate the median. The allowed data types of the expression are <code>int32</code> , <code>int64</code> , <code>float</code> , and <code>timestamp</code> .

Return Value

The return value is the median of the items in the group. The data type of the return value generally matches the data type of `<column_expression>`. If the number of items in the group is even, then median returns the arithmetic means of `<column_expression>` for the two middle items.

Exceptions

The `median()` function raises an SQL processing exception under these conditions:

- No column expressions are provided.
- A column expression has a data type that is not allowed.

Example

The `SELECT` statement returns the median number of bytes for all the records in the specified table:

```
SELECT median(RespSize)
FROM example_webserv_100
DURING ALL;
```

sum()

The `_sum()` aggregate functions return the sum of values in a group.

Synopsis

```
sum( <column_expression> )
```

Description

The `sum()` aggregate returns the sum of the given expression for all the items in a group.

Arguments

Argument	Description
<code><column_expression></code>	The target column with numeric values to sum; data type is <code>timestamp</code> or <code>varchar</code> .

Return Values

The return value of the `sum()` aggregate function has special meaning, depending on the data type of `<column_expression>`:

Data type of <code><column_expression></code>	Meaning of the Return Value
<code>bool</code>	Returns a <code>bool</code> returns <code>false</code> if all the values in the group are <code>false</code> ; otherwise it returns <code>true</code> .
<code>int32</code> <code>int64</code>	Returns an <code>int64</code> containing the sum of all the values in the group.
<code>float</code>	Returns a <code>float</code> containing the sum of all the values in the group.

Exceptions

The `sum()` aggregate function raises a SQL processing exception under any of these conditions:

- No expressions are passed as arguments.
- The data type of `<column_expression>` is `varchar`.

Examples

The following query returns the total number of bytes returned for all the records in the `example_webserv_100` table:

```
SELECT sum(RespSize)
FROM example_webserv_100
DURING all;
```

_first()

The `_first()` aggregate function returns the first value in a group.

Synopsis

```
_first( <column_expression> )
```

Description

Returns the first value of the given expression from the first row in a group.

Generally, you use the `_first()` function in conjunction with an `ORDER BY` clause.

Arguments

Argument	Description
<code><column_expression></code>	The target column in the first row in the group from which the return value is taken.

Return Value

The `_first()` function returns the value of `<column_expression>` from the first row in the group.

_last()

The `_last()` aggregate function returns the last value in a group.

Synopsis

```
_last( <column_expression> )
```

Description

Returns the value of the given expression from the last row in a group.

Generally, you use the `_last()` function in conjunction with an `ORDER BY` clause.

Arguments

Argument	Description
<code><column_expression></code>	The target column in the last row in the group from which the return value is taken.

Return Value

The `_last()` function returns the value of `<column_expression>` from the last row in the group.

_strsum()

The `_strsum()` aggregate function returns the string concatenation of all the values in a group.

Synopsis

```
_strsum( varchar_column_expression )
```

Description

The `_strsum()` aggregate function returns the string concatenation of the given expression for all the items in a group.

Arguments

Argument	Description
<code><varchar_column_expression></code>	Target column expression

Return values

The `_strsum()` aggregate function returns a `varchar` containing the string concatenation of all the values in the group.

STATISTICAL AGGREGATE FUNCTIONS

This section describes functions that perform statistical calculations on the values of a named column. If the query contains a "group by" clause, the calculation is made for each group of rows returned by the query; otherwise, the calculation is performed on all rows returned by the query.

Function	Purpose	Page
<code>var_pop()</code>	Returns the population variance of the values in the named column.	page 120
<code>stddev_pop()</code>	Returns the population standard deviation for the values of named column.	page 121
<code>var_samp()</code>	Returns the sample variance of the named column.	page 122
<code>stddev_samp()</code>	Returns the sample standard deviation for the values of named column.	page 123
<code>variance()</code>	Alias for the <code>var_samp()</code> function.	page 124
<code>stddev()</code>	Alias for the <code>stddev_samp()</code> function.	page 124

`var_pop()`

Returns the population variance of the values in the named column.

Synopsis

```
var_pop(column_expression)
```

Description

The `var_pop()` function returns the population variance of the values in the named column for each grouping of rows that results from a "group by" clause, or for all the rows if there is no "group by" clause.

POPULATION VARIANCE CALCULATION

The population variance of N numbers {x1, x2, x3, ..., xN} is defined as the sum of the squares of the difference between each number and the mean of the numbers, divided by N, as shown in the table below:

Column values	Mean	Difference from mean	Square of difference from mean
10	20	-10	100
20	20	0	0
30	20	10	100
25	20	5	25
15	20	-5	25
Sum of the squares of differences			250
Population variance			50

Arguments

Argument	Description
<code>column_expression</code>	A column expression that evaluates to one of the following datatypes: <ul style="list-style-type: none"> • int32 • int64 • float • timestamp (calculations are made using the number of seconds since epoch)

Return Value

The function returns the population variance as a FLOAT datatype.

`stddev_pop()`

Returns the population standard deviation for the values of named column.

Synopsis

```
stddev_pop(column_expression)
```

Description

The `stddev_pop()` function returns the population standard deviation of the values in the named column for each grouping of rows that results from a "group by" clause, or for all the rows if there is no "group by" clause.

POPULATION STANDARD DEVIATION CALCULATION

The population standard deviation of a group of numbers is the positive square root of the population variance (See “[var_pop\(\)](#)”, on page 120. In the example below, the population variance is 50 and the population standard deviation is 7.07106, the square root of 50.

Column values	Mean	Difference from mean	Square of difference from mean
10	20	-10	100
20	20	0	0
30	20	10	100
25	20	5	25
15	20	-5	25
Sum of the squares of differences			250
Population variance			50
Population standard deviation			7.0710678

Arguments

Argument	Description
<code>column_expression</code>	A column expression that evaluates to one of the following datatypes: <ul style="list-style-type: none"> • int32 • int64 • float • timestamp (calculations are made using the number of seconds since epoch)

Return values

The function returns the population standard deviation as a FLOAT datatype.

`var_samp()`

Returns the sample variance of the named column.

Synopsis

```
var_samp(column_expression)
```

Description

The `var_samp()` function returns the sample variance of the values in the named column for each grouping of rows that results from a "group by" clause, or for all the rows if there is no "group by" clause.

SAMPLE VARIANCE CALCULATION

The sample variance of N numbers {x1, x2, x3, ..., xN} is defined as the sum of the squares of the difference between each number and the mean of the numbers, divided by N-1, as shown in the table below:

Column values	Mean	Difference from mean	Square of difference from mean
10	20	-10	100
20	20	0	0
30	20	10	100
25	20	5	25
15	20	-5	25
Sum of the squares of differences			250
Sample variance			62.5

Arguments

Argument	Description
<code>column_expression</code>	A column expression that evaluates to one of the following datatypes: <ul style="list-style-type: none"> • int32 • int64 • float • timestamp (calculations are made using the number of seconds since epoch)

Return Value

The function returns the sample variance as a FLOAT datatype.

`stddev_samp()`

Returns the sample standard deviation for the values of named column.

Synopsis

```
stddev_samp(column_expression)
```

Description

The `stddev_samp()` function returns the sample standard deviation of the values in the named column for each grouping of rows that results from a "group by" clause, or for all the rows if there is no "group by" clause.

SAMPLE STANDARD DEVIATION CALCULATION

The sample standard deviation of a group of numbers is the positive square root of the sample variance (See “[var_samp\(\)](#)”, on page 122). In the example below, the sample variance is 62.5 and the sample standard deviation is 7.9056, the square root of 62.5.

Column values	Mean	Difference from mean	Square of difference from mean
10	20	-10	100
20	20	0	0
30	20	10	100
25	20	5	25
15	20	-5	25
Sum of the squares of differences			250
Sample variance			62.5
Sample standard deviation			7.9056

Arguments

Argument	Description
<code>column_expression</code>	A column expression that evaluates to one of the following datatypes: <ul style="list-style-type: none"> • int32 • int64 • float • timestamp (calculations are made using the number of seconds since epoch)

Return values

The function returns the sample standard deviation as a FLOAT datatype.

`variance()`

The `variance()` function is an alias for the `var_samp()` function. The functions may be used interchangeably. For details, see “[var_samp\(\)](#)”, on page 122.

`stddev()`

The `stddev()` function is an alias for the `stddev_samp()` function. The functions may be used interchangeably. For details, see “[stddev_samp\(\)](#)”, on page 123.

LOGARITHMIC AND EXPONENTIAL FUNCTIONS

This section describes functions that perform logarithmic and exponential calculations.

Function	Purpose	Page
_log()	Returns the natural logarithm of a value	page 125
_log10()	Returns the base 10 logarithm of a value	page 125
_pow()	Returns a power of a value	page 126
_exp()	Returns a power of e	page 126

_log()

The `_log()` function returns the natural logarithm of a value.

Synopsis

```
_log( <numeric_expression> )
```

Description

The `_log()` function returns the natural logarithm of `<numeric_expression>`.

Arguments

Argument	Description
<code><numeric_expression></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value.

Return Value

The data type of the return value from the `_log()` function is `float`.

_log10()

The `_log10()` function returns the base 10 logarithm of a value.

Synopsis

```
_log10( <numeric_expression> )
```

Description

The `_log10()` function returns the base 10 logarithm of `<numeric_expression>`.

Arguments

Argument	Description
<code><numeric_expression></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value.

Return Value

The data type of the return value from the `_log10()` function is `float`.

_pow()

The `_pow()` function returns the power of a value.

Synopsis

```
_pow( <numeric_expression>, <power> )
```

Description

The `_pow()` function returns the `<power>` of `<numeric_expression>`.

Arguments

Argument	Description
<code><numeric_expression></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value. This value is the base in the computation.
<code><power></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value. This value is the exponent in the computation.

Return Value

The data type of the return value from the `_pow()` function is `float`.

_exp()

The `_exp()` function returns a power of e.

Synopsis

```
_exp( <power> )
```

Description

The `_exp()` function returns e raised to the power specified by `<power>`.

Arguments

Argument	Description
<code><power></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value. This value is the exponent in the computation; e is the base.

Return Value

The data type of the return value from the `_exp()` function is `float`.

NUMERIC ROUNDING FUNCTIONS

This section describes functions that perform numeric rounding.

Function	Purpose	Page
_abs()	Computes the absolute value of an expression	page 127
_ceil()	Rounds a numeric value up to the nearest integer value	page 127
_floor()	Rounds a numeric value down to the nearest integer value	page 128
_round()	Performs fractional rounding to a specified precision	page 128

[_abs\(\)](#)

The `_abs()` function computes the absolute value of an expression.

Synopsis

```
_abs( <expression> )
```

Description

The `_abs()` function computes the absolute value of `<expression>`.

Arguments

Argument	Description
<code><expression></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value.

Return Value

The `_abs()` function returns the absolute value of `<expression>`, in the same data type.

Exceptions

The `_abs()` function raises an SQL processing exception if the data type of `<expression>` is not `int32`, `int64`, or `float`.

[_ceil\(\)](#)

The `_ceil()` function rounds a numeric value up to the nearest integer.

Synopsis

```
_ceil( <expression> )
```

Description

The `_ceil()` function rounds `<expression>` up to the nearest integer.

Arguments

Argument	Description
<code><expression></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value.

Return Value

If the data type of *<expression>* is float, the `_ceil()` function returns a float value that rounds *<expression>* up to the nearest integral value. If the data type is `int32` or `int64`, the `_ceil()` function returns *<expression>* without altering its value.

`_floor()`

The `_floor()` function rounds a numeric value down to the nearest integer.

Synopsis

```
_floor( <expression> )
```

Description

The `_floor()` function rounds *<expression>* down to the nearest integer.

Arguments

Argument	Description
<i><expression></i>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or float value.

Return Value

If the data type of *<expression>* is a float value, The `_floor()` function returns a float value that rounds that *<expression>* down to the nearest integral value. If the data type is `int32` or `int64`, the `floor()` function returns *<expression>* without altering its value.

Example

The following query returns the integral value of the average size of the requests in the log table:

```
SELECT _floor( avg(RespSize) )
FROM example_webserv_100
DURING ALL;
```

`_round()`

The `_round()` function performs rounding to a specified decimal or whole-number precision.

Synopsis

```
_round( <expression> [,<digits>] )
```

Description

The `_round()` function rounds *<expression>* to the decimal or whole-number precision specified by *<digits>*. The function rounds to the nearest integer value when *<digits>* is omitted.

Arguments

Argument	Description
<code><expression></code>	An expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value.
<code><digits></code>	Optional. An <code>int32</code> value in the range -16 through 16. Positive numbers specify decimal precision. Negative numbers specify whole-number precision. The value 0 specifies that rounding does not occur. If this parameters is omitted, rounding occurs to the nearest integer.

Return Value

If the data type of `<expression>` is a `float` value, the `_round()` function returns a `float` the closest to the $10^{(-<digits>)}$ value. If the data type is `int32` or `int64`, the function returns its argument in the same data type respectively.

Exceptions

The `_round()` function raises an SQL processing exception if the data type of `<expression>` is not `int32`, `int64`, or `float`.

STRING FUNCTIONS

This section describes functions that operate on string values.

Function	Purpose	Page
<code>_strlowercase()</code> , <code>_lc()</code>	Converts text to all lower-case letters	page 130
<code>_strupercase()</code> , <code>_uc()</code>	Converts text to all upper-case letters	page 131
<code>_strmd5()</code> , <code>_md5()</code>	Computes the MD5 hash value for a large block of text as a <code>varchar</code>	page 131
<code>_strmd5_64()</code> , <code>_md5_64()</code>	Computes the MD5 hash value for a large block of text as an <code>int64</code>	page 132
<code>_strlen()</code>	Counts the number of characters in a <code>varchar</code> value	page 132
<code>_strstr()</code>	Determines if a text value can be found as part of another text value	page 133
<code>_strmatch()</code>	Returns a portion of text that matches a regular expression	page 132
<code>_strmatchlist()</code>	Parses portions of text that match a regular expression into a list variable	page 135
<code>_strsplit()</code>	Parses text into a list of text values using a single separation character	page 136
<code>_strsplitxsv()</code>	Parses text into a list of text values using complex separation logic	page 136
<code>_strleft()</code>	Returns a specified number of leading characters from text	page 137
<code>_strright()</code>	Returns a specified number of trailing characters from text	page 137
<code>_strmiddle()</code> , <code>substr()</code>	Returns all text from a starting character for a specified length	page 138
<code>_strrepeat()</code>	Returns a <code>varchar</code> expression repeated a specified number of times	page 138

Function	Purpose	Page
_strlpad()	Prepends a space or the value of an optional varchar expression to a varchar expression up to the length specified in an int32 expression.	page 140
_strrpadd()	Appends a space or the value of an optional varchar expression to the varchar expression up to the length specified in an int32 expression.	page 142
_strtrim()	Returns all but a specified number of leading and trailing characters from text	page 144
_strlink()	Creates an HTML anchor tag	page 145
_strcat()	Concatenates a list of text arguments	page 147
_strjoin()	Concatenates a list of text arguments with a separator between each value	page 147
_strformat() , _sprintf()	Formats a string from a list of substitution values	page 148

[_strlowercase\(\), _lc\(\)](#)

The `_strlowercase()` function converts text to all lower-case letters.

Synopsis

```
_strlowercase( <string> )
```

```
_lc( <string> )
```

Description

The `_strlowercase()` function converts each letter in `<string>` to lower case, if possible.

The `_lc()` function is an alias for `strlowercase()`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value containing the string to convert.

Return Value

The `_strlowercase()` function returns a `varchar` value with the result of the conversion.

Examples

The following query returns all the values of the URL column of the `example_webserv_100` table in lower case.

```
SELECT \_strlowercase( Url )
FROM example_webserv_100
DURING ALL;
```

_strupercase(), _uc()

The `_strupercase()` function converts text to all upper-case letters.

Synopsis

```
_strupercase( <string> )
```

```
_uc( <string> )
```

Description

The `_strupercase()` function converts each letter in `<string>` to upper case, if possible.

The `_uc()` function is an alias for `strupercase()`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value containing the string to convert.

Return Value

The `_strupercase()` function returns a `varchar` value that contains the result of the conversion.

_strmd5(), _md5()

The `_strmd5()` function computes the MD5 hash value for a large block of text.

NOTE: These functions are not recommended for cryptographic use.

Synopsis

```
_strmd5( <string> )
```

```
_md5( <string> )
```

Description

The `_strmd5()` function computes the MD5 hash value for a large block of text and returns the hash value as a `varchar` value. MD5 hash values are computed by an algorithm that produces a 128-bit message digest, or “signature” for a text block of arbitrary size. Generally, one assumes that MD5 hash values are unique unless text blocks are identical.

The `_md5_64()` function is an alias for `_strmd5_64()`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value containing the text block to hash.

Return Value

The `_strmd5()` function returns a `varchar` value that contains the hash value.

Exceptions

The `_strmd5()` function raises an SQL processing exception if the data type of `<string>` is not `varchar`.

`_strmd5_64()`, `_md5_64()`

The `_strmd5_64()` function computes the MD5 hash value for a large block of text.

NOTE: These functions are not recommended for cryptographic use.

Synopsis

```
_strmd5_64( <string> )
```

```
_md5_64( <string> )
```

Description

The `_strmd5_64()` function computes the MD5 hash value for a large block of text and returns the hash value as an `int64` value. MD5 hash values are computed by an algorithm that produces a 128-bit message digest, or “signature” for a text block of arbitrary size. Generally, one assumes that MD5 hash values are unique unless text blocks are identical.

The `_md5_64()` function is an alias for `_strmd5_64()`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value containing the text block to hash.

Return Value

The `_strmd5_64()` function returns an `int64` value that contains the first 8 bytes of the 128-bit hash value.

Example

The following query returns the MD5 hash value of the URL column for each record in the `example_webserv_100` table.

```
SELECT _md5( Url )
FROM example_webserv_100
DURING ALL;
```

`_strlen()`

The `_strlen()` function counts the number of characters in a `varchar` value.

Synopsis

```
_strlen( <string> )
```

Description

The `_strlen()` function counts the number of characters in `<string>`.

Arguments

Argument	Description
<code><string></code>	A varchar value.

Return Value

The `_strlen()` function counts the number of characters in `<string>` up to the first 0x0 character.

Examples

The following query returns the length of the largest URL in the `example_webserv_100` table:

```
SELECT max( _strlen(Url) )
FROM example_webserv_100
DURING ALL;
```

_strstr()

Synopsis

```
_strstr( <string>, <string_to_find> )
```

Description

The `_strstr()` function searches `<string>` for the first occurrence of `<string_to_find>`.

TIP: To enhance performance, Hexis recommends that instead of the LIKE operator, you use the `_strstr()` function as in: `_strstr(<<VAL>%COL>,<VAL>)` instead of `<COL> like %<VAL>%`.

Arguments

Argument	Description
<code><string></code>	A varchar value that contains the string to search
<code><string_to_find></code>	A varchar value that contains the string to find

Return Value

The data type of the return value from the `_strstr()` function is `int32`. The return value contains the 0-based offset of the beginning `<string_to_find>` within `<string>`. The return value is -1 if `<string_to_find>` is not found within `<string>`.

Examples

The following query returns the number of records in the table where the `UserAgent` column contains the string 'Windows'.

```
SELECT count(*)
FROM example_webserv_100
WHERE _strstr(UserAgent, 'Windows') <> -1
DURING ALL;
```

The following query returns the number of records in the table where the `Url` column begins with the sequence '/images':

```
SELECT count(*)
FROM example_webserv_100
WHERE _strstr(Url, '/images') = 0
DURING ALL;
```

`_strmatch()`

Synopsis

```
_strmatch( <string>, <pattern>, <default_value> )
```

Description

The `_strmatch()` function searches `<string>` for a sequence of characters that matches `<pattern>`, which you write as a Perl regular expression. The function returns the matching sequence of characters if a match is found; otherwise, the function returns `<default_value>`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value that contains the string to search
<code><pattern></code>	A <code>varchar</code> value that contains the matching pattern, expressed as a Perl regular expression NOTE: When a pattern contains a backslash (<code>\</code>) character, you must escape it with a second backslash character; for example: <code>\\d</code> .
<code><default_value></code>	A <code>varchar</code> value that contains the string to return if the match fails

Return Value

The `_strmatch()` function returns a `varchar` that corresponds to the first parenthesized elements that matched the string. The function returns `<default_value>` if no match is found.

Exceptions

The `_strmatch()` function raises an SQL processing exception when the value of `<pattern>` is not a valid regular expression.

Examples

The following query returns a portion of the `UserAgent` column of each row in the `example_webserve_100` table with the string 'MSIE'. If the `UserAgent` column does not contain the value, the default value of 'MSIE 5.0' is used.

```
SELECT _strmatch(UserAgent, '(MSIE [^ ]*)', 'MSIE 5.0')
FROM example_webserve_100
DURING ALL;
```

The following query uses the `_strmatch()` function in Boolean expression to return rows that contain 'eclipse' anywhere within the `referrer` column:

```
SELECT *
FROM mytable
WHERE _strmatch(referrer, '*.eclipse.*', '-') <> '-'
DURING ALL
;
```

_strmatchlist()

Synopsis

```
_strmatchlist( <string>, <pattern>[, <fallback>] )
```

Description

The `_strmatchlist()` function attempts to match the specified `<string>` against the specified `<pattern>` and makes the matching portions of the string available as an INTO variable.

For more information, see [“INTO Keyword”, on page 91](#).

Arguments

Argument	Description
<code><string></code>	A varchar value that contains the string to match
<code><pattern></code>	A varchar value that contains the matching pattern, expresses as a Perl regular expression
<code><fallback></code>	Optional. A varchar value that contains a string to match against <code><pattern></code> if <code><string></code> fails to match.

Return Values

The return value from the `_strmatchlist()` function is an `int32` that corresponds to the number of parenthesized elements in `<pattern>` that find matches in `<string>`. If there are no parentheses used in `<pattern>`, `_strmatchlist()` returns 1 if there is a match and 0 if the pattern does not match.

Also, the `_strmatchlist()` makes available the actual matches as list elements. You can return the list as a list variable in place of the return value, or you can return it as an INTO variable. If there are no parenthesized elements used in `<pattern>`, the entire matched substring is returned as the first list element.

For more information on accessing return lists in place of return values, see [“Multiple Values as Lists”, on page 89](#).

Example

Here is an example of using INTO to capture multiple matches from `_strmatchlist()`:

```
SELECT match[1], match[2]
FROM example_webserv_100
WHERE _strmatchlist(UserAgent, '([^(]*)[ (][^(]+)') INTO match
DURING time('Feb 01 00:00:00 2002'), time('Mar 31 23:59:59 2002');
```

`_strsplit()`

Synopsis

```
_strsplit( <separator_string>, <expression>[, <throw>] )
```

Description

The `_strsplit()` function parses `<expression>` into a list of strings, with the separator between the elements specified by `<separator_string>`.

The return value of `_strsplit()` is the number of strings parsed from `<expression>`. To access the list, use INTO or @. For more information about lists, see [“List Functions”, on page 97](#).

Arguments

Argument	Description
<code><separator_string></code>	A varchar value that specifies the parsing separator.
<code><expression></code>	A varchar value that contains the string to parse
<code><throw></code>	Optional. A value of any data type that will cause an exception to be thrown with an explanation when <code><splitstring></code> is rejected

Return Values

The `_strsplit()` function returns the number of strings parsed from `<expression>`.

Also, the `_strsplit()` function makes available the actual matches as list elements. You can return the list as a list variable in place of the return value, or you can return the list as an INTO variable.

For more information on accessing return lists in place of return values, see [“Multiple Values as Lists”, on page 89](#).

`_strsplitxsv()`

Synopsis

```
_strsplitxsv( <parsing_options>, <expression>[, <throw>])
```

Description

The `_strsplitxsv()` function is similar to `_strsplit()`, but instead of using a literal string to specify the separator, it allows you to specify more complex parsing options, such as quoting and escapification. The `<parsing_options>` argument contains a semicolon-delimited list of options.

For details about the permitted values in `<parsing_options>`, see [“Basic Parsing Options”](#), on page 101.

Return Values

The `_strsplitxsv()` function returns the number of strings parsed from `<expression>`.

Also, the `_strsplitxsv()` function makes available the parsed strings as list elements. You can return the list as a list variable in place of the return value, or you can return the list as an `INTO` variable.

For more information on accessing return lists in place of return values, see [“Multiple Values as Lists”](#), on page 89.

`_strleft()`

Synopsis

```
_strleft( <string>, <count> )
```

Description

The `_strleft()` function returns the leading `<count>` of characters from `<string>`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value that contains the string to return
<code><count></code>	A non-negative <code>int32</code> value that specifies the number of characters to return, starting from the left of <code><string></code>

Return Value

The `_strleft()` function returns a `varchar` value with the result.

`_strright()`

Synopsis

```
_strright( <string>, <count> )
```

Description

The `_strright()` function returns the trailing `<count>` of characters from `<string>`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value that contains the string to return
<code><count></code>	A non-negative <code>int32</code> value that specifies the number of characters to return, starting from the right of <code><string></code>

Return Value

The `_strright()` function returns a `varchar` value with the result.

`_strmiddle()`, `substr()`

Synopsis

```
_strmiddle( <string>, <offset>, <length> )
```

```
_substr( <string>, <offset>, <length> )
```

Description

The `_strmiddle()` function extracts a substring that starts at `<offset>` in `<string>` and extends the specified `<length>`.

The `_substr()` function is an alias for `_strmiddle()`.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value that contains the string to return
<code><offset></code>	A non-negative <code>int32</code> value that specifies the position of the first character to return.
<code><length></code>	A non-negative <code>int32</code> value that specifies the number of characters to return.

Return Value

The `_strmiddle()` function returns a `varchar` value with the result.

`_strrepeat()`

Synopsis

```
_strrepeat( <varchar_expression>, <int32_expression> )
```

Description

The `_strrepeat()` function repeats the `varchar` expression as many times as the value specified in the integer expression.

Arguments

Argument	Description
<code><varchar_expression></code>	An expression that evaluates to a <code>varchar</code> value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>varchar</code> .
<code><int32_expression></code>	An expression that evaluates to a non-negative <code>int32</code> value that specifies the number of times to repeat the value in the first argument; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>int32</code> . NOTE: If the integer is negative, the EDW treats the negative value as zero (0) and returns an empty <code>varchar</code> value.

Return Value

The `_strrepeat()` function returns a `varchar` value that repeats the `varchar-expression` value as many times as specified by the `int32` value.

Exceptions

The `_strrepeat()` function raises a SQL processing exception when the data type of `<varchar_expression>` is not `varchar` and the data type of `<int32_expression>` is not `int32`.

Examples

The following query repeats "Page" four times in each row:

```
SELECT _strrepeat( 'Page', 4 )
FROM someTable
DURING ALL;
```

The query above returns: PagePagePagePage.

The following query repeats the value of the `OddChar` column as many times as specified by the value of the `Digit` column:

```
SELECT top 1 OddChar, Digit, _strrepeat( OddChar, Digit )
FROM analyzer_types
DURING ALL;
```

The graphic below illustrates the results.

```
+-----+
| Results for SQL file >(standard input)< |
+-----+-----+-----+-----+
| OddChar | Digit | |                               |
| (varchar) | (int32) | |                               |
+-----+-----+-----+-----+
| comma, comma | 7 | comma, commacomma, commacomma, commacomma, commacomma, commacomma, comma |
```

The following query appends a trailing space to the value of the `OddChar` column, which it repeats as many times as specified by the value of the `Digit` column plus 1:

```
SELECT top 1 _strrepeat( OddChar+" ", Digit+ _int32(1))
FROM analyzer_types
DURING ALL;
```

The graphic below illustrates the results.

```

+-----+
| Results for SQL file >(standard input)< |
+-----+
|                                     |
|                                     |
|                                     |
|                                     |
+-----+
| comma,comma comma,comma comma,comma comma,comma comma,comma comma,comma comma,comma |
+-----+

```

To display the full results of the `_strrepeat()` function, the query above includes only the `_strrepeat()` function in the `SELECT` clause. Therefore, it does not return values from the `OddChar` and `Digit` columns.

`_strlpad()`

Synopsis

```
_strlpad( <varchar_expression>, <int32_expression>  
[,<optional varchar expression>])
```

Description

The `_strlpad()` function prepends the value of the optional varchar expression to the varchar expression until the result value becomes the length specified in the int32 expression. If the length of the result varchar expression is longer than the specified length, the function truncates the text from the right. If no optional varchar expression is included, the function prepends a space.

Arguments

Argument	Description
<code><varchar_expression></code>	An expression that evaluates to a <code>varchar</code> value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>varchar</code> .
<code><int32_expression></code>	An expression that evaluates to a non-negative <code>int32</code> value that specifies the length of the returned value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>int32</code> . NOTE: If the integer is negative, the EDW treats the negative value as zero (0) and returns an empty <code>varchar</code> value.
<code><optional_varchar_expression></code>	An expression that evaluates to a <code>varchar</code> value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>varchar</code> .

Return Value

The `_strlpad()` function returns a `varchar` value that includes the value of the `varchar` expression prepended with the value of the optional `varchar` expression, or a space if the optional expression is missing, up to the length specified in the `int32` expression. If the length of this result `varchar` expression is longer than the specified length, the function truncates the text from the right.

Exceptions

The `_strlpad()` function raises a SQL processing exception when the data type of `<varchar_expression>` or `<optional_varchar_expression>` is not `varchar` and the data type of `<int32_expression>` is not `int32`.

Examples

The following query returns "xyxhi":

```
SELECT _strlpad( 'hi', 5, 'xy')
FROM someTable
DURING ALL;
```

The following query prepends the value of the optional expression to the value in the `SrcIP` column until the result value becomes the length specified in the `int32` expression:

```
SELECT top 5 SrcIP, _strlpad(SrcIP,16,"src: ")
FROM analyzer_types
DURING ALL;
```

The graphic below illustrates the results.

```
+-----+
| Results for SQL file >(standard input)< |
+-----+-----+
| SrcIP | strlpad |
| (varchar) | (varchar) |
+-----+-----+
| 192.168.1.7 | src: 192.168.1.7 |
| 192.168.1.1 | src: 192.168.1.1 |
| 192.168.1.8 | src: 192.168.1.8 |
| 192.168.1.13 | src:192.168.1.13 |
| 192.168.1.6 | src: 192.168.1.6 |
+-----+-----+
```

NOTE: In the results above, the fourth row does not include the space before the IP address because this address is longer than the others. If you modify the function to return 18 characters instead of 16, four of the five rows returned repeat the first two characters of the optional `varchar` expression whereas the row with the longer value repeats only the first character, as shown below.

```

+-----+
| Results for SQL file >(standard input)< |
+-----+
| SrcIP | strlpad |
| (varchar) | (varchar) |
+-----+
| 192.168.1.13 | src: s192.168.1.13 |
| 192.168.1.6 | src: sr192.168.1.6 |
| 192.168.1.7 | src: sr192.168.1.7 |
| 192.168.1.1 | src: sr192.168.1.1 |
| 192.168.1.8 | src: sr192.168.1.8 |
+-----+

```

If you modify the function to return only 3 characters, which is shorter than the values in the `SrcIP` column, no additional characters are prepended to the result. Instead, the query truncates the column values from the right to 3 characters, as shown below.

```

+-----+
| Results for SQL file >(standard input)< |
+-----+
| SrcIP | strlpad |
| (varchar) | (varchar) |
+-----+
| 192.168.1.13 | 192 |
| 192.168.1.6 | 192 |
| 192.168.1.7 | 192 |
| 192.168.1.1 | 192 |
| 192.168.1.8 | 192 |
+-----+

```

_strrpad()

Synopsis

```

_strrpad( <varchar_expression>, <int32_expression>
[, <optional_varchar_expression>] )

```

Description

The `_strrpad()` function appends the value of the optional varchar expression to the varchar expression until the result value becomes the length specified in the int32 expression. If the length of the result varchar expression is longer than the specified length, the function truncates the text from the right. If there is no optional varchar expression, the function appends a space.

Arguments

Argument	Description
<code><varchar_expression></code>	An expression that evaluates to a varchar value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to varchar.

Argument	Description
<code><int32_expression></code>	An expression that evaluates to a non-negative <code>int32</code> value that specifies the length of the returned value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>int32</code> . NOTE: If the integer is negative, the EDW treats the negative value as zero (0) and returns an empty <code>varchar</code> value.
<code><optional_varchar_expression></code>	An expression that evaluates to a <code>varchar</code> value; can be a column value, any general expression that includes column values, and/or constants, provided that the data type of the expression evaluates to <code>varchar</code> .

Return Value

The `_strrpadd()` function returns a `varchar` value that includes the value of the `varchar` expression appended with the value of the optional `varchar` expression, or a space if the optional expression is missing, up to the length specified in the `int32` expression. If the length of this result `varchar` expression is longer than the specified length, the function truncates the text.

Exceptions

The `_strrpadd()` function raises a SQL processing exception when the data type of `<varchar_expression>` or `<optional_varchar_expression>` is not `varchar` and the data type of `<int32_expression>` is not `int32`.

Examples

The following query returns "hixyx":

```
SELECT _strrpadd( 'hi', 5, 'xy')
FROM someTable
DURING ALL;
```

The following query appends the value of the optional expression to the value in the `SrcIP` column until the result value becomes the length specified in the `int32` expression:

```
SELECT top 5 SrcIP, _strrpadd(SrcIP,18,"src: ")
FROM analyzer_types
DURING ALL;
```

The graphic below illustrates the results.

```
+-----+
| Results for SQL file >(standard input)< |
+-----+
| SrcIP | strrpadd |
| (varchar) | (varchar) |
+-----+
| 192.168.1.13 | 192.168.1.13src: s |
| 192.168.1.6 | 192.168.1.6src: sr |
| 192.168.1.7 | 192.168.1.7src: sr |
| 192.168.1.1 | 192.168.1.1src: sr |
| 192.168.1.8 | 192.168.1.8src: sr |
+-----+
```

NOTE: In the results above, the first row appends one fewer character from the optional expression because this IP address is longer than the others. If you modify the query so that it does not include an optional expression, the results append spaces after the IP address, as shown below.

```
+-----+
| Results for SQL file >(standard input)< |
+-----+-----+
| SrcIP | strrpap |
| (varchar) | (varchar) |
+-----+-----+
| 192.168.1.13 | 192.168.1.13 |
| 192.168.1.6 | 192.168.1.6 |
| 192.168.1.7 | 192.168.1.7 |
| 192.168.1.1 | 192.168.1.1 |
| 192.168.1.8 | 192.168.1.8 |
+-----+-----+
```

If you modify the function to return only 3 characters, which is shorter than the values in the `SrcIP` column, no additional characters are appended to the result. Instead, the query truncates the column values from the right to 3 characters, as shown below.

```
+-----+
| Results for SQL file >(standard input)< |
+-----+-----+
| SrcIP | strrpap |
| (varchar) | (varchar) |
+-----+-----+
| 192.168.1.13 | 192 |
| 192.168.1.6 | 192 |
| 192.168.1.7 | 192 |
| 192.168.1.1 | 192 |
| 192.168.1.8 | 192 |
+-----+-----+
```

_strtrim()

Synopsis

```
_strtrim( <string>, <left>[, <right>] )
```

Description

The `_strtrim()` function creates a substring by truncating the `<left>` number of characters from the left side of `<string>` and the `<right>` number of characters from the right side.

Arguments

Argument	Description
<code><string></code>	A <code>varchar</code> value that contains the string to return

Argument	Description
<code><left></code>	<p>A non-negative <code>int32</code> value that specifies the number of characters on the left of <code><string></code> to remove.</p> <p>– or –</p> <p>A <code>varchar</code> value that identifies the leading characters to remove. The function scans the left portion of <code><string></code> until it finds a character not in <code><left></code> and then removes the characters up to that point</p>
<code><right></code>	<p>Optional. A non-negative <code>int32</code> value that specifies the number of characters on the right of <code><string></code> to remove.</p> <p>– or –</p> <p>Optional. A <code>varchar</code> value that identifies the trailing characters to remove. The function scans backwards through the right portion of <code><string></code> until it finds a character not in <code><right></code> and then removes the characters up to that point.</p>

Return Value

The `_strtrim()` function returns a `varchar` value containing the result.

Examples

The following query returns all the values of the `Referrer` column of the table with any slashes removed from the left side:

```
SELECT _strtrim( Referrer, '/' )
FROM example_webserv_100
DURING ALL;
```

The following query returns all the values of the `Referrer` column of the table with the trailing character removed:

```
SELECT _strtrim( Referrer, 0, 1 )
FROM example_webserv_100
DURING ALL;
```

The following query returns all the values of the `Url` column of the table with the leading 5 characters removed, if they begin with 'http:':

```
SELECT _strtrim( Url, _if( _strstr(Url, 'http:')=0, 5, 0 ) )
FROM example_webserv_100
DURING ALL;
```

_strlink()

Synopsis

```
_strlink( <url>, <text>[, <new_window>] )
```

Description

The `_strlink()` function returns an HTML anchor tag suitable for use in HTML documents. The returned tag contains `<text>` as the link text to display, with `<url>` as the HREF attribute.

This function is useful for query results you intend to display in your own HTML or other third-party application.

NOTE: Do not use this function to display an active link in a SenSage AP Console report. Although SenSage AP Console provides the special URL data type from a dropdown in the Columns tab, the URL data type does not display the text as clickable links; instead it sorts the text as URLs.

Arguments

Argument	Description
<code><url></code>	A <code>varchar</code> value that contains the URL referenced by the <code>href</code> attribute of the link
<code><text></code>	A <code>varchar</code> value that contains the text content of the link
<code><new_window></code>	Optional. Value of any data type, which if equal to 1, indicates that the link should display the target URL in a new browser window.

Return Value

The `_strlink()` function returns a `varchar` value containing a valid HTML anchor tag.

Examples

Assume an event-log table contains a `varchar url` column, with values similar to the following:

```
Url
-----
http://www.acme.com
```

The following query returns a list of HTML anchor tags for the URLs contained in the `Url` column of the `example_webserv_100` table:

```
SELECT _strlink( Url, 'Click Here' ) AS anchor_tag
FROM example_webserv_100
DURING ALL;
```

The output looks similar to the following:

```
anchor_tag
-----
<a href="http://www.acme.com">Click Here</a>
```

The following query returns a list of HTML anchor tags encoded so that the link destination is displayed in a new browser window instead of replacing the page in the current browser window:

```
SELECT _strlink( Url, 'Click Here', 1) AS anchor_tag
FROM example_webserv_100
DURING ALL;
```

The output looks similar to the following:

```
anchor_tag
-----
<a href="http://www.acme.com" target=new>Click Here</a>
```

_strcat()

Synopsis

```
_strcat ( <argument>[, <argument>[...]] )
```

Description

The `_strcat()` function concatenates the string representations of the `<argument>` values.

Arguments

Argument	Description
<code><argument></code>	A value of any data type. The function converts the value to a <code>varchar</code> before concatenating.

NOTE: The `_strcat()` function may be passed list variables as arguments. For more information, see [“Working with Lists”, on page 89](#).

Return Value

The `_strcat()` function returns a `varchar` value containing the result.

Example

For rows with `Url="/www.acme.com"` and `ClientDNS = "66.127.84.10"`, the following query will return the value `"http://www.acme.com?dns=66.127.84.10"`.

```
SELECT _strcat('http:', Url, '?dns=', ClientDNS)
FROM example_webserv_100
DURING ALL;
```

_strjoin()

Synopsis

```
_strjoin( <separator_string>, <argument>[, <argument>[...]] )
```

Description

The `_strjoin()` function concatenates the string representations of the `<argument>` values, but separates each string with `<separator_string>`. This function is similar to the Perl function of the same name.

Arguments

Argument	Description
<code><separator_string></code>	A <code>varchar</code> value that contains the characters that separate the concatenated string values. The value must be constant; that is, it cannot vary from row to row.

Argument	Description
<code><argument></code>	A value of any data type. The function converts the value to a <code>varchar</code> before concatenating.

NOTE: The `_strjoin()` function may be passed list variables as arguments. For more information, see [“Working with Lists”, on page 89](#).

Return Value

The `_strjoin()` function returns a `varchar` value containing the result.

Example

The following query will return the value “a,b,c,d”

```
SELECT TOP 1 _strjoin(",", "a", "b", "c", "d")
FROM example_webserv_100
DURING ALL;
```

`_strformat()`, `_sprintf()`

Synopsis

```
_strformat( <format_string>, <argument>[, <argument>[...]] )
_sprintf( <format_string>, <argument>[, <argument>[...]] )
```

Description

The `_strformat()` function creates a string by replacing the *format specifiers* in `<format_string>` with the string representations of the corresponding arguments. A format specifier is a special sequence of characters that is replaced by the `varchar` value of a corresponding `<argument>`.

For more information, see [“Format Specifiers”, on page 148](#).

The `_sprintf()` function is an alias for `_strformat()`.

Arguments

Argument	Description
<code><format_string></code>	A <code>varchar</code> value that specifies the format of the output.
<code><argument></code>	A value of any data type to use in place of the corresponding format specifier in <code><format_string></code> . The data type must match the data type its corresponding format specifier.

Format Specifiers

Use format specifiers in the `<format_string>` argument as placeholders for values that you provide as additional arguments. Format specifiers have the following syntax:

```
%[<flags>][<width>][<precision>]<format>
```

The `_strformat()` function replaces each format specifier with a formatted value, leaving the other text unchanged. Use format specifiers that are appropriate for the data type of the corresponding arguments.

FORMATS TO SPECIFY

These are the types of formats you can specify.

%<format>	Allowed Data Types	Meaning
%d	int32, int64, or timestamp	Format the value as a decimal (base 10) number
%x	int32, int64, or timestamp	Format the value as a hexadecimal (base 16) number
%o	int32, int64, or timestamp	Format the value as an octal (base 8) number
%f	float or timestamp	Format the value as a floating-point number
%e %E	float or timestamp	Format the value as a floating-point number with exponential notation
%g	float or timestamp	Format the value as though %f were used, unless the exponent is less than -4 or larger than the specified precision, in which case format the value as though %e were used.
%s	varchar or timestamp	Format the value appropriately
%c	bool, int32, or int64	Format the value as a single character
%%		Replace with a single % character

FLAGS IN FORMAT SPECIFIERS

<flag>	Meaning
-	Minus sign – Left-justify the field
=	Equal sign – Center-justify the field
0	Zero – Pad numeric values with leading zeros
+	Plus sign – Prefix positive numeric values with '+'

FIELD WIDTHS IN FORMAT SPECIFIERS

<width>	Meaning
<n>	A positive, base-10 integer that specifies a field width
*	Asterisk – obtain the width from the corresponding argument

PRECISION OR LENGTH IN FORMAT SPECIFIERS

<precision>	Meaning
.<n>	A period followed by a base-10 integer that specifies the precision for a numeric argument or the maximum number of characters for a text argument

<i><precision></i>	Meaning
. *	A period followed by an asterisk – obtain the precision or length from the corresponding argument

Return Value

The `_strformat()` function returns a `varchar` value containing the result.

Exceptions

The `_strformat()` function raises an SQL processing exception if the format specifier used for an argument is invalid.

Examples

For rows with a `RespSize` equal to 10240, the following query returns the value "size: 10240":

```
SELECT _strformat("size: %d", RespSize)
FROM example_webserve_100
DURING ALL;
```

For rows with a `RespSize` equal to 10240, the following query returns two columns containing the value "size: 10.0k". The first use of `_strformat()` specifies the field width and precision in the format string, and the second specifies them in the argument list.

```
SELECT _strformat("size:%6.1fk", _float(RespSize)/1024.0),
       _strformat("size:%*. *fk", 6, 1, _float(RespSize)/1024.0)
FROM example_webserve_100
DURING ALL;
```

The space after "size:" in the result occurs because the specified length is six, and "10.0k" is only five characters wide. The function expands the value to six characters by padding the left with a space.

TIME FUNCTIONS

This section describes functions that operate on `timestamps`.

Function	Purpose	Page
<code>_now()</code> , <code>now()</code>	Returns the current system time as a timestamp	page 151
<code>_time()</code> , <code>time()</code>	Creates timestamps in various ways	page 151
<code>_timeadd()</code>	Adds units of time to a timestamp	page 153
<code>_timediff()</code>	Computes the difference between two timestamps	page 155
<code>_timeformat()</code> , <code>_timef()</code>	Creates a formatted string from a timestamp	page 155
<code>_timeparse()</code> , <code>_strptime()</code> , <code>_strftime()</code>	Creates a timestamp from a formatted string	page 159
<code>_timestart()</code>	Rounds a timestamp down to specified precision	page 161

_now(), now()

The `_now()` function returns the current system time in GMT as a `timestamp`.

Synopsis

```
_now()
```

```
now()
```

Description

The `_now()` function returns the current system time.

The `now()` function is an alias for `_now()`.

Return Value

The `_now()` function returns a `timestamp` representing the current system time.

_time(), time()

The `_time()` function creates a timestamp from a variety of specifications.

Synopsis

```
_time( <time_specification>, <adjustment>[, <adjustment>[...]] )
```

```
time( <time_specification>, <adjustment>[, <adjustment>[...]] )
```

Description

The `_time()` function returns a timestamp based on the `<time_specification>` you provide. Generally you specify a character representation of the timestamp value you want. The function recognizes character-based timestamps in a fixed set of character formats. In addition, you can specify that you want the current time, or you can specify that you want the minimum or maximum timestamps that the EDW instance allows.

With `<adjustment>` arguments, you can adjust the `<time_specification>` to an earlier or later timestamp.

The `time()` function is an alias for `_time()`.

Arguments

Argument	Description
<code><time_specification></code>	<p>A <code>varchar</code> value that specifies the timestamp you want; for example:</p> <pre><code>_time('Oct 15 07:18:09 1997')</code></pre> <p>For more information, see “Time Specifications for the <code>_time()</code> Function”, next.</p>

Argument	Description
<code><adjustment></code>	<p>A <code>varchar</code> value that specifies an amount and unit of time by which to adjust the <code><time_specification></code>; for example:</p> <pre>_time('now', '-1wk')</pre> <p>The value is expressed as <code>'<n><unit>'</code>, where <code><n></code> is a positive or negative integer and <code><unit></code> is a longhand or shorthand notation for different units of time.</p> <p>For more information, see “Units of Time for the <code>_time()</code> and <code>_timeadd()</code> Functions”, on page 152.</p>

TIME SPECIFICATIONS FOR THE `_TIME()` FUNCTION

The `_time()` functions recognizes the following time specifications.

Time Specification	Pattern and/or Description	Example
Minimum allowed	The earliest timestamp allowed by the EDW instance	'min'
Maximum allowed	The latest timestamp allowed by the EDW instance	'max'
Current system time	The current timestamp	'now'
ISO 8601 format	YYYY-MM-DDTHH:MM:SS.NNNNNNZ The function always returns ISO 8601 timestamps in GMT, regardless of the WITH TIMEZONE directive.	'1997-11-15T07:18:09.000000Z'
Unix date format	MMM DD HH:MM:SS YYYY	'Oct 15 07:18:09 1997'
Unix date format with time zone	MMM DD HH:MM:SS ZZZ YYYY	'Oct 15 07:18:09 GMT 1997'
Unix date format with day of the week	WW MMM DD HH:MM:SS ZZZ YYYY	'Wed Oct 15 07:18:09 GMT 1997'
American century format with time	MM/DD/YYYY HH:MM:SS	'10/15/1997 17:18:09'
American decade format with time	MM/DD/YY HH:MM:SS	'10/15/97 17:18:09'
American century format	MM/DD/YYYY	'10/15/1997'
American decade format	MM/DD/YY	'10/15/97'

UNITS OF TIME FOR THE `_TIME()` AND `_TIMEADD()` FUNCTIONS

The `_time()` and `_timeadd()` functions recognize the following longhand and shorthand notations for specifying units of time. Trailing "s" characters are ignored.

Longhand Time Unit	Shorthand Time Unit	Usage Notes
microsecond	usec, microsec	Specifies one microsecond.

Longhand Time Unit	Shorthand Time Unit	Usage Notes
millisecond	msec, millisec	Specifies one thousand microseconds.
second	sec	Specifies one second worth of microseconds.
minute	min	Specifies one minute worth of microseconds.
hour	hr	Specifies one hour worth of microseconds.
day		Specifies 24 hours worth of microseconds.
week	wk	Specifies seven days worth of microseconds.
month	mon	Specifies 30 days-worth of microseconds.
year	yr	Specifies 365 days worth of microseconds.

Return Value

The `_time()` function returns a timestamp containing the value you specified.

Examples

The following query returns rows with timestamps between the time the query executes and one hour prior:

```
SELECT count(*)
  FROM example_webserv_100
  DURING _time( 'now', '-1hr' ), _now()
;
```

The following query returns rows with timestamps between the time the query executes and three months plus five days prior:

```
SELECT count(*)
  FROM example_webserv_100
  DURING _time( 'now', '-3months', '-5days' ), _now();
```

The following query returns rows with timestamps between the beginning of the current year and the end of the first quarter:

```
WITH $tz AS 'PDT'

SELECT count(*)
  FROM example_webserv_100
  DURING      // current year begin
    _time( 'Jan 1 00:00:00' +
      $tz +
      _timeparse( (_now()), '%C' )),
    // current year 1st quarter end
    _time( _timeparse( (_now()), '%C') + '-03-31T23:59:59:999999' +
      $tz)
;
```

_timeadd()

The `_timeadd()` function adds a length of time to a timestamp.

Synopsis

```
_timeadd( <timestamp>, <amount>, <time_unit>[, <time_zone>] )
```

Description

The `_timeadd()` function adds or subtracts a length of time to or from a `timestamp`.

The `_timeadd()` function accepts negative values for `<amount>`, which may yield timestamps that fall before the earliest timestamp that the EDW can store—generally, January 1, 1970.

Arguments

Argument	Description
<code><timestamp></code>	The timestamp to adjust by adding or subtracting a length of time.
<code><amount></code>	An <code>int32</code> that specifies the number of time units to add or subtract.
<code><time_unit></code>	A <code>varchar</code> value that indicates the kind of time units to add or subtract. The value is expressed in longhand or shorthand notation. For more information, see “Units of Time for the <code>_time()</code> and <code>_timeadd()</code> Functions” , on page 152.
<code><time_zone></code>	Optional. A <code>varchar</code> value that indicates the time zone in which to perform the calculation. Use this argument to override the default time zone. You control the default time zone with the <code>TIMEZONE</code> setting directive. For more information, see “Time-Zone Conversion” , on page 251. For a list of allowed time-zone values, see “Appendix B: Time Zones” in the <i>Administration Guide</i> .

Return Value

The `_timeadd()` function returns a `timestamp` adjusted with the length of time added or subtracted.

Examples

The following query returns rows with timestamps between the time the query executes and one hour prior:

```
SELECT count(*)
FROM example_webserv_100
DURING _timeadd( _now(), -1, 'hr' ), _now()
;
```

The following query returns rows with timestamps between the time the query executes and three months prior, in a specified time zone:

```
WITH $tz AS 'PDT'

SELECT count(*)
FROM example_webserv_100
DURING _time( _now(), '-3', 'mon', $tz ), _now();
```

_timediff()

The `_timediff()` function computes the difference between two timestamps.

Synopsis

```
_timediff( <timestamp_1>, <timestamp_2> )
```

Description

The `_timediff()` function computes the difference in microseconds between `<timestamp_1>` and `<timestamp_2>`.

Arguments

Argument	Description
<code><timestamp_1></code>	The base timestamp
<code><timestamp_2></code>	The timestamp for which the difference with <code><timestamp_1></code> is computed.

Return Value

The `_timediff()` function returns an `int64` value, which is the absolute value of the difference between the two timestamps in microseconds. The return value is always positive. For a signed result, use `"_INT64(ts1) - _INT64(ts2)"`.

Example

The following query searches the table for users who visited a page under `'/company'` two or more times, and returns the elapsed time between their first and last visit.

```
SELECT ClientDNS, _timediff( min(ts), max(ts) )
FROM example_webserve_100
WHERE _strstr(Url, '/company') = 0
GROUP BY 1
HAVING count(*) > 1
DURING ALL
```

```
+-----+
| Results for SQL file >example-time-03.sql< |
+-----+-----+
| ClientDNS | timediff |
| (varchar) | (int64)  |
+-----+-----*
output is post-sorted
+-----+-----*
| 216.239.46.200 | 363000000 |
+-----+-----*
```

_timeformat(), _timef()

The `_timeformat()` function creates a formatted string from a timestamp.

Synopsis

```
_timeformat( <time_format_string>, <timestamp>[, <time_zone>] )
```

```
_timef( <time_format_string>, <timestamp>[, <time_zone>] )
```

Description

The `_timeformat()` function creates a date-and-time-of-day `varchar` from a `timestamp` by replacing the formatting directives in `<time_format_string>` with the appropriate fields from `<timestamp>`.

The `_timef()` function is an alias for `_timeformat()`.

Arguments

Argument	Description
<code><time_format_string></code>	<p>A <code>varchar</code> value that specifies the format of the output date and time-of-day. For example, the following time format string specifies month-day-year format:</p> <pre><code>%m-%d-%y</code></pre> <p>It produces results similar to the following:</p> <pre><code>11-16-06</code></pre> <p>For more information, see “Format Strings and Formatting Directives for the <code>_timeformat()</code> and <code>_timeparse()</code> Functions”, on page 156.</p>
<code><timestamp></code>	A <code>timestamp</code> value to be formatted.
<code><time_zone></code>	Optional. A <code>varchar</code> value that indicates the time zone in which to perform the calculation. You control the default time zone with the <code>TIMEZONE</code> setting directive. For more information and a list of allowed values, see “Time-Zone Conversion” , on page 251.

FORMAT STRINGS AND FORMATTING DIRECTIVES FOR THE `_TIMEFORMAT()` AND `_TIMEPARSE()` FUNCTIONS

A `<time_format_string>` specifies the format you want for character-based timestamps. You can use text literals, such as hyphens (-) and colons (:), and you can use formatting directives, which begin with a percent sign (%). Enclose time format strings in quotation marks when you pass them as arguments to the `_timeformat()` and `_timeparse()` functions.

For example, the following time format string uses two text literals and three directives to specify character-based timestamps in month-day-year format:

```
'%m-%d-%y'
```

These are the formatting directives you can use in time format strings.

Formatting Directive	Meaning
<code>%a</code>	The abbreviated weekday name
<code>%A</code>	The full weekday name
<code>%b</code>	The abbreviated month name

Formatting Directive	Meaning
%B	The full month name
%c	The preferred date and time representation for the current locale
%C	The century number (year/100) as a 2-digit integer; single digits are preceded by a zero. See also %y and %Y.
%d	The day of the month as a decimal number (range 01 to 31)
%D	Equivalent to: %m/%d/%y NOTE: In countries other than the United States, %d/%m/%y is the standard date format. To avoid the ambiguity of these two date formats, use the ISO standard for date formats: %Y/%m/%d. In addition to avoiding ambiguity, the ISO format sorts in a reasonable way.
%e	Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space
%E	POSIX locale extensions. The sequences %Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy are supposed to provide alternate representations. Additionally %OB implemented to represent alternative months names (used standalone, without day mentioned).
%f	The preferred time representation for milliseconds and microseconds. Valid for the following functions: <ul style="list-style-type: none"> • <code>_strptime()</code> - Parse a <i>seconds</i> value with an optional decimal portion, in which the decimal point and the following digits are optional; the %f will consume all digits found after the decimal point if there is a decimal point present in the input string. Note that decimal digits after the sixth one are ignored in the conversion to the SenSage AP internal timestamp datatype with a resolution of 1us. • <code>_strftime()</code> - Output a <i>seconds</i> value with a 6-digit decimal portion in which six decimal digits are always output. If you do not want the decimal portion of the <i>seconds</i> value, use the %S tag.
%G	The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %v). This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.
%g	Like %G, but without century, that is, with a 2-digit year (00-99)
%h	Equivalent to %b
%H	The hour as a decimal number using a 24-hour clock (range 00 to 23)
%I	The hour as a decimal number using a 12-hour clock (range 01 to 12)
%j	The day of the year as a decimal number (range 001 to 366)
%k	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. See also %H
%l	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. See also %I
%m	The month as a decimal number (range 01 to 12)
%M	The minute as a decimal number (range 00 to 59)
%n	A newline character
%O	Same as %E
%p	Either 'AM' or 'PM' according to the given time value, or the corresponding strings for the current locale. Noon is treated as 'PM' and midnight as 'AM'.

Formatting Directive	Meaning
%P	Like %p but in lowercase: 'am' or 'pm' or a corresponding string for the current locale
%r	The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to '%I:%M:%S %p'.
%R	The time in 24-hour notation (%H:%M) For a version including the seconds, see %T below.
%s	The number of seconds since the epoch defined for the EDW instance.
%S	The second as a decimal number (range 00 to 61)
%t	A tab character
%T	The time in 24-hour notation (%H:%M:%S)
%u	The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w
%U	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also %V and %W.
%V	The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also %U and %W.
%w	The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.
%W	The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
%x	The preferred date representation for the current locale without the time
%X	The preferred time representation for the current locale without the date
%y	The year as a decimal number without a century (range 00 to 99)
%Y	The year as a decimal number including the century
%z	The time-zone as a numeric offset from GMT. Required to emit dates that conform with RFC822 (using '%a, %d %b %Y %H:%M:%S %z'). Valid for the following functions: <ul style="list-style-type: none"> • <code>_timeformat()</code> • <code>_timef()</code>
%Z	The time-zone as name or abbreviation. Valid for the following functions: <ul style="list-style-type: none"> • <code>_timeparse()</code> • <code>_strptime()</code> • <code>_strftime()</code> • <code>_timeformat()</code> • <code>_timef()</code>
%+	The date and time in C <date(1)> format
%%	A literal '%' character

Return Value

The `_timeformat()` function returns a `varchar` with a formatted date and time-of-day.

Exceptions

The `_timeformat()` function raises an SQL processing exception if `<time_format_string>` does not conform with the defined syntax.

Example

The following query returns the epoch of the EDW instance, formatted as a string. The epoch is January 1, 1970. It is always represented internally as 0 microseconds.

```
WITH $epoch AS timestamp '0usec'

SELECT _timeformat('%b %e %H:%M:%S %Y', $epoch)
```

```
+-----+
| Results for SQL file >example-time-04.sql< |
+-----+
|      timeformat      |
|      (varchar)      |
+-----+
output is post-sorted
+-----+
| Jan  1 00:00:00 1970 |
+-----+
```

The following query returns the URLs visited by '65.194.51.154', along with the hour and minute the URLs were visited:

```
SELECT ts, _timeformat('%H:%M', ts) AS time, url
FROM example_webserv_100
WHERE ClientDNS = '65.194.51.154'
ORDER BY 1
DURING ALL
```

```
+-----+
| Results for SQL file >example-time-07.sql< |
+-----+
|          ts          |    time    |    url    |
|      (timestamp)     | (varchar)  | (varchar)  |
+-----+
|2002-02-01T08:02:07.000000Z|08:02      |/summary.html|
|2002-02-01T08:17:05.000000Z|08:17      |/summary.html|
|2002-02-01T08:32:22.000000Z|08:32      |/summary.html|
|2002-02-01T09:17:12.000000Z|09:17      |/summary.html|
|2002-02-01T10:16:54.000000Z|10:16      |/summary.html|
|2002-02-01T11:01:51.000000Z|11:01      |/summary.html|
+-----+
```

_timeparse(), _strptime(), _strftime()

The `_timeparse()` function creates a timestamp from a date-and-time-of-day varchar.

Synopsis

```
_timeparse( <date_and_time>, <time_format_string>[, <time_zone>] )

_strptime( <date_and_time>, <time_format_string>[, <time_zone>] )

_strftime( <date_and_time>, <time_format_string>[, <time_zone>] )
```

Description

The `_timeparse()` function creates a timestamp value from a character-based timestamp. It uses the specification in `<time_format_string>` to interpret the date and time-of-day components in `<date_and_time>`.

The `_strptime()` and `_strftime()` functions are aliases for `_timeparse()`.

Arguments

Argument	Description
<code><date_and_time></code>	A <code>varchar</code> value with a date and time-of-day to be parsed and converted to a <code>timestamp</code> .
<code><time_format_string></code>	<p>A <code>varchar</code> value that specifies the format of the input date and time-of-day. For example, the following time format string specifies month day, year hour:minute:second format:</p> <pre>%b %d, %Y %H:%M:%S</pre> <p>It expects the value in <code><date_and_time></code> to be similar to the following:</p> <pre>Jul 8, 2009 19:01:36</pre> <p>For more information, see “Format Strings and Formatting Directives for the <code>_timeformat()</code> and <code>_timeparse()</code> Functions”, on page 156.</p>
<code><time_zone></code>	Optional. A <code>varchar</code> value that indicates the time zone in which to perform the calculation. You control the default time zone with the <code>TIMEZONE</code> setting directive. For more information and a list of allowed values, see “ Time-Zone Conversion ”, on page 251 .

Return Value

The `_timeparse()` function returns a `timestamp`.

Exceptions

The `_timeparse()` function raises an SQL processing exception if:

- `<time_format_string>` does not conform with the defined syntax
- input value and time-format string omit hour, minute, and second

Examples

The following query returns the specified timestamp in ISO 8601 format:

```
WITH $timestamp AS '11-16-06:00:00:00'

SELECT _timeparse($timestamp, '%m-%d-%y:%H:%M:%S')
```

```
+-----+
| Results for SQL file >example-time-05.sql< |
+-----+
|      timeparse      |
|      (timestamp)    |
```

```
+-----*
|2006-11-16T00:00:00.000000Z|
+-----*
```

_timestart()

The `_timestart()` rounds `timestamp` down to a specified unit of precision.

Synopsis

```
_timestart ( <timestamp>, <unit>[, <time_zone>]] )
```

Description

The `_timestart()` function rounds down `<timestamp>` to the `<unit>` of time specified.

Arguments

Argument	Description
<code><timestamp></code>	A <code>timestamp</code> value to be rounded down
<code><unit></code>	A <code>varchar</code> value that indicates the time unit to round to. The value is expressed in longhand (microsecond, millisecond, second, hour, minute, day, week, month, year) or in shorthand (usec, microsec, msec, millisec, sec, min, hr, wk, mon, yr) and must be enclosed within quotation marks. Sunday is the beginning of the week.
<code><time_zone></code>	Optional. A <code>varchar</code> value that indicates the time zone in which to perform the calculation. You control the default time zone with the <code>TIMEZONE</code> setting directive. For more information, see “Time-Zone Conversion” , on page 251. For a list of allowed values, see “Appendix B: Time Zones” in the <i>Administration Guide</i> .

Return Value

The `_timestart()` functions returns a `timestamp`.

Exceptions

The `_timestart()` function raises an SQL processing exception the value of `<unit>` does not conform to the specified syntax.

NETWORK ADDRESS FUNCTIONS

This section describes these functions that operate on network address values.

Function	Purpose	Page
_abbrev()	Abbreviated display format as text	page 162
_broadcast()	Broadcast address for network	page 163
_family()	Extracts family of address: 4 for IPv4, 6 for IPv6	page 164
_host()	Extracts IP address as text	page 165
_hostmask()	Constructs host mask for network	page 165
_masklen()	Extracts netmask length	page 166
_netmask()	Constructs netmask for network	page 167
_network()	Extracts the network part of the address	page 167
_set_masklen()	Sets netmask length for inet value	page 168
_mapto_ipv4() , _mapto_ipv6()	Maps an IPv6 address to an IPv4 address Maps an IPv4 address to an IPv6 address	page 169
_inet_contains()	Returns true if the first inet contains the second inet.	page 169
_inet_contains_or_equal()	Returns true if the first inet contains or equals the second inet.	page 170
_inet_plus() , _inet_minus()	Addition operator function Subtraction operator function	page 171
_inet_and() , _inet_not() , _inet_or() , _inet_xor()	Bitwise AND operator function Bitwise NOT operator function Bitwise OR operator function Bitwise XOR operator function	page 172

_abbrev()

The `_abbrev()` function produces an abbreviated display format of a network address as text.

Synopsis

```
_abbrev(<expression>)
```

Description

The `_abbrev()` function produces an abbreviated display of `<expression>` as text, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_abbrev()` function returns a `varchar` value, which is an RFC5952 string representation of an `inet`, including the `/subnet` suffix unless the address is a *host-only* address (which means subnet is 32 for IPv4 and 128 for IPv6). The value will be `"(invalid)"` if the column expression did not evaluate to a valid `inet` value.

Exceptions

The `_abbrev()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

The following query returns all values of the `addr1` column of the `inet_test` table appended to the string "Network Address: "

```
SELECT _strcat('Network Address:',_abbrev(addr1))FROM inet_test DURING ALL;
```

If the `addr1` column held an `inet` value `10.1.0.0/16` then the `_abbrev()` function returns the `varchar` value `"10.1.0.0/16"`.

_broadcast()

The `_broadcast()` function produces a broadcast address from a specified network address.

Synopsis

```
_broadcast(<expression>)
```

Description

The `_broadcast()` function produces a broadcast address from `<expression>` as an `inet` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_broadcast()` function returns an `inet` value.

Exceptions

The `_broadcast()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

If the column expression held an `inet` value `192.168.1.0/16`, then the `_broadcast()` function returns the `inet` value `0.0.255.255`.

`_family()`

The `_family()` function returns the family of the specified network address.

Synopsis

```
_family(<expression>)
```

Description

The `_family()` function returns the family of the specified network address: 4 for IPv4, 6 for IPv6 address, 0 for 'none' (when the empty string was converted to `inet`), 1 for 'invalid' (non-empty string that did not contain a valid IPv4/IPv6 address was converted to `inet`).

`<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_family()` function returns one of the following `int32` values:

Value	Description
0	the <code><expression></code> evaluated to an empty <code>inet</code> value.
1	invalid (the <code><expression></code> evaluated to a value that was neither a valid IPv4 nor IPv6 representation)
4	IPv4 address
6	IPv6 address

Exceptions

The `_family()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

If the column expression held an `inet` value `192.168.2.1`, then the `_family()` function returns the `int32` value `4`.

_host()

The `_host()` function extracts just the IP address as text from a specified network address.

Synopsis

```
_host(<expression>)
```

Description

The `_host()` function extracts an IP address from `<expression>` as a `varchar` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_host()` function returns a `varchar` value, containing the RFC5952 representation of the address information without the `"/subnet"` suffix.

Exceptions

The `_host()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

The following query returns all values of the `addr1` column of the `inet_test` table appended to the string "Host Address: "

```
SELECT _strcat('Host Address: ',_host(addr1)) FROM inet_test DURING ALL;
```

If the `addr1` column held an `inet` value `10.1.0.0/16` then the `_host()` function returns the `varchar` value `"10.1.0.0"`.

_hostmask()

The `_hostmask()` function produces a host mask for the specified network address.

Synopsis

```
_hostmask(<expression>)
```

Description

The `_hostmask()` function produces a host mask from `<expression>` as an `inet` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_hostmask()` function returns an `inet` value of the same family, which contains a bitmask, and when AND'ed against the original `inet`, will set all the subnet address bits to zero.

Exceptions

The `_hostmask()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

If the column expression held an `inet` value `192.168.1.0/16`, then the `_hostmask()` function returns the `inet` value `0.0.255.255`.

_masklen()

The `_masklen()` function extracts the network mask length for a specified network address.

Synopsis

```
_masklen(<expression>)
```

Description

The `_masklen()` function extracts the network mask length from `<expression>` as an `int32` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_masklen()` function returns an `int32` value.

Exceptions

The `_masklen()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

If the column expression held an `inet` value `192.168.1.0/16`, then the `_masklen()` function returns the `int32` value `16`.

_netmask()

The `_netmask()` function produces a network mask for a specified network address.

Synopsis

```
_netmask(<expression>)
```

Description

The `_netmask()` function produces a network mask from `<expression>` as an `inet` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_netmask()` function returns an `inet` value of the same family, which contains a bitmask which, when AND'ed against the original `inet`, will set all the host address bits to zero.

Exceptions

The `_netmask()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

If the column expression held an `inet` value `192.168.1.0/16`, then the `_netmask()` function returns the `inet` value `255.255.0.0`.

_network()

The `_network()` function extracts the network part of the address.

Synopsis

```
_network(<inet>)
```

Description

The `_network()` function extracts the network part of the address from the supplied `inet` value.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_network()` function returns an `inet` value..

Exceptions

The `_network()` function raises a SQL processing exception if the argument passed to the function is not evaluated to an `inet` value.

Examples

If the column expression held an `inet` value `192.168.1.0/16`, then the `_network()` function returns the `inet` value `255.255.0.0`.

`_set_masklen()`

The `_set_masklen()` function sets the network mask length for a specified network address.

Synopsis

```
_set_masklen(<expression>,<length>)
```

Description

The `_set_masklen()` function sets the network mask length for `<expression>` as an `inet` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4 or IPv6 network address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.
<code><length></code>	A column expression that evaluates to an <code>int32</code> value, representing the network mask length

Return Value

The `_set_masklen()` function returns an `inet` value of the same family, which contains the same address bits as the supplied `inet`, but whose subnet mask length is set to the specified value. Setting the mask length to `-1`, for example, sets the mask length equal to the size of the `inet`'s address data either to 32 (IPv4 or 128 IPv6) and is used to create a "host-only" address

Examples

If the column expression held an `inet` value `192.168.1.5/24`, then the `_set_masklen()` function returns the `inet` value **192.168.1.5/16** if the `<length>` column expression contained the `int32` value 16.

_mapto_ipv4(), _mapto_ipv6()

The `_mapto_ipv4()` and `_mapto_ipv6()` functions return a specified network address as an IPv4 or IPv6 address.

Synopsis

```
_mapto_ipv4(<expression>)
_mapto_ipv6(<expression>)
```

Description

The `_mapto_ipv4()` function produces an IPv4 address from `<expression>` as an `inet` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4-mapped IPv6 network address (see <http://tools.ietf.org/html/rfc4291#section-2.5.5.2>).

The subnet mask is computed by examining how far the IPv6 subnet extends into the last 32-bits of the IPv6 address.

NOTE: This function only works with the current `::ffff:a.b.c.d` style address, not the deprecated `::a.b.c.d` style.

The `_mapto_ipv6()` function produces an IPv4-mapped IPv6 address of the form `::ffff:a.b.c.d`, from `<expression>` as an `inet` value, where `<expression>` is a column expression that evaluates to an `inet` value representing a valid IPv4- network address.

The subnet mask is computed by extending the IPv4 subnet mask to cover the entire range of the extended IPv6 address.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.

Return Value

The `_mapto_ipv4()` and `_mapto_ipv6()` functions returns an `inet` value. It returns "invalid" if a IPv4-mapped-IPv6 (`_mapto_ipv4 ()`) address or an IPv4 (`_mapto_ipv4 ()`) address is not provided.

_inet_contains()

The `_inet_contains()` function returns true if the first `inet` *contains* the second `inet`.

Synopsis

```
_inet_contains(<expression1>,<expression2>)
```

Description

The first inet *contains* the second inet if and only if:

- the inets are of the same family, and
- the subnet mask of the first inet is shoreter than the second, and
- the common subnet bits of the two address are equal.

Arguments

Argument	Description
<code><expression1></code>	A column expression that evaluates to an inetvalue.
<code><expression2></code>	A column expression that evaluates to an inetvalue.

Return Value

The `_inet_contains()` function returns an true if the first inet *contains* the second inet.

`_inet_contains_or_equal()`

The `_inet_contains_or_equal()` function returns true if the first inet *contains or equals* the second inet.

Synopsis

```
_inet_contains_or_equal(<expression1>,<expression2>)
```

Description

The first inet *contains or equals* the second inet if and only if:

- the inets are of the same family, and
- the subnet mask of the first inet is shoreter than the second, and
- the common subnet bits of the two address are equal.

Arguments

Argument	Description
<code><expression1></code>	A column expression that evaluates to an inetvalue.
<code><expression2></code>	A column expression that evaluates to an inetvalue.

Return Value

The `_inet_contains_or_equal()` function returns an true if the first inet *contains or equals* the second inet.

_inet_plus(), _inet_minus()

The `_inet_plus()` and `_inet_minus` functions provide the addition and subtraction operators for inet values.

Synopsis

```
_inet_plus(<operand1>,<operand2>)
_inet_minus(<operand1>,<operand2>)
```

Description

The `_inet_plus()` and `_inet_minus()` functions return an inet value with the same subnet as the supplied inet, but whose address bits reflect the addition and subtraction respectively of the indicated ineger to the original address bits. These functions provide addition and subtraction capability in the absence of support for the "+" and "-" operators on inet values within AP SQL

Arguments

Argument	Description
<code><operand1></code>	A column expression that evaluates to an inet value.
<code><operand2></code>	A column expression that evaluates to an int32 or int64 value

Return Value

The `_inet_plus()` and `_inet_minus()` functions return an inet value unless both operands are inet values, that is, we are adding or subtracting two inet values, in which case an int64 value is returned. Consequently, adding or subtracting two inet values means that the int64 value cannot properly represent the delta between all pairs of IPv6 addresses. Under this condition, AP SQL does not produce an overflow error but simply wraps the integers/addresses and proceeds (this is consistent with how math operations on simple integers are performed in AP SQL).

Examples

If the first operand held an inet value 127.0.0.1, then the `_inet_plus()` function returns the inet value 127.0.1.2 if the second operand contained the int32 value 257.

If the first operand held an inet value 192.168.1.2, then the `_inet_minus()` function returns the inet value 192.168.0.1 if the second operand contained the int32 value 257.

If the first operand held an inet value 192.168.1.2, then the `_inet_minus()` function returns the int64 value 257 if the second operand contained the inet value 192.168.0.1.

_inet_and(), _inet_not(), _inet_or(), _inet_xor()

The `_inet_and()`, `_inet_not()`, `_inet_or()`, and `_inet_xor` functions provide the bitwise AND, NOT, OR, and XOR operators for `inet` values.

Synopsis

```
_inet_and(<expression>,<operand>)
_inet_not(<expression>,<operand>)
_inet_or (<expression>,<operand>)
_inet_xor(<expression>,<operand>)
```

Description

The `_inet_and()`, `_inet_not()`, `_inet_or()`, and `_inet_xor()` functions provide the bitwise AND, NOT, OR, and XOR operator capability in the absence of support for the "&", "^", "|", and "⊕" operators on `inet` values within AP SQL.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>inet</code> value.
<code><operand></code>	A column expression that evaluates to an <code>int32</code> , <code>int64</code> or <code>inet</code> value

Return Value

The `_inet_and()`, `_inet_not()`, `_inet_or()`, and `_inet_xor()` functions return an `inet` value.

Examples

If the first operand held an `inet` value `127.0.0.1`, then the `_inet_plus()` function returns the `inet` value `127.0.1.2` if the second operand contained the `int32` value `257`.

MISCELLANEOUS FUNCTIONS

This section describes miscellaneous functions.

Function	Purpose	Page
_quantize()	Performs distribution analysis	page 173
_fifo()	Pushes a value onto a queue	page 177
_lms_taskid()	Returns the task ID of the SQL request	page 177
_lms_buildinfo()	Returns information about the version of the EDW	page 179

Function	Purpose	Page
_fromname()	Returns the name of the table that is being queried	page 180
_fromindex()	Returns the index of the table that is being queried	page 181

_quantize()

The `quantize()` performs distribution analysis on the values in a column.

Synopsis

```
_quantize( <expression>, <low_format>, <limit>, <high_format> )

_quantize( <expression>, <low_format>[, <start>, <format>, <increment>
      [, <start>, <format>, <increment> [...]], <limit>, <high_format> )
```

Description

The `_quantize()` function performs distribution analysis on numeric values returned in a column of a result set. The function returns formatted strings that describe the distribution ranges in which the numeric values fall. You specify the distribution ranges with `<limit>` and optionally with `<start>` and `<increment>` arguments. These arguments specify the thresholds that delimit one distribution range from the next.

You can specify thresholds with:

- a single value that delimits two ranges
- or—
- a set of values that delimit multiple ranges

The function returns a formatted string that is derived from one of the format arguments. Each format argument must be a `varchar` that includes a threshold variable. The threshold variable is expressed as `%1` for low thresholds and `%2` for high thresholds. For example, a low-range format argument might be "below %1", a high-range format argument might be "%2 and above", and a mid-range format argument might be "%1 up to %2".

The formatted string that `_quantize()` returns indicates the range in which `<expression>` falls. The function compares the value of each expression in the result set against values specified for `<limit>`, `<start>`, and `<increment>`. When the shorter syntax is used, the function operates with only two ranges. When the longer syntax is used, the function operates with three or more ranges.

Arguments

Argument	Description
<code><expression></code>	A column expression that evaluates to an <code>int32</code> , <code>int64</code> , or <code>float</code> value
* The optional arguments operate as a set. You cannot specify one without specifying the others. Multiple sets of the optional arguments are allowed.	

Argument	Description
<low_format>	A varchar that includes %1 as the low-value threshold
<start>*	Optional. An int32, int64, or float value
<format>*	Optional. A varchar that includes %1 as low-value threshold and %2 as high-value threshold
<increment>*	Optional. An int32, int64, or float value
<limit>	An int32, int64, or float value
<high_format>	A varchar that includes %2 as the high-value threshold
* The optional arguments operate as a set. You cannot specify one without specifying the others. Multiple sets of the optional arguments are allowed.	

DISTRIBUTING DATA BETWEEN TWO RANGES

When only <limit> is specified, the `_quantize()` function compares the value of <expression> to the value of <limit>. If <expression> is less than <limit>, the function replaces %1 in the <low_format> argument with <limit> and returns the formatted string. If <expression> is equal to or greater than <limit>, the function replaces %2 in the <high_format> argument with <limit> and returns the formatted string.

For example, assume you run the following query:

```
SELECT RespSize, _quantize(RespSize, "under %1", 16000, "%2 or above") as range
FROM example_webserv_100
DURING ALL;
```

The example below illustrates a subset of the rows that the `_quantize()` function returns.

```
+-----+
| Results for SQL file >(standard input)< +
+-----+-----+
|RespSize|    range    |
| (int32) | (varchar)   |
+-----+-----+
| 37361 | 16000 or above |
| 261 | under 16000 |
| 0 | under 16000 |
| 7121 | under 16000 |
| 37361 | 16000 or above |
| 3172 | under 16000 |
| 17252 | 16000 or above |
| 12203 | under 16000 |
| 0 | under 16000 |
| 7121 | under 16000 |
```

DISTRIBUTING DATA AMONG MULTIPLE RANGES

When <start>, <format>, and <increment> arguments are specified, in addition to <limit>, the `_quantize()` function compares the value of <expression> to <limit> and the other intermediate thresholds, which it computes. The additional <format> argument must include both %1 as the low threshold variable and %2 as the high threshold variable; for example, it might be "%1 up to %2".

The `_quantize()` function evaluates the syntax with one start value as follows:

- If `<expression>` is equal to or greater than `<limit>`, `_quantize()` replaces “%2” in `<high_format>` with `<limit>` and returns the resulting string.
- If `<expression>` is less than `<start>`, `_quantize()` replaces “%1” in `<low_format>` with `<start>` and returns the resulting string.
- If `<expression>` is greater than `<start>` but less than `<limit>`, `_quantize()` operates with a set of intermediate ranges to which it applies `<format>`.

The first intermediate range begins with the start value and ends with the threshold computed by adding the increment value to the start value. The `quantize()` function computes additional intermediate ranges by sequentially adding the increment value to the upper threshold of the previous range. It then compares each expression to the values in these intermediate ranges. When `<expression>` is greater than `<start>` but less `<limit>`, `_quantize()` determines which intermediate range it falls within. It replaces %1 in `<format>` with the lower threshold of the range and replaces %2 with the higher threshold and returns the resulting string.

For example, assume you run the following query:

```
SELECT RespSize, _quantize(RespSize, "under %1",
    1000, "%1 up to %2", 5000,
    16000, "%2 or above") as range
FROM example_webserv_100
DURING ALL;
```

The example below illustrates a subset of the rows that the `_quantize()` function returns.

```
+-----+
| Results for SQL file >(standard input)< +
+-----+-----+-----+
|RespSize|      range      |
| (int32) | (varchar)       |
+-----+-----+-----+
|    37361|16000 or above  |
|      261|under 1000      |
|        0|under 1000      |
|    7121|6000 up to 11000|
|    37361|16000 or above  |
|    3172|1000 up to 6000 |
|   17252|16000 or above  |
|   12203|11000 up to 16000|
|        0|under 1000      |
|    7121|6000 up to 11000|
```

IMPORTANT: The `_quantize()` function returns the high-format string for any value that equals or exceeds `<limit>`. To avoid confusing results, ensure that `<start>` and `<increment>` define intermediate ranges such that an upper threshold eventually coincides with `<limit>`.

SPECIFYING MULTIPLE START VALUES

You can specify more than one set of `<start>`, `<format>`, and `<increment>` arguments to the `_quantize()` function. The `_quantize()` function evaluates the syntax with multiple start values as follows:

- If `<expression>` is equal to or greater than `<limit>`, `_quantize()` replaces “%2” in `<high_format>` with `<limit>` and returns the resulting string.

- If `<expression>` is less than the first `<start>`, `_quantize()` replaces “%1” in `<low_format>` with the first `<start>` and returns the resulting string.
- If `<expression>` is greater than the first `<start>` but less than the second `<start>`, `_quantize()` places the expression value within the appropriate intermediate range between the two.
- If `<expression>` is greater than the second `<start>` but less than the third `<start>`, or `<limit>` if there are no additional `<start>` arguments, `_quantize()` places the expression value within the appropriate intermediate range.

IMPORTANT: If there are multiple start values, you must list them in increasing order for the result to be meaningful.

For example, assume you run the following query:

```
SELECT RespSize, _quantize(RespSize, "under %1",
    1000, "%1 up to %2", 500,
    5000, "%1 up to %2", 1000,
    16000, "%2 or above") as range
FROM example_webserv_100
DURING ALL;
```

The graphic below illustrates the purpose of the numeric values in the above query.

```
SELECT RespSize, _quantize(RespSize, "under
    %1",
First <start> _____ First <increment>
    1000, "%1 up to %2", 500,
Second _____ Second
    5000, "%1 up to %2", 1000,
<limit> _____
    16000, "%2 or above") as range
```

The example below illustrates a subset of the rows that the `_quantize()` function returns.

```
+-----+
| Results for SQL file >(standard input)< +
+-----+
|RespSize|      range      |
| (int32) | (varchar)        |
+-----+
| 37361 | 16000 or above |
| 261   | under 1000     |
| 0     | under 1000     |
| 8132  | 8000 up to 9000 |
| 37361 | 16000 or above |
| 3172  | 3000 up to 3500 |
| 17252 | 16000 or above |
| 5208  | 5000 up to 6000 |
| 12203 | 12000 up to 13000 |
| 43    | under 1000     |
| 3894  | 3500 up to 4000 |
| 2421  | 2000 up to 2500 |
| 4765  | 4500 up to 5000 |
| 1313  | 1000 up to 1500 |
```

GRAPHING THE RESULTS

Because the `_quantize()` function labels each range, you can graph its result set in SenSage AP Console. To make the graph meaningful, put the statement with the `_quantize()` function in a subquery and then count the ranges in the main query. Run the query in SenSage AP Console and display the results as a bar graph.

For example, the following query enables graphing in SenSage AP Console:

```
WITH subquery as (
  SELECT RespSize, _quantize(RespSize, "under %1",
    5000, "%1 to %2", 10000,
    15000, "%1 to %2", 25000,
    30000, "%2 or above") as range
  FROM example_webserv_100
  DURING ALL
)
SELECT range, count(*)
FROM subquery
GROUP BY range
ORDER BY 1
```

Return Value

The return type of the `_quantize()` function is a `varchar`.

Exceptions

The `_quantize()` function raises a SQL processing exception under any of these conditions:

- When the data types of the `<expr>` or `<limit>` arguments are not `int32`, `int64`, or `float` values.
- When the data types of the `<expr>` and `<limit>` arguments are not identical.
- When the data type of format arguments are not `varchar`.
- If you pass an incorrect number of arguments.

_fifo()

The `_fifo()` function pushes a value onto a queue.

Synopsis

```
_fifo( <list_name>, <value>, <default> )
```

Description

If the queue designated by `<list_name>` contains at least one element, `_fifo()` shifts off the first value and adds the specified `<value>` to the end of the queue, returning the removed value.

If the queue designated by `<list_name>` is empty, `_fifo()` adds the specified `<value>` to the end of the queue and returns the value of the `<default>` argument, if specified.

If the queue designated by `<list_name>` is empty and no `<default>` is specified, `_fifo()` raises an exception.

This function is often used with the SLICE BY clause. For more information, see [“SLICE BY Clauses”, on page 56](#).

Arguments

Argument	Description
<code><list_name></code>	The name of a list such as 'myfifo'. The name <code><list_name></code> may be derived from a column so that each group (those rows with the same value in a certain column) has its own FIFO queue.
<code><value></code>	A value of any data type that designates the value to be pushed into the queue
<code><default></code>	Mandatory. The value to be returned when the designated queue is empty.

Return Value

The return type is the type of the `<default>` argument.

Exceptions

The `_fifo()` function raises an SQL processing exception if the data types of `<value>` and `<default>` are not the same.

_lms_taskid()

The `_lms_taskid()` function returns the task ID of the SQL request.

Synopsis

```
_lms_taskid()
```

Description

The `_lms_taskid()` function returns the internal task ID generated by the system when it receives a SQL request.

Return Value

The `_lms_taskid()` function returns a `varchar` value that contains the internal task ID.

Example

The following query returns the value of the task ID:

```
SELECT TOP 1 _lms_taskid()
FROM example_webserv_100
DURING ALL
```

Results:

```

+-----+
| Results for SQL file >example-lms-01.sql< |
+-----+
|          lms_taskid          |
|          (varchar)          |
+-----*
output is post-sorted
+-----*
| 955BC97C2E68F83F797E0AEFF3BC0307 |
+-----*

```

_lms_buildinfo()

The `_lms_buildinfo()` function returns information about the version of the EDW.

Synopsis

```
_lms_buildinfo()
```

Description

The `_lms_buildinfo()` function returns a string describing the version of the EDW.

Return Value

The `_lms_buildinfo()` function returns a varchar that contains the build date, the ID of the most recent committed software change, the name of the build client specification, the name and IP address of the machine on which the build occurred, the directory containing the build files, and a list of the files compiled with VERBOSE logging enabled.

Example

The following query returns the build information for the EDW.

```

SELECT TOP 1 _lms_buildinfo()
  FROM example_webserv_100
  DURING ALL;

```

The following shows an example of the output:

```

2008-09-25T03:44:06+0000 Change 56459 Client name:
source.quattroloop.2006.main.56460 Client host: eng.hq..com Client root: /
export/services/buildloop/tmp/67clae309a349577d03d1973ae9d7d21/ootb-analytics
verbose:

```

`_fromname()`

The `_fromname()` function returns the fully qualified name of each table or view that is being queried.

Synopsis

```
_fromname()
```

Description

SenSage AP supports a `FROM @<list_expression>` construct that concatenates the rows of one or more tables and/or views and provides the result as the source data for the `SELECT` statement. The result of the query is equivalent to the results of a `UNION ALL` query.

You can use the `_fromname()` function to identify the source object (table or view) of each row in the result set of a `SELECT` statement whose `FROM` clause uses the `@<list_expression>` construct. The `_fromname()` function evaluates to a string that represents the name of the object in the `FROM @<list_expression>` construct for each returned row.

For more information, see:

- [“FROM Clauses”, on page 47](#)
- [“Working with Lists”, on page 89](#)
- [“_tablematch\(\)”, on page 110](#)

Return Value

The `_fromname()` function returns a `varchar` with the name of each table or view that is being queried.

Examples

The following statement queries two tables and returns the earliest timestamp, the latest timestamp, and the number of records in each table, grouped by tablename. The name of the table is returned in the first column of the result set.

```
SELECT _fromname(), min(ts), max(ts), count(*)
FROM @_list('example_webserv_100', 'example_syslog')
GROUP BY 1
DURING ALL;
```

The following statement queries all tables in the default namespace and returns a count of each url in each table returned. In this case, the table names are not known at the time the query is written.

```
SELECT _fromname(), url, count(*)
FROM @_tablematch('.*')
GROUP BY 1,2
DURING ALL;
```

_fromindex()

The `_fromindex()` function returns the index of the table or view that is being queried.

Synopsis

```
_fromindex()
```

Description

SenSage AP supports a `FROM @<list_expression>` construct that concatenates the rows of one or more tables and/or views and provides the result as the source data for the `SELECT` statement. The result of the query is equivalent to the results of a `UNION ALL` query.

You can use the `_fromindex()` function to identify the source object (table or view) of each row in the result set of a `SELECT` statement whose `FROM` clause uses the `@<list_expression>` construct. The `_fromindex()` function evaluates to an `int64`. The position is zero-based, which means that `0` represents the first name, `1` the second, and so on.

For more information, see:

- [“FROM Clauses”, on page 47](#)
- [“Working with Lists”, on page 89](#)
- [“_tablematch\(\)”, on page 110](#)

Return Value

The `_fromindex()` function returns an `int64` with the 0-based index of the table that is being queried.

Examples

The following statement queries two tables and returns the earliest timestamp, the latest timestamp, and the number of records in each table, grouped by the position of the table in the result set. The position of the first table is represented as `0`.

```
SELECT _fromindex(), min(ts), max(ts), count(*)
FROM @_list('example_webserv_100', 'example_syslog')
GROUP BY 1
DURING ALL;
```

The graphic below illustrates the results of this query.

Results for SQL file >(standard input)<			
fromindex	min	max	count
(int64)	(timestamp)	(timestamp)	(int64)
0	2002-01-30T07:44:37.000000Z	2002-03-17T18:08:45.000000Z	60191
1	2002-01-30T07:44:37.000000Z	2002-03-17T18:08:45.000000Z	60191

Elapsed time: 1.4 sec Number of rows: 2

The following statement queries all tables in the default namespace and returns a count of each url in each table, grouped by the position of the table in the result set.

```
SELECT _fromindex(), url, count(*)
FROM @_tablematch('.*')
GROUP BY 1,2
DURING ALL;
```

NOTE: In this case, the table names are not known, either at the time the query is written or in the result set. This query, which enables you to aggregate the results by table position rather than by the full result set, is useful for aggregating the result data when you do not need to know the table names.

The graphic below illustrates a few of the results of this query.

Results for SQL file >(standard input)<		
-----+		
fromindex		url
count		
(int64)		(varchar)
-----+		
-----*		
0 /		
4787		
0 /%20target=_top		
1		
0 ../company/index.html		
1		
0 ../contact/index.html		
2		
0 ../product/index.html		

Configuring and Managing Open Access Extension (OAE)

This chapter includes these sections:

- “Overview”, next
- “Managing EDW External Tables in Postgres”, on page 190
- “Understanding OAE Security”, on page 194

OVERVIEW

Open Access Extension (OAE) allows users to access EDW data with such open standards as ANSI SQL, ODBC, and JDBC. OAE uses the Postgres database query processing interface to provide access to SenSage AP EDW data.

Although PostgreSQL is the official product name, the product is conventionally referred to as "Postgres". Hexis documentation uses Postgres.

Also, while SenSage AP OAE uses the Postgres interface to access EDW data, Postgres commands are not documented. For Postgres commands, refer to Postgres documentation at:

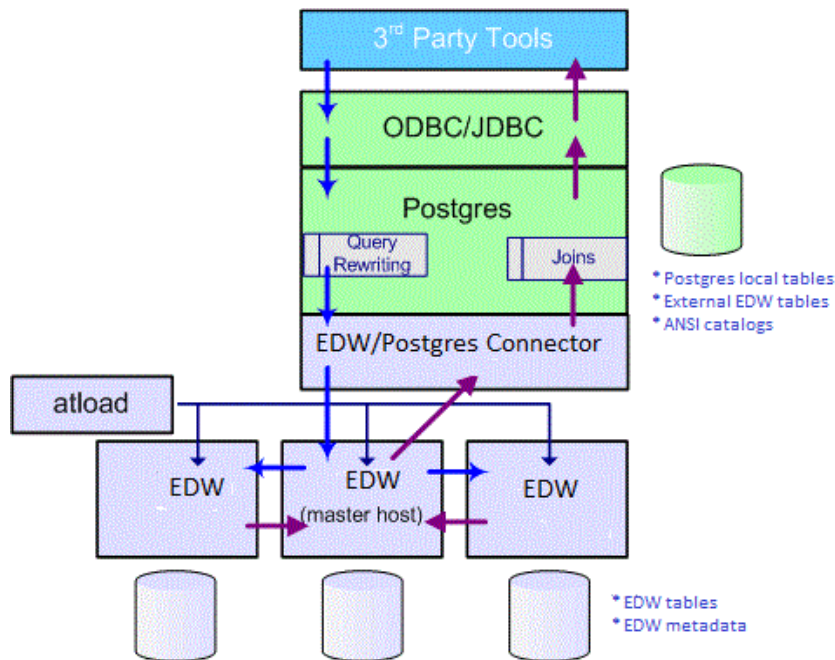
<http://www.postgresql.org/docs>

This section covers the following topics:

- “OAE Architecture and Process Flow”, next
- “Handling Failover and Replication”, on page 186

OAE Architecture and Process Flow

Figure 5-1: Architecture that Supports OAE



As illustrated above, OAE enables any third-party tool that uses standard ODBC or JDBC to query EDW tables by sending the queries through Postgres, whose SQL is ANSI SQL. Postgres uses a SenSage AP-provided connector to send data-access requests to and receive result sets from an EDW instance.

When you use OAE to query data in the EDW, you use Postgres SQL (PSQL) to write your queries. OAE uses a SenSage AP plugin named OAE (Open Access Extension) to rewrite these PSQL queries into SenSage AP SQL (AP SQL) queries. The EDW head node receives each query from Postgres and distributes it to all other hosts in the cluster. The head node receives query results from each EDW host and returns all results to Postgres. OAE processes the requests without any loss of data or data integrity, and uses the OAE plugin to process joins on the data returned to Postgres. This distribution exploits the EDW Massively Parallel Processing (MPP) architecture by executing the EDW query in parallel.

The EDW is responsible for all data loading into EDW tables and stores EDW tables and metadata. Postgres maintains metadata about its own tables in its system catalogs. Through the OAE plugin, OAE makes EDW tables and views available through the Postgres system catalogs. Storing information about the EDW tables in the Postgres system catalogs enables users and third-party tools to use these catalogs to build queries and applications.

To enable EDW tables and views to be available in Postgres, OAE uses *external tables*, which define each table and view in an EDW instance as a view in Postgres. These views that represent EDW tables and views are called external tables.

In addition to the oae plugin, OAE provides the oae schema, which contains all the SQL code associated with OAE. A schema in Postgres is similar to an EDW namespace in that they both contain tables and views. However:

- A Postgres schema can also contain objects other than tables and views; these objects include functions.
- A Postgres schema cannot be nested.

Among the functions that the oae schema contains, one defines every table and view in an EDW instance as an external table in Postgres. A single call to this function creates a view definition for each table and view in the instance. This process occurs automatically during the SenSage AP installation process.

IMPORTANT: The installation process defines all EDW tables and views in Postgres. However, over time the system catalog for the specified instance might change. By default, OAE automatically updates system-catalog changes that occur when database object(s) are renamed or the schema has changed in the EDW. For more information, see “[Managing EDW External Tables in Postgres](#)”, on page 190.

NOTE: If the metadata gets out of sync with the EDW:

- Use the CREATE, DROP, and RENAME statements for reconciliation.
- Use the following function to reload the metadata, where *<db>* is the name of the EDW database:

```
oae.reload_db(<db>)
```

WARNING: Be sure no custom views or tables were created under the OAE external schema. The `oae.reload_db()` function will drop the existing external schema and re-create it by pulling the latest metadata on the EDW database. Therefore, any existing custom views or tables created under the OAE external schema will not be re-created because neither the OAE external schema or the EDW database is aware of them.

If you simply want the latest updates to the EDW database without re-creating the entire OAE external schema, then use:

```
oae.force_sync(<db>)
```

This function will not drop the current OAE external schema; instead just apply the latest updates from the EDW database.

Obtaining Drivers and Client Software

To take advantage of OAE, third-party tools use Postgres as their database and use ODBC and JDBC drivers to connect to the Postgres database. For more information on configuring these drivers, see the following Postgres documentation:

<http://jdbc.postgresql.org/>

and

<http://psqlodbc.projects.postgresql.org/>

JDBC and ODBC drivers for Postgres are included with your SenSage AP distribution along with some client software. You can also download the drivers by accessing the following URL from a Web browser:

`https://<hostname of SenSage AP Application Manager>/downloads`

The following software is available:

- SenSage AP pgAdmin3 tool- 1.8.4
- ODBC-Windows-64bit
- ODBC-Windows-32bit
- JDBC4-JVM 1.6+

IMPORTANT: When accessing the Postgres database using JDBC, you'll need to use the SSL-secured URL, as follows:

```
jdbc:postgresql://xxx.sensage.com:5432/  
controller?ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory
```

For an example of using ODBC and the Postgres database query processing interface to access SenSage AP EDW data from Microsoft® Office Excel 2003, see ["Using OAE over ODBC to Return Data to Excel: An Example"](#), on page 210.

Handling Failover and Replication

OAE supports failover by designating a primary EDW host connection and a list of secondary, failover connections. If the primary host is not reachable, OAE automatically connects to the secondary EDW host connection, and so on, until the connection is established to an EDW host and the query can be executed.

To set up the failover servers, specify the connection string in the format:

```
'host1:port1 | host2:port | ...'
```

OAE does not provide Postgres replication. A customer site can use one of the many Postgres replication solutions and have a warm standby Postgres server in case the active Postgres server goes down, but Hexis Cyber Solutions does not test or support these solutions.

Mapping Tables and Views from the EDW to Postgres

The mapping from EDW tables and views to Postgres is based on the following:

- the connection identifier

OAE users use this connection ID to send queries from Postgres to the EDW.

- a schema for each namespace in the EDW instance

Although the EDW allows namespace hierarchies, Postgres allows only `"database.schema.table"`. Postgres represents an EDW namespace that consists of nested

namespaces by separating the individual namespaces with an underscore rather than the period used in the EDW.

When an EDW instance contains multiple namespaces, OAE creates a separate schema for each one. For more information, see [“Mapping Namespaces and Table Names Between the EDW and Postgres”](#), on page 191.

- a view for each table and view in the instance

Each view defines one external table and provides the table definition to Postgres.

Because there are Postgres views for each EDW table and view in the EDW instance, schemas for these data objects are readily accessible through Postgres system catalogs. These catalogs enable all third-party tools to discover EDW data.

Managing Users and Roles in Postgres

OAE uses the Postgres database named **controller** and creates all of its schemas (including the **oae** schema) in that database. By default, Postgres is configured to have three users:

- **controller**— used internally by SenSage AP components. Regular users should not log in using this username. If they do login, they may accidentally modify system metadata that could potentially bring down the application.
- **sls**— used internally by SenSage AP components. Regular users should not log in using this username as it has limited privileges.
- **SenSage AP user**—this username has the same name as the SenSage AP user configured on your system. The SenSage AP user does not have a password and is set up so only the SenSage AP Linux user or root can log in.

In order to log in as this user, log into the Linux session on the node that runs Postgres using ssh. Login as the SenSage AP user. After logging in as the SenSage AP user, you can connect to Postgres as any user (without a password) and you can use the Postgres command-line tools without a password.

The Postgresql command-line tools are located in the following directory:

```
<HawkEye AP Home>/bin.
```

Hexis Cyber Solutions recommends that you create your own administrative user with your own password in order to manage Postgres. To create your own administrative user, log into Linux on the machine that is running Postgres (as the user that owns Postgres) and run the following command:

```
createuser -srP <new user name>
```

You will be prompted for a password and you may also be prompted for the password for the Linux SenSage AP user.

After you create this administrative role, you can log into Postgres using the administrative role to perform other administrative activities such as creating more users. Hexis Cyber Solutions recommends that you use the pgAdmin tool for these tasks. This tool is included with your SenSage AP distribution. You can download the tool by accessing the following URL from a Web browser:

`https://<hostname of SenSage AP Application Manager>/downloads`

Using the Postgres Command-Line Tool to Log Into Postgres

To log into Postgres

- 1 Run the following command:

```
psql controller <user_name>
```

If prompted, enter your password.

NOTE:

- To facilitate using Postgres command-line tools, add `<HawkEye AP Home>/bin/` to your PATH variable.

Configuring OAE

OAE provides a set of configuration variables that enable you to customize your environment. OAE configuration options are described in the sections below.

NOTE:

- To set a configuration variable, use the Postgres `set` command, as in:

```
SET oae.display_translation = true;
```

- Alternatively, an administrator can edit the following file:

```
<HawkEye AP Home>/data/postgresql.conf
```

After you save this file, any new sessions that start will use the new settings. However, to get the settings read into existing sessions, you should run the Postgres shell command `pg_ctl reload`. See the Postgres documentation for more information (<http://www.postgresql.org/docs/8.3/static/config-setting.html>).

- To view the value of a configuration variable, use the Postgres `show` command:

```
SHOW oae.version;
```

Displaying version information

Use the read-only configuration variable, `'SHOW oae.version'` to display version information

Displaying OAE Translations

- To display OAE translations in the Postgres log file set `oae.translation_logging` to true.
- To display OAE translations to the client, set `oae.display_translation` to true.

NOTE: Translation messages log at the INFO level (see section 18.7.2 in the Postgres documentation at <http://www.postgresql.org/docs/8.3/static/runtime-config-logging.html>),

Allowing for Correlated External Subqueries

To allow correlated external subqueries, set `oe.allow_correlated_external_subqueries` to `true`. This option allows users to write queries with subqueries that:

- reference columns from the outer query
- and
- reference external tables.

Because these queries are very inefficient, Hexis Cyber Solutions does not recommend using them. Therefore, this variable defaults to `false`, which causes OAE to return an error for such queries.

Example:

The subquery in the following query is a correlated external subquery:

```
SELECT * FROM WeblogView WHERE EXISTS (SELECT ClientDNS FROM Pubs.Weblog Weblog
WHERE WeblogView.ClientDNS=Weblog.ClientDNS)
```

Note that in the above, `Pubs Weblog` is an EDW table.

Allowing only for Queries Sent to EDW

To allow only queries that are sent to the EDW, set `oe.require_whole_query_translation` to `true`. By default, if a query references external tables but also has features that prevent it from being sent to the external server, OAE does its best to retrieve the minimal amount of data from any external tables and then processes the remaining part of the query in Postgres.

This option prevents this partial solution and allows only queries that are sent entirely to the EDW.

Example: The external-table query below uses the Postgres cube-root function, which EDW does not recognize:

```
select * from myslsinstance_pubs.weblog where
  RespTime > 0 and cbrt(RespSize) < 10
```

OAE would send the following translated query to the EDW:

```
SELECT * FROM Pubs.Weblog WHERE
  RespTime > 0
```

Postgres handles the `cbrt(RespSize)`. Typically the user would not know where the function is processed. However, if this configuration variable is set to `true`, this query would error out because the whole query is not translated.

Defaults to `false`.

Returning EDW Query Error for a Call that Omits OAE "During" Clause

To return an error for any EDW external-table query that does not include a call to the `oe.during()` function, set `oe.require_DURING_clause` to `true`. To prevent unwanted return of every row in a table, the EDW requires a DURING clause in every query.

If this variable is set to `false` and you write an external-table query without a call to `oe.during()`, OAE sets the value of the DURING clause to `'all'`.

However, if this variable is set to true and you write an external-table query without a call to `oae.during()`, Postgres returns an error.

If this option is set to true but you want to query over all timestamps, use `oae.during('all')` in your EDW external-table query.

NOTE: A pass-through query does not accept a call to `oae.during()`.

Defaults to `false`.

Limiting Row Maximum Returned from an EDW Query

To avoid overloading a single, small Postgres server, OAE limits the number of rows returned to Postgres. By default the limit is 1,000,000 rows. This is an arbitrary limit and may be changed to correspond with the sizing of your Postgres server deployment (consult Professional Services for sizing analysis).

To set the limit for rows returned, locate the following file:

```
/opt/hexis/hawkeye-ap/etc/atpgsql/postgresql.conf
```

In this file, add or change the value of the following property:

```
oae.returned_rows_limit=
```

If this property is missing from the file, the default limit of 1,000,000 rows is in effect. To remove this limit set the property to **0**:

```
oae.returned_rows_limit=0
```

To set any other limit for this property, specify any positive integer (without commas) that represents the maximum number of rows you want returned.

If a query returns more than the specified number of rows, OAE generates an error message and no rows are returned.

NOTE: The `oae.returned_rows_limit` property also applies to pass-through queries.

MANAGING EDW EXTERNAL TABLES IN POSTGRES

As mentioned above in [“OAE Architecture and Process Flow”](#), on page 184, OAE defines all tables and views in your EDW instance as Postgres views. Because the names of objects such as namespaces, tables, and views can be longer in the EDW than in Postgres, the installation process truncates the EDW names as required when it defines them in Postgres.

This section describes the naming truncation and documents how to discover the truncated names. It also documents how an administrator can synchronize schema changes to Postgres when EDW tables and views are created or deleted.

To create or modify the definition of external tables in Postgres, a user must have the following privileges:

- **EDW**— a minimum of read privilege on all namespaces in the EDW instance
- **Postgres**—be the owner of OAE and have CREATE privilege on the current database.

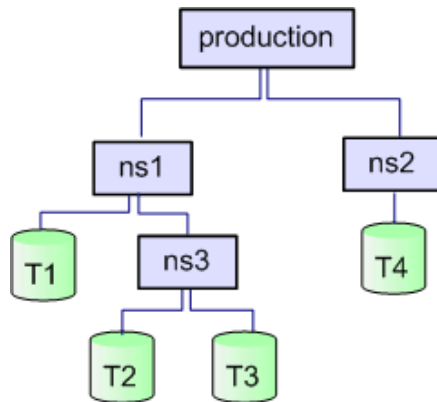
This section includes the following topics:

- “Mapping Namespaces and Table Names Between the EDW and Postgres”, next
- “Synchronizing EDW Schema Changes to Postgres”, on page 194

Mapping Namespaces and Table Names Between the EDW and Postgres

EDW allows namespaces to be arranged hierarchically, as illustrated in [Figure 5-2](#).

Figure 5-2: Example Namespace Hierarchy in the EDW



The example above illustrates the production root namespace with three namespaces and four tables below it. This example illustrates four namespaces in the EDW:

- production
- production.ns1
- production.ns1.ns3
- production.ns2

If OAE were to create the external tables for the example above, it would create the following three schemas in Postgres:

- production
- production_ns1
- production_ns1_ns3
- production_ns2

Postgres maps each namespace to a single schema. It creates the schema name by substituting an underscore for each period that separates the individual namespaces in each namespace and it converts all uppercase letters to lowercase. Then it prepends the name of the connection ID to the beginning of the schema name, followed by another "_".

For a more real-world example, assume the following view exists in an EDW instance named MyCompany:

```
investigation__senske_collectorTransaction_syslogng
```

Assume further that the namespace in which the table is located is:

```
production.eastern.analytics.intellischema.__connectors
```

Postgres would create the following schema name:

```
mycompany_production_eastern_analytics_intellischema__connectors
```

IMPORTANT: Although Postgres parses identifiers of any length, it uses only the first 63 characters when comparing names. It compares the table or view name separately from the schema name. Therefore the table name and the schema name can each be a maximum of 63 characters long.

In the example above, the view name (`investigation__sensation_collectorTransaction_syslogng`) is only 53 characters. However, the schema name is 66 characters.

If the created schema name is longer than the longest identifier that Postgres allows, OAE reduces the leftmost component of the namespace to a single-letter abbreviation. If that name is still too long, it reduces the next component to its initial letter, and so on up to but not including the last namespace represented in the schema name.

Given the example above, with a 66-character schema name, OAE would truncate the schema name to:

```
mycompany_p_eastern_analytics_intellischema__connectors
```

The highlighted letter above indicates the name that was truncated. If the name is still too long, OAE truncates "eastern" to its initial "e". If the name is still too long, it continues truncating the name of each namespace, from left to right, to a single letter until it reaches the namespace on the right.

If the name is still too long after OAE has truncated every namespace but the one on the right, it truncates more characters from the right until it is short enough.

After this base name has been created, OAE looks for duplicates. If it finds any duplicates, it begins a process of appending numbers until it creates a unique name. If the name now exceeds the length limit, it removes enough other characters to make room for the numbers.

Working with Schema Names

This topic contains the following subtopics:

- [“Schemas Names and Namespaces”, next](#)
- [“Discovering the Default Schema Name”, on page 193](#)
- [“Changing the Schema Name”, on page 193](#)

SCHEMAS NAMES AND NAMESPACES

Assume that your EDW instance contains multiple namespaces, as illustrated in [Figure 5-2](#). For example, assume your Production instance also contains the `investigation__sensation_collectorTransaction_syslogng` view in a second namespace:

```
production.western.analytics.intellischema.__connectors
```

OAE creates a schema for the eastern namespace, this western namespace, and all other namespaces in the EDW instance. The schema name for the western namespace would be:

```
mycompany_p_western_analytics_intellischema__connectors
```

The highlighted text above indicates where the example western schema name would differ from the example eastern schema name.

You can discover the default name of each schema and, if desired, change the default names. These processes are described below.

DISCOVERING THE DEFAULT SCHEMA NAME

A user can use the following function to discover the default schema name of each namespace:

```
oe.namespace_to_schema(<connection_name>, <namespace>)
```

This function returns the Postgres schema name currently associated with the specified namespace. For example, given a connection name of MySLInstance, a user could use the following function to determine the schema name of the specified namespaces:

```
select oe.namespace_to_schema('MySLInstance',
    'production.eastern.analytics.intellischema.__connectors');
```

The function above would return:

```
mycompany_p_eastern_analytics_intellischema__connectors
```

NOTE: As described below, you can change the default schema name. After you do so, you can run this function again to verify the name change.

DISCOVERING THE NAMESPACE

A user can use the following function to discover the namespace for a given schema name:

```
oe.schema_to_namespace(_schema_name_text)
```

This function returns the Postgres namespace currently associated with the specified schema name. For example, given a connection name of "mycompany_p_eastern_analytics", a user could use the following function to determine the schema name of the specified namespaces:

```
select oe.schema_to_namespace('mycompany_p_eastern_analytics__connectors');
```

The function above would return:

```
production.eastern.analytics.intellischema.__connectors
```

CHANGING THE SCHEMA NAME

After the schemas have been created, the administrator can make the names more convenient by running the replace schema function:

```
oe.replace_namespace_translation(<eds_name_text>, <namespace>,
    <new_schema_name>)
```

The administrator must run this function separately for each schema name to be replaced. For example, to replace the following schema name:

```
mycompany_p_eastern_analytics_intellischema__connectors
```

For example:

```
select oae.replace_namespace_translation('MySQLInstance',  
    'production.eastern.analytics.intellischema.__connectors',  
    'mycompany_prod_east_analytics_intellischema__connectors');
```

The function above keeps the schema name short while also making the element names meaningful.

Working with Table, View, and Column Names

As mentioned above, Postgres translates table and view names separately from schema names. Table and view names, as well as column names, must also each be a maximum of 63 characters long. OAE shortens these names differently from the way it shortens schema names. It truncates characters from the right side of the name and appends a number to make the name unique.

When OAE renames a table that exists in more than one namespace, or a column that exists in more than one table, it always shortens the name identically.

Synchronizing EDW Schema Changes to Postgres

If the EDW system catalog changes, the changes will be synchronized to Postgres. If the changes come close together, the user may experience a delay before the changes take effect.

OAE and EDW IP Address Support

The OAE sends IP address data (IPv6 /IPv4) to and from the EDW, converting EDW inet data to Postgres. OAE also provides support for SenSage AP SQL functions that are using the inet datatype and converts EDW inet data to Postgres IP data. The inet data type interacts with the Postgres engine in the following way:

- The engine is able to detect the EDW IP data type column value in an EDW table schema that is defined as an IPv4/IPv6 address
- When a query is issued in OAE, the Postgres engine detects if the query was run with the "ANSI mode" option (provided with the **atquery** command). When this mode is in effect, the engine is able to detect if the inet data type family code of the inet data type column has a special value of either "none" or "invalid". These values mean that either the IPv6/IPv4 IP address in the column does not apply (none) or a non-empty string that did not contain a valid IPv6/IPv4 address (invalid) was converted to an inet address.
- If either of these two special values are detected, the engine transforms them into "0.0.0.0/0" so that the original inet values are never displayed.

UNDERSTANDING OAE SECURITY

Postgres connects to the EDW as a privileged user. The trusted connection has read access to all data but does not have write access. To ensure proper access from Postgres to the EDW data, the administrator who configures OAE must create similar users, roles, and user/role relationships

in Postgres as has been created for the EDW instance. In other words, the Postgres administrator must protect that data by securing it appropriately for each Postgres user.

As described in “[Overview: Pass-Through Queries](#)”, on page 197, OAE provides a function that enables users to query the EDW from Postgres. The function `oae.sls_qry()`, has the same privileges as the login used to create the connection identifier. Typically, these privileges are:

- **EDW**—a minimum of read privilege on all namespaces in the EDW instance
- **Postgres**—be the owner of OAE and have CREATE privilege on the current database.

Hexis Cyber Solutions recommends that you limit access to EDW data with Postgres roles and privileges.

Security Issues

- Assume that there are two external schemas, `windows_local` and `unix_local`, that link to the EDW windows and unix namespaces, respectively. Assume further that a user should have access only to the windows namespace. In this case, the administrator gives the user access to `windows_local` and the objects in `windows_local`.

If the administrator creates a view in `windows_local` that accesses an EDW table in the unix namespace and if the administrator gives SELECT permission on the view to the user above, the administrator is giving this user access to data in the unix namespace. In this case, the administrator did not follow security policy when creating the view.

- The `oae.sls_qry()` function is called with a connection ID. Each connection ID has an associated username.. Therefore, any user who can call one of those functions can access any data in the EDW that can be accessed with the trusted username.

In general, this username pair has privileges that should not be given to ordinary users. Therefore, an OAE site should not allow ordinary users to execute these functions directly. An ordinary user should only be given privilege to SELECT from views that use these functions.

Encryption and Single Sign-On

EDW data encryption is not available in Postgres tables. To cache sensitive data from the EDW in Postgres, users must rely on existing Postgres data-security facilities.

NOTE: OAE does not cache EDW data any longer than required to run the query.

Also, Postgres does not support the EDW version of single sign-on (SSO).

Using Open Access Extension to Query the EDW

This chapter includes these sections:

- “Overview: Pass-Through Queries”, next
- “Joins and Subqueries”, on page 198
- “Running Queries Against the EDW”, on page 199
- “Transaction Isolation Levels”, on page 207
- “Running DDL and DML Commands”, on page 208
- “Setting up OAE for EDW Multi-cluster Access”, on page 208
- “Group Tables and EDW Multi-Cluster Support”, on page 209
- “Using OAE over ODBC to Return Data to Excel: An Example”, on page 210

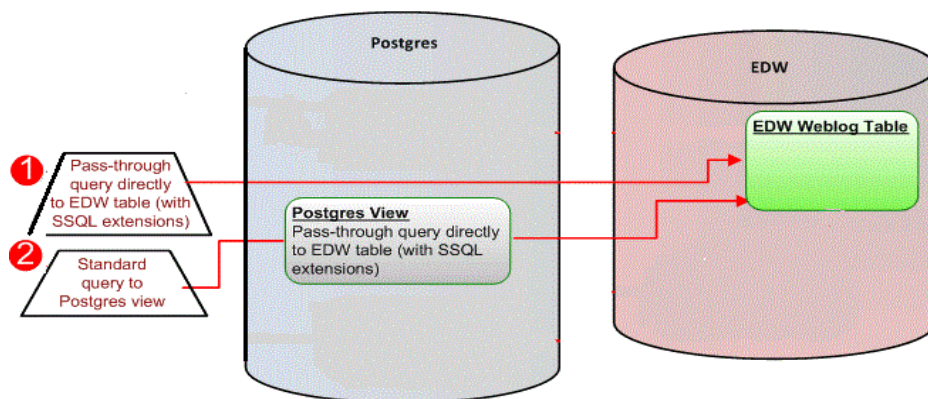
OVERVIEW: PASS-THROUGH QUERIES

Open Access Extension (OAE) supports *pass-through* queries. A pass-through query uses the `oe.sls_qry()` function, which passes AP SQL directly to the EDW and accepts results from the EDW. This query type enables a sophisticated user to take advantage of the full power of the EDW and use EDW extensions (such as the `_fifo` function and the `SLICE BY` clause) in AP SQL when querying the EDW from Postgres.

IMPORTANT: OAE does not modify or optimize a pass-through query.

For information about how to create your own pass-through queries, see “Additional Ways to Use EDW Extensions When Querying EDW Data”, on page 205.

Figure 6-1: Two Ways to Query the Same EDW Table



In Figure 6-1, The first pass-through query uses AP SQL extensions and queries the EDW table directly. Although the second query uses standard SQL, it is run against a Postgres view that uses a pass-through query.

- **Query 1**— illustrates a pass-through query that passes AP SQL directly to the EDW Weblog table. This query does *not* take advantage of OAE optimization. For this query, an administrator must write the AP SQL manually.

This query must be written by an administrator who understands AP SQL extensions and the `oae.sls_qry()` function. If it includes a WHERE clause, the query can take advantage of OAE optimization only if it pushes WHERE-clause processing to the EDW. In this case, this query should perform faster than Query 3. If, however, WHERE-clause processing is not pushed to the EDW, this query performs no faster than Query 3. For more information, see [“Query 1: Example Pass-Through Query”, on page 206](#).

- **Query 2**— illustrates a standard query run against a view created manually in Postgres. Although the view was defined in Postgres, it uses AP SQL extensions. Therefore, this query behaves like a pass-through query and does *not* take advantage of OAE optimization. After the view has been created by an administrator, this query can be written by a user who does not understand AP SQL extensions.

If the query includes a WHERE clause, OAE does not push it to the EDW; instead the EDW returns all rows to Postgres, and the restrictions are applied from within Postgres.

For an example of the code in the two queries above, see [“Running Queries Against the EDW”, on page 199](#).

NOTE:

- Pass-through queries can take advantage of the standard ODBC/JDBC connection.
- A query can combine a reference to a Postgres table and to a pass-through query. Postgres query engine handles the necessary joins. It also joins an EDW table to an EDW table when two EDW tables are referenced in the join.

IMPORTANT: Only the OAE administrator can write a pass-through query. OAE will prohibit and grant access in queries based on permissions. Thus, all objects referred to in a query must have necessary Postgres permissions required to execute the query.

JOINS AND SUBQUERIES

All joins and most subqueries are implemented in Postgres rather than in the EDW. When you write a join query between an EDW table and a Postgres table, OAE sends a query to the EDW table to get all of the candidate rows and then joins the result to the Postgres table.

The examples in this section use the following tables:

- EDW table: `sls_t1(x1 int, y1 int, z1 int)`
- EDW table: `sls_t2(x2 int, y2 int, z2 int)`
- Postgres table: `pg_t3(x3 int, y3 int, z3 int)`

For the following join query:

```
SELECT x1, x3 FROM sls_t1, pg_t3 WHERE y1=y3 AND z1=0 and z3=0
```

OAE sends the following query to the EDW:


```
SELECT x1, y1 FROM sls_t1 WHERE z1=0
```

and then executes the join in Postgres.

If the query involves a join between two EDW tables, for example:

```
SELECT x1, x2 FROM sls_t1, sls_t2 WHERE y1=y2 AND z1=0 AND z2=0
```

then OAE sends the following two queries to the EDW:

```
SELECT x1, y1 FROM sls_t1 WHERE z1=0
SELECT x2, y2 FROM sls_t2 WHERE z2=0
```

and, again, the join is done in Postgres. The user should be aware of the sizes of the candidate result sets and how many rows will be sent over the network.

Subqueries are similar. Most subquery expressions fetch data from EDW and do the work in Postgres, but there are a few exceptions where OAE tries to optimize the query. An "IN" or expression subquery will perform the subquery first and then push the data to EDW. For example:

```
SELECT x1 FROM sls_t1 WHERE y1 = (SELECT max(y3) FROM pg_t3)
```

The expression subquery is executed in Postgres and the result is sent to the EDW as part of the query. For example: If `(select max(y3) from pg_t3)` returns the value 1000, then OAE sends the following query to the EDW:

```
SELECT x1 FROM sls_t1 WHERE y1 = 1000
```

For a query with an "IN" subquery, the following query,

```
SELECT x1 FROM sls_t1 WHERE y1 IN (select y3 FROM pg_t3)
```

evaluates the Postgres subquery and sends a list to the EDW. Suppose the column `y3` in the `pg_t3` table contains the values 10, 100, 1000, then OAE sends the following query to EDW:

```
SELECT x1 FROM sls_t1 WHERE y1 IN (10, 100, 1000)
```

Because EDW tables are expected to be much larger than Postgres tables, this technique will generally send less data over the network.

RUNNING QUERIES AGAINST THE EDW

The external-table views created during SenSage AP installation enable users to write standard queries in Postgres against the EDW tables. OAE translates the PSQL queries into AP SQL queries that the EDW understands.

This section documents the following topics:

- [“Timestamp Restriction and the DURING Clause”, next](#)
- [“Mapping Data Types”, on page 204](#)
- [“Additional Ways to Use EDW Extensions When Querying EDW Data”, on page 205](#)

TIMESTAMP RESTRICTION AND THE DURING CLAUSE

Although EDW is designed to access large amounts of data quickly, entire database searches still result in lengthy queries. To compensate, an EDW feature speeds up queries to perform faster searches within a restricted timestamp range. A special clause is used to implement the feature, the DURING clause, which reduces the problem space that EDW addresses. In the EDW, the DURING clause is required in all queries to restrict the range of timestamps that are searched.

Postgres does not have a DURING clause so OAE provides two methods to restrict searches by timestamp: inequalities on the timestamp column, and the `oae.during()` function call.

Every EDW table has a timestamp column named "ts". In a Postgres query, you can simply restrict the range of the "ts" column named in any external table using the following SQL inequality relations: BETWEEN, <, <=, >, and >=. The EDW will optimize the search space for faster queries when you use these inequalities in any combination with "ts" on one side and a timestamp constant on the other. For BETWEEN, it must be "ts" BETWEEN **x1** and **x2** *where* **x1** and **x2** are timestamp constants. Timestamp constants include timestamp literals and other expressions that can be evaluated to a timestamp by Postgres or the EDW before running the query. For example the query:

```
SELECT * FROM prod_windows.sumetable
WHERE ts BETWEEN timestamp'Jan 01, 2000' AND timestamp'Feb 01, 2000'
AND ts < timestamp'Jan 15, 2000'
```

will restrict the EDW search to rows that have a timestamp in the first 15 days of January, 2000. Even though the BETWEEN clause would accept dates through the end of January, the second inequality restricts the top end of the range further and each endpoint of the search is chosen to be the most restrictive possible.

Using `oae.during()`

The other method that OAE provides to restrict searches by timestamp is by way of the function `oae.during()`. This function comes in three forms:

```
oae.during(<text>)
```

```
oae.during(<timestamp>, <timestamp>)
```

```
oae.during(<timestamp_with_time_zone>, <timestamp_with_time_zone>)
```

The three forms of `oae.during()` are explained in the following sections, but they all have the following rules in common:

- First, `oae.during()` only applies to external tables; it does not effect Postgres tables or SLS tables in pass-through queries.
- Second, `oae.during()` can only occur in the WHERE clause and there can be only one in a given WHERE clause.
- Finally, a call to `oae.during()`, unlike an inequality on the timestamp column, can effect multiple tables as detailed below.

You can use the function `oae.during(t1, t2)` to restrict a query to only those rows between timestamps `t1` and `t2` inclusive in a manner similar to `ts` between `t1` and `t2` but not identical. The difference is:

Unlike an inequality on the `ts` column, `oae.during()` can effect multiple tables. It applies to every external table in the `FROM` clause, whether that table appears at the top level or down inside a view or a derived-table subquery, so long as it is not shadowed by another `oae.during()` call. The following examples show how this works.

EXAMPLES:

Example 1:

```
SELECT c FROM sls_tbl
WHERE oae.during(timestamp'1/1/08',timestamp'1/1/09');
```

this is equivalent to:

```
SELECT c FROM sls_tbl
WHERE ts BETWEEN timestamp'1/1/08',timestamp'1/1/09';
```

However, if `sls_tbl` is in a view that does not export a timestamp, then there is a difference:

```
CREATE VIEW v1 AS SELECT c, d FROM sls_tbl;
```

Example 2:

```
SELECT c FROM v1
WHERE oae.during(timestamp'1/1/08',timestamp'1/1/09');
```

this query will be the same as as the ones in Example 1, but:

```
SELECT c FROM sls_tbl
where ts between timestamp'1/1/08',timestamp'1/1/09';
```

will fail with an error message because there is no column `ts` (because `v1` does not return any columns other than `c` and `d`). In the first query of Example 2, the condition gets pushed down into the view so it does not matter that `v1` does not return a `ts` column.

```
CREATE VIEW v1 AS SELECT c, d FROM sls_tbl;
```

Example 3:

If there are multiple tables in the `FROM` clause, `oae.during()` can be pushed down into all of them:

```
SELECT * FROM sls_tbl, sls_tbl2, sls_tbl3
WHERE oae.during(timestamp'1/1/08',timestamp'1/1/09');
```

If you tried to write this with `BETWEEN`, then you would need a different `BETWEEN` clause for each table:

```
SELECT * FROM sls_tbl, sls_tbl2, sls_tbl3
WHERE sls_tbl.ts BETWEEN timestamp'1/1/08',timestamp'1/1/09'
AND sls_tbl2.ts BETWEEN timestamp'1/1/08',timestamp'1/1/09'
AND sls_tbl3.ts BETWEEN timestamp'1/1/08',timestamp'1/1/09'
```

The `oae.during()` is also pushed down into complex `FROM` clauses and derived tables. In the following query, the `oae.during()` gets applied to both `sls_tbl` (through `v1`) and `sls_tbl2`.

Example 4:

```
SELECT *
  FROM (SELECT * FROM v1 UNION SELECT * FROM sls_tbl2);
  WHERE oae.during(timestamp 1/1/08',timestamp'1/1/09');
```

However, if a view or derived table in the FROM clause already has an oae.during() call, that call takes precedence:

```
CREATE VIEW v2 AS SELECT c FROM sls_tbl WHERE oae.during('all');
```

Example 5:

```
SELECT * FROM v1, v2;
  WHERE oae.during(timestamp 1/1/08',timestamp'1/1/09');
```

In the Example 5 query, v1 is searched using oae.during(timestamp 1/1/08',timestamp'1/1/09') but v2 is searched using oae.during('all') because the lower call takes precedence. This happens no matter how many levels there are:

```
CREATE VIEW v3 AS SELECT * FROM v1;
CREATE VIEW v4 as SELECT * FROM v3
  WHERE oae.during('all');
CREATE VIEW v5 AS SELECT * FROM v4
  WHERE oae.during(timestamp'2/1/08', timestamp'2/1/09')
CREATE VIEW v6 AS SELECT * FROM v5;
```

Example 6:

```
SELECT * FROM v6;
  WHERE oae.during(timestamp 1/1/08',timestamp'1/1/09');
```

In the Example 6 query, sls_tbl is searched using oae.during('all') because that is the "closest" one. Another way to view this is that oae.during('all') is pushed down into the FROM clause, but it is never pushed past another oae.during().

Because this behavior is a bit complex and sometimes hard to visualize, it is recommended that you think of oae.during() as mostly a way to restrict searches rather than as a way to filter rows. In complex queries with multiple views or derived tables, we recommend using ts between t1 and t2 if you specifically want to filter rows. The BETWEEN clause will not get pushed down to more than one external table, but it does get passed on to the EDW to limit the search as well as filtering rows in an easy-to-predict way.

Specifying DURING ALL

The second form of the oae.during() function takes one text argument that it pastes literally into the DURING clause of the query sent to the EDW. Therefore, to specify "DURING ALL", you write oae.during('ALL'). For example:

```
SELECT * FROM prod_windows.sometable WHERE column1 < 0 and oae.during('all');
```

OAE translates the query above into the following EDW query:

```
SELECT * FROM windows.sometable WHERE column1 < 0 DURING ALL;
```

Specifying a Series of Time Periods

You can also use the text argument to specify a DURING clause that includes a series of time frames. The example below provides three alternate time periods:

```
SELECT * FROM prod_windows.sometable WHERE column1 < 0 and oae.during($$
[time('Jan 01 00:00:00 2005'),time('Mar 31 00:00:00 2005')] OR DURING [time('Jan
01 00:00:00 2006'),time('Mar 31 00:00:00 2006')] OR DURING [time('Jan 01 00:00:00
2007'),time('Mar 31 00:00:00 2007')]$$);
```

The query above uses \$\$ quoting to preclude the need to escape the single quotation marks inside the string. OAE expands this query into the following EDW query:

```
SELECT * FROM windows.sometable WHERE column1 < 0 DURING [time('Jan 01 00:00:00
2005'), time('Mar 31 00:00:00 2005')] OR DURING [time('Jan 01 00:00:00 2006'),
time('Mar 31 00:00:00 2006')] OR DURING [time('Jan 01 00:00:00 2007'), time('Mar
31 00:00:00 2007')];
```

For more information about a DURING clause that includes a series of time frames, see [Alternative Formats for DURING Clauses in Chapter 3, “SenSage AP SQL”](#).

Specifying the Postgres Timestamp Function with Explicit Time Zone

The third form of the oae.during() function takes two arguments that are both of type `TIMESTAMP WITH TIME ZONE`, which Postgres abbreviates as `timestamptz`. This function conveniently supports the use of time constants with explicit time zones as well as other expressions of type `TIMESTAMP WITH TIME ZONE`. For example, the function in the clause below includes `timestamptz` to specify a time in a specific time zone:

```
oae.during(timestamptz'2000-01-01 00:00:00-05', timestamptz'2000-02-01 00:00:00-
05')
```

If you call `oae.during()` with two text arguments, the function casts the two text arguments into `timestamp` with time zone. For example, Postgres interprets the call below similarly to the one above:

```
oae.during('2000-01-01 00:00:00-05', '2000-02-01 00:00:00-05')
```

IMPORTANT: However, because the text values that represent the time do not specify a time zone, the function below uses the local time zone:

```
oae.during('2000-01-01 00:00:00', '2000-02-01 00:00:00')
```

OAE interprets the clause above as:

```
oae.during(cast('2000-01-01 00:00:00' as timestamptz), cast('2000-02-01
00:00:00' as timestamptz))
```

This interpreted value is usually a different time value from the one below, which explicitly converts the text values to timestamps:

```
oae.during(cast('2000-01-01 00:00:00' as timestamp), cast('2000-02-01 00:00:00'
as timestamp))
```

The clause above converts to the GMT time zone instead of your local time zone.

Mapping Data Types

OAE supports only EDW data types for EDW tables. When you write a query that calls the `oae.sls_qry()` function, you must provide the types of the return columns in the Postgres version of the data type rather than the EDW version. The table below illustrates the mapping between data types.

EDW Type	Range	Postgres Type	Range
bool	TRUE/FALSE	bool boolean	TRUE/FALSE
int32	-2,147,483,648 to +2,147,483,647	int4 int integer	32-bit integer -2,147,483,648 to +2,147,483,647
int64	signed 64-bit integer -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	int8 bigint	signed 64-bit integer -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	64-bit floating-point -1.797693e+308 to 1.797693e+308	float8 float double precision	64-bit floating-point -1.797693e+308 to 1.797693e+308
timestamp	date-and-time-of-day values, expressed as the number of microseconds since January 1, 1970, GMT. The latest timestamp that can be represented falls in the year 2038.	timestamp timestamp without time zone	The period between 4713 BC and 5874897AD with a resolution of one microsecond
varchar	maximum length of a varchar value is 2147483646 bytes, which is two bytes fewer than two gigabytes	text	Approximately 100 bytes fewer than one gigabyte
inet	IP host address that carries the 32- or 128-bit of IP address information in "network order (same as Big-Endian) and optionally, identifies the subnet where the host resides, all in a single field. If the subnet mask is 32 (IPv4) or 128 (IPv6), then the result is host only. If an empty string was converted to inet then the result is an empty string, and if a non-empty string that did not contain a valid IPv4/IPv6 address was converted to an inet, then the result is invalid.	inet	The inet type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. It's 7 or 19 bytes.

OAE automatically maps the data types for external tables.

Additional Ways to Use EDW Extensions When Querying EDW Data

A user who knows AP SQL can take advantage of the full power of the EDW extensions when querying the EDW from Postgres. OAE provides three ways to include AP SQL extensions by using one of the following types of queries:

- **Query 1**--A pass-through query, which queries the EDW directly.
- **Query 2**--A standard query that queries a Postgres view written as a pass-through query

For an overview of query types 1, 2 refer to [“Overview: Pass-Through Queries”, on page 197](#).

Writing a Pass-Through Query

For Query 1 or Query 2 to include AP SQL extensions, the query or the view must be a pass-through query that calls the `oae.sls_qry()` function.

Pass-through queries use the Postgres `oae.sls_qry()` table function, which returns a set of records.

You can use the `oae.sls_qry()` function in place of a table in the FROM clause. Therefore, Postgres allows other tables in the FROM clause if you want to join to the results of an EDW query or simulate an EDW subquery.

Typically, a pass-through call uses a format like the one below:

```
SELECT <expression_list> FROM oae.sls_qry($$, '<connection_ID>' $$
    <sls_query>)
    AS <derived_table_name>(<column_declarations>);
```

where:

- `<expression_list>` is a list of expressions that can reference column names in the `<column_declarations>`.
- `<connection_ID>` specifies the connection between Postgres and the EDW instance.
- `<derived_table_name>` specifies a name for the table. Typically you specify the same name as the EDW table but you can use any legal Postgres name that is not currently used or leave this argument empty.
- `<column_declarations>` specifies a list of `<column_name>` `<column_type>` pairs.
 - Each `<column_name>` specifies the name of a column declared in the EDW query.
 - Column names must be identical to the column names declared in the EDW query and must be in the same order as the columns declared in the EDW query.
 - Each `<column_type>` specifies the Postgres data type of the column.

NOTE:

- The column names returned by the EDW must be the same as the column names in the Postgres alias list. If you are not sure what column names the EDW will return, use column aliases in the EDW query. For example, if the SELECT statement uses an expression in the SELECT list, specify a column alias for the expression.

- If an EDW column name begins with an underscore (`_`), the EDW strips off the underscore in the result set. Use a column alias to retain the underscore. For example:

```
SELECT * FROM oae.sls_gry('SELECT ts, _uploadid AS _uploadid FROM
default.syslog DURING ALL', 'MySLSinstance') AS (ts timestamp, _uploadid text);
```

- If a column name alias contains upper case letters, the column name must be enclosed in double quotation marks. For example:

```
SELECT * FROM oae.sls_gry('SELECT ts, _uploadid AS Upid FROM default.syslog
DURING ALL', 'MySLSinstance') AS (ts timestamp, "Upid" text);
```

Query 1: Example Pass-Through Query

The following statement illustrates a pass-through query written directly against the EDW Weblog table:

```
SELECT * from oae.sls_gry($$
  SELECT ClientDNS, count(*)
  FROM Pubs.Weblog
  GROUP BY 1
  SLICE BY _int32(ts) - _fifo(ClientDNS, _int32(ts), _int32(ts)) > 10
  DURING ALL;$$, 'MySLSinstance') as
  ("ClientDNS" text, count int);
```

NOTE:

- The SLICE BY clause is used with the GROUP BY clause to separate records that would normally be aggregated in a single group into multiple groups. The SLICE BY clause is commonly used with the `_fifo()` function to split a group of records based on the previous value of an expression. For more information on these SQL extensions, see [Chapter 3: SenSage AP SQL](#).
- Because ClientDNS includes uppercase letters, it must be enclosed within double quotation marks when it appears in the column-declaration section. Without the quotation marks, OAE would look for a column named clientdns.
- The query above encloses MySLSInstance within single quotation marks and ClientDNS within double quotation marks. This is because SQL delimits string literals by single quotation marks and object identifiers by double quotation marks.

Delimiting Quotation Marks

The `$$ <select_string> $$` notation is a Postgres string-quoting syntax that is convenient when the string contains quotation marks or is multi-line. For more information, see section 4.1.2.2 Dollar-Quoted String Constants in the Postgres documentation:

<http://www.postgresql.org/docs/8.3/interactive/sql-syntax-lexical.html#SQL-SYNTAX-DOLLAR-QUOTING>

The example query above uses the `$$` delimiter because the query surrounds the date values within quotation marks. If the quoted string itself contains two consecutive dollar signs (`$$`), you can identify the quotation delimiter by including an arbitrary string between each set of dollar

signs. In this case, the delimiter syntax would be: `$<arbitrary_string>$ <select_string>`
`$<arbitrary_string>$`. For example:

```
SELECT <expression_list> FROM oae.sls_qry($foo$
  SELECT <column_declarations> FROM <sls_namespace>.<sls_table>
  DURING '<start_time>', '<end_time>' $foo$, '<connection_ID>')
  AS <derived_table_name>(<column_declarations>)
  WHERE <where_clause>;
```

Query 2: Example Standard Query Against a Postgres Pass-Through View

The following statement illustrates the manually created Postgres view that uses the AP SQL_fifo function and SLICE BY clause and runs directly against the SLS Weblog table:

```
CREATE VIEW PGWeblogSliceBy AS SELECT * FROM oae.sls_qry($$
  SELECT ClientDNS, count(*)
  FROM Pubs.Weblog
  GROUP BY 1
  SLICE BY _int32(ts) - _fifo(ClientDNS, _int32(ts), _int32(ts)) > 10
  DURING ALL;$$, 'MySLSinstance') as
  ("ClientDNS" text, count int);
```

The view created above enables the same query illustrated in [“Query 1: Example Pass-Through Query”, on page 206](#). However, after the administrator creates the view, an end user who knows nothing about AP SQL extensions can query the EDW with these extensions. The user writes a standard EDW query against the view, like one of the queries below:

```
SELECT * FROM PGWeblogSliceBy;
```

NOTE: When you create a Postgres view that includes a pass-through query to the EDW, Postgres does not validate the query until the view is used. In other words, even if the view contains an error, such as referencing an invalid EDW table or column or an invalid EDW instance, view creation completes successfully. It is only when a query runs against the view that Postgres displays an error.

TRANSACTION ISOLATION LEVELS

Postgres supports the ANSI transaction isolation levels. However, because the EDW does not have cursors or transactions, it does not support these isolation levels. Therefore, Postgres does not enforce transaction isolation levels when a query accesses data in external tables.

Postgres supports setting isolation levels at the transaction level rather than the statement level. The table below illustrates the levels that Postgres supports.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

When a query involves only EDW table(s), Postgres applies none of the isolation levels and end users get the effect of Read Committed isolation levels. When a query involves a Postgres table and EDW table, Postgres applies isolation levels only to the Postgres table while the EDW table(s) get Read Committed behavior.

RUNNING DDL AND DML COMMANDS

OAE does not enable a Postgres user to create or modify EDW tables or to insert, update, or delete data in EDW tables. In other words, OAE supports no usage of Data Definition Language (DDL) and the only Data Manipulation Language (DML) command it supports on EDW tables is SELECT.

SETTING UP OAE FOR EDW MULTI-CLUSTER ACCESS

SenSage AP supports a special configuration setup for a EDW multi-cluster environment distributed data among different cluster nodes. Through OAE it is possible to access data spread across multiple EDW clusters. To do this, you have to configure OAE so that all the EDW clusters in your network are accessible and can be queried.

NOTE: While planning a cluster federation, decide on the complete set of clusters belonging to the federation. Note that this procedure is for a static, one-time federation set-up only.

IMPORTANT: EDW cluster names have to be unique for this setup to work.

Setup Instructions:

- 1 Set up authentication and access for each EDW cluster that resides on the OAE machine.
- 2 Create a directory for each cluster:

```
mkdir -p <HawkEye_AP_prefix>/etc/sls/instance/<cluster_name>
```

Example: `mkdir -p /opt/hexis/hawkeye-ap/etc/sls/instance/edw_cluster`

- 3 Copy the shared-secret key of the EDW cluster:

```
scp <EDW_host_name>:<HawkEye_AP_prefix>/etc/sls/instance/<EDW_cluster_name>/  
shared_secret.asc <HawkEye_AP_prefix>/etc/sls/instance/<EDW_cluster_name>/
```

Example: `scp edw_host:/opt/hexis/hawkeye-ap/etc/sls/instance/edw_cluster/
shared_secret.asc /opt/hexis/hawkeye-ap/etc/sls/instance/edw_cluster/`

- 4 Change the file ownership (only if commands above were not run as the SenSage AP user):

```
chown -R <hawkeye_AP_user>:<hawkeye_AP_group> <HawkEye_AP_prefix>/etc/sls/  
instance/
```

Example:

```
chown -R hexis:hexis /opt/hexis/hawkeye-ap/etc/sls/
```

- 5 Create the external schema of each EDW cluster by running this command:

```
/opt/hexis/hawkeye-ap/bin/psql -U <PG_user> -d controller -c "SELECT
oea.create_external_sls('<EDW_cluster_name>', '<EDW_host_name>:<port>', '<OAE_us
er>');"

```

Example:

```
SELECT oae.create_external_sls('edw_cluster','edw_host:8072','oea');
```

6 Configure automatic schema updates from the EDW cluster:

a On each EDW host in the EDW cluster, edit the OpenSLS section of `/var/lib/ambari-agent/puppet/modules/hdp-sensage/templates/athttpd.conf.erb` EDW configuration file to point to OAE host.

b `DDLNotification=1`

c `PGConnectionString=<OAE_host>:5432`

7 Restart the EDW cluster from the Deployment Manager. You will find a restart button on the summary page.

8 If you find the OAE external schema and EDW schema are out of sync, then to get the latest EDW schema updates, run the following command:

```
/opt/hexis/hawkeye-ap/bin/psql -U <PG_user> -d controller -c "SELECT
oea.force_sync(<EDW_cluster_name>)"

```

NOTE: This function will get latest updates to the EDW database.

9 To create group tables for your EDW for multi-cluster access, see the following section below, *Group Tables and EDW Multi-Cluster Support*.

GROUP TABLES AND EDW MULTI-CLUSTER SUPPORT

SenSage AP supports a special deployment for extra-large applications in a EDW multi-cluster environment distributing data among many nodes. Through a single external OAE table, known as the group table, it is possible to reference data distributed among multiple EDW clusters. To do this, you write queries against a group table just like any other external table; OAE will get the data from all of the clusters and join that data together to answer the queries correctly.

Creating and Using Group Tables

A group table can only be created in a group schema. The group schema is used to define the set of clusters that the group tables will be distributed among. In other words, every group table in a particular group schema is distributed among the same group of EDW clusters. You create a group schema with the function:

```
oea.create_group (<schema-name>,<list-of-clusters>)
```

where *<schema-name>* is a string representing a new schema (that is, the schema must not exist and it must be in lowercase characters) and *<list-of-clusters>* is a string of comma-separated names. Each name is the name of an OAE external database.

Each table in the schema is distributed across all of the EDW clusters that were named in the `oae.create_group()` function call. Because creating a group schema does not create the actual tables, you have to configure the tables themselves on all EDW clusters in the group, with the same namespace and schema.

Once you have an identical table on all EDW clusters in the group schema, you can create a group table with the function:

```
oae.create_group_table <group-schema>, <EDW-namespace>, <table-name>)
```

This function adds `<table-name>` as a new group table in the group schema named `<group-schema>`.

When a group table appears in a query it is optimized like other queries, but the EDW query is sent in parallel to all clusters in the group, with the results sent to Postgres. In a query with aggregate function or a GROUP BY where the FROM clause contains only one group table and no joins, the optimized query causes each EDW to do a local aggregation and GROUP BY, then do a local aggregation and GROUP BY to merge the results. This can only be done with the regular standard functions: max, min, sum avg and count.

Pass-through queries also work on group tables:

```
oae.sls_qry (<connection>. <edw-query>)
```

If the `<connection>` parameter is a group name then the EDW query is sent to all clusters in the group and the results are merged as if with UNION ALL.

IMPORTANT: In 6.1.0, the group name for the `<connection>` parameter must be in lowercase characters for federated cluster environments.

Once created, the group schema and group tables are permanently available to all users with the appropriate permissions. To remove these schemas and tables, execute the following:

```
oae.drop_group(<group-schema>)
```

USING OAE OVER ODBC TO RETURN DATA TO EXCEL: AN EXAMPLE

This section provides an example that illustrates how you can use ODBC and the Postgres database query processing interface to access SenSage EDW data from Microsoft® Office Excel 2003.

This section includes the following topics:

- [“Verify psqLODBC is Installed on Your System or Download & Install It”, next](#)
- [“Create a DSN and Point it to the Server that Runs OAE”, on page 211](#)
- [“Import Data Into Microsoft Office Excel from OAE EDW”, on page 213](#)

Verify psqLODBC is Installed on Your System or Download & Install It

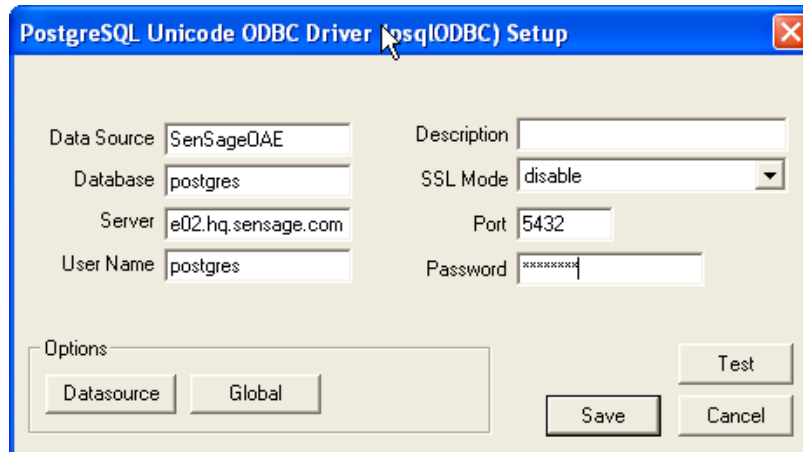
To take advantage of OAE, third-party tools use Postgres as their database and use ODBC and JDBC drivers to connect to the Postgres database. For details, see See “Obtaining Drivers and Client Software” on page 185.

Create a DSN and Point it to the Server that Runs OAE

To use ODBC to connect to the EDW, you must create a DSN (Data Source Name) that points to the server on which OAE is installed. This section illustrates how to configure a User DSN that connects applications on your computer to the Postgres database. After you configure the DSN, you can return specific data from the EDW to your applications.

When you configure the DSN from the Data Sources (ODBC) tool accessed from Administrative Tools on a Microsoft Windows XP system, select a Postgres SQL driver from the list of drivers. This example uses the PostgreSQL Unicode driver.

After you select the driver for which you are configuring the data source, the Setup dialog for the driver displays. [Figure 6-2](#) illustrates driver configuration on Windows XP.

Figure 6-2: Configuring the Driver

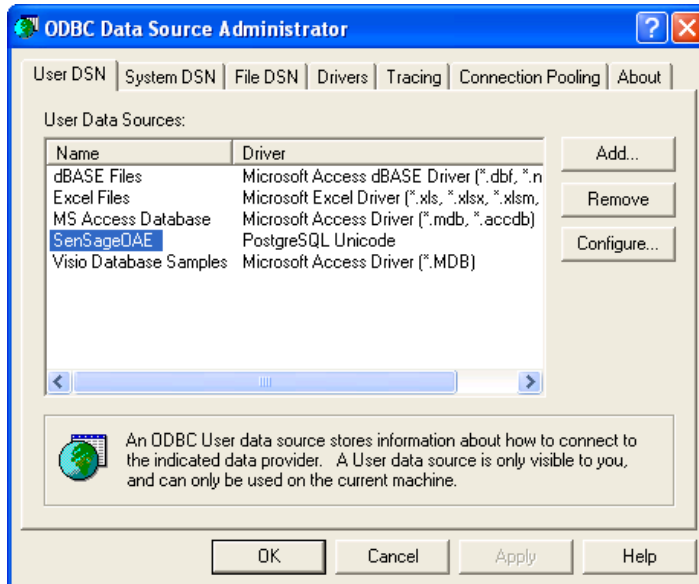
As illustrated above, specify values for the following fields:

- **Data Source** —name the DSN
- **Database**—specify the name of your Postgres database (defaults to postgres)
- **Server** —identify the server on which OAE is installed; this should be the EDW node you want to query
- **User Name**—specify the name of your Postgres user; ask your system administrator for the Postgres login and password
- **Password**—specify the name of your Postgres user's password

Click **Test** to ensure that your connection works as expected. If it does, click **Save**.

After you save the configuration, your new DSN will display among your existing DSNs. [Figure 6-3](#) illustrates this list on Windows XP.

Figure 6-3: List of Data Sources



Import Data Into Microsoft Office Excel from OAE EDW

You are now ready to import data directly into Microsoft Office Excel from the EDW. The procedure below describes the steps.

To import EDW data into Microsoft Office Excel

- 1 Open Microsoft Office Excel.
- 2 From the **Data** menu, select **Import External Data > New Database Query....**

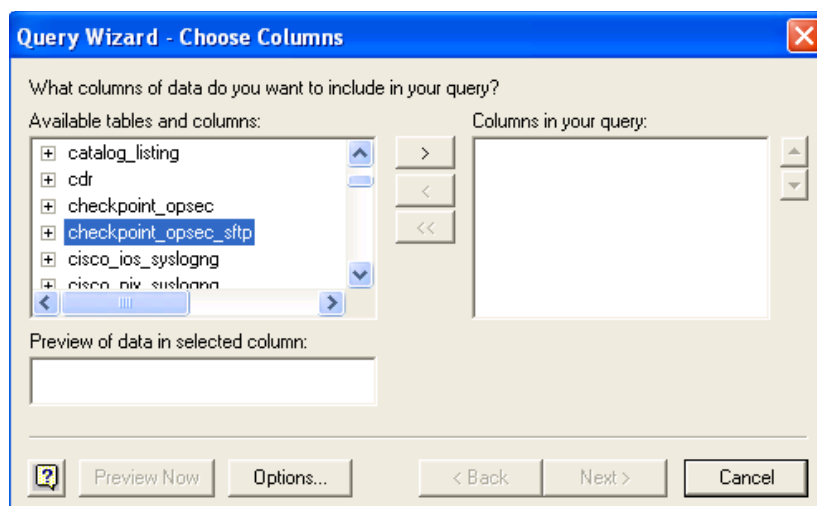
The Choose Data Source dialog displays.

- 3 From the Databases tab, select the DSN you just created that connects to Postgres, and click **OK**.

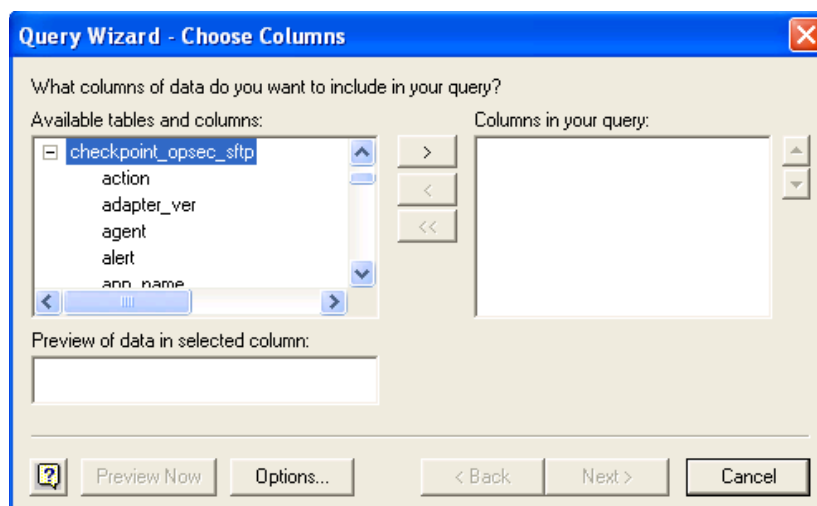
The Query Wizard - Choose Columns dialog displays.

- 4 Locate and select the desired EDW table in the list of **Available tables and columns** field.

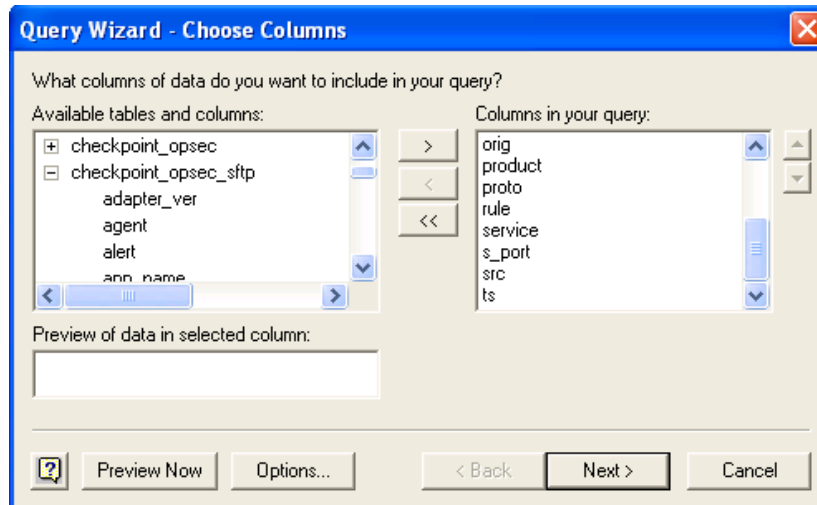
This example uses the `checkpoint_opsec_sftp` table. The graphic below illustrates this dialog for Microsoft® Office Excel 2003.



5 Expand the table to display its columns, as illustrated below.



6 Select each desired column and move it from the **Available tables and columns** field to the **Columns in your query** field, as illustrated below.



7 Click Next.

The Query Wizard - Filter Data dialog displays.

8 If desired, use this dialog to build a WHERE clause, and click Next.

The Query Wizard - Sort Order dialog displays.

9 If desired, use this dialog to specify sort order for the returned data, and click Next.

The Query Wizard - Finish dialog displays.

10 Keep the default setting (Return Data to Microsoft Office Excel), and click Finish.

The Import Data dialog displays with the cursor blinking in one cell.

11 If the cursor is in the desired cell, keep the default option (Existing worksheet) and click OK. If not, move the cursor to the desired cell before you click OK.

Each column of the EDW data displays in separate columns in Excel, as illustrated below.

Microsoft Excel - Book1

File Edit View Insert Format Tools Data Window Help Adobe PDF

SnagIt Window

A70 accept

	A	B	C	D	E	F	G	H	I	J	K
1	action	app_vendor	bytes	dst	elapsed	filename	if_dir	orig	product	proto	rule
2	accept	Checkpoint	901	63.150.183.38	0:05:30	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	33
3	accept	Checkpoint	915	63.150.183.38	0:05:30	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	33
4	accept	Checkpoint	903	63.150.183.38	0:05:30	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	33
5	accept	Checkpoint	1295	155.199.128.244	0:02:21	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	33
6	accept	Checkpoint	8470	206.242.150.107	0:00:32	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
7	accept	Checkpoint	11018	206.242.150.106	0:00:28	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
8	accept	Checkpoint	52798	206.242.150.107	0:00:19	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
9	accept	Checkpoint	2009	206.242.150.106	0:00:22	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
10	accept	Checkpoint	2324	206.242.150.106	0:00:14	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
11	accept	Checkpoint	8784	206.242.150.107	0:00:15	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
12	accept	Checkpoint	47797	206.242.150.107	0:00:12	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
13	accept	Checkpoint	15164	206.242.150.106	0:00:08	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
14	accept	Checkpoint	7725	206.242.150.106	0:00:10	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
15	accept	Checkpoint	3123	206.242.150.107	0:00:05	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
16	accept	Checkpoint	10222	206.242.150.106	0:00:07	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
17	accept	Checkpoint	30554	206.242.150.107	0:00:07	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82
18	accept	Checkpoint	3478	206.242.150.106	0:00:06	fw.log	inbound	206.242.150.67	VPN-1 & FireWall-1	tcp	82

Running and Using SenSage AP on VMware

This chapter discusses recommended configurations for running a SenSage AP instance under VMware and contains the following sections:

- “Overview”, next
- “Hardware Configuration for ESX Server”, on page 219
- “Virtual Machine Configuration”, on page 220

OVERVIEW

You can run a SenSage AP instance under VMware using either the ESX or ESXi products. These products provide a "bare-metal" implementation of VMware that you install directly on server hardware and do not require a "host" operating system.

See <http://www.vmware.com/products/esxi/> for more information on the ESX and ESXi products from VMware.

Planning a deployment using VMware requires careful consideration of how system and network resources are utilized. There are many ways you can configure a SenSage AP instance to run under VMware. This document provides some general recommendations for hardware and virtual machine configurations that you can use to help plan your deployment.

IMPORTANT: If you intend to use VMWare, carefully craft your VM deployment to realize the full performance of the SenSage AP product:

- Be sure to maximize the availability of the server's hardware resources to the VMs running the SenSage AP product. Overbooking resources, throttling VS, and employing VM migration technologies will typically reduce performance.
- While you can use VMs to run the SenSage AP product, most of the standard benefits of using virtualization (overbooking of hardware resources, load balancing, etc) do not apply.
 - When running the EDW (which is a resource intensive, distributed application) it is best that each node has full access to all of a server's resources and that disk I/O is decentralized to maximize throughput.
- Keep in mind that virtualized environments typically restrict a VMs access to the physical resources of the server and typically centralize the disk I/O.

SUPPORT FOR ADDITIONAL FUNCTIONALITY OF VIRTUALIZATION PRODUCTS

The following is outside the scope of the SenSage AP certification process:

- Virtualization functionality that operates without the knowledge or cooperation of the operating system or applications executing within the virtual machine such as but not limited to:
 - live migration of virtual machine clusters
 - memory ballooning
 - virtual machine fault tolerance

There are no industry standards to follow in implementing such features. By design, the operating system or application is not required to be cognizant of such functionality, which is part of the virtualization product. Since the virtualization product operates independently of the operating system or application, there is no practical method of testing its product features.

For this reason, the certification process does not test these features or functions and leaves the testing and support for them the sole responsibility of the product virtualization vendor. Note that any certification the virtualization vendor has performed in regard to these features or functions is outside the scope of the SenSage AP certification process.

If you are running a supported version of Red Hat Server on VMware and experience issues after using third party virtualization features (such as live migration) that operate independently of Hexis products, then contact VMware support for assistance in resolving the issue.

Virtual machine snapshots that do not use VSS volume snapshots are not supported by SenSage AP. Any snapshot technology that does a behind-the-scenes save of a VM's point-in-time memory, disk, and device state without interacting with applications on the guest using VSS may leave SenSage AP in an inconsistent state.

SENSAGE AP VIRTUALIZED ARCHITECTURE CONSIDERATIONS

Because the EDW is a distributed, resource-intensive application, when a task is passed to the EDW (that is, a query, a load, etc.) *every* instance of the EDW simultaneously becomes busy. A critical architecture consideration is to avoid overload of any other component of this physical infrastructure when all EDW nodes go active simultaneously.

The following are the principle issues to consider when creating the VM architecture:

- **CPU usage**—do not put more cores of EDW instances on a physical server than there are available physical cores. Overbooking of CPU resources will result in degraded EDW performance.
- **Network usage**—plan appropriately the allocation of EDW network traffic over physical NICs.
- **VMKernel NIC**—be aware of how much network traffic can be generated over the VMKernel NIC by the EDW nodes and plan appropriately.
- **Disk usage**—the allocation of virtual disks to VMs running the EDW needs to be able to accommodate the aggregate throughput generated when all the EDW nodes go active simultaneously.

Remember when planning your VM deployment that when EDW becomes active, every VM running an instance of the EDW becomes simultaneously active and will want to consume as much of the system resources as possible. This is an atypical deployment scenario for most VM architectures and requires thorough planning.

VMWARE MIGRATION

SenSage AP does not completely support "Fully Automated" Virtual Machine migration. This means you must configure all *guest* VMs (that are part of a SenSage AP deployment) for manual migration at all times.

Manual movement of guest VMs is especially critical in a disaster recovery scenario; SenSage AP has successfully tested the VMware Distributed Resource Scheduler (DRS) feature in "Manual" mode and the VMware High Availability (HA) turned ON for guest VMs participating in the cluster. In this configuration, if an ESX host becomes inoperable, the guest VMs may be manually migrated to another ESX server for temporary use as part of disaster recovery.

VMWARE AND LINUX VERSIONS

IgniteTech used the following versions of VMware when testing the recommendations included in this chapter:

- 64-bit VMWare ESX (or ESXi) 4.0 and above

IgniteTech used the following versions of Linux as the guest operating system when testing the recommendations included in this chapter:

- RedHat 5.5, 64-bit and RedHat 5.7, 64-bit

The VMware Tools application was installed on all guest operating systems. See [“Install VMware Tools”, on page 223](#).

Hardware Configuration for ESX Server

SenSage AP recommends the hardware configurations described in this section for servers running ESX or ESXi.

ESX Server CPU

- Four CPU cores for each virtual machine
- Faster CPUs provide better performance.
- Minimum 2.5Ghz is recommended; 3Ghz is preferred.

ESX Server RAM

- Minimum physical RAM should be equal to the RAM allocated for each virtual machine plus 1 GB for ESX overhead.
- Additional RAM generally improves performance.
- IgniteTech recommends 4GB per core.

Network Interface Card (NIC)

- One physical Gigabit Ethernet NIC for each virtual machine.
- Clusters containing more than 20 EDW nodes may require additional physical Gigabit Ethernet NICs for each virtual machine.

Storage

Use a Storage Area Network (SAN) to host virtual drives. Each virtual machine running should have access to its own LUN (logical unit numbers identify storage volumes in the SAN).

- Aggregate bandwidth from the physical ESX server to the SAN server should be at least 100MB/second multiplied by the number of virtual machines.
- The number of LUNs available to the physical ESX server should be equal to the number of virtual machines.
- The amount of available storage required for each LUN depends on customer data retention requirements.

VIRTUAL MACHINE CONFIGURATION

This section describes recommend virtual machine configurations and includes the following sections:

- [“Deploying EDW and non-EDW components”, next](#)
- [“Recommended Virtual Machine Configuration for SenSage AP Hosts”, on page 221](#)
- [“Reserving CPU Resources for the ESX Server to Improve Performance”, on page 221](#)
- [“Install VMware Tools”, on page 223](#)

Deploying EDW and non-EDW components

IgniteTech recommends that you deploy each EDW node on a virtual machine configured as described in the table below ([“Recommended Virtual Machine Configuration for SenSage AP Hosts”, on page 221](#)).

IgniteTech advises you to deploy the following non-EDW components on virtual machines that are *NOT* running the EDW. ([“Recommended Virtual Machine Configuration for SenSage AP Hosts”, on page 221](#)).

- LDAP
- Application Manager
- Postgres
- Collector
- Parser

NOTE: Disk space requirements for these components may vary based on the amount of cached data retained by SenSage AP, the frequency of data collection, and other factors.

RECOMMENDED VIRTUAL MACHINE CONFIGURATION FOR SENSAGE AP HOSTS

Hexis Cyber Solutions used the following versions of VMware when testing the recommendations included in this chapter:

System	Recommended configuration per virtual machine	Comments
Number of virtual CPUs	4 virtual processors.	Faster CPUs provide better performance. Use of resource pools to ensure sufficient hypervisor performance is recommended. See the next section below for details on resource pools.
Virtual RAM	Minimum 2GB per virtual processor. SenSage AP recommends 4GB of RAM per virtual processor. (8GB minimum, 16 GB recommended)	Additional RAM generally improves performance.
Virtual disk drive	Use raw device map.ping (RDM) to map the virtual machine's virtual disk to a LUN in a SAN device.	The amount of storage available for each LUN depends on customer data retention requirements. Each LUN should be dedicated to a specific virtual machine.
Network Interface Card	Each virtual machine should map one virtual Gigabit Ethernet card to one physical Gigabit Ethernet NIC.	Clusters having more than 20 nodes may require additional NICs and more complex network configurations to handle the intra-cluster network traffic.

IMPORTANT: Do not over commit physical resources. When you run additional virtual machines on the same physical host with virtual machines running SenSage AP hosts, make sure that there are sufficient physical resources available to support the additional virtual machines.

Reserving CPU Resources for the ESX Server to Improve Performance

The VMWare ESX Hypervisor manages the user/allocation of the physical resources to VMs and other logics related to the operation of the virtualization platform. If the Hypervisor does not have sufficient server resources to meet processing demands (by the VMs various forms of atypical behavior and/or performance), this can degrade its own performance.

For example and in particular, operation of the virtual network switches can start to degrade to the point that they are overwhelmed and return ICMP host "unreachable" packets in response to incoming network packets. The end result of this behavior is that applications running inside the VMs start to experience random networking faults related to the issue "no route to host."

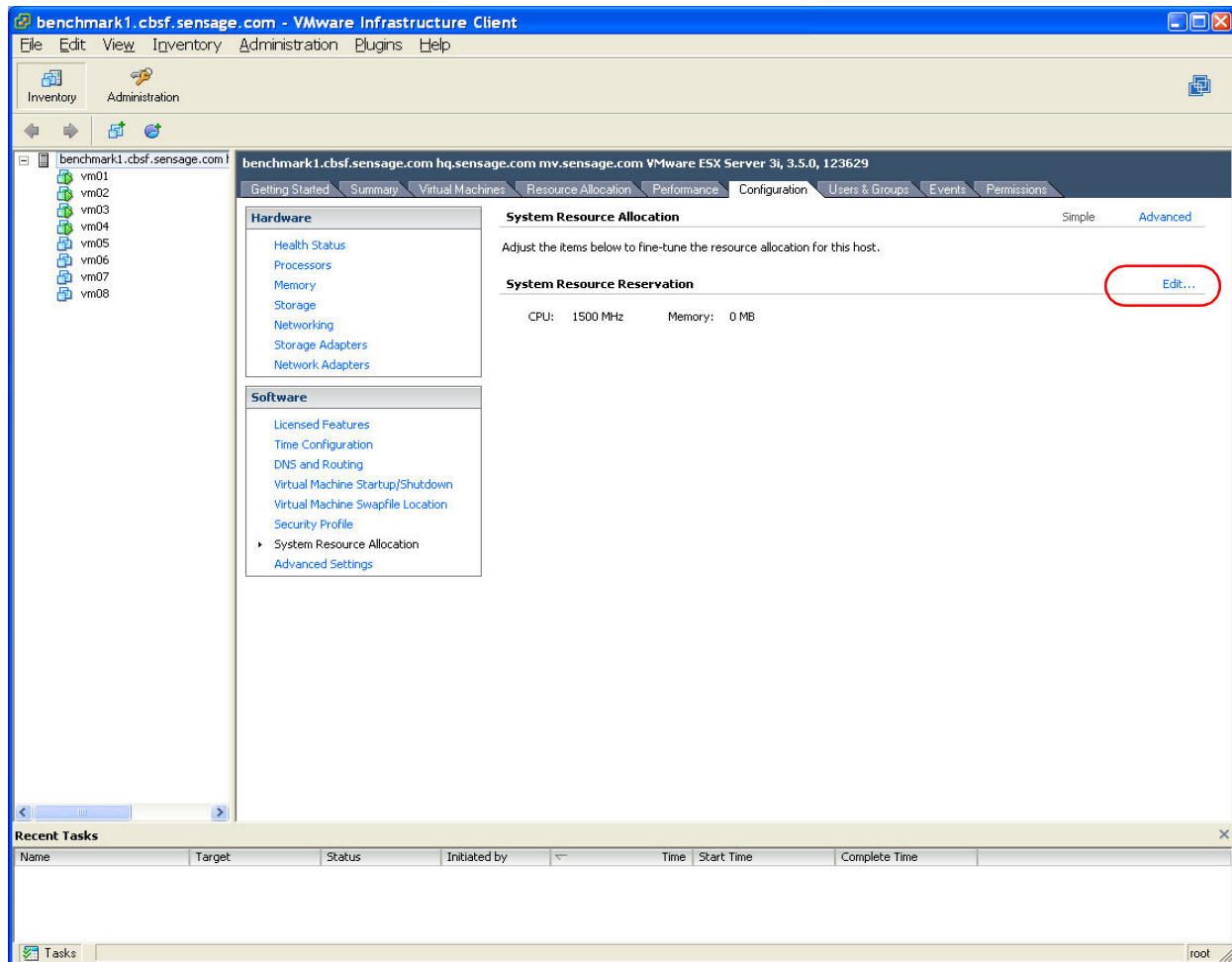
The easiest way to ensure that the ESX Hypervisor has sufficient resources to keep pace with the demands from the operating VMs is to configure the 'System Resource Pool' to ensure that a sufficient number of CPU cycles are reserved for the operation of the Hypervisor.

To reserve CPU resources for the ESX host, perform the following steps on each ESX host in your SenSage AP deployment:

- 1 Open the ESX infrastructure management tool.
- 2 Select the ESX server you want to configure from the list in the left panel.
- 3 Open the **Configuration** tab.
- 4 In the **Software** section, click the **System Resource Allocation** link.

System resource allocation and reservation information displays, as shown in [Figure 7-1](#).

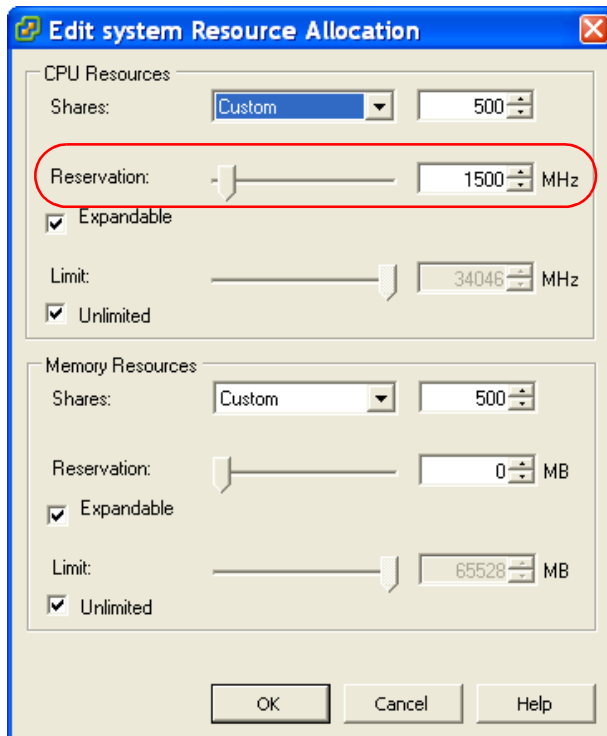
Figure 7-1: Resource Allocation



- 5 Click the **Edit...** link (outlined in red, in [Figure 7-1](#)) located to the right of the **System Resource Reservation** line.

The **Edit System Resource Allocation Properties** window displays.

Figure 7-2: Edit System Resource Allocation Properties



6 In the **CPU Resources** section, set the value for **Reservation** to the number of CPU cycles you want to reserve for the ESX server. IgniteTech recommends you set this value to 1600 Mhz, but some experimentation may be required to determine the correct value. If EDW operations in your SenSage AP cluster sometimes fail with "no route to host" errors that cannot be explained by networking failures, increase the **Reservation** value.

7 Click **OK**.

Install VMware Tools

IgniteTech recommends that you install VMware tools on each instance of the guest operating system (Linux). Installing VMware tools improves performance and clock accuracy. You can install VMware tools from the VMware Infrastructure Client or the vSphere Client.

Perl Subroutines

SenSage AP SQL can perform simple transformation and analysis of event-log data, but it has limitations in performing complex transformations, also known as *business logic*. The SenSage AP SQL engine includes a Perl subsystem that lets you declare and use Perl code in your SenSage AP SQL statements.

This chapter includes these sections:

- “About Perl Subroutines in SenSage AP SQL”, next
- “Declaring Perl Functions”, on page 226
- “Declaring Perl Aggregates”, on page 227
- “Perl Execution Environment”, on page 229
- “Accessing External Modules”, on page 232
- “Testing and Debugging Perl Subroutines”, on page 233
- “Installing Perl Modules”, on page 235

ABOUT PERL SUBROUTINES IN SENSAGE AP SQL

Perl subroutines can be declared as either *functions* or *aggregates*. Perl Functions are passed arguments for each row being processed, and they return a single value for each row. For example, `_strformat()` is an SQL function. You pass in arguments and it returns a string. With Perl functions, you can write your own routines like `_strformat()`.

Perl aggregates behave like SQL aggregation functions, such as `sum()` and `avg()`. Perl aggregates are passed arguments for each group created by a `GROUP BY` clause, and they return one result per group. Perl aggregates are useful for analyzing multiple records, such as when performing session analysis. For example, you might declare a Perl aggregate to detect suspicious user behavior. Use a `GROUP BY` clause to create a group for each user, with subgroups for each session. Then apply your Perl aggregate to match against suspicious patterns.

How Perl Processing Works

The basic purpose of the SQL engine is to consume streams of columns from a stream parser and send streams of columns to a stream formatter. Without displaying its processing to the user, the SQL engine rewrites calls to Perl functions to call a built-in function named `_perl()`. This function invokes the Perl interpreter once per column value.

For each query, the Perl engine is initialized, which compiles the Perl code and checks for errors. Assuming there are no errors, the Perl code is evaluated, registering the Perl functions and initializing any global variables.

For a Perl function, the SQL engine performs these steps

- 1 The SQL engine evaluates each parameter to the Perl function (per row).
- 2 For each argument, the value is converted into a `varchar` if necessary.
- 3 The `_perl()` SQL function is called with these arguments.

- 4 The result of the `_perl()` expression is a column of `varchar` values corresponding to the values returned by the Perl function.

For a Perl aggregate, the SQL engine performs these steps

- 1 The SQL engine saves up argument-records for each unique group (buffering).
- 2 When enough records have been buffered, `_perlagg()` is called with these arguments, similar to `_perl()`, including the conversion to `varchar` when necessary.
- 3 The `_perlagg()` SQL function loops through each argument record and invokes the Perl aggregate.
- 4 After all values for a group have been processed (not just the last value for this one buffer of records), the routine whose name is derived by adding `_final` to the name specified by the first argument to the `_perlagg()` expression is invoked.
- 5 The result of the `_perlagg()` expression is a column of `varchar` values that correspond to the values returned by the final Perl subroutine.

DECLARING PERL FUNCTIONS

You declare Perl functions as processing directives in a `WITH` clause. For example, the following SQL `Select` statement declares a Perl function named `maxarg()`, which returns the larger of two values.

```
-- declare a Perl function
WITH maxarg AS BUILTIN 'perl5' FUNCTION <<EOF
  sub maxarg {
    my($x,$y) = @_;

    if ($x > $y) {
      return $x;
    }

    return $y;
  }
EOF

-- use the Perl function in the query
SELECT maxarg( _strlen(Url) , _strlen(Referrer) ),
         _strlen(Url),
         _strlen(Referrer)
FROM example_webserv_100
DURING ALL;
```

Because the Perl declaration spans lines, it is bounded by here document syntax `<<EOF . . . EOF`. Here documents tell the SQL parser to treat the contents as a single `varchar` literal, including the newline characters.

For more information on declaring Perl functions, see [“User-Defined Subroutines”, on page 82](#).

DECLARING PERL AGGREGATES

You declare Perl aggregates as processing directives in a `WITH` clause. Within a Perl aggregate declaration, you specify a pair of subroutines and global variables that the two subroutines can access. One subroutine manages the incremental state of groups in an aggregate, and the other subroutine returns the final aggregate value for a group. The second subroutine, which returns the final value, has the same name as the first, but with a suffix of `_final`. The SQL query engine calls each subroutine when appropriate.

The following SQL `Select` statement implements a custom Perl aggregate named `my_max()`:

```
WITH my_max AS BUILTIN 'perl5' AGGREGATE <<EOF
# -- global variables
my %state;

# -- incremental subroutine, called for each value in a group
sub my_max {
    my $call = $_[0]; # -- is this the first call?
    my $group = $_[1]; # -- which group is the value part of?
    my $value = $_[2]; # -- what is the value being passed?

    if (!defined($state{$group})) { # -- save the first value in the group
        $state{$group} = $value;
    } elsif ($value > $state{$group}) { # -- or save a higher value in the group
        $state{$group} = $value;
    }
}

# -- final subroutine, called after the end of a group is reached
sub my_max_final {
    my $group = $_[1];
    return $state{$group}; # -- return the highest value from the group
}

EOF

SELECT ClientDNS, my_max(RespSize)
FROM example_webserv_100
GROUP BY 1
DURING ALL;
```

The `my_max()` subroutine keeps track of the incremental state, and the `my_max_final()` subroutine returns the final value. Note that `my_max()` does not return any value; instead, it manages the global list variable `%state`.

The arguments to the custom Perl aggregate are saved in the global variables `$call`, `$group`, and `$value`. You reference arguments to custom Perl aggregates by their ordinal numbers.

The first aggregate argument, `$_[0]`, the *call* argument, indicates with the string values `first` and `other` whether this is the first invocation of the aggregate. Many custom Perl aggregates, like `my_max()`, do not use the call argument. The call argument has the string value `final` whenever the final subroutine is called.

The second aggregate argument, `$_[1]`, the *group* argument, indicates which values are combined in the `GROUP BY` clause. The third argument, `$_[2]`, the *value* argument, has a value from a row in the group identified by `$_[1]` when the incremental subroutine is called.

The `%state` variable is a Perl hash table. This table receives its key values from the group values in `$_[1]`, which the Perl aggregate passes into the SQL statement. Each entry in the hash table represents one GROUP BY group. The subroutines access entries in `%state` with the expression `$state{$group}`.

IMPORTANT: Because the EDW uses parallel processing when aggregating data, when an aggregate function is called, the EDW creates multiple instances of the code. Each instance of the code receives a subset of the group keys. Therefore, you cannot write functions that perform calculations *among* groups.

For more information on declaring Perl aggregates, see [“User-Defined Subroutines”](#), on page 82.

PERL EXECUTION ENVIRONMENT

This section describes these topics:

- [“Exiting from Perl Subroutines”, next](#)
- [“List Support and Perl Functions”, on page 229](#)
- [“Using Macros in Perl Subroutines”, on page 230](#)
- [“Understanding Parallelism and Side Effects”, on page 231](#)

Exiting from Perl Subroutines

The SQL query engine prohibits the use of the Perl `exit` operation. Subroutines and modules that include it may fail to load. Use the Perl `die` function instead. For example:

```
WITH Digit AS BUILTIN 'perl5' FUNCTION <<EOF

my %digits = (
    "0" => "zero", "1" => "one",  "2" => "two", "3" => "three",
    "4" => "four",  "5" => "five", "6" => "six",  "7" => "seven",
    "8" => "eight", "9" => "nine"
);

sub Digit {
    return $digits{ $_[0] } || die "invalid value $_[0]";
}

EOF
```

When a Perl function or aggregate calls `die` during query execution, an SQL processing exception with the message passed to `die` is raised, and the query terminates.

List Support and Perl Functions

Perl subroutines can accept lists as arguments. In doing so, the `@_` argument is an array filled with the list values. Perl subroutines can also return lists of values. Use the `addamark::setInto()` function to populate the entries in the returned list. The function has the following syntax:

```
addamark::setInto( <index>, <value> )
```

The `<value>` is stored in a temporary list variable, at the entry specified by `<index>`. The temporary list is returned as the INTO variable when the function is invoked with the INTO keyword:

```
<list_function>( <arguments> ) INTO <list_variable>
```

For more information on INTO variables, see [“Working with Lists”, on page 89](#).

Under normal evaluation, the following `test()` function returns its first argument. With the INTO keyword, the function also returns a list. The list contains three entries: the second argument, the third argument, and the constant `'three'`.

```

WITH Test AS BUILTIN 'perl5' FUNCTION <<EOF
  sub Test {

    # -- make the list available
    addamark::setInto(1, $_[1]);
    addamark::setInto(2, $_[2]);
    addamark::setInto(3, 'three');

    # -- return a single value
    return $_[0];
  }
EOF

SELECT TOP 1
  , foo[1]
  , foo[2]
  , foo[3]
  , Test('a', 'b', 'c') INTO foo
FROM example_webserv_100
DURING ALL;

```

The result of running this query is a four-column row containing the values 'b', 'c', 'three', and 'a':

```

+-----+
| Results for SQL file >example-ref-perl-02.sql< |
+-----+-----+-----+-----+
| foo_1 | foo_2 | foo_3 |perl_into_foo|
| (varchar) | (varchar) | (varchar) | (varchar) |
+-----+-----+-----+-----+
output is post-sorted
+-----+-----+-----+-----+
|b      |c      |three  |a      |
+-----+-----+-----+-----+

```

Using Macros in Perl Subroutines

To facilitate the sharing of information between Perl subroutines and SQL expressions, the SQL query engine defines global Perl variables for each expression macro declared in the SELECT statement. The names of these global variables have the following form:

```
$sql_<macro_identifier>
```

In the following statement, the `$sql_start` and `$sql_end` Perl variables correspond to the `$start` and `$end` expression macros used in the DURING clause:

```

WITH $start AS _timeadd( _now(), -1, 'hour' )
WITH $end as _now()

WITH Last2 AS BUILTIN 'perl5' AGGREGATE <<EOF
  my %state;
  sub Last2 {
    my $group = $_[1];
    my $value = $_[2];
    $state{$group} = $value;
  }
  sub Last2_final {
    my $group = $_[1];

```



```

        return "From $sql_start to $sql_end: $state{$group}";
    }
EOF

```

```

SELECT _perlagg('Last2', ClientDNS)
FROM example_webserv_100
DURING $start, $end

```

```

+-----+
| Results for SQL file >example-ref-perl-03.sql< |
+-----+
|                perlagg                |
|                (varchar)                |
+-----+
output is post-sorted
+-----+
|From 1012558131000000 to 1012561731000000: 212.9.190.79|
+-----+

```

For more information, see [“Macros”, on page 78](#).

Understanding Parallelism and Side Effects

While a single `_perl()` expression is free to make use of side effects, the combination of the distributed nature of the SQL query engine and the column-wise evaluation order make it very difficult for two separate `_perl()` expressions to share state in a meaningful way.

For example, consider the following simple query:

```

WITH Count AS BUILTIN 'perl5' FUNCTION <<EOF
    my $count = 0;
    sub Count {
        $count++;
        return $count;
    }
EOF

SELECT _perl('Count') as cnt1, _perl('Count') as cnt2
FROM Table
DURING ALL;

```

When this query runs against a table that contains a few thousand rows spread across a three-host instance, the following results might be returned:

```

cnt1 cnt2
---- ----
1     24
1     322
1     366
10    33
10    331
10    375
100   421
100   465
1000  1300
1001  1301

```

```

1002 1302
1003 1303
101  422
101  466
...

```

The output is interesting for several reasons:

- Duplicate count values occur because the SQL engine distributes the query across then hosts in the EDW instance. Separate processes on each host have their own private copies of `$count`.
- The differences between values returned by the first and second `_perl()` expressions occur because the SQL query engine makes an arbitrary number of calls to the 'cnt1' expression before it makes any calls to the 'cnt2' expression.
- The actual number of column values processed at any given instant is based on network buffering and is not generally predictable.

ACCESSING EXTERNAL MODULES

This section describes these topics to help you access external modules from within Perl subroutines in SenSage AP SQL:

- [“The use Directive”, next](#)
- [“The @INC Variable”, on page 233](#)
- [“The Inline.pm Perl Module”, on page 233](#)

The use Directive

The Perl `use` directive loads external Perl modules into the Perl interpreter that is embedded in the EDW. For example, the following statement declares a Perl function named `FormatBytes()` that formats numbers as bytes. The statement invokes the function on the sum of the `RespSize` column, which contains byte counts.

The statement uses the `Number::Format` Perl module to implement the custom Perl function. The Perl `use` directive makes the module available within the custom Perl function. Depending on the number of bytes in the expression `sum(RespSize)`, the result of `FormatBytes()` might be `744.6K`.

```

WITH FormatBytes AS BUILTIN 'perl5' FUNCTION <<EOF
    use Number::Format; # -- access the external Perl module from the Perl library
    my $fmt = Number::Format->new;

    sub FormatBytes {
        return $fmt->format_bytes($_[0])
    }
EOF

SELECT _perl('FormatBytes', sum(RespSize) )
FROM example_webserv_100
DURING ALL;

```

NOTE: As an alternative, you can access Perl modules that have been installed in the Perl interpreter. For more information, see [“Installing Perl Modules”, on page 235](#).

The @INC Variable

The `Perl use` directive causes queries to fail if the Perl interpreter cannot load the specified module from one of the places identified by the `@INC` variable. The following query will show you the contents of the `@INC` variable:

```
WITH inc AS BUILTIN 'perl5' FUNCTION <<EOF
  sub inc {
    return join " ", @INC;
  }
EOF

SELECT TOP 1 _perl('inc')
  FROM example_webserv_100 -- use any table containing at least one record
  DURING ALL;
```

If `@INC` does not reflect where your Perl modules are located, use the `perldir=` directive in the `athttpd.conf` file to specify up to 36 additional places where perl modules may be found. Separate locations with colons (:). For example:

```
perldir=/usr/local/lib/perl5/site_perl:/usr/lib/perl5/site_perl
```

You can find the `athttpd.conf` file in this location:

```
<SenSage_Home>latest/etc/sls/instance/<instance_name>
```

The Inline.pm Perl Module

The `Inline.pm` module allows you to call functions written in other languages, including C, C++, and Java, from within your custom Perl subroutines. Languages like C/C++ provide greater performance than Perl, and they allow you to reuse existing code written in these other languages.

TESTING AND DEBUGGING PERL SUBROUTINES

This section describes these topics:

- [“Running Perl Subroutines in Test Scripts”, next](#)
- [“Printing Debugging Messages in Utility Logs”, on page 234](#)
- [“Printing Debugging Messages to External Files”, on page 235](#)

Running Perl Subroutines in Test Scripts

You can test the code of your Perl routines by placing it in command line scripts. For example, to test the `Mean()` Perl aggregate, copy the body of the aggregate declaration as shown below to a file named `mean.pl`, and run the command `perl mean.pl`.

```
Use Statistics::Descriptive;
my %state;
sub Mean {
  my $call = $_[0];
  my $group = $_[1];
```

```

my $value = $_[2];

if ($call eq "first") {
    $state{$group} = Statistics::Descriptive::Full->new();
}
$state{$group}->add_data( $value );
}

sub Mean_final {
    my $group = $_[1];
    return $state{$group}->mean();
}

# simulate the operation of _perlagg()
#
Mean("first", 0, 1000);
Mean("other", 0, 2000);
Mean("other", 0, 3000);
print Mean_final("final", 0), "\n";

```

The last six lines in the test-script version simulate how the SQL query engine invokes the final subroutine when your Perl aggregate runs within a SELECT statement.

Printing Debugging Messages in Utility Logs

Use the `addamark::dbgPrint()` Perl function to print debugging messages that appear in the log files of the `atquery` and `atload` EDW utilities. The debug printing function has the following syntax:

```
addamark::dbgPrint( <message> )
```

For example:

```
WITH sleeper AS BUILTIN 'perl5' FUNCTION <<EOF
```

```

my $counter=0;

sub sleeper {
    $counter++;
    if ($counter % 2 == 0) { sleep(1); }    # -- sleep a second every other call

    my $t = localtime();
    addamark::dbgPrint("$t: $counter");
}
EOF

```

NOTE:

- The EDW runs the debug printing function in parallel on several computers at once, interleaving the messages in the results. This causes messages to appear in a different orders each time.
- If you have many debugging messages to print, it may be easier to write them to your own log files. For more information, see [“Printing Debugging Messages to External Files”, next](#).

Printing Debugging Messages to External Files

Use Perl `open()` and `print()` functions to print debugging messages to your own, external files. For example:

```
WITH MyCount AS BUILTIN 'perl5' function <<EOF
  my $count = 0;
  sub MyCount {
    open(LOG, ">>/tmp/log.$$txt");
    $count++;
    print LOG "Count is ", $count, "\n" ;
    close(LOG);
    return $count;
  }
EOF
SELECT _perl('MyCount')
FROM example_webserv_100
DURING ALL;
```

LOGFILE LOCATIONS AND FILENAMES

Generally, you should open and write debugging messages to files in the `/tmp` directory. This well-known directory exists on every host in an EDW instance. Perl subroutines run in parallel on each host in the instance, so each host will have a version of your log file with a unique set of messages.

Generally, you should include the special symbols `$$` in the filenames of your log files. The symbols expand to the current process ID. Embedding the process ID in filenames prevents multiple processes from overwriting the log files of each other.

INSTALLING PERL MODULES

The EDW embeds a fully-functional Perl interpreter, `atperl`, within its SQL query engine. The interpreter lets you use Perl code in SenSage AP SQL Select statements while loading or querying the EDW. As with any Perl interpreter, you can install Perl modules into `atperl` to extend its functionality.

Perl modules are distributed in files with names like the following:

```
<module_name>.tar.gz
```

The EDW distribution includes these Perl modules:

Module	Distribution File	Description
<code>addamark::dbgPrint</code>		Pre-installed. Prints debugging messages in the log files of the <code>atload</code> and <code>atquery</code> commands.
<code>addamark::setInto</code>		Pre-installed. Supports implementing Perl subroutines that work with the <code>INTO</code> keyword. For more information, see Chapter 8: Perl Subroutines in the <i>Reporting Guide</i> .
<code>Archive::Tar</code>	<code>Archive-Tar-0.22.tar.gz</code>	Pre-installed. Load logs from <code>.tar</code> files.

Module	Distribution File	Description
Archive::Zip	Archive-Zip-1.01.tar.gz	Pre-installed. Load logs from .zip files.
Compress::Zlib	Compress-Zlib-1.19.tar.gz	Pre-installed. Archive loaded logs in .tar files.

In addition, you can download and install third-party Perl modules, such as the `Crypt::Rijndael` module for strong encryption. Rijndael is the U.S. government standard for strong encryption, designed to replace DES.

To install a Perl module in the EDW

1 Download or copy the distribution file to `/usr/local/src`.

2 Run `clsync` to copy the file to every host in the EDW cluster:

```
clsync <module_filename>.tar.gz
```

3 Run the following `clssh` command to install the module on every host in the EDW cluster:

```
clssh "cd /usr/local/src; \
tar zxf <module_filename>; cd <module_filename>;
..HawkEye AP Home/bin/atperl Makefile.PL; make; make install"
```

4 If the Perl module supports test, create a test by running the following `clssh` command:

```
clssh "cd //usr/local/src; \
cd <module_filename>; \
..HawkEye AP Home/bin/atperl make test"
```

NOTE: When you install Perl modules you must also have installed the GCC compiler.

CHAPTER 9

Using the DBD Driver

The DBD driver (DBD::Addamark) implements the DBI (Database Independent) API over the Perl API of the EDW. Only queries (that is, SQL SELECT statements) are supported.

This chapter contains these sections:

- “Installation”, next
- “Sample DBI Program”, on page 238
- “Explanation of the Sample DBI Program”, on page 238
- “DBI elements Supported by SenSage AP”, on page 240
- “SenSage AP DBD attributes”, on page 241

NOTE: For more information about the DBI API, including full documentation, visit:

<http://search.cpan.org/~timb/>

and click the DBI link under Registered Modules near the bottom of the page.

INSTALLATION

Requirements:

- Perl 5.x — (note that there is an issue with 5.8.2 as described in the `Bundle::DBI` README)
- DBI 1.32 or later — download the distribution file for the `Bundle::DBI` Perl module from:

<http://search.cpan.org/~timb/>

and click the DBI link under Registered Modules near the bottom of the page.

- DBD::Addamark — provided in the SenSage AP software distribution in the following location:
<HawkEye AP Home>

Install the DBI and DBD modules according to the instructions for “Installing Perl Modules”, on page 235.

INTERNATIONAL SUPPORT IN THE DBD DRIVER

The DBD driver does not support international characters in text data. The driver operates on ASCII text data only.

SAMPLE DBI PROGRAM

```
#!/usr/bin/env perl
use strict;
use v5.6.0;
use DBD::Addamark;
#DBI->trace(3);

my $port = shift;
# print "$port\n";

my $namespace = shift;
# print "$namespace\n";

my $sqlfile = shift;
# print "$sqlfile\n";

my $sql = "";
open(FILE, $sqlfile) || die "could not open $sqlfile";

while (<FILE>) {
    $sql .= $_;
}
close(FILE);

my $dbh = DBI->connect('dbi:Addamark:host.domain.com:1234',
                      'username', 'password');
my $sth = $dbh->prepare("select * from syslog during all");
$sth->execute();

while(my $d = $sth->fetch) {
    print "@$d\n";
}
```

EXPLANATION OF THE SAMPLE DBI PROGRAM

Perl Code	Explanation
#!/usr/bin/env perl	A special syntax that tells the shell to run the rest of the script using Perl.
use strict; use v5.6.0;	Set up the Perl environment (v5.6.0) and use strict to restrict unsafe constructs and limit constructs to using Perl version 5.6.0.
use DBD::Addamark;	Specify that DBI should use the Addamark DBD driver.
#DBI->trace(3);	A comment line. Uncomment to set DBI tracing to level three. Trace level three traces DBI calls returning with results or errors, method entries with parameters and returning with results, and adds high-level information from the DBD driver and internal information from DBI.
my \$port = shift;	Declares \$port as a local variable and sets the value. The \$port variable stores the TCP/IP port to be used when connecting to the database server.

Perl Code	Explanation
<code># print "\$port\n";</code>	A comment line. Uncomment to display the port number to be used.
<code>my \$namespace = shift;</code>	Declares <code>\$namespace</code> as a local variable and sets the value. The <code>\$namespace</code> variable stores the namespace to be used when connecting to the database server.
<code># print "\$namespace\n";</code>	A comment line. Uncomment to display the Addamark namespace.
<code>my \$sqlfile = shift;</code>	Declares <code>\$sqlfile</code> as a local variable and sets the value. The <code>\$sqlfile</code> variable stores the name of the SQL file. The SQL file contains the SQL statement or statements (query or queries) to send to the database server.
<code># print "\$sqlfile\n";</code>	A comment line. Uncomment to display the name of the sqlfile.
<code>my \$sql = "";</code>	Declares <code>\$sql</code> as a local variable and initializes (clears) the value. The <code>\$sql</code> variable stores a SQL statement to be passed to the database server after the connection is made.
<code>open(FILE, \$sqlfile) die "could not open \$sqlfile";</code>	Try to open the SQL file. If the file cannot be opened, exits the program and displays an error message saying, "could not open <i><sqlfile></i> ", where <i><sqlfile></i> is the name of the SQL file.
<code>while (<FILE>) { \$sql .= \$_; }</code>	Iteratively read the contents of the SQL file and puts the content into the <code>\$_</code> global variable.
<code>close(FILE);</code>	Closes the SQL file.
<code>my \$dbh = DBI->connect('dbi:Addamark:' . \$port, '', '');</code>	Declares <code>\$dbh</code> as a local variable and sets the value to <code>DBI->connect('dbi:Addamark:' . \$port, '', '')</code> , where <code>port</code> is the value previously stored in the <code>\$port</code> variable. This entire variable is used to construct the database handle, which contains the information needed to connect with the database.
<code>\$dbh->{addamark_tableNamespace} = \$namespace;</code>	The completely constructed database handle connects to the Addamark database in the <code>addamark_tableNamespace</code> . The connection to the database server uses the TCP/IP port contained in the <code>\$port</code> variable, and DBI uses the <code>DBD::Addamark</code> driver.
<code>my \$sth = \$dbh->prepare(\$sql);</code>	Declares <code>\$sth</code> as a local variable, which is used to prepare the SQL statement from the SQL file for execution. A statement handle is returned, and that handle contains the SQL statement that may be executed later using the <code>execute</code> method.
<code>\$sth->execute warn \$sth->errstr;</code>	Executes the SQL statement (from the SQL file) contained in the statement handle. If an error occurs, an error message is displayed, showing the errors from the last SQL statement executed.
<code>while(my \$d = \$sth->fetch) { print "@\$d\n"; }</code>	Iteratively retrieve the results from the query and stores them in the local variable <code>\$d</code> , then displays the results, one to a line.

DBI ELEMENTS SUPPORTED BY SENSAGE AP

DBD::Addamark supports most of the database and statement handle attributes; however, not all the attributes are relevant to this driver. As of DBI v.1.40, the supported attributes are:

- **Database handle attributes**

- AutoCommit
- Driver
- Name
- Statement
- RowCacheSize
- Username

For more information about these attributes, see:

http://search.cpan.org/~timb/DBI-1.40/DBI.pm#Database_Handle_Attributes

- **Statement handle attributes**

- NUM_OF_FIELDS
- NUM_OF_PARAMS
- NAME
- NAME_lc
- NAME_uc
- NAME_hash
- NAME_lc_hash
- NAME_uc_hash
- TYPE
- CursorName
- Database
- Statement

For more information about these attributes, see

http://search.cpan.org/~timb/DBI-1.40/DBI.pm#Statement_Handle_Attributes

UNSUPPORTED DBI FEATURES

The following DBI features *are not* supported by SenSage AP:

- Question marks (?) as placeholders (and the ParamValues statement handle attribute will return “undef”) are not supported.
- The following database schema related methods are not supported:
 - `table_info()`
 - `column_info()`
 - `primary_key_info()`
 - `primary_key()`
 - `foreign_key_info()`
 - `tables()`
 - `type_info_all()`
 - `type_info()`
 - `get_info()`

For more information about these methods, see

http://search.cpan.org/~timb/DBI-1.40/DBI.pm#Database_Handle_Methods

SENSAGE AP DBD ATTRIBUTES

In addition to the previously mentioned attributes, DBD::Addamark implements the following private driver attributes:

- `addamark_tableNamespace`
- `addamark_rawOutputHandle`
- `addamark_dbgprintRequest`
- `addamark_oob_callback`
- `addamark_tableNamespace`

addamark_tableNamespace

String, required. Sets the `tableNamespace` value in the request.

```
$dbh->{addamark_tableNamespace} = "default";
```

addamark_rawOutputHandle

IO handle, optional. Sets the `rawOutputHandle` for dumping internal request data. Useful for debugging. Must be specified when `addamark_dbgprintRequest` is enabled.

addamark_dbgprintRequest

Boolean, optional, default is "F" if not specified. When true, the request to the server is printed out, and the program terminates. Requires `addamark_rawOutputHandle` to be specified.

addamark_oob_callback

Subroutine reference, optional. A subroutine that is called with any OOB data sent back during the execution of the request. The subroutine should have the following structure:

```
sub oob_callback {  
    my $code = shift;  
    my $msg = shift;  
    my $type = shift;  
    my $data = shift;  
    .... do any processing here ....  
}
```

Use a callback subroutine to code progress meters, for example.

MAPPING OPERATORS AND FUNCTIONS

This appendix lists all of the Postgres functions on types that EDW supports. This appendix documents the following operators and functions:

- [“Logical Operators”, next](#)
- [“Comparison Operators”, on page 244](#)
- [“Mathematical Operators”, on page 244](#)
- [“Mathematical Functions”, on page 245](#)
- [“String Functions and Operators”, on page 246](#)
- [“Conversion Functions”, on page 247](#)
- [“Pattern Matching”, on page 248](#)
- [“Formatting Functions”, on page 248](#)
- [“Date/Time Operators”, on page 248](#)
- [“Inet Functions”, on page 249](#)
- [“Aggregate Functions for Statistics”, on page 250](#)

LOGICAL OPERATORS

PG Operator or Function	# of Arguments	Translation to EDW
AND	2	AND
OR	2	OR
NOT	1	NOT

COMPARISON OPERATORS

PG Operator or Function	# of Arguments	Translation to EDW
<	2	<
>	2	>
<=	2	<=
>=	2	>=
=	2	=
<>	2	<>
BETWEEN	3	BETWEEN (but change AND to ,)
BETWEEN SYMMETRIC	3	no translation to SLS (but executes in OAE)
IS [NOT] NULL	2	true or false
DISTINCT FROM	2	<>
IS [NOT] {TRUE FALSE UNKNOWN}	2	one of true, false, =, <>
IN		IN

MATHEMATICAL OPERATORS

NOTE: N/A in the third column means this operator has no translation in EDW, but executes in OAE.

PG Operator or Function	# of Arguments	Translation to EDW
+	2	+
+	1	+
-	2	-
-	1	-
*	2	*
/	2	/
%	2	%
^ (exponentiation)	2	N/A
/ (square root)	1	N/A
/ (cube root)	1	N/A
! (factorial)	1	N/A
!! (factorial)	1	N/A
@ (abs)	1	N/A
&	2	N/A
	2	N/A

PG Operator or Function	# of Arguments	Translation to EDW
# (xor)	2	N/A
~	1	N/A
<<	2	
>>	2	

MATHEMATICAL FUNCTIONS

NOTE: N/A in the third column means this function has no translation in EDW but executes in OAE..

PG Operator or Function	# of Arguments	Translation to EDW
abs	1	_abs
cbirt	1	N/A
ceil	1	_ceil
ceiling	1	_ceil
degrees	1	N/A
exp	1	_exp
floor	1	_floor
ln	1	N/A
log (base 10)	1	_log10
log	2	N/A
mod	2	N/A
pi	0	N/A
power	2	N/A
radians	1	N/A
random	0	N/A
round	1	N/A
round	2	_round
setseed	1	N/A
sign	1	N/A
sqrt	1	N/A
trunc	1	N/A
trunk	2	N/A
width_bucket	4	N/A
acos	1	N/A
asin	1	N/A
atan	1	N/A
atan2	2	N/A

PG Operator or Function	# of Arguments	Translation to EDW
cos	1	N/A
cot	1	N/A
sin	1	N/A
tan	1	N/A

STRING FUNCTIONS AND OPERATORS

NOTE: N/A in the third column means this function/operator has no translation in EDW, but executes in OAE

PG Operator or Function	# of Arguments	Translation to EDW
	2	+
bit_length	1	8*_strlen(%1)
char_length	1	_strlen
lower	1	_strlowercase
octet_length	1	_strlowercase
overlay	3	N/A
position	2	_strstr(%2,%1)
substring	3	_substr(%1,(%2)-1,%3)
substring (string from pattern)	2	_substr(%1,(%2)-1)
substring	3	_substr(%1,(%2)-1,%3)
trim	2	N/A
upper	1	_strupercase
ascii	1	N/A
btrim	2	_strtrim(%1,%2,%2)
chr	1	N/A
convert	3	N/A
convert_from	2	N/A
convert_to	2	N/A
decode	2	N/A
encode	2	N/A
initcap	1	N/A
length	1	N/A
lpad	3	_strlpad
ltrim	2	strtrim(%1,%2,0)?
md5	1	_strmd5_64
quote_ident	1	N/A
quote_literal	1	N/A

PG Operator or Function	# of Arguments	Translation to EDW
regexp_matches	3	N/A
regexp_replace	4	N/A
repeat	2	_strrepeat
rpad	3	_strlpad
rtrim	2	_strtrim(%1,0,%2)
split_apart	3	N/A
strpos	2	_strstr
substr	3	_substr(%1,(%2)-1,%3)
to_ascii	2	N/A
to_hex	2	N/A
translate	3	N/A

CONVERSION FUNCTIONS

PG Operator or Function	# of Arguments	Translation to EDW
bool	N/A	_bool
int4	N/A	_int32
int8	N/A	_int64
float8	N/A	_float
timestamp	N/A	_timestamp
text	N/A	_varchar
CAST	N/A	_bool, _int32, _int64, _float, _timestamp, _varchar OAE translates CAST only when the EDW has a corresponding type; for more information, see “Mapping Data Types” , on page 204.

PATTERN MATCHING

PG Operator or Function	# of Arguments	Translation to EDW
LIKE	N/A	LIKE
SIMILAR	N/A	N/A

FORMATTING FUNCTIONS

PG Operator or Function	# of Arguments	Translation to EDW
to_char	N/A	N/A
to_date	N/A	N/A
to_number	N/A	N/A
to_timestamp	N/A	N/A

DATE/TIME OPERATORS

NOTE: N/A in the third column means this function/operator has no translation in EDW, but executes in OAE.

PG Operator or Function	# of Arguments	Translation to EDW
+	2	N/A
-	2	N/A
*	2	N/A
/	2	N/A
OVERLAPS	4	N/A
EXTRACT	1	N/A

INET FUNCTIONS

PG Operator or Function	# of Arguments	Translation to EDW
broadcast	1	_broadcast
family	1	_family
host	1	_host
hostmask	1	_hostmask
masklen	1	_masklen
netmask	1	_netmask
set_masklen	1	_set_masklen
abbrev	1	_abbrev
oae.mapto_ipv4	1	_mapto_ipv4
oae.mapto_ipv6	1	_mapto_ipv6

INET OPERATORS.

PG Operator or Function	# of Arguments	Translation to EDW
&	2	_inet_and
	2	_inet_or
+	2	_inet_plus
-	2	_inet_minus
~	1	_inet_not

CONDITIONAL EXPRESSIONS

PG Operator or Function	# of Arguments	Translation to EDW
CASE	N/A	CASE (but only if there is an ELSE clause)
COALESCE	N/A	CASE

AGGREGATE FUNCTIONS

PG Operator or Function	# of Arguments	Translation to EDW
avg	N/A	avg (complex translation)
bool_and	N/A	N/A
bool_or	N/A	N/A
count(*)	N/A	count(*)

PG Operator or Function	# of Arguments	Translation to EDW
count	N/A	count
max	N/A	max
min	N/A	min
sum	N/A	sum

AGGREGATE FUNCTIONS FOR STATISTICS

PG Operator or Function	# of Arguments	Translation to EDW
corr	N/A	min
covar_pop	N/A	N/A
covar_samp	N/A	N/A
regr_avgx	N/A	N/A
regr_avgy	N/A	N/A
regr_count	N/A	N/A
regr_intercept	N/A	N/A
regr_r2	N/A	N/A
regr_slope	N/A	N/A
regr_sxx	N/A	N/A
regr_sxy	N/A	N/A
regr_syy	N/A	N/A
stddev	N/A	stddev
stddev_pop	N/A	stddev_pop
stddev_samp	N/A	stddev_samp
variance	N/A	variance
var_pop	N/A	var_pop
var_samp	N/A	var_samp

APPENDIX B

TIME ZONES

This appendix lists every time zone supported by the EDW. Because the EDW handles raw data from logs, which does not always use the supported time zones, it must accommodate some shortenings of time zone indicators found in that data (such as PDT for Pacific Daylight Time and CST for Central Standard Time). For more information, see [“Time-Zone Conversion”, next](#).

TIME-ZONE CONVERSION

SenSage AP SQL provides functions that recognize specific time-zone shortenings and maps them to standard time-zone strings, as shown in the table below.

Shortened Time Zone	Standard Time Zone	Description
PST	PST8PDT	Pacific Standard Time and Pacific Daylight Time are interpreted as either standard time or daylight savings depending on the time of year of the actual date.
PDT	PST8PDT	
MDT	MST7MDT	Mountain Daylight Time is interpreted as either standard time or daylight savings depending on the time of year of the actual date.
CST	CST6CDT	Central Standard Time and Central Daylight Time are interpreted as either standard time or daylight savings depending on the time of year of the actual date.
CDT	CST6CDT	
EDT	EST5EDT	Eastern Daylight Time is interpreted as either standard time or daylight savings depending on the time of year of the actual date.

NOTE:

- Because Arizona does not support MDT, Mountain Standard Time is NOT converted to Mountain Daylight Time.
- Because Indiana does not consistently support EDT, Eastern Standard Time is NOT converted to Eastern Daylight Time.

SUPPORTED TIME ZONES

Africa/Addis_Ababa
Africa/Algiers
Africa/Asmera
Africa/Bangui
Africa/Blantyre
Africa/Brazzaville
Africa/Bujumbura
Africa/Cairo
Africa/Ceuta
Africa/Dar_es_Salaam
Africa/Djibouti
Africa/Douala
Africa/Gaborone

Africa/Harare
Africa/Johannesburg
Africa/Kampala
Africa/Khartoum
Africa/Kigali
Africa/Kinshasa
Africa/Lagos
Africa/Libreville
Africa/Luanda
Africa/Lubumbashi
Africa/Lusaka
Africa/Malabo
Africa/Maputo
Africa/Maseru
Africa/Mbabane
Africa/Mogadishu
Africa/Nairobi
Africa/Ndjamena
Africa/Niamey
Africa/Porto-Novo
Africa/Tripoli
Africa/Tunis
Africa/Windhoek
America/Adak
America/Anchorage
America/Anguilla
America/Antigua
America/Araguaina
America/Argentina/Buenos_Aires
America/Argentina/Catamarca
America/Argentina/ComodRivadavia
America/Argentina/Cordoba
America/Argentina/Jujuy
America/Argentina/La_Rioja
America/Argentina/Mendoza
America/Argentina/Rio_Gallegos
America/Argentina/San_Juan
America/Argentina/Tucuman
America/Argentina/Ushuaia
America/Aruba
America/Asuncion
America/Atka
America/Bahia
America/Barbados
America/Belem
America/Belize
America/Boa_Vista
America/Bogota
America/Boise
America/Buenos_Aires
America/Cambridge_Bay
America/Campo_Grande
America/Cancun
America/Caracas
America/Catamarca
America/Cayenne
America/Cayman
America/Chicago
America/Chihuahua

America/Coral_Harbour
America/Cordoba
America/Costa_Rica
America/Cuiaba
America/Curacao
America/Dawson
America/Dawson_Creek
America/Denver
America/Detroit
America/Dominica
America/Edmonton
America/Eirunepe
America/El_Salvador
America/Ensenada
America/Fort_Wayne
America/Fortaleza
America/Glace_Bay
America/Godthab
America/Goose_Bay
America/Grand_Turk
America/Grenada
America/Guadeloupe
America/Guatemala
America/Guayaquil
America/Guyana
America/Halifax
America/Havana
America/Hermosillo
America/Indiana/Indianapolis
America/Indiana/Knox
America/Indiana/Marengo
America/Indiana/Vevay
America/Indianapolis
America/Inuvik
America/Iqaluit
America/Jamaica
America/Jujuy
America/Juneau
America/Kentucky/Louisville
America/Kentucky/Monticello
America/Knox_IN
America/La_Paz
America/Lima
America/Los_Angeles
America/Louisville
America/Maceio
America/Managua
America/Manaus
America/Martinique
America/Mazatlan
America/Mendoza
America/Menominee
America/Merida
America/Mexico_City
America/Miquelon
America/Monterrey
America/Montevideo
America/Montreal
America/Montserrat

America/Nassau
America/New_York
America/Nipigon
America/Nome
America/Noronha
America/North_Dakota/Center
America/Panama
America/Pangnirtung
America/Paramaribo
America/Phoenix
America/Port-au-Prince
America/Port_of_Spain
America/Porto_Acre
America/Porto_Velho
America/Puerto_Rico
America/Rainy_River
America/Rankin_Inlet
America/Recife
America/Regina
America/Rio_Branco
America/Rosario
America/Santiago
America/Santo_Domingo
America/Sao_Paulo
America/Shiprock
America/St_Johns
America/St_Kitts
America/St_Lucia
America/St_Thomas
America/St_Vincent
America/Swift_Current
America/Tegucigalpa
America/Thule
America/Thunder_Bay
America/Tijuana
America/Toronto
America/Tortola
America/Vancouver
America/Virgin
America/Whitehorse
America/Winnipeg
America/Yakutat
America/Yellowknife
Antarctica/Casey
Antarctica/Davis
Antarctica/DumontDUrville
Antarctica/Mawson
Antarctica/McMurdo
Antarctica/Palmer
Antarctica/Rothera
Antarctica/South_Pole
Antarctica/Syowa
Antarctica/Vostok
Arctic/Longyearbyen
Asia/Aden
Asia/Almaty
Asia/Amman
Asia/Anadyr
Asia/Aqtau

Asia/Aqtobe
Asia/Ashgabat
Asia/Ashkhabad
Asia/Baghdad
Asia/Bahrain
Asia/Baku
Asia/Bangkok
Asia/Beirut
Asia/Bishkek
Asia/Brunei
Asia/Calcutta
Asia/Choibalsan
Asia/Chongqing
Asia/Chungking
Asia/Colombo
Asia/Dacca
Asia/Damascus
Asia/Dhaka
Asia/Dili
Asia/Dubai
Asia/Dushanbe
Asia/Gaza
Asia/Harbin
Asia/Hong_Kong
Asia/Hovd
Asia/Irkutsk
Asia/Istanbul
Asia/Jakarta
Asia/Jayapura
Asia/Jerusalem
Asia/Kabul
Asia/Kamchatka
Asia/Karachi
Asia/Kashgar
Asia/Katmandu
Asia/Krasnoyarsk
Asia/Kuala_Lumpur
Asia/Kuching
Asia/Kuwait
Asia/Macao
Asia/Macau
Asia/Magadan
Asia/Makassar
Asia/Manila
Asia/Muscat
Asia/Nicosia
Asia/Novosibirsk
Asia/Omsk
Asia/Oral
Asia/Phnom_Penh
Asia/Pontianak
Asia/Pyongyang
Asia/Qatar
Asia/Qyzylorda
Asia/Rangoon
Asia/Riyadh
Asia/Riyadh87
Asia/Riyadh88
Asia/Riyadh89

Asia/Saigon
Asia/Sakhalin
Asia/Samarkand
Asia/Seoul
Asia/Shanghai
Asia/Singapore
Asia/Taipei
Asia/Tashkent
Asia/Tbilisi
Asia/Tehran
Asia/Tel_Aviv
Asia/Thimbu
Asia/Thimphu
Asia/Tokyo
Asia/Ujung_Pandang
Asia/Ulaanbaatar
Asia/Ulan_Bator
Asia/Urumqi
Asia/Vientiane
Asia/Vladivostok
Asia/Yakutsk
Asia/Yekaterinburg
Asia/Yerevan
Atlantic/Bermuda
Atlantic/Canary
Atlantic/Cape_Verde
Atlantic/Faeroe
Atlantic/Jan_Mayen
Atlantic/Madeira
Atlantic/South_Georgia
Atlantic/Stanley
Australia/ACT
Australia/Adelaide
Australia/Brisbane
Australia/Broken_Hill
Australia/Canberra
Australia/Currie
Australia/Darwin
Australia/Hobart
Australia/LHI
Australia/Lindeman
Australia/Lord_Howe
Australia/Melbourne
Australia/NSW
Australia/North
Australia/Perth
Australia/Queensland
Australia/South
Australia/Sydney
Australia/Tasmania
Australia/Victoria
Australia/West
Australia/Yancowinna
Brazil/Acre
Brazil/DeNoronha
Brazil/East
Brazil/West
CET
CST6CDT

Canada/Atlantic
Canada/Central
Canada/East-Saskatchewan
Canada/Eastern
Canada/Mountain
Canada/Newfoundland
Canada/Pacific
Canada/Saskatchewan
Canada/Yukon
Chile/Continental
Chile/EasterIsland
Cuba
EET
EST
EST5EDT
Egypt
Eire
Europe/Amsterdam
Europe/Andorra
Europe/Athens
Europe/Belfast
Europe/Belgrade
Europe/Berlin
Europe/Bratislava
Europe/Brussels
Europe/Bucharest
Europe/Budapest
Europe/Chisinau
Europe/Copenhagen
Europe/Dublin
Europe/Gibraltar
Europe/Helsinki
Europe/Istanbul
Europe/Kaliningrad
Europe/Kiev
Europe/Lisbon
Europe/Ljubljana
Europe/London
Europe/Luxembourg
Europe/Madrid
Europe/Malta
Europe/Mariehamn
Europe/Minsk
Europe/Monaco
Europe/Moscow
Europe/Nicosia
Europe/Oslo
Europe/Paris
Europe/Prague
Europe/Riga
Europe/Rome
Europe/Samara
Europe/San_Marino
Europe/Sarajevo
Europe/Simferopol
Europe/Skopje
Europe/Sofia
Europe/Stockholm
Europe/Tallinn

Europe/Tirane
Europe/Tiraspol
Europe/Uzhgorod
Europe/Vaduz
Europe/Vatican
Europe/Vienna
Europe/Vilnius
Europe/Warsaw
Europe/Zagreb
Europe/Zaporozhye
Europe/Zurich
GB
GB-Eire
GMT
HST
Hongkong
Indian/Antananarivo
Indian/Chagos
Indian/Christmas
Indian/Cocos
Indian/Comoro
Indian/Kerguelen
Indian/Mahe
Indian/Maldives
Indian/Mauritius
Indian/Mayotte
Indian/Reunion
Iran
Israel
Jamaica
Japan
Kwajalein
Libya
MET
MST
MST7MDT
Mexico/BajaNorte
Mexico/BajaSur
Mexico/General
Mideast/Riyadh87
Mideast/Riyadh88
Mideast/Riyadh89
NZ
NZ-CHAT
Navajo
PRC
PST8PDT
Pacific/Apia
Pacific/Auckland
Pacific/Chatham
Pacific/Easter
Pacific/Efate
Pacific/Enderbury
Pacific/Fakaofo
Pacific/Fiji
Pacific/Funafuti
Pacific/Galapagos
Pacific/Gambier
Pacific/Guadalcanal

Pacific/Guam
Pacific/Honolulu
Pacific/Johnston
Pacific/Kiritimati
Pacific/Kosrae
Pacific/Kwajalein
Pacific/Majuro
Pacific/Marquesas
Pacific/Midway
Pacific/Nauru
Pacific/Niue
Pacific/Norfolk
Pacific/Noumea
Pacific/Pago_Pago
Pacific/Palau
Pacific/Pitcairn
Pacific/Ponape
Pacific/Port_Moresby
Pacific/Rarotonga
Pacific/Saipan
Pacific/Samoa
Pacific/Tahiti
Pacific/Tarawa
Pacific/Tongatapu
Pacific/Truk
Pacific/Wake
Pacific/Wallis
Pacific/Yap
Poland
Portugal
ROK
Singapore
Turkey
US/Alaska
US/Aleutian
US/Arizona
US/Central
US/East-Indiana
US/Eastern
US/Hawaii
US/Indiana-Starke
US/Michigan
US/Mountain
US/Pacific
US/Samoa
UTC
W-SU
WET

OPEN ACCESS EXTENSION CONSIDERATIONS

This appendix lists all known issues regarding Open Access Extension (OAE).

TRANSLATION FAILURE

OAE cannot handle an external-table query whose GROUP BY clause references an expression other than a column number or name. For example, the following query will fail translation and will return an error message:

```
SELECT columnA+columnB, sum(columnC)
  FROM someTable
 GROUP BY columnA+columnB
```

You can workaround this problem by writing the above query as follows:

```
select columnA+ColumnB as a, sum(columnC) from someTable
Group BY a
```

ARCHITECTURE MODEL SUPPORT

OAE supports the client-server model, The server-server model is unsupported.

SENSAGE APSYSTEM TABLES

This appendix contains the schema of each system table.

- “system.properties”, on page 263
- “system.cluster_properties”, on page 264
- “system.task_list”, on page 264
- “system.users”, on page 265
- “system.users2”, on page 265
- “system.userroles”, on page 265
- “system.userroles2”, on page 266
- “system.roles”, on page 266
- “system.roles2”, on page 266
- “system.permissions”, on page 267
- “system.rolepermissions”, on page 267
- “system.rolepermissions2”, on page 267
- “system.upload_info”, on page 268
- “system.raw_upload_info”, on page 268
- “< namespace>.storage.metadata”, on page 269
- “< namespace>.storage.metadata_history”, on page 270
- “< namespace>.storage.raw_metadata”, on page 270
- “< namespace>.storage.raw_metadata_with_ts”, on page 271
- “< namespace>.<tablename>.storage.metadata”, on page 272

IMPORTANT:

- To enable the EDW to integrate with an external authentication authority (such as Active Directory), several system tables required additional columns. To prevent incompatibility with existing scripts that reference these tables, SenSage AP did not add columns to the existing tables. Instead, SenSage AP created a new expanded version of these tables. The name of each expanded table ends in "2".
- The values in authentication tables are retrieved from the external authentication authority. You do not directly enter these values into the EDW by running `atmanage`.

system.properties

This table provides configuration information about a specific host in your EDW instance. For example, run `atquery` against this table to return the version of SenSage AP running and the date that SenSage AP version was installed.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row
basic_property_name	varchar	property name

Column Name	Data Type	Description
basic_property_value	varchar	property value

system.cluster_properties

This table provides configuration information about an entire EDW instance. For example, run `atquery` against this table to return the name, host (node), port, and block size of every host in your EDW instance.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row, for example. columns or tables
node_index	int32	identifies the host in the cluster
dsm_sib_idx	int32	identifies sibling relationships between hosts
cluster_name	varchar	the EDW instance name
node_name	varchar	the host "name" (typically the same as the DNS host name, but not required to be)
node	varchar	the DNS host name for a host, can vary if the self-reported name differs from the name that other hosts use to address it
port	int32	port number the host is listening on
device	varchar	device number, for accurately detecting disk-sharing, and avoid misreporting free space
block_size	int32	size (in bytes) of a block on this device
num_blocks	int64	total capacity (in blocks) of this device
num_bytes	int64	total capacity (in bytes) of this device
count	int32	number of the instances of this type of item

system.task_list

This table provides information about tasks running in your EDW instance. For example, run `atquery` against this table to return the start time of running tasks.

Column Name	Data Type	Description
row_type	varchar	type of information being returned: NODE or TASK
node	varchar	host the record relates to
port	int32	port of the host the record relates to
_systaskid	varchar	internal ID that identifies the task in the system
method_name	varchar	name of the internal method being run
runqueue	varchar	priority level of the queue that is running
purpose	varchar	additional task information (where known)

Column Name	Data Type	Description
_extid	varchar	external ID that the client program has given this task
at_top	bool	if true, this is the process running as the "master" for this operation
start_time	varchar	time the task started
finish_time	timestamp	time the task ended (flushes every few minutes)
state	timestamp	current state, for example, <code>RUNNING</code>

system.users

This table provides the unique identifier of each user in your EDW instance. It also returns each user's status (enabled or disabled).

Column Name	Data Type	Description
uid	varchar	unique user identifier of every user in the system, including administrator, guest, system, and individual users
isEnabled	boolean	indicates whether the user has been enabled

system.users2

This table provides information about the users in your EDW instance. The source of the information is an external authentication authority.

Column Name	Data Type	Description
name	varchar	login name
local_id	varchar	user identifier that is unique within a specific EDW instance
guid	varchar	user identifier that is unique across all EDW instances
isEnabled	boolean	indicates whether the user has been enabled to use the system
real_name	varchar	user's full name
mail	varchar	user's email address
pager	varchar	user's pager number
description	varchar	text description of the user

system.userroles

This table returns user/role relationships: every role in your EDW instance and the users assigned to each role. Roles with multiple users display multiple times in the role column (as many rows as the number of users assigned to them).

Column Name	Data Type	Description
role	varchar	name of every role in the system, including administrator, guest, system, and roles created for individual users
uid	varchar	unique identifier of each user assigned to a specific role

system.userroles2

This table returns user/role relationships: every role in your EDW instance, the users assigned to each role, and the group identifiers of each role and user. The source of the information is an external authentication authority.

Column Name	Data Type	Description
role_name	varchar	name of every role in the system, including administrator, guest, system, and roles created for individual users
role_id	varchar	role identifier that is unique within a specific EDW instance
role_guid	varchar	role identifier that is unique across all EDW instances
user_name	varchar	login name
user_id	varchar	user identifier that is unique within a specific EDW instance
user_guid	varchar	user identifier that is unique across all EDW instances

system.roles

This table provides the unique identifier of each role in your EDW instance. It also returns each role's status (enabled or disabled).

Column Name	Data Type	Description
role	varchar	unique identifier of every role in the system, including administrator, guest, system.
isEnabled	boolean	indicates whether the role has been enabled

system.roles2

This table returns every role in your EDW instance and its local and global identifiers. It also returns each role's status (enabled or disabled). The source of the information is an external authentication authority.

Column Name	Data Type	Description
name	varchar	name of every role in the system, including administrator, guest, system, and roles created for individual users
local_id	varchar	role identifier that is unique within a specific EDW instance

Column Name	Data Type	Description
guid	varchar	role identifier that is unique across all EDW instances
isEnabled	boolean	indicates whether the role has been enabled

system.permissions

This table returns every permission that can be granted to a role in your EDW instance.

Column Name	Data Type	Description
permission	varchar	name of every permission in the system
mixMethod	boolean-or string-set	indicates whether the permission has been enabled

system.rolepermissions

This table returns role/permission relationships: every role in your EDW instance and the permissions assigned to each role. Roles with multiple permissions display multiple times in the role column (as many rows as the number of permissions assigned to them). The table also indicates whether each permission has been enabled for each role.

Column Name	Data Type	Description
role	varchar	name of every role in the system
permission	varchar	name of every permission in the system
value	varchar	this column has internal significance only

system.rolepermissions2

This table returns role/permission relationships: every role in your EDW instance and the permissions assigned to each role. Roles with multiple permissions display multiple times in the role column (as many rows as the number of permissions assigned to them). The table also returns the globally unique role identifier and indicates whether each permission has been enabled for each role. The source of the information is an external authentication authority.

Column Name	Data Type	Description
role	varchar	name of every role in the system, including administrator, guest, system, and roles created for individual users
role_id	varchar	role identifier that is unique within a specific EDW instance
role_guid	varchar	role identifier that is unique across all EDW instances
permission	varchar	name of every permission in the system
value	varchar	this column has internal significance only

system.upload_info

This table allows administrators to check on upload status in the system. When the system is healthy, this table returns one row for each upload (using its unique identifier, **uploadid**). When the upload data is not consistent across all the hosts in the instance, this table returns rows with a **consistent** value of `false`. In this case, the table may return multiple rows for a given **uploadid** (all of which are marked as inconsistent).

Column Name	Data Type	Description
uploadid	varchar	The unique identifier for a particular load
original_tablename	varchar	The fully qualified name of the table that was the target of this load at the time of the load operation
ptl_signature	varchar	The digital signature of the PTL Information used for this load
started	timestamp	The time at which the load was started
completed	timestamp	The time at which the load completed
user	varchar	The username of the person who performed the load
min_ts	timestamp	The minimum timestamp in the loaded data
max_ts	timestamp	The maximum timestamp in the loaded data
line_count	int64	The number of lines in the source log data
parse_count	int64	The number of lines that were successfully parsed by the PTL's regular expressions (regexes)
load_count	int64	The number of rows that were loaded into the table
successful	boolean	Indicates whether the load completed successfully
client_signature	varchar	The client-side signature that was generated for this load
client_signature_method	varchar	Indicates how the client generated the signature (a value of <code>PRIVATE</code> indicates it is client-specific)
client_blob	varchar	The additional data that the client wanted associated with this load
current_tablename	varchar	The fully qualified name of the current name of the table that received this load
consistent	boolean	Indicates whether information about this upload was consistent across the cluster

system.raw_upload_info

This table allows a support person to see the raw data that is distributed across the system in order to troubleshoot an 'inconsistency' problem in the `upload_info` data.

Column Name	Data Type	Description
uploadid	varchar	The unique identifier for a particular load
original_tablename	varchar	The fully qualified name of the table that was the target of this load at the time of the load operation

Column Name	Data Type	Description
ptl_signature	varchar	The digital signature (MD5) of the PTL Information used for this load
started	timestamp	The time at which the load was started
completed	timestamp	The time at which the load completed
user	varchar	The username of the person who performed the load
min_ts	timestamp	The minimum timestamp in the loaded data
max_ts	timestamp	The maximum timestamp in the loaded data
line_count	int64	The number of lines in the source log data
parse_count	int64	The number of lines that were successfully parsed by the PTL's regular expressions (regexes)
load_count	int64	The number of rows that were loaded into the table
successful	boolean	Indicates whether the load completed successfully
client_signature	varchar	The client-side signature that was generated for this load
client_signature_method	varchar	Indicates how the client generated the signature (a value of <code>PRIVATE</code> indicates it is client-specific)
client_blob	varchar	The additional data that the client wanted associated with this load
partition	varchar	Specifies either the Primary or Secondary directory
current_tablename	varchar	The fully qualified name of the current name of the table that received this load
node	varchar	The logical name of the host that this row of data came from

<namespace>.storage.metadata

This table provides information about the tables and columns in the specified namespace. If you specify an empty namespace ("") when you run `atquery` against this table, the table returns all tables and columns throughout your EDW instance.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row. 'COLUMN_INFO' =
cluster_name	varchar	the installation name
namespace	varchar	the complete namespace name for this table
table_name	varchar	the table name
column_name	varchar	the column name
column_type	varchar	the data type for this column
primary_key	boolean	is this the primary key? typically, only the 'ts' (timestamp) column is a key
min_value	varchar	the minimum value for this column (currently, only available for the 'ts' column)

Column Name	Data Type	Description
max_value	varchar	the maximum value for this column (currently, only available for the 'ts' column)
error_type	varchar	message(s) in case there are inconsistencies across hosts in the cluster
column_filter	varchar	the name of each filter on the column, separating multiple filters with commas

<namespace>.storage.metadata_history

This table provides historical information about the tables and columns in the specified namespace. If you specify an empty namespace ("") when you run `atquery` against this table, the table returns all tables and columns throughout your EDW instance.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row. 'COLUMN_INFO' =
cluster_name	varchar	the installation name
namespace	varchar	the complete namespace name for this table
table_name	varchar	the table name
column_name	varchar	the column name
column_type	varchar	the data type for this column
primary_key	boolean	is this the primary key? typically, only the 'ts' (timestamp) column is a key
min_value	varchar	the minimum value for this column (currently, only available for the 'ts' column)
max_value	varchar	the maximum value for this column (currently, only available for the 'ts' column)
error_type	varchar	message(s) in case there are inconsistencies across hosts in the cluster
column_filter	varchar	the name of each filter on the column, separating multiple filters with commas

<namespace>.storage.raw_metadata

This table provides configuration information about an entire EDW instance. For example, you can query this table to return the name, host (node), port and relationship of every host in your EDW instance.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row. 'COLUMN_INFO' =

Column Name	Data Type	Description
node_index	int32r	identifies the host in the cluster
dsm_sib_idx	int32	identifies sibling relationships between hosts
cluster_name	varchar	the SLS instance name
node_name	varchar	the host "name" (typically the same as the DNS host name, but not required to be)
node	varchar	the DNS host name for a host, can vary if the self-reported name differs from the name that other hosts use to address it
port	int32	port number the host is listening on.
partition	varchar	Specifies either the Primary or Secondary directory
namespace	varchar	the complete namespace name for this table
table_name	varchar	the table name
column_name	varchar	the name of each filter on the column, separating multiple filters with commas
column_type	varchar	the column name
primary key	boolean	False because there is no TS
column filter	varchar	the name of each filter on the column, separating multiple filters with commas
error_type	varchar	message(s) in case there are inconsistencies across hosts in the cluster

<namespace>.storage.raw_metadata_with_ts

This table provides configuration information about an entire EDW instance. For example, you can query this table to return the name, host (node), port and relationship of every host in your EDW instance.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row. 'COLUMN_INFO' =
node_index	int32r	identifies the host in the cluster
dsm_sib_idx	int32	identifies sibling relationships between hosts
cluster_name	varchar	the SLS instance name
node_name	varchar	the host "name" (typically the same as the DNS host name, but not required to be)
node	varchar	the DNS host name for a host, can vary if the self-reported name differs from the name that other hosts use to address it
port	int32	port number the host is listening on.
partition	varchar	Specifies either the Primary or Secondary directory
namespace	varchar	the complete namespace name for this table
table_name	varchar	the table name

Column Name	Data Type	Description
column_name	varchar	the name of each filter on the column, separating multiple filters with commas
column_type	varchar	the column name
primary key	boolean	T if TS value is present, F if TS value is not present.
column filter	varchar	the name of each filter on the column, separating multiple filters with commas
min_value	varchar	the minimum value for this column
max_value	varchar	the maximum value for this column
error_type	varchar	message(s) in case there are inconsistencies across hosts in the cluster

<namespace>.<tablename>.storage.metadata

For each table, there exists a virtual table named *<tablename>.storage.metadata* which contains information about the columns in this table. Prefix the table with a namespace to query a table from a specific namespace other than the default.

Column Name	Data Type	Description
row_type	varchar	type of data represented by this row. 'COLUMN_INFO' =
cluster_name	varchar	the SLS instance name
namespace	varchar	the complete namespace name for this table
table_name	varchar	the table name
column_name	varchar	the column name
column_type	varchar	the data type for this column
primary_key	boolean	indicates whether this the primary key; typically, only the 'ts' (timestamp) column is a key
min_value	varchar	the minimum value for this column (currently, only available for the 'ts' column)
max_value	varchar	the maximum value for this column (currently, only available for the 'ts' column)
error_type	varchar	message(s) in case there are inconsistencies across hosts in the cluster

INDEX

Symbols

__ prefix in SELECT for invisible targets 45
 _fifo() 177
 _fifo() and SLICE BY 52
 _first() 52, 113
 _fromindex() 181
 _fromname() 180
 _if() 95
 _iftable() 96
 _into() 89
 _last() 52, 113
 _lc() 130
 _list() 97
 _lms_buildinfo() 179
 _lms_taskid() 178
 _lookup() 89, 99
 _md5_64() 131
 _md5() 131
 _now() 151
 _nth() 98
 _perl() 89
 _perl(), example 229
 _perlagg() example 233
 _quantize() 173
 _rev_dns() 109, 110
 _sprintf() 148
 _strcat() 147
 _strformat() 148
 _strjoin() 147
 _strleft() 137
 _strlen() 132
 _strlink() 145
 _strlowercase() 130
 _strmatch_strstr() 134
 _strmatchlist() 89, 135
 _strmd5_64() 131
 _strmd5() 131
 _strmiddle() 137
 _strright() 137
 _strsplit() 89, 136
 _strsplitxsv() 89, 136
 _strsum() 113
 _strtrim() 137
 _strupercase() 130
 _substr() 137
 _tablematch() 110
 _time() 151
 _timeadd() 151
 _timediff() 151

_timestart() 151
 _uc() 130
 @_ argument 229
 @INC 232
 * argument 68

A

addamark::dbgPrint() 233
 addamark::setInto() 229
 Aggregate
 functions 52, 113
 partitioning 52
 Aggregates, defining in Perl 226
 architecture 184
 Arrays, list support 229
 AS, named targets 45
 athttpd.conf, listing where Perl modules are located 232
 atmanage
 deleting tables 38
 renaming tables 38
 avg 113
 avg() 113

B

backslash
 syntax usage explained 20
 BETWEEN and IN 68
 bool 68
 BOTTOM 45
 buildinfo 179
 BUILTIN example 226

C

C, calling functions 232
 C++, calling functions 232
 Case expression 68, 77
 Characters
 removing from a string 137
 clauses
 defined 44
 cluster_properties table 264

- Column expressions 65
- column filters
 - defining 25
 - dropping 30
- column names
 - and underscores 206
- Command Line Utilities
 - HawkEye AP SQL 21
- Concatenating strings 147
- Conditional evaluation based on current table 96
- Conditional evaluation with `_if` 95
- configuration variables
 - `xqo.allow_correlated_external_subqueries` 189
 - `xqo.display_translation` 188
 - `xqo.require_whole_query_translation` 189
 - `xqo.translation_logging` 188
 - `xqo.version` 188
- Constants, literal 65
- Conversions
 - data processing expressions 68
- Convert strings to upper or lower case 130
- `CONVERT()` 68, 76
- `count` 113
- `count()` 113
- `count(*)` 52, 68
- creating
 - external tables 190, 197

D

- data types 68, 204
- Debugging messages, printing 233
- defining
 - column filters 25
 - tables 24
 - views 30
- deleting
 - tables 38
- delimiters 206
- Die 229
- `DISTINCT` 45, 68
- DNS
 - reverse lookup 109, 110
- documentation
 - roadmap 17
- dropping
 - column filters 30
 - tables 25
 - views 36
- DURING
 - clause explained 49
- DURING clause 200

E

- Elements
 - picking from a list with `_nth` 98
- encryption 195
- ESX
 - hardware configuration 219
- ESX, ESXi 217
- Evaluation, conditional with `_if` 95
- `example_webserv_100` 43
- Exit and die 229
- EXPLODE 89
 - limitations 89
- EXPLODE BY 89
- expressions
 - CASE 77
 - column 65
 - conversions 68
 - data processing 68
 - data source 65
 - formatting 173
 - in SELECT statements 45
- external modules
 - loading 232
- external query 189
- external tables
 - creating 190
 - creating manually 197

F

- failover 186
- fifo 177
- filters. See column filters
- FIRST 45, 113
- float 68
- Floating point literals 65
- Format
 - strings 148
- Format an expression
 - using a list of bucket specifications 173
- `format_bytes` example 232
- FROM
 - constant list expression 89
 - described 47
- fromindex 181
- fromname 180
- Function syntax 68
- Functions
 - aggregates 52, 113
 - calling those not written in Perl 232
 - concatenating strings 147
 - conditional evaluation based on current table 96
 - convert strings to upper or lower case 130
 - creating a URL from string data 145
 - current table 180

- data processing expressions 68
- formatting expressions 173
- formatting strings 148
- function-style conversions 68
- lookups 99
- MD5 hashes 131
- `oae.sls_qry()` 194, 197, 198, 204, 205, 206, 207
- `oae.tbl()` 194, 204
- parsing a string into a list of fields 136
- picking an element from a list with `_nth` 98
- queue with `_fifo` 177
- removing characters from a string 137
- returning internal information with `_lms_buildinfo` 179
- returning internal information with `_lms_taskid` 178
- reverse dns lookup 109, 110
- storing expressions with `_list` 97
- string length 132
- string matching 134, 135

G

- GROUP BY 52
 - multi-column 52

H

- Hashes
 - creating an MD5 hash from a string 131
- HAVING
 - described 57
- HawkEye AP SQL
 - introduced 21, 23
 - Writing queries 23
- HEREdocs 65, 226
- host operating system 217
- Hostnames
 - converting IPs to 109, 110

I

- identifier
 - defined 44
- if 95
- itable 96
- IN and BETWEEN 68
- IN n PASSES 52
- Including Perl modules 232
- Inline
 - C 232
- Inline.pm 232
- installing
 - Perl modules 235

- int32 68
- int64 68
- Integer literals 65
- INTO 89
- INTO, array and list support 229
- IPs
 - converting to hostname 109, 110
- isolation levels 207

J

- Java, calling functions 232

K

- keywords
 - defined 44

L

- LAST 45, 113
- lc 130
- list 97
- List support 89, 229
- Lists 89
 - acceptors 89
 - generators 89
- literals
 - constants 65
 - defined 44
- Loading an external module into the interpreter 232
- Logging, debugging info to a server-side file 233
- look ups
 - DNS from IP address 109
 - from external data sources 99
 - table names 110
- lookup 99
- Lowercase 130

M

- macros
 - using in Perl subroutines 229
- mapping
 - data types 204
 - namespaces & table names 191
- Math operators 68
- max 113
- MAX example 226
- `max()` 52, 113

- md5 131
- mean.pl 233
- median () 117
- min 113
- min() 52, 113
- Modules, including 232
- Modules, loading external with 'use' 232
- M-ops=exit 229
- Multiple return values 89

N

- namespaces 43
 - mapping 191
 - specifying 186
 - storage.metadata table 269
- now() 151
- nth 98
- Number
 - Format example 232

O

- OAE functions
 - oae.sls_qry() 194, 197, 198, 204, 205, 206, 207
 - oae.tbl() 194, 204
- OAE security 194
- open() 233
- OpenSLS
 - architecture 184
 - process flow 186
 - security 194
- Operators 68
 - comparison 68
 - logical 68
 - math 68
 - precedence 68
- ORDER BY 58
- Overview 197

P

- Parallelism and side effects 229
- Partitioning, aggregates 52
- Perl
 - installing modules 235
- Perl functions
 - about using 21, 225
- Perl modules
 - installing 235
- Perl processing, how it works 225
- perldir= 232

- permissions table 267
- Pick an element from a list 98
- Precedence
 - for operators 68
- Prefixes
 - __ 45
- Printing, debugging messages 233
- Processing Perl in SQL engine 225

Q

- quantize 173
- queries
 - complex 205
- query
 - basic syntax 43
 - required parts 43
- Queue function _fifo 177
- quotation marks 206

R

- raw_upload_info table 268
- renaming
 - tables 38
 - views 36
- replication 186
- RespSize example 232
- Return internal information with _lms_buildinfo 179
- Returning internal information with _lms_taskid 178
- rev_dns 109, 110
- Reverse DNS lookup 109
- rolepermissions table 267
- rolepermissions2 table 267
- roles table 266
- roles2 table 266

S

- security 194
- SELECT
 - BOTTOM 45
 - computed expressions 45
 - DISTINCT 45
 - FIRST 45
 - invisible targets __ 45
 - LAST 45
 - named targets AS 45
 - TOP 45
- Server-side, logging debugging info to 233
- single sign-on 195
- SLICE BY 52

- SLICE BY and `_fifo()` 52
- SORT BY 68
- split 136
- Splitting a group of records with SLICE BY and `_fifo()` 52
- sprintf 148
- SQL 23
 - basic syntax 43
 - command line utilities 21
 - macros 229
- SQL reports
 - `oae.qry()` 197, 206
 - `oae.sls_qry()` 194, 198, 204, 205, 207
 - `oae.tbl()` 194, 204
- `storage.metadata` table 269
- Store expressions into a list with `_list` 97
- strcat 147
- strformat 148
- String length function 132
- String literals 65
- String matching 134, 135
- Strings
 - concatenating 147
 - converting to upper or lower case 130
 - create from time 155
 - creating an MD5 hash from a string 131
 - formatted 148
 - parsing into a list of fields 136
 - removing characters from 137
- strjoin 147
- strleft 137
- strlen 132
- strlink 145
- strlowercase 130
- strmatch 134
- strmatchlist 135
- strmd5 131
- strmiddle 137
- `_strptime()` 159
- strright 137
- strsplit 136
- strsplitxsv 136
- strstr 134
- strsum 113
- strtrim 137
- struppercase 130
- substr 137
- sum() 52
- system
 - `cluster_properties` table 264
 - `permissions` table 267
 - `properties` table 263
 - `raw_upload_info` table 268
 - `rolepermissions` table 267
 - `rolepermissions2` table 267
 - `roles` table 266
 - `roles2` table 266
 - `task_list` table 264
 - `upload_info` table 268

- `userroles` 265
- `userroles2` 266
- `users` table 265
- `users2` table 265
- `system.properties` table 263

T

- Table functions 180
- table names
 - mapping 191
- `tablematch()` 110
- tables
 - conditional evaluation 96
 - defining 24
 - deleting 38
 - renaming 38
 - `storage.metadata` 269
 - `system.cluster_properties` 264
 - `system.permissions` 267
 - `system.properties` 263
 - `system.raw_upload_info` 268
 - `system.rolepermissions` 267
 - `system.rolepermissions2` 267
 - `system.roles` 266
 - `system.roles2` 266
 - `system.task_list` 264
 - `system.upload_info` 268
 - `system.userroles` 265
 - `system.userroles2` 266
 - `system.users` 265
 - `system.users2` 265
- targets
 - in view definitions 32
 - named with AS 45
- `task_list` table 264
- `taskid` 178
- TBLDEF*.xml files 38
- Test scripts 233
- Time functions 151
- time zones
 - conversion functions 251
 - supported 251
- `time()` 151
- `_timef()` 155
- `_timeformat()` 155
- `_timeparse()` 159
- timestamps
 - conversion functions 155
 - performing calculations on 151
- TOP 45
- transaction isolation 207
- Typecast literals 65
- Typecasts
 - alternative syntax 68

U

- uc 130
- upload_info table 268
- Uppercase 130
- URL, creating from string data 145
- use 232
- userroles table 265
- userroles2 table 266
- users table 265
- users2 table 265
- Using 197

V

- varchar 68
- views
 - defining 30

- dropping 36
- renaming 36
- VMware 217

W

- WHERE 51
- WITH, using to declare Perl subroutines 226

X

- xqo.allow_correlated_external_subqueries 189
- xqo.display_translation 188
- xqo.require_whole_query_translation 189
- xqo.translation_logging 188
- xqo.version 188