CCL プログラマーズ・ガイド

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01742-01-0500-01
改訂： 2011 年 12 月
Copyright © 2011 by Sybase, Inc. All rights reserved.
このマニュアルは Sybase ソフトウェアの付属マニュアルであり、新しいマニュアルまたはテクニカル・ノートで特に示されないかぎりは、後続のリリースにも付属します。このマニュアルの内容は予告なしに変更されることがあります。このマニュアルに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、無断で使用することはできません。
このマニュアルの内容を弊社の書面による事前許可を得ずに、電子的、機械的、手作業、光学的、またはその他のいかなる手段によっても、複製、転載、翻訳することを禁じます。
Sybase の商標は、Sybase の商標リスト (http://www.sybase.com/detail?id=1011207) で確認できます。Sybase およびこのリストに掲載されている商標は、米国法人 Sybase, Inc. の商標です。® は、米国における登録商標であることを示します。
このマニュアルに記載されている SAP、その他の SAP 製品、サービス、および関連するロゴは、ドイツおよびその他の国における SAP AG の商標または登録商標です。
Java および Java 関連の商標は、米国およびその他の国における Sun Microsystems, Inc. の商標または登録商標です。
Unicode と Unicode のロゴは、Unicode, Inc. の登録商標です。
このマニュアルに記載されている上記以外の社名および製品名は、当該各社の商標または登録商標の場合があります。
Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.
Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

第 1 章：Sybase Event Stream Processor の概要 1

- イベント・ストリーム 2
- Event Stream Processor とデータベースの比較 2
- データフロー・プログラミング 3
- ESP Projects：アダプタ、ストリーム、ウィンドウ、継続クエリ 4
- ストリームとウィンドウ 5
- スキーマ 6
- 挿入、更新、削除 6
- 製品のコンポーネント 7
- 入力アダプタと出力アダプタ 8
  - カスタム・アダプタ 9
- オーサリング手法 10
- Continuous Computation Language 10
- SPLASH 11

第 2 章：CCL プロジェクトについて 13

- 要素の順序 13
- CCL でのプロジェクトの開発 13
- ストリーム、ウィンドウ、デルタ・ストリームについて 14
  - ストリーム、ウィンドウ、デルタ・ストリームの比較 14
    - 暗黙的なカラム 15
    - 入力／出力／ローカル 16
  - ウィンドウの設定とタイプ 17
    - 名前付きウィンドウ 17
    - 名前なしウィンドウ 18

保持 19
デルタ・ストリームの関数と例 20
スキーマ 21
ストア 22
アダプタの操作 23
入力アダプタを介してのデータ受信 23
出力アダプタを介したデータのパブリッシュ 24

第 3 章：言語コンポーネント 25

大文字と小文字の区別 25
データ型 26
間隔 30
演算子 31
式 35
CCL コメント 36

第 4 章：CCL クエリの構築 39

フィルタリング 39
union 40
例：ストリームまたはウィンドウからのデータのマージ 41
ジョイン 41
キー・フィールド・ルール 43
ジョインの例：ANSI 構文 44
ジョインの例：カンマ区切りの構文 46
パターン一致 47
集約操作 48

第 5 章：高度な CCL プログラミング手法 51

DECLARE ブロック 51
typedef 52

パラメータ 53
変数 54
プロジェクトの変数、パラメータ、データ型、関数の宣言 56
フレックス演算子 57
モジュール性 58
モジュールの作成と使用 59
例：モジュールの作成と使用 59
例：モジュール内のパラメータ 61
永続性 63
ログ・ストアの最適化手法 64
エラー・ストリーム 64
エラー・ストリームのモニタリング 66

第 6 章：文 69

ADAPTER START 文 69
ATTACH ADAPTER 文 70
CREATE DELTA STREAM 文 72
CREATE ERROR STREAM 文 74
CREATE FLEX 文 75
CREATE LIBRARY 文 79
CREATE LOG STORE 文 80
CREATE MEMORY STORE 文 82
CREATE MODULE 文 84
CREATE SCHEMA 文 85
DECLARE 文 86
CREATE STREAM 文 87
CREATE WINDOW 文 89
IMPORT 文 91
LOAD MODULE 文 92

第 7 章：句 95
AGING 句 95
AS 句 97
CASE 句 98
IN 句 98
KEEP 句 100
OUT 句 101
PARAMETERS 句 102
PRIMARY KEY 句 104
SCHEMA 句 105
STORE 句 106
STORES 句 107

第 8 章：クエリ 109
FROM 句 111
FROM 句：カンマ区切りの構文 111
FROM 句：ANSI 構文 112
GROUP BY 句 114
GROUP FILTER 句 115
GROUP ORDER BY 句 116
HAVING 句 117
MATCHING 句 118
ON 句：ジョインの構文 121
SELECT 句 121
UNION 演算子 123
WHERE 句 125

第 9 章：関数 129
ユーザ定義の SPLASH 関数 130

ユーザ定義の外部関数 130
外部 C/C++ 関数の要件 131
例：外部 C/C++ 関数の使用 133
例：Java 関数の使用 134
集合関数 136
any() 136
avg() 137
corr() 138
covar_pop() 139
covar_samp() 140
count() 141
count(distinct) 142
exp_weighted_avg() 142
first() 144
first_value() 145
last() 145
last_value() 146
lwm_avg() 146
max() 147
meandeviation() 148
median() 148
min() 149
nth() 150
recent() 151
regr_avgx() 152
regr_avgy() 153
regr_count() 154
regr_intercept() 154
regr_r2() 155
regr_slope() 156
regr_sxx() 157
regr_sxy() 158
regr_syy() 159
stddev() 160
stddeviation() 160

stddev_pop() 160
stddev_samp() 161
sum() 162
valueinserted() 163
var_pop() 164
var_samp() 164
vwap() 165
weighted_avg() 166
xmlagg() 167
スカラ関数 168
数値関数 168
acos() 168
asin() 169
atan() 169
atan2() 169
avgof() 170
bitand() 171
bitclear() 171
bitflag() 172
bitflaglong() 172
bitmask() 172
bitmasklong() 173
bitnot() 173
bitor() 174
bitset() 174
bitshiftleft() 175
bitshiftright() 175
bittest() 176
bittoggle() 176
bitxor() 177
cbrt() 178
ceil() 178
compare() 178
cos() 179
cosd() 179

cosh() 180
distance() 180
distancesquared() 181
floor() 181
isnull() 182
length() 182
ln() 183
log2() 183
log10() 183
logx() 184
maxof() 184
minof() 185
nextval() 185
pi() 186
power() 186
random() 186
round() 187
sign() 187
sin() 188
sinh() 188
sqrt() 188
tan() 189
tand() 189
tanh() 190
文字列関数 190
int32() 190
left() 191
like() 191
lower() 192
ltrim() 192
patindex() 193
real() 194
regexp_firstsearch() 194
regexp_replace() 195
regexp_search() 196

replace() 196
right() 197
rtrim() 197
string() 198
substr() 198
trim() 199
trunc() 199
upper() 200
変換関数 200
ascii() 200
base64_binary() 201
base64_string() 201
cast() 202
char() 203
concat() 203
extract() 204
fromnetbinary() 204
hex_binary() 205
hex_string() 205
msecToTime() 206
secToTime() 206
timeToMsec() 207
timeToUsec() 207
timeToSec() 208
to_binary() 208
to_bigdatetime() 209
to_boolean() 210
to_date() 210
to_float() 211
to_integer() 212
to_interval() 212
to_long() 213
to_money() 213
to_xml() 214
tonetbinary() 214

to_string() 215
to_timestamp() 217
usecToTime() 217
XML 関数 218
xmlconcat() 218
xmlelement() 218
xmlparse() 219
xmlserialize() 219
日付と時刻の関数 220
business() 220
businessday() 220
date() 221
dateceiling() 221
datefloor() 223
datename() 225
datepart() 225
dateround() 226
dayofmonth() 227
dayofweek() 228
dayofyear() 229
hour() 229
intdate() 230
makebigdatetime() 230
microsecond() 231
minute() 232
month() 232
second() 233
sysdate() 234
systimestamp() 234
unbigdatetime() 234
undate() 235
weekendday() 235
year() 236
その他の関数 236
cacheSize() 236

coalesce() 238
dateint() 238
deleteCache() 239
exp() 241
firstnonnull() 241
get*columnbyindex() 241
get*columnbyname() 242
getCache() 243
getData() 245
getmoneycolumnbyindex() 246
getmoneycolumnbyname() 247
getrowid() 247
now() 248
rank() 248
recordDataToRecord 249
recordDataToString 249
sind() 250
sysbigdatetime() 250
totimezone() 251

付録 A :キーワードのリスト 253

付録 B :日付と時刻のプログラミング 255

タイム・ゾーン 255

タイム・ゾーンのデフォルトの変更 256
タイム・ゾーンのリスト 257

日付/時刻のフォーマット・コード 264

カレンダ・ファイル 268

索引 271

第 1 章　Sybase Event Stream Processor の概要

Sybase® Event Stream Processor を使用すると、独自の複雑なイベント処理 (CEP) アプリケーションを作成および実行し、イベント・データのストリームから連続する情報をリアルタイムに取り出せます。

イベント・ストリーム処理と CEP

イベント・ストリーム処理は CEP の 1 つの形態で、状況を把握するために、イベントについての情報をリアルタイムに分析する手法です。大量のイベント・メッセージが発生した場合、状況を全体的に把握することは困難です。イベント・ストリーム処理を使用すると、イベントがストリームに流入した時点で分析でき、新しく出現する脅威と機会が発生した時点で特定できます。Event Stream Processor サーバで、データのフィルタ処理、集約、要約が行われるので、より完全で時宜を得た情報に基づいてより良い意思決定ができるようになります。

Event Stream Processor はエンドユーザ・アプリケーションではなく、単純なプロジェクトと複雑なプロジェクトの両方の開発および展開を簡易化するツールを提供する実現技術です。これらのプロジェクトを展開する、高度にスケーラブルなランタイム環境を提供します。

開発プラットフォームとしての Event Stream Processor

Event Stream Processor は、CEP プロジェクトを開発するためのプラットフォームとして、イベントの処理方法と解析方法を定義するための高いレベルのツールを提供します。開発者は、ビジュアル指向またはテキスト指向のオーサリング環境で作業できます。受信イベントに適用されるロジックを定義して、以下を実行できます。

- 複数のソースからのデータを組み合わせて、より豊富でより完全な情報を含む、派生イベント・ストリームを生成する。
- 価値の付加された情報を計算して、迅速な意思決定を可能にする。
- 特定の状態またはパターンを監視して、瞬時対応を可能にする。
- 要約データ、統計量、傾向情報などの高いレベルの情報を生成し、多くの個々のイベントの全体像または最終的な影響を把握する。
- 受信データの複雑な分析に基づいて、主要な運用値を連続的に計算する。
- 生データおよび結果データを収集して履歴データベースに格納し、履歴分析および法令遵守に活用する。

Event Stream Processor のランタイム環境

Event Stream Processor はイベント駆動型アーキテクチャ (EDA) のエンジンとして、イベントの吸収、集約、相互の関連付け、分析ができ、応答をトリガできる高い

レベルの新しいイベントと、ビジネスの現状を示す高いレベルの情報を生成できます。 Event Stream Processor では以下のことが行われます。

- 到着したデータを連続して処理する。
- ディスクに格納される前にデータを処理する。これによって、非常に高速なスループットと少ない遅延時間が達成され、より完全で時宜を得た情報に基づくより良い意思決定が可能になります。
- ビジネス・ロジックをデータ管理から分離する。これによって、ビジネス・ロジックの保守が容易になり、総所有コストが削減されます。
- エンタープライズ・クラスのスケーラビリティ、信頼性、セキュリティを提供する。

イベント・ストリーム

ビジネス・イベントは、発生した実際のビジネス・イベントに関する情報を含むメッセージです。多くのビジネス・システムでは、何かが発生すると、イベントが生成されます。

次は、イベント・メッセージのストリームとして送信されることが多いビジネス・イベントの例です。

- 取引イベントと相場イベントを送信する金融市場データ・フィード
- RFID (Radio Frequency Identification System) タグを近くで検知したことを示すイベントを送信する RFID センサ
- ユーザが Web サイトのリンク、ボタン、またはコントロールをクリックするたびにメッセージ (クリック イベント) を送信するクリック・ストリーム
- レコードがデータベースに追加されたりデータベース内のレコードが更新されたりするたびに発生するトランザクション・イベント

多くのアプリケーションは、リアルタイムでイベントを生成するように設計されており、通常、イベントをメッセージ・バスに発行します。このように設計されていないアプリケーションは、Sybase® Replication Server® などのツールを使用してイベントに対応可能です。これらのツールは、トランザクション・ログをモニタし、アプリケーション・データベースの更新に基づいてイベント・ストリームをリアルタイムで生成できます。

Event Stream Processor とデータベースの比較

Sybase Event Stream Processor は従来のデータベースを補完して、連続した、イベント駆動型のデータ分析が必要な新しい種類の問題を解決するのを支援します。

Event Stream Processor は、データベースの代替となるものではありません。データベースは、静的データの格納とクエリ、信頼性の高い方法でのトランザクション

処理に優れています。ただし、データベースは、高速のデータ・ストリームの連続的な分析には効果的ではありません。

- 従来のデータベースでは、すべてのデータをディスク上に格納した後に、処理を開始する必要があります。
- データベースは、事前に登録された継続クエリを使用しません。データベースのクエリは、「1 回かぎり」のクエリです。1 秒間に質問を 10 回行うには、クエリを 1 秒間に 10 回発行する必要があります。このモデルは、多くのクエリを連続して評価する必要がある場合には機能しません。
- データベースは、段階的な処理を使用しません。Event Stream Processor は、データが到着するごとに、クエリを段階的に評価できます。

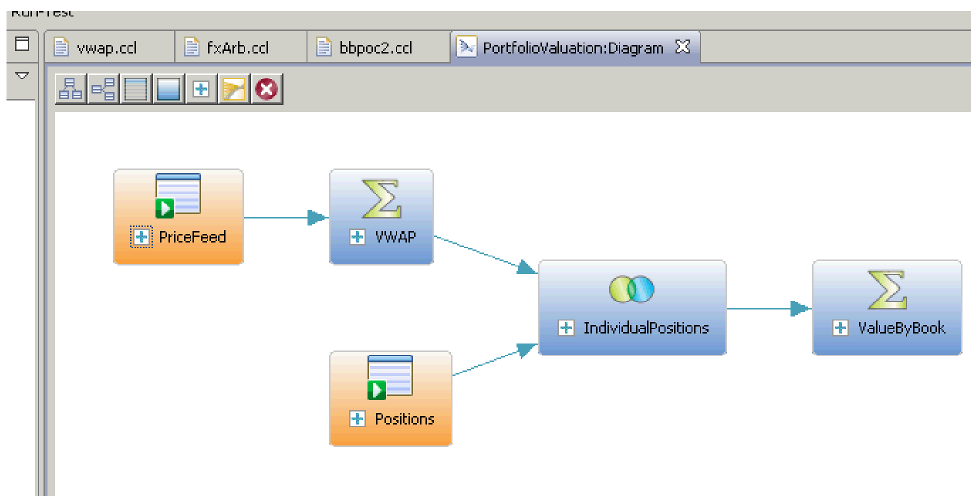
Event Stream Processor は、インメモリ・データベースではありません。インメモリ・データベースは、要求される処理速度を達成するために、メモリ内で動作し、すべてのデータをメモリに保持します。Event Stream Processor は、インメモリ・データベースにいくつかの点で類似していますが、インメモリ・データベースと異なり、オンデマンド・クエリを効果的に処理するように設計されており、連続するイベント駆動型クエリに最適化されたデータフロー・アーキテクチャを使用します。

データフロー・プログラミング

データフロー・プログラミングでは、一連のイベント・ストリームとそれらの間の接続を定義し、データがソースから出力に流れるときの操作をデータに適用します。

データフロー・プログラミングは、複雑になる可能性のある計算を、1 つの操作から次の操作に流れるデータに伴う一連の操作に分割します。 この方法では、各操作がイベント駆動型であり、独立して適用されるので、スケーラビリティと潜在的な並列化がもたらされます。 各操作は異なるスレッドで実行し、他の操作から受信したイベントのみを処理します。操作間で他の調整は必要ありません。

図 **1 :** データフロー・プログラミング



ESP Projects：アダプタ、ストリーム、ウィンドウ、継続クエリ

ESP プロジェクトは、イベント・ストリーム、他の必要なデータソース、結果を生成するために受信イベント・データに適用されるビジネス・ロジックのセットを定義します。

プロジェクトは、最も基本的なレベルで、以下で構成されます。

- **入力ストリームと入力ウィンドウ** – ここから、入力データがプロジェクトに流れ込みます。入力ストリームは、イベント駆動型を基本として受信イベント・データを受信でき、1 回ロードされるか定期的にリフレッシュされるデータの静的セットまたは疑似静的セットも受信できます。
- **アダプタ** – 入力ストリームまたは入力ウィンドウをデータソースに接続します。 Sybase Event Stream Processor には、多くの組み込みアダプタと、カスタム・アダプタを作成するのに使用できる SDK が同梱されています。アダプタは、出力ストリームまたは出力ウィンドウを送信先に接続することもできます。
- **派生ストリームと派生ウィンドウ** – 1 つ以上のストリームまたはウィンドウからデータを取得し、継続クエリを適用して新しいストリームまたはウィンドウを生成します。

ESP プロジェクトからの結果の取得

Event Stream Processor では、実行中のプロジェクトからの出力を以下の 4 つの方法で取得できます。

- アプリケーションは、プロジェクトの作成時にストリームにアタッチされた内部出力アダプタからの情報を自動的に受信します。
- アプリケーションは、外部サブスクライバの手段によってデータ・ストリームにサブスクライブできます。外部サブスクライバは、製品と共に提供されるサブスクリプション API を使用して作成できます。
- ユーザは、実行中のプロジェクトを再設定することなく、そのプロジェクト内のストリームにバインド (接続) する新しいプロジェクトを起動できます。
- ユーザは、esp_query ツールを使用してオンデマンド・クエリを実行し、実行中の ESP プロジェクト内の出力ウィンドウにクエリできます。詳細については、『ユーティリティ・ガイド』を参照してください。

参照：

- *ストリーム、ウィンドウ、デルタ・ストリームについて*(14 ページ)
- *アダプタの操作* (23 ページ)

ストリームとウィンドウ

ストリームとウィンドウの両方が、イベントを処理します。これらの違いは、ウィンドウでは状態は維持されてデータの保持と格納が行えますが、ストリームはステートレスでデータの保持と格納が行えないことです。

ストリームは、ストリームにアタッチされている継続クエリに従って、受信イベントを処理し、出力イベントを生成しますが、データは保持しません。

それとは対照的に、ウィンドウには、受信イベントがローの追加、既存のローの更新、またはローの削除を行うことができるテーブルがあります。ウィンドウのサイズを、時間または記録されるイベントの数に基づいて設定できます。たとえば、過去 20 分間のすべてのイベント、または最新の 1,000 イベントを保持するように、ウィンドウを設定できます。すべてのイベントを保持するようにウィンドウを設定することもできます。この場合、受信イベント・ストリームは自己管理を行い、ウィンドウへのローの挿入とウィンドウからのローの削除の両方を行うイベントを含む必要があります。こうすることで、ウィンドウが無制限に大きくなるのを防げます。

入力、出力、ローカルのストリームとウィンドウ
ストリームとウィンドウは、入力、出力、またはローカルとして指定できます。入力ストリームは、データが外部ソースからアダプタを介してプロジェクトに入力されるポイントです。 プロジェクトには、任意の数の入力ストリームを設定できます。入力ストリームには、継続クエリをアタッチできません。フィルタは定義できます。

ローカルと出力のストリートとウィンドウは、アダプタからではなく、他のストリームまたはウィンドウから入力を取得し、継続クエリを適用して出力を生成します。ローカル・ストリームとローカル・ウィンドウは、外部のサブスクライバ

から隠蔽されていることを除いて、それぞれ出力ストリームと出力ウィンドウに同じです。このため、サブスクライバがサブスクライブ先となるストリームまたはウィンドウを選択する場合に、出力ストリームと出力ウィンドウのみが利用できます。

注意： ビジュアル・オーサリングのパレットには、ローカルと出力のストリームが派生ストリームとして示され、ローカルと出力のウィンドウが派生ウィンドウとして示されます。

参照：

- *ストリーム、ウィンドウ、デルタ・ストリームの比較* (14 ページ)

スキーマ

ストリームまたはウィンドウには、それぞれにスキーマがあります。スキーマは、ストリームまたはウィンドウによって生成されるイベント内のカラムを定義します。

各カラムには、名前とデータ型の属性があります。単一のストリームまたはウィンドウから出力されるすべてのイベントは、同じカラムのセットを持ちます。例を示します。

- RFID リーダから受信される、RFIDRaw と呼ばれる入力ストリームには、文字列データを含む ReaderID と TagID のカラムが含まれることがあります。
- 証券取引所から受信される、Trades と呼ばれる入力ストリームには、Symbol (文字列)、Volume (整数値)、Price (浮動小数点値)、Time (日時値) のカラムが含まれることがあります。

参照：

- *スキーマ*(21 ページ)
- *入力アダプタを介してのデータ受信* (23 ページ)
- *出力アダプタを介したデータのパブリッシュ* (24 ページ)

挿入、更新、削除

オペレーション・コード (opcode) によって、挿入、更新、削除のイベントがウィンドウと関連付けられます。オペレーション・コードは、これらのイベントを自動的に適用することによって、開発を簡略化し、パフォーマンスを向上させます。

Event Stream Processor の多くのユース・ケースでは、イベントは相互に独立しています。各イベントは、発生した事柄に関する情報を伝達します。これらの場合、イベントのストリームは、一連の独立したイベントで構成されます。このタイプ

のイベント・ストリームでウィンドウを定義する場合、受信イベントがそれぞれウィンドウに挿入されます。ウィンドウをテーブルと考えるならば、新規イベントがウィンドウに新しいローとして追加されます。

その他のユース・ケースでは、イベントは以前のイベントに関する新規の情報を伝達します。ESP サーバでは、受信イベントがビューを継続的に更新するため、情報セットのビューを最新の状態に維持する必要があります。2 つの一般的な例を挙げるならば、資本市場における証券の注文帳簿、または充当システムにおける未出荷注文です。両方のアプリケーションで、受信イベントが以下を実行する必要があることを示す場合があります。

- 注文を未出荷注文のセットに追加する。
- 既存の未出荷注文のステータスを更新する。
- キャンセルされた注文または充当された注文を未出荷注文のセットから削除する。

受信イベントによって更新された情報セットを処理するために、Event Stream Processor は受信イベントに関連付けられた挿入、更新、削除の操作を認識します。イベントに opcode (イベントが挿入イベント、更新イベント、または削除イベントのいずれであるかを示す特別なフィールド) を含むタグを追加できます。マッチング・キーを使用して既存のレコードを更新するか、または新規レコードを挿入するアップサート opcode もあります。

入力ウィンドウは、イベントが到着したときに、挿入、更新、削除のイベントをウィンドウのデータに直接適用します。挿入、更新、削除はクエリ・グラフ、つまりすべての下流となるストリーム派生ウィンドウで送信されます。イベントが入力ウィンドウのレコードを更新または削除すると、その操作は自動的にすべての下流となるストリーム派生ウィンドウに適用されます。更新と削除のこのネイティブ処理によって、高いパフォーマンスと簡略化を実現できます。受信イベントを調べ、その受信イベントをウィンドウに適用する方法を決定するために、ユーザが手動でロジックを定義する必要はありません。

製品のコンポーネント

Event Stream Processor には、データのストリームを処理し、相互に関連付けるためのサーバ・コンポーネント、サーバで実行するアプリケーションの開発、テスト、起動ができるスタジオ環境、および管理ツールが同梱されています。

コンポーネントとして以下があります。

- **ESP サーバ** – データのストリームを実行時に処理し、相互に関連付けるソフトウェア。Event Stream Processor は 1 秒間で数十万件のメッセージを処理し、分析できます。クラスタリングによって、ESP サーバのスケールアウト・サポートが提供されます。サーバ・クラスタを使用すると、複数のプロジェクトを同

時に実行できます。また、高い可用性とフェールオーバ機能が達成され、クラスタ接続を管理するためのセキュリティとサポートを一元的に適用できます。

- **ESP スタジオ –** ESP プロジェクトの作成、変更、テストができる統合開発環境です。
- **CCL コンパイラ –** ESP サーバで処理されるようにするためにプロジェクトを翻訳し、最適化するコンパイラです。ESP スタジオまたはコマンド・ラインから起動できます。
- **入力アダプタと出力アダプタ –** Event Stream Processor とデータソースとの間の接続と、ESP サーバと Event Stream Processor からの出力を受け取るコンシューマとの間の接続を確立するコンポーネントです。
- **統合 SDK –** カスタム関数ライブラリを統合し、ライブ・プロジェクトの管理と監視を行うために、カスタム・アダプタを C/C++、Java、.NET で作成するための一連の API です。
- **ユーティリティ –** 多くの管理機能、プロジェクト開発、発行とサブスクリプション、他の機能にコマンド・ラインからアクセスできるようにする一連の実行プログラムです。

入力アダプタと出力アダプタ

Event Stream Processor は、入力アダプタを使用して動的外部ソースと静的外部ソースからメッセージを受信し、出力アダプタを使用して動的外部送信先と静的外部送信先にメッセージを送信します。

外部ソースまたは外部送信先には、以下があります。

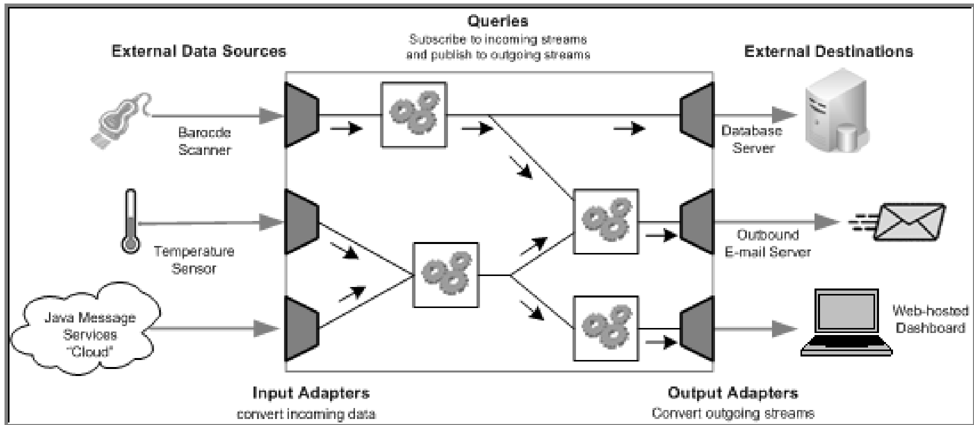
- データ・フィード
- センサ・デバイス
- メッセージング・システム
- RFID (Radio frequency identification) リーダ
- 電子メール・サーバ
- リレーショナル・データベース

入力アダプタは外部データソースに接続し、外部ソースからの受信メッセージを、ESP サーバによって受け付けられるフォーマットに変換します。出力アダプタは、Event Stream Processor によって処理されたローを外部送信先と互換性のあるメッセージ・フォーマットに変換し、それらのメッセージをダウンストリームに送信します。

以下の図は、温度センサ、バー・コード・スキャナ、JMS (Java Message Service) クラウドからのメッセージを、Event Stream Processor と互換性のあるフォーマットに変換する、一連の入力アダプタを示しています。データが Event Stream Processor 内のさまざまなクエリを使用して処理されると、出力アダプタが生成されたロー

を外部のデータベース・サーバ、電子メール・サーバ、Web サービス・ダッシュボードに送信されるアップデートに変換します。

図 2 : Event Stream Processor 内のアダプタ



Event Stream Processor に同梱されているアダプタの完全なリストについては、『アダプタ・ガイド』を参照してください。

カスタム・アダプタ

Event Stream Processor で提供されるアダプタ以外に、独自のアダプタを作成してサーバに統合できます。標準のアダプタが管理できないさまざまな外部要件を処理するように、アダプタを設計できます。

Event Stream Processor ではさまざまな SDK が用意されており、以下を含む多くのプログラム言語でアダプタを作成できます。

- C
- C++
- Java
- .NET (C#、Visual Basic など)

カスタム・アダプタを作成する方法の詳細については、『アダプタ・ガイド』を参照してください。これらの SDK でサポートされているバージョンについては、『インストール・ガイド』を参照してください。

オーサリング手法

Event Stream Processor スタジオは、ビジュアル・オーサリング環境とテキスト・オーサリング環境を提供します。

ビジュアル・オーサリング環境では、グラフィック・ツールを使用してプロジェクトを開発し、ストリームとウィンドウの定義、それらの接続、入力アダプタと出力アダプタの統合、さまざまな単純なクエリの作成が行えます。

テキスト・オーサリング環境では、任意のテキスト・エディタ内と同様に、CCL (Continuous Computation Language) でプロジェクトを開発できます。データのストリームとウィンドウの作成、クエリの開発、階層モジュールとプロジェクト内でのそれらの編成が行えます。

ビジュアル・エディタと CCL エディタは、いつでも簡単に切り替えられます。一方のエディタで加えた変更は、もう一方のエディタに反映されます。プロジェクトをスタジオ内でコンパイルすることもできます。

ビジュアル・オーサリングとテキスト・オーサリングのコンポーネントに加えて、スタジオには、サンプル・プロジェクトで作業するための環境や、さまざまなデバッグ・ツールを使用してアプリケーションを実行したり、テストしたりするための環境が用意されています。スタジオではまた、プロジェクトのアクティビティの記録と再生、ファイルからのデータのアップロード、手動による入力レコードの作成、サーバへのコマンドの発行、サーバに対するアドホック・クエリの実行が行えます。

コマンド・ラインから作業する場合は、**esp_server**、**esp_client**、**esp_compiler** のコマンドを使用してプロジェクトの開発と実行が行えます。Event Stream Processor のユーティリティの全一覧は、『ユーティリティ・ガイド』を参照してください。

Continuous Computation Language

CCL (Continuous Computation Language) は、Event Stream Processor の主要なイベント処理言語です。ESP プロジェクトは、CCL で定義されます。

CCL は SQL (Structured Query Language) を基本としており、イベント・ストリーム処理用に変更されています。

CCL は高度なデータ選択能力と計算能力をサポートし、以下の機能を提供します。データのグループ化、集約、ジョイン。さらに、CCL には、データ・ストリーム上のウィンドウや、パターンとイベントの一致処理などのリアルタイム連続処理時のデータ操作に必要な機能も用意されています。

CCL を特徴付ける重要な機能は、動的データを連続的に処理する能力です。SQL クエリは通常、データベース・サーバに発行されるごとに 1 回のみ実行され、ユーザまたはアプリケーションがクエリの再実行を必要とするごとに再発行される必要があります。それとは対照的に、CCL クエリは連続しています。プロジェクト内で CCL クエリを定義すると、連続実行として登録され、いつまでもアクティブな状態に維持されます。プロジェクトを ESP サーバで実行すると、登録されているクエリが、そのデータソースの 1 つからデータが到着するごとに実行されます。

CCL では SQL 構文を使用して継続クエリを定義しますが、ESP サーバは SQL クエリ・エンジンを使用しません。その代わりに、CCL を効率の高いバイト・コードにコンパイルします。このコードは、ESP サーバによってデータフロー・アーキテクチャ内の継続クエリを構築するために使用されます。

CCL クエリは、CCL コンパイラによって実行可能な形式に変換されます。一般的に、コンパイルは Event Stream Processor スタジオ内で実行されますが、コマンド・ラインから CCL コンパイラを呼び出しても実行できます。

SPLASH

Stream Processing LAnguage SHell (SPLASH) は、CCL に拡張性をもたらすスクリプト言語で、標準の SQL では提供されないカスタム演算子やカスタム関数の作成を可能にします。

SPLASH スクリプトを CCL に埋め込めるので柔軟性が非常に高まり、それを CCL エディタ内で行うことができるので、ユーザの生産性が大きく向上します。SPLASH を使用すると、手続き型のロジックを使用して複雑な計算を定義できるため、リレーションのパラダイムを使用するより容易に定義を行うことができます。

SPLASH は簡易なスクリプト言語で、他の値から値を計算するために使用される式、変数、ループ構成体で構成されます。また、関数内で命令を編成する機能もあります。SPLASH の構文は、C と Java に似ており、比較的小さなプログラミング問題を解決する言語 (AWK や Perl など) との類似性も持ち合わせています。

参照：

- *DECLARE ブロック* (51 ページ)
- *ユーザ定義の SPLASH 関数* (130 ページ)
- *CREATE FLEX 文* (75 ページ)

第 2 章　CCL プロジェクトについて

プロジェクトを開始する前に、文と句が出現する順序、プロジェクト開発、ストリームとウィンドウの基本について理解しておく必要があります。

要素の順序

句と文の構文の定義と制約に基づいて、CCL プロジェクト要素の順序を決定します。

他の文または句で参照される CCL 要素を、それらの文や句を使用する前に定義します。これを行わないと、コンパイル・エラーが発生します。

たとえば、**CREATE STREAM** でスキーマを名前で参照する前に、**CREATE SCHEMA** 文を使用してそのスキーマを定義します。同じく、CCL の文または句でパラメータまたは変数を参照する前に、DECLARE ブロック内でそれらのパラメータや変数を宣言します。

CCL の文または句の中ではサブ句要素の順序を変更できません。

CCL でのプロジェクトの開発

ESP スタジオで CCL エディタを使用するか、他のサポートされるエディタを使用して、CCL コードの作成、変更を行います。単純なプロジェクトの開発から始めて、複雑な機能を追加するごとにテストを繰り返します。

高いレベルの手順の詳細については、この『CCL プログラマーズ・ガイド』の関連する部分、『スタジオ・ユーザーズ・ガイド』、『アダプタ・ガイド』、『SPLASH プログラマーズ・ガイド』を参照してください。

1. `.ccl` ファイルを作成します。

   ESP スタジオでプロジェクトを作成すると、`.ccl` ファイルは自動的に作成されます。
2. 入力ストリームと入力ウィンドウを追加します。
3. 単純な継続クエリを備えた出力ストリームと出力ウィンドウを追加します。
4. アダプタをストリームとウィンドウにアタッチして、外部ソースにサブスクライブするか、または出力をパブリッシュします。
5. CCL コードをコンパイルします。

6. ESP スタジオのデバッグ・ツールとコマンド・ライン・ユーティリティを使用して、コンパイル済みプロジェクトをテスト・データに対して実行します。
   必要に応じて、この手順を繰り返します。
7. クエリをプロジェクトに追加します。単純クエリを起動し、段階的に複雑な関数を追加します。
8. (オプション) 継続クエリの関数を使用して、算術演算、集約、データ型変換、他のコマンド・タスクを実行します。
   - 多くの共通操作のための組み込み関数
   - SPLASH プログラミング言語で記述されているユーザ定義関数
   - C/C++ または Java で記述されているユーザ定義外部関数
9. (オプション) 名前付きスキーマを作成して、ストリームとウィンドウ用の再利用可能なデータ構造体を定義します。
10. (オプション) メモリ・ストアまたはログ・ストアを作成して、メモリ内またはディスク上にデータ・ウィンドウの状態を保持します。
11. (オプション) モジュールを作成して、プロジェクト内で複数回ロードされる可能性のある再利用可能な CCL を組み込みます。

ストリーム、ウィンドウ、デルタ・ストリームについて

CCL プロジェクトを成功させるには、ストリーム、ウィンドウ、デルタ・ストリームの特徴と違いについて理解することが必須となります。

ストリーム、ウィンドウ、デルタ・ストリームの比較

ストリーム、ウィンドウ、デルタ・ストリームにはさまざまな特性と特徴が用意されていますが、共通の指定、アクセス性、カラム・パラメータもあります。

一般に、「ステートレス」と「ステートフル」という用語が、ウィンドウとストリームの間の最も顕著な違いを説明します。ステートフル要素は情報を格納できますが、ステートレス要素は格納できません。

特徴となる機能	ストリーム	ウィンドウ	デルタ・ストリーム
要素のタイプ	ステートレス	保持機能と格納機能を持つため、ステートフル	ステートレス
データの保持	なし	はい。ロー (保持ポリシーに基づく)	なし

特徴となる機能	ストリーム	ウィンドウ	デルタ・ストリーム
利用可能なストアのタイプ	適用されない	メモリ・ストアまたはログ・ストア	適用されない
この要素から抽出可能な要素のタイプ	集約句 (GROUP BY) を指定したストリームまたはウィンドウ	ストリーム、ウィンドウ、デルタ・ストリーム	ストリーム、ウィンドウ、デルタ・ストリーム
必須プライマリ・キー	なし	あり。明示的または推定的	あり。明示的または推定的
集約操作のサポート	不可	可	不可
update の受信時の動作	受信し、insert を生成	受信し、update を生成	受信し、update を生成
insert の受信時の動作	受信し、insert を生成	受信し、insert を生成	受信し、insert を生成
delete の受信時の動作	受信するが無視	受信し、delete を生成	受信し、delete を生成

ストリーム、ウィンドウ、デルタ・ストリームには、明暗黙的なカラムとアクセス性のルールなど、いくつかの共通する重要な特性があります。

参照：

- *ウィンドウの設定とタイプ*(17 ページ)
- *ストリームとウィンドウ*(5 ページ)

暗黙的なカラム

すべてのストリーム、ウィンドウ、デルタ・ストリームは、ROWID、ROWTIME、BIGROWTIME と呼ばれる 3 つの暗黙的なカラムを使用します。

- ROWID – 受信データの各ローの一意なロー識別番号。
- ROWTIME – 秒単位の精度の日付で表される最後の変更時間。
- BIGROWTIME – マイクロ秒単位の精度で表される最後の変更時間。これらのカラムに基づいて、フィルタと選択を実行できます。たとえば、営業時間外に発生したデータ・ローをすべて除外できます。

入力／出力／ローカル

ストリーム、ウィンドウ、デルタ・ストリームを、入力、出力、またはローカルとして指定できます。

入力／出力のストリームとウィンドウ

入力ストリームと入力ウィンドウは、入力アダプタを使用するか、外部パブリッシャに接続することによって、プロジェクト外のソースからデータを取得できます。入力ウィンドウ、または入力ストリームに直接に、出力アダプタをアタッチするか、外部サブスクライバを接続できます。 また、SQL インタフェースを使用して、入力ウィンドウのローに対する **SELECT**、入力ストリームのローに対する **INSERT**、または、入力ウィンドウのローに対する **INSERT**/**UPDATE**/**DELETE** を実行できます。

出力のウィンドウ、ストリーム、デルタ・ストリームは、データを出力アダプタまたは外部のサブスクライバにパブリッシュできます。 SQL インタフェースを使用して、出力ウィンドウのローに対してクエリ (**SELECT**) を実行できます。

ローカルのストリーム、ウィンドウ、デルタ・ストリームは、プロジェクト外からアクセスできません。また、それらに入力アダプタまたは出力アダプタをアタッチすることもできません。ローカルのストリーム、ウィンドウ、デルタ・ストリームの内容に、サブスクライブまたは SQL インタフェースを使用してクエリできません。

サポートされている SQL 文の詳細については、『ユーティリティ・ガイド』を参照してください。

例

次は、フィルタが指定されている入力ストリームです。

```
CREATE SCHEMA mySchema (Col1 INTEGER, Col2 STRING);
CREATE INPUT STREAM IStr2 SCHEMA mySchema
    WHERE IStr2.Col2='abcd';
```

次は、出力ストリームです。

```
CREATE OUTPUT STREAM OStr1
    AS SELECT l.Col1 col1, l.Col2 col2
    FROM LStr1 l;
```

次は、入力ウィンドウです。

```
CREATE SCHEMA mySchema (Col1 INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE INPUT WINDOW IWin1 SCHEMA mySchema
    PRIMARY KEY(Col1)
    STORE myStore;
```

次は、出力ウィンドウです。

```
CREATE SCHEMA mySchema (Col1 INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE OUTPUT WINDOW OWin1
    PRIMARY KEY (Col1)
    STORE myStore
    AS SELECT  l.Col1 col1, l.Col2 col2
    FROM LWin1 i;
```

ローカルのストリームとウィンドウ
ストリームがアダプタを必要としない場合、または外部接続を許可しない場合は、ローカルのストリーム、ウィンドウ、またはデルタ・ストリームを使用します。ローカルのストリーム、ウィンドウ、デルタ・ストリームは、それらを含む CCL プロジェクトの内部でのみアクセスでき、CCL コンパイラによってより高度に最適化されます。修飾子のないストリームとウィンドウは、ローカルです。

注意： ローカル・ウィンドウはデバッグできません。

例
次は、ローカル・ストリームです。

```
CREATE SCHEMA mySchema (Col1 INTEGER, Col2 STRING);
CREATE LOCAL STREAM LStr1
    AS SELECT i.Col1 col1, i.Col2 col2
    FROM IStr1 i;
```

次は、ローカル・ウィンドウです。

```
CREATE SCHEMA mySchema (Col1 INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE LOCAL WINDOW LWin1
    PRIMARY KEY (Col1)
    STORE myStore
    AS SELECT i.Col1 col1, i.Col2 col2
    FROM IStr1 i;
```

ウィンドウの設定とタイプ

ウィンドウは、名前付きまたは名前なしのステートフル要素で、定義されている保持ポリシーに基づいてローを維持します。

参照：

- *ストリーム、ウィンドウ、デルタ・ストリームの比較* (14 ページ)

名前付きウィンドウ

名前付きウィンドウは、**CREATE WINDOW** 文を使用して明示的に作成され、他のクエリから参照できます。

名前付きウィンドウは、入力、出力、またはローカルのいずれかとして分類できます。入力ウィンドウは、データをアダプタまたはコネクタと他のストリームに送信でき、アダプタまたはコネクタからデータを受信できます。 出力ウィンドウは、データをアダプタまたはコネクタと他のストリームに送信でき、他のスト

リームからのみデータを受信できます。入力ウィンドウと出力ウィンドウは、両方共に外部からアクセスでき、サブスクライブまたはクエリの対象になります。ローカル・ウィンドウはプライベートであり、外部からアクセスできません。ウィンドウに修飾子が指定されていない場合、ローカルのタイプであると想定されます。

表 1 : 名前付きウィンドウの機能

名前付きウィンドウのタイプ	データの送信元	データの送信先	外部からのアクセス
入力	アダプタまたはコネクタ	アダプタまたはコネクタと他のストリーム	可
出力	他のストリーム	アダプタまたはコネクタと他のストリーム	可
ローカル	他のストリーム	他のストリーム	不可

参照：

- *CREATE WINDOW 文* (89 ページ)

名前なしウィンドウ

名前なしウィンドウは、暗黙的に作成されるステートフル要素で、プロジェクトの他の部分では、参照または使用できません。

名前なしウィンドウは、**KEEP** 句が、文の **FROM** 句内でソース名と一緒に使用されると、暗黙的に作成されます。

名前なしウィンドウは、特に、それ自身がステートフルではない、基になるデルタ・ストリーム上に作成されると、メモリを大量に使用する可能性があります。

注意： 名前なしウィンドウは、ソース・ストリームに対して使用できません。ソースがデルタ・ストリームまたはウィンドウである場合のみ使用できます。

例

この例は、`MaxTradePrice` ウィンドウの入力 `Trades` 上に名前なしウィンドウを作成します。このウィンドウは、最新の 10,000 取引に見られるすべての証券コードの取引最高値を追跡します。

```
CREATE WINDOW MaxTradePrice
PRIMARY KEY DEDUCED
STORE S1
AS SELECT trd.Symbol, max(trd.Price) MaxPrice
FROM Trades trd KEEP 10000 ROWS
GROUP BY trd.Symbol
```

次の例は、`Trades` 上に名前なしウィンドウを作成し、`MaxTradePrice` は、過去 10 分間の取引のすべての証券コードの取引最高値を追跡します。

```
CREATE WINDOW MaxTradePrice
PRIMARY KEY DEDUCED
STORE S1
AS SELECT trd.Symbol, max(trd.Price) MaxPrice
FROM Trades trd KEEP 10 MINUTES
GROUP BY trd.Symbol
```

両方の例で、`Trades` は、デルタ・ストリームまたはウィンドウです。

参照：

- *CREATE WINDOW 文* (89 ページ)
- *KEEP 句* (100 ページ)

保持

保持ポリシーは、ステートフル要素に保持されるデータと、その保持期間を定義します。

保持とは、ステートフル要素によってレコードが保持されることを意味します。CCL では、ウィンドウのみがレコードを保持できます。ストリームとデルタ・ストリームはレコードを保持しないので、これらには保持ポリシーを指定できません。

保持は、**KEEP** 句を介して指定します。ウィンドウ内のレコード数を、ウィンドウ内のレコードの数または経過時間のいずれかに基づいて制限できます。これらの手法はそれぞれ、カウント基準の保持ポリシーおよび時間基準の保持ポリシーと呼ばれます。別の手法として、すべてのローの保持、または最後のレコードのみの保持を明示的に選択できます。保持ポリシーが指定されていない場合、ウィンドウは KEEP ALL ポリシーを使用してすべてのレコードを保持します。

注意： 保持は、メモリに基づくストアを使用するウィンドウのみに指定できます。ログ・ファイルに基づくストアの保持は、入力ウィンドウでのみサポートされます。

カウント基準の保持

カウント基準の保持ポリシーでは、整数の定数が、ウィンドウに保持されるローの最大数を指定する式になります。数の式でパラメータを使用できます。

カウント基準の保持ポリシーでもオプションのスラック値を定義できます。このオプションを定義することによって、メモリ・ストアの必要なクリーニングの回数が減るので、パフォーマンスを向上できます。これは、スラック値を使用することによって、ウィンドウ内のロー数が N + S に制限されるためです。ここで、N は保持サイズで、S はスラック値です。ウィンドウ内のロー数が N + S に達すると、システムは S 個のローを消去します。スラックの値が大きいほど、必要なクリーニングの回数が減るので、パフォーマンスが向上します。

スラックのデフォルト値は 1 です。スラックが 1 の場合は、ウィンドウ内のレコードが最大数に達すると、新しいレコードが挿入されるたびに最も古いレコードが削除されます。これは、パフォーマンスに重大な影響を与えます。スラックの値が大きいほど、ローを絶えず削除する必要が減るので、パフォーマンスが向上します。

次の例は、フィルタ条件に一致する最新の 100 レコードを保持するフィルタ・ウィンドウを作成します。スラック値は、ウィンドウに最大 110 のレコードを格納できることを意味します。

```
CREATE WINDOW Last100Trades PRIMARY KEY DEDUCED
KEEP 100 ROWS SLACK 10
AS
SELECT *
FROM Trades
WHERE Trades.Volume > 1000
```

時間基準の保持

時間基準の保持ポリシーでは、定数の間隔式によって、ウィンドウに保持されるローの最大経過時間が指定されます。

次の例は、過去 10 分間に受信したレコードを保持するウィンドウを作成します。

```
CREATE WINDOW RecentPositions PRIMARY KEY DEDUCED
KEEP 10 MINS
AS
SELECT * FROM Positions;
```

参照：

- *永続性* (63 ページ)
- *KEEP 句* (100 ページ)

デルタ・ストリームの関数と例

デルタ・ストリームはステートレス要素で、すべての opcode を処理できます。

デルタ・ストリームは、計算、フィルタ、またはユニオンを使用する場所であればどこでも使用できますが、ステータスを維持する必要はありません。デルタ・ストリームは、ステータスを維持しないので、メモリの使用量が少なく、速度が向上します。このため、これらの演算をウィンドウよりも効率的に実行します。

例

この例は、名前が `DeltaTrades` のデルタ・ストリームを作成します。ここでは、**getrowid** 関数と **now** 関数が使用されています。

```
CREATE LOCAL DELTA STREAM DeltaTrades
    SCHEMA (
        RowId long,
        Symbol STRING,
        Ts bigdatetime,
```

```
        Price MONEY(2),
        Volume INTEGER,
        ProcessDate bigdatetime )
    PRIMARY KEY (Ts)
AS SELECT  getrowid ( TradesWindow) RowId,
        TradesWindow.Symbol,
         TradesWindow.Ts Ts,
         TradesWindow.Price,
         TradesWindow.Volume,
         now() ProcessDate
    FROM TradesWindow

CREATE OUTPUT WINDOW TradesOut
    PRIMARY KEY DEDUCED
AS SELECT * FROM DeltaTrades ;
```

スキーマ

スキーマは、ストリームまたはウィンドウ内のデータ・ローの構造を定義します。

ストリームまたはウィンドウの各ローは同じ構造またはスキーマを持つ必要があります。これには、カラムの名前、カラムのデータ型、カラムの配列順序が含まれます。複数のストリームまたはウィンドウが同じスキーマを使用できますが、1 つのストリームまたはウィンドウは 1 つのスキーマしか持つことができません。

CREATE SCHEMA 文を使用してスキーマを作成し、特定のストリームまたはウィンドウに関連付けます。使用する構文内にインライン・スキーマを作成してストリームおよびウィンドウを作成したり、以降で参照するために名前付きスキーマを個別に作成したりできます。

単純なスキーマ CCL の例

次は、**CREATE SCHEMA** 文の例です。TradeSchema はスキーマの名前を表し、bigdatetime システム・カラムが、ローの挿入された時間を取得します。

```
CREATE SCHEMA TradeSchema (
        Ts bigdatetime,
        Symbol STRING,
        Price MONEY(4),
        Volume INTEGER
);
```

次の例は、**CREATE SCHEMA** 文を使用してインライン・スキーマを作成します。

```
CREATE STREAM trades SCHEMA (
        Ts bigdatetime,
        Symbol STRING,
        Price MONEY(4),
        Volume INTEGER
);
```

参照：

- *CREATE SCHEMA 文* (85 ページ)

ストア

ストアのデフォルトを設定するか、ログ・ストアまたはメモリ・ストアを選択して、ウィンドウからのデータが保存される方法を指定します。

CREATE DEFAULT STORE 文を使用してデフォルトのストアを設定しない場合、各ウィンドウにはデフォルトのメモリ・ストアが割り当てられます。特定のストア・タイプに新しいウィンドウを割り当てない場合、ストア・タイプと場所についてデフォルトのストア設定を使用できます。

ログ・ストア
永続性を実装するにはログ・ストアを使用して、すべてのデータをディスクに記録します。これによって、障害が発生した場合にデータを確実にリカバリできます。

ログ・ストアを作成するには、**CREATE LOG STORE** 文を使用します。**CREATE DEFAULT STORE** 文を使用して、ログ・ストアをデフォルトのストアに設定できます。この設定は、デフォルトのメモリ・ストアを上書きします。

ログ・ストアの依存関係ループは、コンパイル・エラーを発生させるので、ログ・ストアの使用時に問題となることがあります。ログ・ストアのループは、プロジェクト内で複数のログ・ストアを使用し、ウィンドウをこれらのストアに割り当てると発生することがあります。ログ・ストアを使用する場合は、ログ・ストアをソース・ウィンドウのみに割り当てるか、または同じストリーム・パス内のすべてのウィンドウを同じストアに割り当てることをおすすめします。`logstore1` をそれらのウィンドウの `n` に対して使用する場合、別のウィンドウに対しては `logstore2` を使用し、チェーンの以降で再び `logstore1` を使用しないでください。たとえば、Logstore B に割り当てられている Window Y が Logstore A に割り当てられている Window X からデータを取得する場合、Window Y からデータを (直接的または間接的に) 取得するウィンドウは、いずれも Logstore A に割り当てないようにする必要があります。

メモリ・ストア
メモリ・ストアでは永続性は使用されず、データはすべてメモリに保存されます。メモリ・ストアには、プロジェクトが実行されているかぎり、前回のサーバの起動時からのプロジェクトのクエリの状態が保持されます。 クエリの状態はディスクではなくメモリに保持されるため、メモリ・ストアへのアクセスはログ・ストアより高速です。

メモリ・ストアを作成するには、**CREATE MEMORY STORE** 文を使用します。デフォルト・ストアが定義されていない場合、新しいウィンドウは自動的にメモリ・ストアに割り当てられます。

参照：

- *永続性* (63 ページ)
- *CREATE MEMORY STORE 文* (82 ページ)
- *CREATE LOG STORE 文* (80 ページ)

アダプタの操作

Event Stream Processor が提供するアダプタとカスタム・アダプタの前提条件を決定します。

個別のアダプタの構成、データ型のマッピング、スキーマ検出の詳細については、『アダプタ・ガイド』を参照してください。

参照：

- *ATTACH ADAPTER 文* (70 ページ)
- *CREATE SCHEMA 文* (85 ページ)
- *パラメータ* (53 ページ)

入力アダプタを介してのデータ受信

入力アダプタを使用して、システムへの情報を外部データ・ソースから取得します。

ここでは、入力アダプタを ESP サーバにアタッチする前に実行する典型的なタスクの概要と、**ATTACH ADAPTER** 文の概要について説明します。

1. 入力データを評価します。ESP サーバに取り込むデータのセットまたはサブセットを決定します。
2. 入力アダプタを選択します。

   データ・ソースが ESP サーバによってサポートされていないデータ型を使用する場合、ESP サーバはデータを有効なデータ型にマップします。アダプタの関連するマッピングの説明については、『アダプタ・ガイド』を参照してください。
3. アダプタを設定します。
4. 入力ストリームまたは入力ウィンドウを作成し、SCHEMA 句で受信データの構造体を定義します。

5. **ATTACH ADAPTER** 文を使用して、アダプタをサーバ・ストリームまたはサーバ・ウィンドウにアタッチし、アダプタ・プロパティの値を設定します。

   ATTACH ADAPTER 文には、サーバ実行時にアダプタ・プロパティの変更を可能にするパラメータがあります。

   注意： モジュールまたはプロジェクトがロードされたときに、宣言したパラメータのみを新しい値にバインドできます。

   定義したストリームまたはウィンドウを介してソースからの入力が利用できるという想定の下で、モデルの構築を継続します。

出力アダプタを介したデータのパブリッシュ

以下のタスクを実行してから、出力アダプタを外部データ送信先にアタッチします。

1. 出力データを評価します。外部データ送信先に送信するデータのセットまたはサブセットを決定します。
2. 出力アダプタを選択します。

   出力送信先が ESP サーバによってサポートされていないデータ型を使用する場合、ESP サーバはデータを有効なデータ型にマップします。アダプタの関連するマッピングの説明については、『アダプタ・ガイド』を参照してください。
3. アダプタを設定します。
4. 出力ストリームまたは出力ウィンドウを作成し、出力データの構造体を定義します。
5. **ATTACH ADAPTER** 文を使用して、アダプタを出力ストリームまたは出力ウィンドウにアタッチし、アダプタ・プロパティの値を設定します。

   ATTACH ADAPTER 文には、サーバ実行時にアダプタ・プロパティの変更を可能にするパラメータがあります。

   注意： モジュールまたはプロジェクトがロードされたときに、宣言したパラメータのみを新しい値にバインドできます。

   データは出力ストリームまたは出力ウィンドウを通して流れ、アダプタによって外部データ送信先に送信されます。

第 3 章　言語コンポーネント

CCL プロジェクトで言語を適切に使用するために、CCL での大文字と小文字の区別、サポートされるデータ型、演算子、式について理解を深めます。

大文字と小文字の区別

一部の CCL 構文要素には、大文字と小文字が区別される名前があります。その他では区別されません。

すべての識別子で、大文字と小文字が区別されます。これには、ストリーム、ウィンドウ、パラメータ、変数、スキーマ、カラムの名前があります。キーワードには大文字と小文字の区別はありません。キーワードは識別子の名前として使用できません。アダプタのプロパティにも、大文字と小文字の制限があります。

ほとんどの組み込み関数の名前 (キーワードであるものを除きます) とユーザ定義の関数で、大文字と小文字が区別されます。次の組み込み関数では、大文字と小文字が区別されません。

- setOpcode、setopcode
- getOpcode、getopcode
- setRange、setrange
- setSearch、setsearch
- copyRecord、copyrecord
- deleteIterator、deleteiterator
- getIterator、getiterator
- resetIterator、resetiterator
- businessDay、businessday
- weekendDay、weekendday
- expireCache、expirecache
- insertCache、insertcache
- keyCache、keycache
- getNext、getnext
- getParam、getparam
- dateInt、dateint
- intDate、intdate
- uniqueId、uniqueid

- LeftJoin、leftjoin
- valueInserted、valueinserted

例
'aVariable' と 'AVariable' の 2 つの変数は、異なる変数として扱われるので、同じコンテキスト内に存在できます。同様に、大文字と小文字の異なる同じ名前を使用して、異なるストリームまたはウィンドウを定義できます。

参照：

- *付録 A、「キーワードのリスト」*(253 ページ)

データ型

Sybase Event Stream Processor は、すべてのコンポーネントで整数、浮動小数点数、文字列、通貨、長整数、タイムスタンプのデータ型をサポートします。

データ型	説明
`integer`	32 ビット符号付き整数。許可される値の範囲は、-2147483648 ～ +2147483647 (-2^{31} ～ 2^{31-1}) です。この範囲を超える定数値は、自動的に長整数データ型として処理されます。 変数、パラメータ、またはカラムを -2147483648 の値で初期化するには、(-2147483647) -1 を指定して CCL コンパイラ・エラーの発生を防ぎます。
`long`	64 ビット符号付き整数。許可される値の範囲は、-9223372036854775808 ～ +9223372036854775807 (-2^{63} ～ 2^{63-1}) です。 変数、パラメータ、またはカラムを -9223372036854775808 の値で初期化するには、(-9223372036854775807) -1 を指定して CCL コンパイラ・エラーの発生を防ぎます。
`float`	倍精度の 64 ビット浮動小数点数。許可される値の範囲は、約 -10^{308} ～ $+10^{308}$ です。
`string`	UTF-8 でエンコードされたバイト値で表される可変長の文字列。最大文字列長は、プラットフォームによって異なりますが、最大で 65,535 バイトです。
`money`	世界的規模の符号付き 64 ビット整数。入力データ・ストリームでは、通貨記号とカンマはサポートされません。

データ型	説明
money(n)	小数点以降に 1 ～ 15 桁のさまざまな精度をサポートする符号付き 64 ビット整数。入力データ・ストリームでは、通貨記号とカンマはサポートされません。小数点はサポートされます。 サポートされる値の範囲は、指定される精度によって異なります。 money(1)： -922337203685477580.8 ～ 922337203685477580.7 money(2)： -92233720368547758.08 ～ 92233720368547758.07 money(3)： -9223372036854775.808 ～ 9223372036854775.807 money(4)： -922337203685477.5808 ～ 922337203685477.5807 money(5)： -92233720368547.75808 ～ 92233720368547.75807 money(6)： -92233720368547.75808 ～ 92233720368547.75807 money(7)： -922337203685.4775808 ～ 922337203685.4775807 money(8)： -92233720368.54775808 ～ 92233720368.54775807 money(9)： -9223372036.854775808 ～ 9223372036.854775807 money(10)： -922337203.6854775808 ～ 922337203.6854775807 money(11)： -92233720.36854775808 ～ 92233720.36854775807 money(12)： -9223372.036854775808 ～ 9223372.036854775807 money(13)： -922337.2036854775808 ～ 922337.2036854775807 money(14)： -92233.72036854775808 ～ 92233.72036854775807 money(15)： -9223.372036854775808 ～ 9223.372036854775807 変数、パラメータ、またはカラムを -92233.72036854775807 の値で初期化するには、(-9...7) -1 を指定して CCL コンパイラ・エラーの発生を防ぎます。 通貨型定数の精度を明示的に指定するには、**Dn** 構文を使用します (**n** は精度を表します)。たとえば、100.1234567D7、100.12345D5 というように指定します。 money(n) 型間の暗黙的な変換は、範囲または精度が失われる可能性があるのでサポートされません。異なる精度を持つ通貨型で作業するには、**cast** 関数を実行します。

データ型	説明
bigdatetime	マイクロ秒精度のタイムスタンプ。デフォルト・フォーマットは、YYYY-MM-DDTHH:MM:SS:SSSSSS です。 すべての数値データ型は、暗黙的に bigdatetime にキャストされます。 変換の規則は、一部のデータ型で異なります。 • すべての boolean 値、integer 値、long 値は、元のフォーマットから bigdatetime に変換される。 • money(n) 値と float 値では、整数部分のみが bigdatetime に変換される。精度を維持して money(n) 値と float 値を bigdatetime に変換するには、**cast** 関数を使用する。 • すべての date 値は bigdatetime フォーマットを満たすために、1000000 で乗算されてから、マイクロ秒単位に変換される。 • すべての timestamp 値は bigdatetime フォーマットを満たすために、1000 で乗算されてから、マイクロ秒単位に変換される。
timestamp	ミリ秒精度のタイムスタンプ。デフォルト・フォーマットは、YYYY-MM-DDTHH:MM:SS:SSS です。
date	ミリ秒精度の日付。デフォルト・フォーマットは、YYYY-MM-DDTHH:MM:SS:SSS です。

データ型	説明
interval	2 つのタイムスタンプ間のマイクロ秒数を表す符号付き 64 ビット整数。interval は、複数の単位をスペースで区切って、たとえば、"5 Days 3 hours 15 Minutes" と指定します。間隔カラムに送信される外部データは、マイクロ秒単位であることが想定されます。単位指定は、string データとの間で変換される interval 値ではサポートされていません。 interval を指定する場合、その間隔は、マイクロ秒数に変換された場合に 64 ビット整数 (long) で表される必要があります。各 interval 単位について、マイクロ秒に変換された場合に長整数で表される最大値は、以下のとおりです。 • MICROSECONDS (MICROSECOND、MICROS)： +/- 9223372036854775807 • MILLISECONDS (MILLISECOND、MILLIS)： +/- 9223372036854775 • SECONDS (SECOND、SEC)： +/- 9223372036854 • MINUTES (MINUTE、MIN)： +/- 153722867280 • HOURS (HOUR、HR)： +/- 2562047788 • DAYS (DAY)： +/- 106751991 カッコ内の値は、interval 単位の代替名です。 単位の最大値を指定した場合、他の単位は指定できません。指定すると、オーバフローが発生します。 各単位は 1 回のみ指定できます。
binary	生のバイナリ・バッファを表す。値の最大長はプラットフォームによって異なりますが、最大で 65535 バイトです。 NULL 文字を使用できます。
boolean	値は、true または false。boolean の許可される範囲外の値のフォーマットは、0/1/false/true/y/n/on/off/yes/no です。これらは大文字と小文字が区別されません。

注意： サポートされないデータ型はサポートされるデータ型にマップされますが、各データ型マッピングは、各アダプタ・タイプによって異なります。使用しているアダプタのデータ型のマッピングについては、『アダプタ・ガイド』の「データ型のマッピング」を参照してください。

参照：

- *cast()* (202 ページ)

間隔

間隔構文は、日、時、分、秒、ミリ秒、マイクロ秒の値をサポートします。

間隔は、64 ビットの精度を使用して、2 つのタイムスタンプ間の経過時間を測定します。すべての間隔は、次の定義を参照します。

```
value | {value [ {DAY[S] | {HOUR[S] | HR} | MIN[UTE[S]] | SEC[OND[S]]
| {MILLISECOND[S] | MILLIS} | {MICROSECOND[S] | MICROS} ]  [...]}
```

value のみが指定されている場合、タイムスタンプのデフォルトは、MICROSECOND[S] です。スペースで区切ることによって複数の時間単位を指定できますが、各単位は 1 回しか指定できません。たとえば、HOUR[S]、MIN[UTE[S]]、SEC[OND[S]] の値を指定すると、これらの値は間隔構文で再び指定できません。

各単位には、別の単位と組み合わされていない場合の最大値があります。

時間単位	許可される最大値
MICROSECOND[S] ｜ MICROS	9,223,372,036,854,775,807
MILLISECOND[S] ｜ MILLIS	9,233,372,036,854,775
SEC[OND[S]]	9,223,372,036,854,775
MIN[UTE[S]]	153,722,867,280,912
HOUR[S] ｜ HR	2,562,047,788,015
DAY[S]	106,751,991,167

これらの最大値は、単位を組み合わせると小さくなります。

時間単位を使用して value を指定する場合、正の値である必要があります。value が負の場合、式として扱われます。つまり、間隔構文での -10 MINUTES は -(10 MINUTES) として扱われます。同じく、10 MINUTES-10 SECONDS は、(10 MINUTES)-(10 SECONDS) として扱われます。

時間単位は、CCL 内でのみ指定できます。間隔カラムの値を API またはアダプタを使用して指定する場合、数値だけをマイクロ秒単位でのみ指定できます。

例

```
3 DAYS, 1 HOUR, 54 MINUTES
```

```
2 SECONDS, 12 MILLISECONDS, 1 MICROSECOND
```

演算子

CCL では、さまざまなタイプの数値演算子、非数値演算子、論理演算子がサポートされています。

算術演算子
算術演算子は、数値の符号切り替え、加算、減算、乗算、または除算に使用します。数値型に適用できますが、数値型の混在もサポートされます。算術演算子では 1 つまたは 2 つの引数を指定できます。単項演算子は引数と同じデータ型を返します。二項演算子は、数値的に優先順位が最も高い引数を選び、残りの引数をそのデータ型に暗黙的に変換して、その型を返します。

演算子	意味	使用例
+	加算	3+4
-	減算	7-3
*	乗算	3*4
/	除算	8/2
%	剰余 (余り)	8%3
^	指数	4^3
-	符号切り替え	-3
++	インクリメント 前置インクリメント (++*argument*) 値は、増分されてから引数として渡されます。 後置インクリメント (*argument*++) 値は、渡されてから増分されます。	++a (前置インクリメント) a++ (後置インクリメント)
−−	デクリメント 前置デクリメント (--*argument*) 値は、減分されてから引数として渡されます。 後置デクリメント (*argument*--) 値は、渡されてから減分されます。	--a (前置デクリメント) a-- (後置デクリメント)

比較演算子

比較演算子は、2 つの式を比較します。比較の結果は、TRUE、FALSE、または null となります。

比較演算子は、次の構文を使用します。

```
expression1 comparison_operator expression2
```

演算子	意味	使用例
=	等しい	a0=a1
!=	等しくない	a0!=a1
<>	等しくない	a0<>a1
>	より大きい	a0!>a1
>=	以上	a0!>=a1
<	より小さい	a0!<a1
<=	以下	a0!<=a1
IN	値のリストのメンバ。値が式リストの値にある場合、結果は TRUE となります。	a0 IN (a1, a2, a3)

論理演算子

演算子	意味	使用例
AND	すべての式が TRUE の場合は TRUE を、その他の場合は FALSE を返す。	(a < 10) AND (b > 12)
NOT	すべての式が FALSE の場合は TRUE を、その他の場合は TRUE を返す。	NOT (a = 5)
OR	いずれかの式が TRUE の場合は TRUE を、その他の場合は FALSE を返す。	(b = 8) OR (b = 6)
XOR	1 つの式が TRUE でその他が FALSE の場合は、TRUE を返す。両方共に TRUE または FALSE の場合、FALSE を返します。	(b = 8) XOR (a > 14)

文字列演算子

演算子	意味	使用例
+	文字列を連結し、別の文字列を返す。 **注意：** + 演算子は、データ型の混在 (整数と文字列など) をサポートしません。	'go' + 'cart'

LIKE 演算子

カラム式と **WHERE** 句式で使用できます。LIKE は LIKE 演算子と REGEXP_LIKE 演算子の使用をサポートします。これらの演算子は、文字列式を、相互に非常に類似しているが正確には一致していない文字列に一致させます。

演算子	構文と意味	使用例
LIKE	**WHERE** 句の文字列式を、相互に非常に類似しているが正確には一致していない文字列に一致させる。 `compare_expression LIKE pattern_match_expression` LIKE 演算子は、**compare_expression** が **pattern_match_expression** に一致する場合は TRUE を、一致しない場合は FALSE を返します。式にはワイルドカードを使用できます。ここで、パーセント記号 (%) は任意の長さの文字列に一致し、アンダースコア (_) は単一の文字に一致します。	Trades.StockName LIKE "%Corp%"

[] 演算子

[] 演算子は、辞書とベクトルのコンテキストでのみサポートされます。

演算子	構文と意味	使用例
[]	ストリームまたはウィンドウ内の現在のローではないローを対象に関数を実行することを可能にする。 `stream-or-window-name[index].column` **stream-or-window-name** は、ストリームまたはウィンドウの名前で、**column** はストリームまたはウィンドウ内のカラムを示します。**index** はリテラル、パラメータ、または演算子を指定できる式で、整数に評価されます。この整数は、現在のローまたはウィンドウのソート順を基準とする、ストリームまたはウィンドウのローを示します。	MyNamedWindow[1].MyColumn

演算子の評価順序

複数の演算子のある式を評価する場合、優先度の高い演算子が低い演算子よりも先に評価されます。優先度が同じである場合、式内で左側から右側への順で評価されます。カッコを使用して演算子の優先度を変更できます。エンジンは、カッコの内側の式をその外側の式よりも先に評価します。

注意： ^ 演算子は、右から左に評価されます。したがって、a ^ b ^ c = a ^ (b ^ c) であり、(a ^ b) ^ c ではありません。

演算子の優先度を以下に示します。同一行の演算子は、同じ優先度を持ちます。

- +.- (単項演算子の場合)
- ^
- *、/、%
- +、- (二項演算子と連結の場合)
- =、!=、<>、<、>、<=、>= (比較演算子)
- LIKE、IN、IS NULL、IS NOT NULL
- NOT
- AND
- OR、XOR

式

式は、値に評価される 1 つ以上の値、演算子、組み込み関数の組み合わせです。

式は、多くの場合、そのコンポーネントのデータ型を想定します。式は、次の場所で使用できます。

- **SELECT** 句の select-list
- **WHERE** 句または **HAVING** 句の条件

式には、単純式と複合式の 2 つのタイプがあります。**length()** または **pi()** などの組み込み関数も、式として扱われます。

単純式

CCL の単純式は、定数、NULL、またはカラムを指定します。定数は、数またはテキスト文字列です。リテラル NULL は null 値を意味します。NULL は他の式の一部にはなりませんが、NULL 自体は式です。

カラム名を単独で、またはそのストリームまたはウィンドウの名前と一緒に指定できます。カラムとストリームまたはウィンドウの両方を指定するには、"stream_name.column_name" という形式を使用します。

単純式の例を次に示します。

- `stocks.volume`
- `'this is a string'`
- `26`

複合式

CCL の複合式は、単純式または複合式の組み合わせです。複合式には、演算子、関数、CCL の単純式 (定数、カラム、NULL) を記述できます。

カッコを使用して、式のコンポーネントの優先順位を変更できます。

複合式の例を次に示します。

- `sqrt (9) + 1`
- `('example' + 'test' + 'string')`
- `( length ('example') *10 ) + pi()`

条件式

CCL の条件式は、一連の条件を評価して、結果を決定します。条件式の出力は、設定されている条件に基づいて評価されます。CCL で、キーワード **CASE** がこれらの式の先頭に配置され、**WHEN-THEN-ELSE** 構造体がこれに続きます。

基本的な構造体を次に示します。

```
CASE
WHEN expression THEN expression
[...]
ELSE expression
END
```

最初の **WHEN** 式は、ゼロまたはゼロ以外のいずれかに評価されます。ゼロは条件が false であることを意味し、ゼロ以外は true を示します。**WHEN** 式が true に評価されると、以降の **THEN** 式が実行されます。条件式は、指定されている順に基づいて評価されます。最初の式が false に評価されると、以降の **WHEN** 式がテストされます。いずれの **WHEN** 式も true に評価されないと、 **ELSE** 式が実行されます。

CCL の有効な条件式の例を次に示します。

```
CASE
WHEN mark>100 THEN grade:=invalid
WHEN mark>49 THEN grade:=pass
ELSE grade:=fail
END
```

参照：

- *HAVING 句* (117 ページ)
- *SELECT 句* (121 ページ)
- *WHERE 句* (125 ページ)

CCL コメント

他のプログラミング言語と同様に、CCL でも、コードの説明を記述するためのコメントを追加できます。

CCL では、2 つのタイプのコメントを使用できます。ドキュメントコメントと通常の複数行コメントです。ドキュメントコメントは、CREATE SCHEMA や CREATE INPUT WINDOW など、CCL のトップレベル文のみで使用します。トップレベル文の直前ではない場所に記述されているドキュメントコメントは、ESP スタジオのビジュアル・エディタによってエラーとして扱われます。

複数行コメントは任意のコンテキストでサポートされているので、ドキュメントコメントよりも複数行コメントの使用をおすすめします。

複数行コメントは `/*` で始まり、`*/` で終了します。例を示します。

```
/*
This is a multi-line comment.
All text within the begin and end tags is treated as a comment.
*/
```

ドキュメントコメントは、`/**` で始まり、`*/` で終了します。例を示します。

```
/**
This is a doc-comment. Note that it begins with two * characters
instead of one. All text within the begin and end tags is treated
as a comment.
*/
CREATE SCHEMA S1 ...
```

ドキュメントコメントが CREATE SCHEMA 文の前にあることに注意してください (これはサンプルとしてのみ提供されており、完全な構文ではありません)。

一般的に、アスタリスク (*) の行を使用して、コード・セクションの境界を示します。例を示します。

```
/**************************************************************
Do not modify anything beyond this point without authorization
**************************************************************/
```

この表記は `/**` で始まるので、CCL によってドキュメントコメントとして扱われます。複数行コメントを使用して同じ効果を得るには、最初の 2 つのアスタリスクをスペースで区切ります (`/* *`)。

第 4 章 CCL クエリの構築

CCL クエリを使用して、どのデータを派生要素 (ストリーム、ウィンドウ、デルタ・ストリーム) に受け入れるかを指定します。クエリを構築して、データのフィルタリング、複数クエリの組み合わせ、複数データソースのジョイン、パターン一致ルールの使用、データの集約を実行できます。

クエリは派生要素に対してのみ使用でき、1 つのクエリのみを派生要素にアタッチできます。CCL クエリは、派生要素の該当する情報を示す複数の句の組み合わせで構成されます。クエリは、派生要素のデータを指定するために、AS 句と一緒に使用されます。

フィルタリング

派生要素 (ストリーム、ウィンドウ、またはデルタ・ストリーム) によって処理されるデータをフィルタするには、CCL クエリで **WHERE** 句を使用します。

WHERE 句とフィルタ式を使用して、派生要素によって受け入れられる受信データをフィルタできます。**WHERE** 句は、**SELECT** 句によって取得されるデータを制限し、生成される結果の数を減らします。**WHERE** 句で指定した値に一致するデータのみが、派生要素に送られます。

派生要素の出力は、入力からのレコードのサブセットから構成されます。 各入力レコードは、フィルタ式に対して評価されます。フィルタ式が false (0) と評価される場合、そのレコードは派生要素の一部にはなりません。

この例では、`IBMTrades` という新しいウィンドウを作成します。このウィンドウに出力されるローは、`Trades` からの「`IBM`」という証券コードを持つローのみです。

```
CREATE WINDOW IBMTrades
    PRIMARY KEY DEDUCED
    AS SELECT * FROM Trades WHERE Symbol = 'IBM';
```

参照：

- *union* (40 ページ)
- *ジョイン* (41 ページ)
- *パターン一致* (47 ページ)
- *集約操作* (48 ページ)
- *WHERE 句* (125 ページ)

• *式*(35 ページ)

union

2 つ以上のクエリの結果を 1 つの結果に結合するには、CCL クエリで **UNION** 演算子を使用します。

2 つ以上のクエリを結合するとき、重複するローは、指定しないかぎり、結果セットから削除されます。

UNION 演算子の入力は、1 つまたは複数のストリームまたはウィンドウから取得されます。その出力は、入力のユニオンを表すレコードのセットです。 次の例は、`InStocks` と `InOptions` という 2 つのウィンドウの間での単純なユニオンを示します。

```
CREATE INPUT WINDOW InStocks
    SCHEMA StocksSchema
    Primary Key (Ts)
;

CREATE INPUT WINDOW InOptions
    SCHEMA OptionsSchema
    Primary Key (Ts)
;
CREATE output  Window  Union1
    SCHEMA OptionsSchema
    PRIMARY KEY DEDUCED
    AS SELECT s.Ts as Ts, s.Symbol as StockSymbol,
           Null as OptionSymbol, s.Price as Price, s.Volume as
Volume
     FROM InStocks s
UNION
    SELECT s.Ts as Ts, s.StockSymbol as StockSymbol,
           s.OptionSymbol as OptionSymbol,  s.Price as Price,
           s.Volume as Volume
    FROM InOptions s
;
```

参照：

• *フィルタリング*(39 ページ)
• *ジョイン*(41 ページ)
• *パターン一致* (47 ページ)
• *集約操作* (48 ページ)
• *UNION 演算子* (123 ページ)

例：ストリームまたはウィンドウからのデータのマージ

UNION 句を使用して、2 つのストリームまたはウィンドウからのデータをマージし、派生要素 (ストリーム、ウィンドウ、またはデルタ・ストリーム) を生成します。

1. 新しいウィンドウを作成します。

```
CREATE WINDOW name
```

   新しいストリームまたはデルタ・ストリームも作成できます。

2. プライマリ・キーを指定します。

```
PRIMARY KEY (…)
```

3. ユニオンの最初の派生要素を指定します。

```
SELECT * FROM StreamWindow1
```

4. **UNION** 句を追加します。

```
UNION
```

5. ユニオンの 2 つ目の派生要素を指定します。

```
SELECT * FROM StreamWindow2
```

ジョイン

複数のデータソースを 1 つのクエリに結合するには、CCL クエリでジョインを使用します。

ストリーム、ウィンドウ、またはデルタ・ストリームは、ジョインを構成できます。ただし、デルタ・ストリームをジョインに構成できるのは、 **KEEP** 句。1 つのジョインには、任意の数のウィンドウとデルタ・ストリームを (対応する **KEEP** 句を使用して) 構成できますが、ストリームは 1 つしか構成できません。セルフ・ジョインもサポートされます。たとえば、各インスタンスにエイリアスを指定することによって、同じウィンドウまたはデルタ・ストリームを 1 つのジョインに複数回構成できます。

ストリームとウィンドウのジョインでは、ターゲットとして、集約が指定されたストリームまたはウィンドウを指定できます。ストリームとウィンドウのジョインはキーを指定せず、ウィンドウはキーを必要とするので、ウィンドウをターゲットとして使用するには集約が必要です。集約時に、**GROUP BY** カラムが、ターゲット・ウィンドウのキーを自動的に生成します。この制限は、デルタ・ストリームとウィンドウのジョインには適用されません。これは、 **KEEP** 句 を使用すると、デルタ・ストリームが名前なしウィンドウに変換されるためです。

ジョインはペアで実施されますが、複数のジョインを結合して、複数テーブルの複雑なジョインを生成できます。ジョインの複雑性と特性に応じて、コンパイラ

は中間ジョインを作成することがあります。カンマ・ジョイン構文は内部ジョインのみをサポートしており、この構文での **WHERE** 句はオプションです。省略すると、**FROM** 句内のストリーム間に多対多の関係があることを意味します。

Event Stream Processor では、次のすべてのジョイン・タイプがサポートされます。

ジョイン・タイプ	説明
内部ジョイン	ジョインがレコードを生成するには、ジョインの両側から 1 つずつのレコードが必要である。
左外部ジョイン	レコードが右側 (内側) にあるかどうかにかかわらず、ジョインの左側 (外側) からレコードが生成される。右側にレコードが存在しない場合、内側からのカラムは NULL 値になります。
右外部ジョイン	左外部ジョインとは反対に、右側がジョインの外側で、左側がジョインの内部である。
全外部ジョイン	ジョインの右側または左側に一致があるかどうかにかかわらず、レコードが生成される。

Event Stream Processor では、以下のカーディナリティもサポートされます。

タイプ	説明
1 対 1	ジョインの一方の側のキーは、ジョインの他方の側のキーに完全にマッピングされる。1 つの受信ローから出力として生成されるのは、1 つのローのみです。
1 対多	「1」の側からの 1 つのレコードは、「多」の側の複数のレコードとジョインする。ジョインの「1」の側とは、すべてのプライマリ・キーがジョインの他方の側にマッピングされる側です。レコードがジョインの「1」の側に来ると常に、多数のローが出力として生成されます。
多対多	ジョインの両側のキーは、ジョインの他方の側のキーに完全にはマッピングされない。ジョインのいずれかの側にローが来ると、複数のローが出力として生成される可能性があります。

この例では、**FROM** 句を ANSI 構文で使用して、2 つのウィンドウ (`InStocks` と `InOptions`) をジョインします。結果は出力ウィンドウです。

```
CREATE INPUT Window InStocks SCHEMA StocksSchema Primary Key (Ts)  ;

CREATE INPUT Window InOptions SCHEMA OptionsSchema Primary Key (Ts)
KEEP ALL;

CREATE Output Window OutStockOption  SCHEMA OutSchema
    Primary Key (Ts)
```

```
    KEEP ALL
AS
    SELECT InStocks.Ts Ts,
        InStocks.Symbol Symbol,
        InStocks.Price StockPrice,
        InStocks.Volume StockVolume,
        InOptions.StockSymbol StockSymbol,
        InOptions.OptionSymbol OptionSymbol,
        InOptions.Price OptionPrice,
        InOptions.Volume OptionVolume
    FROM InStocks JOIN InOptions
      ON
        InStocks.Symbol = InOptions.StockSymbol and
                    InStocks.Ts = InOptions.Ts ;
```

参照：

- *フィルタリング*(39 ページ)
- *union* (40 ページ)
- *パターン一致* (47 ページ)
- *集約操作* (48 ページ)
- *FROM 句：カンマ区切りの構文* (111 ページ)
- *WHERE 句* (125 ページ)
- *FROM 句：ANSI 構文* (112 ページ)
- *ON 句：ジョインの構文* (121 ページ)
- *KEEP 句* (100 ページ)

キー・フィールド・ルール

キー・フィールド・ルールは、ローの挿入が重複しているかキー・フィールドが null の場合に、ローが拒否されないようにします。

- ターゲットのキー・フィールドは常に、ジョインの「多」の側のキーから完全に抽出される。多対多の関係では、キーはジョインの両側のキーから抽出されます。
- 1 対 1 の関係では、キーはジョインのいずれかの側のキーから完全に抽出される。
- 外部ジョインでは、キー・フィールドはジョインの外側から抽出される。ジョインの外側が関係の「多」の側ではない場合、エラーが生成されます。
- 全外部ジョインでは、すべてのソースとターゲットでキー・カラムの数と型が同一であることが必要である。また、キー・カラムには、対応するキー・カラムをソースに含む **FIRSTNONNULL** 式が必要です。

ジョインの結果がウィンドウである場合、特定のルールによって、ターゲット・ウィンドウのプライマリ・キーを形成するカラムが決定されます。複数テーブルのジョインでは、概念上は各ジョインがペアで生成されてから、ジョインの結果

が別のストリームまたはウィンドウとジョインされ、以下同様の処理が行われるため、同じルールが適用されます。

次の表では、この情報を、ジョイン・タイプに関連して説明しています。

	1 対 1	1 対多	多対 1	多対多
INNER	少なくとも一方の側からのキーが射影リスト (キーが複合キーである場合は、射影リストの組み合わせ) に含まれる必要あり。	右側からのキーが射影リストに含まれる必要あり。	左側からのキーが射影リストに含まれる必要あり。	両側からのキーが射影リストに含まれる必要あり。
LEFT	左側からのキーのみが含まれる必要あり。	不可。	左側からのキーが射影リストに含まれる必要あり。	不可。
RIGHT	右側からのキーのみが含まれる必要あり。	右側からのキーが射影リストに含まれる必要あり。	不可。	不可。
OUTER	キーは、両側からのキーの各ペアに対して **FIRSTNONNULL**() を使用して生成される必要あり。	不可。	不可。	不可。

参照：

- *ジョインの例：ANSI 構文* (44 ページ)
- *ジョインの例：カンマ区切りの構文* (46 ページ)

ジョインの例：ANSI 構文

ANSI 構文を使用したさまざまなジョイン・タイプの例。

以下の例のこれらの入力を参照してください。

```
CREATE INPUT STREAM S1 SCHEMA (Val1S1 integer, Val2S1 integer, Val3S1
string);
CREATE INPUT WINDOW W1 SCHEMA (Key1W1 integer, Key2W1 string, Val1W1
integer, Val2W1 string) PRIMARY KEY (Key1W1, Key2W1);
CREATE INPUT WINDOW W2 SCHEMA (Key1W2 integer,  Key2W2 string, Val1W2
integer, Val2W2 string) PRIMARY KEY (Key1W2, Key2W2);
CREATE INPUT WINDOW W2 SCHEMA (Key1W3 integer,  Val1W3 integer,
Val2W3 string) PRIMARY KEY (Key1W3);
```

簡単な内部ジョイン：1 対 1

このジョインでは、W1 か W2 のいずれかからキーを抽出できます。

```
CREATE OUTPUT WINDOW OW1
PRIMARY KEY  (Key1W2, Key2W2)
SELECT W1.*, W2.*
FROM W1 INNER JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W1.Key2W2
```

簡単な左ジョイン：1 対 1

キーは、左ジョインの外側から抽出されます。値が null である可能性があるので、内側からキーを抽出するのは誤っています。

```
CREATE OUTPUT WINDOW OW1
PRIMARY KEY (Key1W1, Key2W1)
SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W1.Key2W2
```

簡単な全外部ジョイン：1 対 1

すべてのキー・カラム内に、必須の **FIRSTNONNULL** 式があります。

```
CREATE OUTPUT WINDOW OW2
PRIMARY KEY (Key1, Key2)
SELECT FIRSTNONNULL(W1.Key1W1, W2.Key1W2) Key1,
FIRSTNONNULL(W1.Key2W1, W2.Key2W2) Key2, W1.*, W2.*
FROM W1 FULL JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W1.Key2W2
```

簡単な左ジョイン：1 対多

このジョインでは、W2 のキーはすべてマッピングされますが、W1 のキーは 1 つしかマッピングされません。「多」の側は W1 で、「1」の側は W2 です。キーは、「多」の側から抽出されます。

```
CREATE OUTPUT WINDOW OW3
PRIMARY KEY (Key1W1, Key2W1)
SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Val2W1 =
W1.Key2W2
```

簡単な内部ジョイン：多対多

どちら側のキーも完全にマッピングされないので、これは多対多のジョインです。ターゲットのキーは、ジョインを構成するすべてのウィンドウのキーである必要があります。

```
CREATE OUTPUT WINDOW OW3
PRIMARY KEY (Key1W1, Key2W1, Key2W1, Key2W2)
SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Val1W1 = W2.Val11W2 AND W1.Val2W1 =
W1.Val22W2
```

ストリームとウィンドウの簡単な左ジョイン：
左ジョインにストリームが構成されるとき、そのストリームは外側にあります。集約操作も実施する場合を除いて、ターゲットをウィンドウにすることはできません。

```
CREATE OUTPUT STREAM OSW1
SELECT S1.*, W2.*
FROM S1 LEFT JOIN W2 ON S1.Key1S1 = W2.Key1W2 AND W1.Val2W1 =
W1.Key2W2
```

複雑なジョイン
2 つのテーブル間の内部ジョインであるため、OW4 のキーは W1 または W2 のいずれかから抽出できます。

```
CREATE OUTPUT WINDOW OW4
PRIMARY KEY DEDUCED
SELECT S1.*, W1.*, W2.*, W3.*
FROM W1 INNER JOIN (W2 LEFT  JOIN W3 ON W2.Key1W2 = W3.Key1W3) ON
W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2;
```

ストリームとウィンドウの複雑なジョイン
このジョインでは、レコードが S1 に到着した場合のみ、ジョインがトリガされます。また、集約操作があるので、ターゲットは、ストリームに限定するのではなく、ウィンドウにする必要があります。

```
CREATE OUTPUT WINDOW OW5
PRIMARY KEY DEDUCED
SELECT S1.* W1.*, W2.*, W3.*   //Some column expression.
FROM S1 LEFT JOIN (W1 INNER JOIN (W2 LEFT  JOIN W3 ON W2.Key1W2 =
W3.Key1W3) ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2) ON
S1.Val1S1 = W1.Key1.Val1
WHERE W2.Key1W2 = 'abcd'
GROUP BY W1.Key1W1, W1.Key2W2
HAVING SUM(W3.Val1W3) > 10;
```

参照：

- *キー・フィールド・ルール* (43 ページ)
- *ジョインの例：カンマ区切りの構文* (46 ページ)

ジョインの例：カンマ区切りの構文

カンマ区切りの構文を使用した複雑なジョインの例。

このジョインは、カンマ区切りの構文を使用した、3 つのウィンドウからなる複雑なジョインです。**WHERE** 句には、レコードをジョインする条件を指定します。

```
CREATE OUTPUT WINDOW OW4
PRIMARY KEY DEDUCED AS
SELECT W1.*, W2.*, W3.*
FROM W1 w1, W2 w2, W3 w3
```

```
WHERE w1.Key1W1 = w2.Key1W2 AND w1.Key2W2 = w2.Key2W2  AND w1.Key1W1
= w3.Key1W3;
```

参照：

- *キー・フィールド・ルール* (43 ページ)
- *ジョインの例：ANSI 構文* (44 ページ)

パターン一致

CCL クエリで **MATCHING** 句を使用することによって、1 つ以上の派生要素 (ストリーム、ウィンドウ、またはデルタ・ストリーム) から入力を取得し、事前に定義されたパターンが入力データに見つかった場合にレコードを生成します。

パターン一致を使用して、1 つ以上のストリームのレコード間の複雑な関係を定義します。パターンを使用することによって、特定の期間内にイベントが発生したかどうかをチェックし、発生している場合にレコードをダウンストリームのストリームに送信できます。

注意： パターン・ルール・エンジンは、opcode がパターン一致条件の一部として含まれていないかぎり、入力レコードの opcode に関係なくパターン一致を実行します。

次の例は、2 つのパターンの結果のユニオンである出力ストリーム ThreeConsecTrades を生成します。各パターンは、異なる証券取引所を対象にして、同じ株式について相互に 5 秒以内に発生した連続する 3 つの取引を検索します。このストリームの出力は、取引された株式の証券コードと、最新の 3 つの価格です。

```
CREATE OUTPUT STREAM ThreeConsecTrades
AS
SELECT
        T1.Symbol,
        T1.Price Price1,
        T2.Price Price2,
        T3.Price Price3
FROM QTrades T1, QTrades T2, QTrades T3
MATCHING[5 SECONDS: T1, T2, T3]
ON T1.Symbol = T2.Symbol = T3.Symbol
;
```

参照：

- *フィルタリング* (39 ページ)
- *union* (40 ページ)
- *ジョイン* (41 ページ)
- *集約操作* (48 ページ)

- *MATCHING 句* (118 ページ)

集約操作

集約操作は、 **GROUP BY** 句によって設定されたカラムの値に基づいて入力レコードをグループ化し、`min`、`max`、`sum`、`count` などの集合関数を適用して、グループごとに 1 つの出力ローを生成します。

グループ内のレコードは、**GROUP BY** 句で指定されているカラムと同じ値を持ちます。**GROUP BY** 句で指定されたカラムはターゲットのキーを形成するので、**SELECT** 句にも指定されている必要があります。 このため、出力要素のプライマリ・キーは、プライマリ・キーを明示的に指定する代わりに **PRIMARY KEY DEDUCED** 句を使用する必要があります。

GROUP BY 句に加えて、**GROUP FILTER** 句と **GROUP ORDER BY** 句を指定できます。**GROUP ORDER BY** 句を指定すると、指定されたカラムに基づいてグループ内のレコードを並び替えてから、**GROUP FILTER** 句と集合関数が適用されます。レコードを並び替えることによって、`first`、`last`、`nth` などレコードの順序が重要性を持つ集合関数を有効に使用できるようになります。

GROUP FILTER 句は、**GROUP ORDER BY** 句の後に実行され、フィルタ条件を満たさないローをグループから削除します。指定されるフィルタ条件は、**WHERE** 句のフィルタ条件に似ています。ただ 1 つの例外は、特別なランク付け関数を指定できることです。ランク付け関数は、**GROUP ORDER BY** 句と組み合わせて使用されます。**GROUP ORDER BY** 句の実行後、グループ内のすべてのローは、1 から N にランク付けされます。ここで、**GROUP FILTER** 句に `rank() < 11` が指定されているとします。これは、**GROUP ORDER BY** 句で指定されたカラムに基づいて並び替えられた後のグループの最初の 10 ローのみに集合関数が適用されることを意味します。

最後に、オプションで **HAVING** 句も指定できます。**HAVING** 句は、指定されたグループのレコードに集合関数を適用した結果に基づいて、レコードをフィルタします。主な違いは、**HAVING** 句では集約操作が許可されますが、**WHERE** 句では許可されないことです。

注意： **GROUP ORDER BY** 句、**GROUP FILTER** 句、**HAVING** 句は、**GROUP BY** 句と組み合わせた場合のみ指定できます。

例
次の例は、すべての `Symbol` ごとに総取引件数、取引最高値、出来高を計算します。ターゲット・ウィンドウには、出来高が 5000 を超える `Symbols` のみが格納されます。

```
CREATE INPUT STREAM Trades
SCHEMA (TradeId integer, Symbol string, Price float, Shares integer);

CREATE OUTPUT WINDOW TradeSummary
PRIMARY KEY DEDUCED
AS
   SELECT trd.Symbol, count(trd.TradeId) NoOfTrades, max(trd.Price)
MaxPrice, sum(trd.Shares) TotalShares
    FROM Trades trd
    GROUP BY trd.Symbol
    HAVING sum(trd.Shares) > 5000;
```

参照：

- *フィルタリング* (39 ページ)
- *union* (40 ページ)
- *ジョイン* (41 ページ)
- *パターン一致* (47 ページ)
- *GROUP BY 句* (114 ページ)
- *GROUP FILTER 句* (115 ページ)
- *GROUP ORDER BY 句* (116 ページ)
- *HAVING 句* (117 ページ)

第 5 章　高度な CCL プログラミング手法

洗練された複雑なプロジェクトを開発するには、高度な CCL 手法を使用します。

DECLARE ブロックを使用して、変数、定数、SPLASH 関数を定義し、データ型の独自の名前を追加します。

モジュールを作成して、再利用可能なコードをカプセル化します。

メモリ・ストアを使用して、ステートフル要素の内容をメモリに保持します。ログ・ストアを使用して、ステートフル要素の内容をディスクに保持し、障害イベント時のリカバリを可能にします。

DECLARE ブロック

モデル設計者は DECLARE ブロックを使用することによって、CCL データ・モデルの変数、パラメータ、typedef、関数など、関数型プログラミングの要素を組み込むことができます。

CCL はグローバルとローカルの DECLARE ブロックをサポートします。

- **グローバル DECLARE ブロック –** プロジェクト全体からアクセス可能。ただし、各モジュール用の個々のグローバル DECLARE ブロックも設定できます。

  注意： グローバル DECLARE ブロックは、他の CCL ファイルからさらにインポートされると、それらと一緒にマージされます。1 つのプロジェクトで 1 つのみが可能です。

- **ローカル DECLARE ブロック –** CREATE 文内で宣言され、この宣言文を含むストリームまたはウィンドウの **SELECT** 句内でのみアクセス可能。

  注意： ローカル DECLARE ブロックで定義された変数と関数は、**SELECT** 句内と、フレックス演算子内の任意の場所のみでアクセスできます。

CCL 変数は、モデルの実行時に変化する可能性のある変数の記憶領域として利用できます。変数は、SPLASH 構文を使用して、DECLARE ブロックで定義されます。SPLASH 構文の詳細については、『SPLASH プログラマーズ・ガイド』を参照してください。

CCL typedef は、サポートされるデータ型のユーザ定義名です。長い型名は、typedef を使用して短くできます。DECLARE ブロックで定義された typedef は、モデル全体を通してすべての SPLASH 文内でデータ型の代わりに使用できます。

CCL パラメータは、モデルの実行時に値を設定できる定数です。プロジェクト内でリテラル値の代わりにこれらのパラメータを使用することによって、ウィンドウ保持ポリシーやストア・サイズなどの動作を実行時に変更できます。また、プロジェクトを変更せずに実行時に容易に変更できる他の同様の変更も実行できます。CCL パラメータはグローバル DECLARE ブロックで定義し、プロジェクト設定ファイルで初期化します。また、パラメータの宣言時にデフォルト値を設定することもできます。このため、サーバ起動時の初期化はオプションです。

DECLARE ブロック内で SPLASH 関数を作成できます。これによって、手続きアプローチを使用して操作をより簡単に処理できます。これらの SPLASH 関数は、プロジェクト全体で、ストリームのクエリや他の関数から呼び出せます。

参照：

- *フレックス演算子* (57 ページ)
- *typedef* (52 ページ)
- *パラメータ* (53 ページ)
- *変数* (54 ページ)
- *CREATE FLEX 文* (75 ページ)

typedef

既存のデータ型の新しい名前を宣言します。

構文

```
typedef existingdatatypeName newdatatypeName;
```

コンポーネント

existingdatatypeName	元のデータ型。
newdatatypeName	上記のデータ型の新しい名前。

使用法

typedef を使用すると、既存のデータ型に新しい名前を指定できます。この名前は、新しい変数とパラメータを定義し、関数の戻り値の型を指定するのに使用できます。typedef は、DECLARE ブロック、UDF、フレックス・プロシージャ内で宣言できます。typedef で宣言された型は、単純な型に解決される必要があります。

注意： サポートされていないデータ型については、DECLARE ブロック内で typedef を使用し、サポートされているデータ型のエイリアスを作成します。

例

次の例は、money(2) データ型の別名として euros を宣言します。

```
typedef money(2) euros;
```

euros typedef を定義したら、次のように使用できます。

```
euros price := 10.80d2;
```

これは、次と同じです。

```
money(2) price := 10.80d2;
```

参照：

- *データ型* (26 ページ)
- *プロジェクトの変数、パラメータ、データ型、関数の宣言* (56 ページ)

パラメータ

サーバコマンド名またはプロジェクト設定ファイルを使用してプロジェクトのセットアップ時に設定された定数。

構文

```
parameter typeName parameterName1 [:= constant_expression]
[,parameterName2 [:= constant_expression],…];
```

コンポーネント

typeName	宣言済みパラメータのデータ型。
parameterName	宣言済みパラメータの名前。
constant_expression	定数に評価される式。

使用法

パラメータは、修飾子 **parameter** を使用して定義されます。オプションで、デフォルト値を指定できます。デフォルト値は、サーバ起動時に値がパラメータに提供されない場合のみ使用されます。

パラメータは、基本的なデータ型のみを使用して、プロジェクトまたはモジュールのグローバル **DECLARE** ブロックで宣言する必要があります。パラメータは複雑なデータ型としては宣言できません。パラメータは定数ですので、その値はモデル内で変更できません。

複雑なデータ型の詳細については、『SPLASH プログラマーズ・ガイド』を参照してください。

プロジェクト・セットアップ時のパラメータ

パラメータは、プロジェクトとモジュールのグローバル DECLARE ブロック内で定義できます。プロジェクトレベルのパラメータは、サーバの起動時にバインドされます。モジュールレベルのパラメータは、モジュールのロード時にバインドされます。

パラメータの値は、サーバの起動時に、サーバを起動するために使用するコマンド・ラインに値を指定することによって、またはプロジェクト設定ファイルを通して割り当てできます。デフォルト値が設定されていないすべてのプロジェクト・パラメータに値を割り当てる必要があります。パラメータを新しい値にバインドできるのは、モジュールまたはプロジェクトのロード時のみです。

パラメータの宣言時に、デフォルト値を指定できます。デフォルト値は、プロジェクトまたはモジュールのロード時にパラメータが新しい値にバインドされない場合に使用されます。パラメータにデフォルト値が指定されていない場合、モジュールまたはプロジェクトのロード時にパラメータを新しい値にバインドする必要があります。バインドしないとエラーが発生します。

パラメータが式を使用して初期化されると、式はコンパイル時にのみ評価されます。 その結果が、パラメータにデフォルト値として割り当てられます。

実行時に、間隔のデータ型として宣言されているパラメータに値を指定する場合、間隔値は CCL の単位表記で、プロジェクト設定ファイルのマイクロ秒値を使用して指定されます。プロジェクト設定とプロジェクト設定ファイルのパラメータの詳細については、『スタジオ・ユーザーズ・ガイド』を参照してください。

参照：

- *プロジェクトの変数、パラメータ、データ型、関数の宣言* (56 ページ)

変数

変数は、プロジェクトの実行時に変化する可能性のある、特定の情報です。変数は、SPLASH 構文を使用して宣言されます。

構文

```
typeName {variableName[:=any_expression] [, ...]}
```

使用法

変数は、任意の DECLARE ブロック、SPLASH UDF、またはフレックス・プロシージャ内で宣言できます。複数の変数を単一行で宣言できます。

変数の宣言には、オプションで初期値を定数式で指定できます。初期値が指定されていない変数は、null で初期化されます。

変数には、複雑な型を指定できます。ただし、複雑な型の変数は、ローカル DECLARE ブロックとフレックス・ストリーム内の DECLARE ブロックでのみ使用できます。

ローカル DECLARE ブロックで宣言された変数は、以降の **SELECT** 句で使用できますが、**WHERE** 句で使用するとコンパイラ・エラーが発生します。

SPLASH 言語の詳細については、『SPLASH プログラマーズ・ガイド』を参照してください。

例
次の例は、変数を定義し、通常のストリームとフレックス・ストリームの両方でその変数を使用します。

```
declare
 INTEGER ThresholdValue := 1000;
end;
//
// Create Schemas
Create Schema TradeSchema(
    Ts bigdatetime,
    Symbol STRING,
    Price MONEY(4),
    Volume INTEGER
);

Create Schema ControlSchema (
    Msg STRING,
    Value   INTEGER
); //
// Input  Trade Window
//

CREATE  INPUT  WINDOW TradeWindow
  SCHEMA TradeSchema
  PRIMARY KEY (Ts);

//
// Input Stream for Control Messages
//

CREATE INPUT STREAM ControlMsg SCHEMA ControlSchema ;

//
// Output window, only has rows that were greater than the
thresholdvalue
// was when the row was received
CREATE  Output  WINDOW OutTradeWindow
    SCHEMA (Ts bigdatetime, Symbol STRING, Price MONEY(4), Volume
INTEGER)
    PRIMARY KEY (Ts)
as
select *
    from TradeWindow
    where TradeWindow.Volume > ThresholdValue;

//
//Flex Stream to process the control message
CREATE FLEX FlexControlStream
  IN ControlMsg
  OUT OUTPUT WINDOW SimpleOutput
  SCHEMA ( a integer, b string, c integer)
     PRIMARY KEY ( a)
```

```
BEGIN
    ON ControlMsg
    {
        // change the value of ThresholdValue
        if ( ControlMsg.Msg = 'set')
{ThresholdValue:=ControlMsg.Value;}
        // The following is being populate so you can see that the
ThresholdValue is being set
        output [a=ControlMsg.Value; b=ControlMsg.Msg;
c=ThresholdValue; |];
    }
    ;
END
;
```

参照：

- *DECLARE ブロック* (51 ページ)
- *SELECT 句* (121 ページ)
- *プロジェクトの変数、パラメータ、データ型、関数の宣言* (56 ページ)

プロジェクトの変数、パラメータ、データ型、関数の宣言

グローバルとローカルの両方の DECLARE ブロックで変数、パラメータ、typedef、関数を宣言します。

1. プロジェクトのグローバル DECLARE ブロックを、メイン・プロジェクト・ファイルの **DECLARE** 文を使用して作成します。
2. パラメータ、変数、ユーザ定義の SPLASH 関数をグローバル DECLARE ブロックに追加します。

   この DECLARE ブロックで定義される要素は、モジュールの内側ではないプロジェクトの任意の要素からアクセスできます。
3. ローカル DECLARE ブロックを派生ストリーム、派生ウィンドウ、または両方内の **DECLARE** 文を使用して作成します。
4. 変数、パラメータ、ユーザ定義の SPLASH 関数をローカル DECLARE ブロックに追加します。

   これらの要素は、ブロックが定義されているストリーム、ウィンドウ、またはフレックス演算子の内側からのみアクセスできます。

参照：

- *typedef* (52 ページ)
- *パラメータ* (53 ページ)
- *変数* (54 ページ)

フレックス演算子

フレックス演算子は、CCL を拡張し、SPLASH で記述されたカスタム・イベント・ハンドラが派生ストリームまたは派生ウィンドウを生成できるようにします。

フレックス演算子は、**CREATE** 文でストリーム、ウィンドウ、またはデルタ・ストリームの各派生要素が生成されるのと同じ方法で、これらの要素を生成します。しかし、**CREATE** 文では、入力から新しいウィンドウを派生させるために CCL クエリを使用しますが、フレックス演算子は SPLASH スクリプトを使用します。

フレックス演算子は CCL を拡張可能にし、宣言の **SELECT** 文で実装するのが困難なイベント処理ロジックを実装できるようにします。SPLASH は、複雑な処理を制御できるようにし、イベント間で状態を保持できるデータ構造を提供します。

SPLASH の全機能は、以下を含むフレックス演算子で使用できます。

データ構造	• 変数 • EventCache (ウィンドウ) • 辞書 • ベクトル
制御構造	• While • If • For

フレックス演算子は任意の数の入力を受け取ることができ、この入力には、ストリーム、デルタ・ストリーム、またはウィンドウを組み合わせて指定できます。入力ごとに SPLASH イベント・ハンドラを記述できます。イベントがその入力に到着すると、関連する SPLASH スクリプトまたはメソッドが呼び出されます。

あらゆる入力に対してメソッドを用意する必要はありません。入力の中には、その他の入力に関連するメソッドで使用するデータを提供するだけの入力もあります。関連するメソッドがない入力では、受信イベントはアクションをトリガしませんが、同じフレックス演算子のその他のメソッドにアクセスできます。

フレックス・メソッドを記述する方法の詳細については、『SPLASH プログラマーズ・ガイド』を参照してください。

参照：
- *CREATE FLEX 文* (75 ページ)

モジュール性

Sybase Event Stream Processor のモジュールは再利用できます。モジュールはプロジェクト全体で複数回ロードしたり使用したりできます。

モジュール性とは、プロジェクトの要素をモジュールと呼ばれる自己完結型の再利用可能なコンポーネントに編成することです。これによって、入力と出力が適切に定義され、よく繰り返されるデータ処理プロシージャをカプセル化できます。

モジュールは、インポート・ファイルやメイン・プロジェクトなどの他のオブジェクトと共に独自のスコープを持ちます。スコープは、変数または定義へのアクセスが許可される範囲を定義します。スコープで宣言された変数、オブジェクト、または定義は、スコープ内でのみアクセス可能です。親スコープと呼ばれる外側のスコープや、その他の外部スコープからはアクセスできません。親スコープは、モジュールでも、メイン・プロジェクトでも可能です。たとえば、モジュール A がモジュール B をロードし、メイン・プロジェクトがモジュール A をロードする場合、モジュール A のスコープはモジュール B の親スコープです。モジュール A の親スコープはメイン・プロジェクトです。

モジュールには、明示的に宣言された入力と出力があります。モジュールへの入力は、親スコープのストリームまたはウィンドウと関連付けられ、モジュールの出力は、識別子を使用して親スコープに公開されます。モジュールを再利用すると、そのモジュール内のストリーム、変数、パラメータ、またはその他のオブジェクトがレプリケートされるので、モジュールの各バージョンは他のバージョンとは別個に存在します。

モジュールは他のモジュール内にロードできるので、モジュール A はモジュール B をロードでき、モジュール B はモジュール C をロードできる、というようになります。ただし、モジュール依存性ループは無効になります。たとえば、モジュール A がモジュール B をロードし、モジュール B がモジュール A をロードすると、CCL コンパイラは、モジュール A と B 間の依存性ループを示すエラーを生成します。

CREATE MODULE 文はモジュールを作成します。モジュールは、プロジェクトに複数回ロードでき、モジュールの入力と出力はより大きいプロジェクトのさまざまなパーツにバインドできます。**LOAD MODULE** 文を使用すると、定義済みのモジュールをプロジェクト全体で 1 回以上再利用できます。

注意： モジュール関連のコンパイル・エラーはすべて致命的です。

参照：

- *モジュールの作成と使用* (59 ページ)
- *CREATE MODULE 文* (84 ページ)

- *IMPORT 文* (91 ページ)
- *LOAD MODULE 文* (92 ページ)
- *IN 句* (98 ページ)
- *OUT 句* (101 ページ)
- *PARAMETERS 句* (102 ページ)
- *STORES 句* (107 ページ)
- *例：モジュール内のパラメータ* (61 ページ)
- *例：モジュールの作成と使用* (59 ページ)

モジュールの作成と使用

再利用可能なモジュールを作成するには **CREATE MODULE** 文を、以前に作成したモジュールをロードするには **LOAD MODULE** を使用します。

モジュールをロードしたら、その入力ストリームまたは入力ウィンドウをプロジェクトのストリームに接続またはバインドできます。モジュールの出力は親のスコープに公開され、**LOAD MODULE** 文で提供されているエイリアスを使用して、そのスコープ内で参照できます。

モジュール内のパラメータは、親スコープのパラメータまたは定数式にバインドされます。モジュール内のストアは、親スコープのストアにバインドされます。モジュール内のストアをモジュール外のストアにバインドすることは、モジュールのストアを使用するすべてのウィンドウが、バインドされたストアを使用することを意味します。

参照：

- *モジュール性* (58 ページ)
- *CREATE MODULE 文* (84 ページ)
- *IMPORT 文* (91 ページ)
- *LOAD MODULE 文* (92 ページ)
- *例：モジュールの作成と使用* (59 ページ)

例：モジュールの作成と使用

生の株式取引情報を処理し、価格が 1.00 を超える取引のリストを出力するモジュールを、モジュール性の基本概念を使用して作成します。

1. インポート・ファイルを作成してスキーマをグループ化し、プロジェクト全体で再利用できるようにします。
   この例では、`schemas.ccl` の名前のインポート・ファイルに、次の文が格納されています。

```
CREATE SCHEMA TradesSchema (
    Id integer,
```

```
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
);
```

注意： モジュールまたはプロジェクト内でスキーマを直接に定義できますが、この例では、インポート・ファイルを使用してコードの重複を削減し、CCL の保守性を向上させます。

2. プロジェクト内で、**CREATE MODULE** 文を使用してモジュールを作成し、**IMPORT** 文を使用してインポート・ファイル (schemas.ccl) をインポートします。

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
    IMPORT 'schemas.ccl';

    CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
    CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema
    AS SELECT * FROM TradeData WHERE TradeData.Price > 1.00;
END;
```

モジュールは、入力ストリーム (TradeData) を使用して株式市場から生フィードを取得し、フィルタされた結果を出力ストリーム (FilteredTradeData) で提供します。モジュール内で **IMPORT** 文を使用することによって、schemas.ccl ファイルにグループ化されているスキーマのすべてをモジュールのストリームで使用できます。

3. **LOAD MODULE** 文を使用して、モジュールをメイン・プロジェクトにロードします。
この例は、モジュールを株式市場のストリームに接続する方法を示しています。

```
IMPORT 'schemas.ccl';

CREATE INPUT STREAM NYSEData SCHEMA TradesSchema;

LOAD MODULE FilterByPrice AS FilterOver1 IN TradeData = NYSEData
OUT FilteredTradeData = NYSEPriceOver1Data;
```

- プロジェクト・ファイルの最初の行は、schemas.ccl をインポートする。これによって、モジュールと同じスキーマが使用できるようになります。
- 入力ストリーム NYSEData は、ニューヨーク証券取引所からの情報を表している。
- **LOAD MODULE** 文は、モジュール FilterByPrice をロードする。このモジュールは、インスタンス名 FilterOver1 で識別されます。

- モジュールの入力ストリーム TradeData を入力ストリーム NYSEData にバインドする。これによって、NYSEData ストリームからの情報がモジュールに流れ込むようになります。
- モジュールの出力は、プロジェクト (NYSEPriceOver1Data) に公開される。
- モジュールの出力にアクセスするには、NYSEPriceOver1Data ストリームからの情報を選択する。

参照：

- *モジュールの作成と使用* (59 ページ)
- *CREATE MODULE 文* (84 ページ)
- *IMPORT 文* (91 ページ)
- *LOAD MODULE 文* (92 ページ)
- *モジュール性* (58 ページ)

例：モジュール内のパラメータ

ここの説明は、パラメータのバインドについて理解を深めることを目的としています。親スコープの式または別のパラメータにバインドできるパラメータを定義するモジュールを作成します。

モジュール FilterByPrice は、受信したすべての取引を価格に基づいてフィルタし、価格が minimumPrice パラメータの値を超える取引のみを出力します。

minimumPrice は、FilterByPrice のロード時に設定できます。また、プロジェクト内の別のパラメータにバインドすることによって、minimumPrice を、プロジェクトがサーバにロードされるときに設定できます。

モジュール定義を次に示します。

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
    IMPORT 'schemas.ccl';

    DECLARE
        parameter money(2) minimumPrice := 10.00d2;
    END;

    CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
    CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema AS
SELECT * FROM TradeData WHERE TradeData.Price > minimumPrice;
END;
```

式へのパラメータのバインド

式へのパラメータのバインドに関して、ロード時に minimumPrice は式にバインドされます。

```
LOAD MODULE FilterByPrice AS FilterOver20 IN TradeData = NYSEData OUT
FilteredTradeData = NYSEPriceOver20Data PARAMETERS minimumPrice =
20.00d2;
```

このタイプのパラメータ・バインドでは、モジュールは価格が 20.00 を超える株式のみを出力します。

親スコープのパラメータへのモジュールのパラメータのバインド
このタイプのバインドでは、モジュール内のパラメータは、メイン・プロジェクトで宣言されているパラメータにバインドされます。このため、フィルタ対象の取引価格を実行時に変更できます。これを行うには、プロジェクトの **DECLARE** ブロックでパラメータを作成し、モジュール内のパラメータ (`minimumPrice`) を新しいパラメータにバインドします。

```
DECLARE
    parameter money(2) minProjectPrice := 15.00d2;
END;

LOAD MODULE FilterByPrice AS FilterOverMinProjPrice IN TradeData =
NYSEData OUT FilteredTradeData = NYSEPriceOverMinProjPrice
PARAMETERS minimumPrice = minProjectPrice;
```

実行時にプロジェクトのパラメータ (`minProjectPrice`) の値が指定されていない場合、モジュールは、プロジェクトのパラメータのデフォルト値 (15.00) に基づいてフィルタします。ただし、実行時に `minProjectPrice` の値が指定されると、モジュールはその値に基づいてフィルタします。

パラメータのバインドなし
この例では、モジュール定義でデフォルトが `minimumPrice` に設定されており、モジュールのロード時にパラメータをバインドする必要はありません。モジュールは、次のようにロードできます。

```
LOAD MODULE FilterByPrice AS FilterOver10 IN TradeData = NYSEData OUT
FilteredTradeData = NYSEPriceOver10Data;
```

LOAD MODULE 文でバインドが指定されていないので、モジュールはデフォルト値の 10.00 を使用してフィルタします。

参照：

- *モジュール性* (58 ページ)
- *モジュールの作成と使用* (59 ページ)
- *PARAMETERS 句* (102 ページ)
- *LOAD MODULE 文* (92 ページ)

永続性

ログ・ストアは、サーバで障害が発生したり、サーバがシャットダウンしたりした場合に、ウィンドウ内でのデータ・リカバリを可能にします。

ログ・ストアは、プロジェクトの永続性を提供します。適切に指定されたログ・ストアは、障害時にステートフル要素をリカバリし、サーバで障害が発生したり、サーバが再起動したりした場合にデータを正しくリストアします。ログ・ストアは、保持ポリシーのないウィンドウで使用できます。ステートレス要素に対しては使用できません。

ログ・ストアは以下のように機能します。

- ログ・ストアは、ウィンドウの内容のみを格納する。
- ログ・ストアは、変数などの中間ステータスを直接に格納しない。
- ローカル・フレックス・ストリームの変数とデータ構造体は、直接に格納されない。ただし、ソース・データが永続ストレージの場合は、ソース・データから再生成できます。
- ログ・ストアは、opcode 情報を維持しない (ログ・ストアの定期的な圧縮とチェックポイント時には、現在のウィンドウ・ステータスのみが維持されます。レコードは、挿入としてリストアされます)。
- ローの到着順は保持されない。いずれのストリームでも、ログ・ストアの圧縮時に複数の操作が 1 つのレコードにまとめられ、到着順が変化する可能性があります。ストリーム間の到着順は維持されません。
- 1 つのプロジェクトに 1 つ以上のログ・ストアを定義可能。複数のストアを使用する場合は、ログ・ストアのループが発生するのを防ぐ必要があります。ログ・ストアのループは、たとえば、`Logstore1` の `Window1` が、`Logstore1` の `Window3` にフィードする `Logstore2` の `Window2` にフィードすると発生します。ログ・ストアのループは、コンパイル・エラーとなります。
- ログ・ストア・ウィンドウから直接にデータを受信するメモリ・ストア・ウィンドウの内容は、ログ・ストア・ウィンドウがディスクからリストアされると再計算される。
- 他のメモリ・ストア・ウィンドウを介してログ・ストア・ウィンドウからデータを受信するメモリ・ストア・ウィンドウの内容は、入力ウィンドウの内容が再計算されると、再計算される。

注意： メモリ・ストア・ウィンドウが、デルタ・ストリームやストリームなどのステートレス要素を介してログ・ストア・ウィンドウからデータを受信する場合、その内容はサーバ・リカバリ時にリストアされません。

ログ・ストアは定期的に圧縮され、その時点で、ストアに累積されたすべてのデータはチェックポイントされ、同じキーの複数操作はまとめられます。チェッ

クポイント処理が完了すると、ストアは次のチェックポイントまで受信データ・ローをストアの末尾に継続して追加します。

注意： ストアに書き込まれているがチェックポイントされていないデータのリカバリは、入力ウィンドウに対してのみ利用できます。ウィンドウをログ・ストアに割り当てる場合には、そのすべての入力ウィンドウもログ・ストアに割り当てることをおすすめします。そうしないと、最後のチェックポイント以降にウィンドウに書き込まれたデータは、リストアされません。

メモリ・ストアと異なり、ログ・ストアは自動的に拡張されません。CCL の **maxfilesize** プロパティを使用して、ログ・ストアのサイズを指定します。ログ・ストアのサイズは、非常に重要です。ログ・ストアのサイズが小さすぎると、オーバフローが発生して処理が停止することがあります。クリーニング・サイクルが頻繁に発生するので、パフォーマンスが大きく悪化することもあります。ログ・ストアのサイズが大きすぎる場合も、大量のディスク容量とメモリを要求するので、パフォーマンスが悪化することがあります。

参照：

- *ストア* (22 ページ)
- *CREATE MEMORY STORE 文* (82 ページ)
- *CREATE LOG STORE 文* (80 ページ)
- *保持* (19 ページ)
- *KEEP 句* (100 ページ)

ログ・ストアの最適化手法

データ・モデルを最適化して最大のパフォーマンスを達成するために、永続性を指定します。

- 可能な場合は常に、静的 (次元) データの格納用には小さなログ・ストアを、動的 (事実) データの格納用には 1 つ以上の大きなログ・ストアを作成する。
- 大きくて、急速に変化する、動的 (事実) データを格納するために複数のログ・ストアを使用している場合は、異なる RAID ボリュームにストアを編成する。
- ログ・ストアのサイズを適切に設定することは、非常に重要である。詳細については、『管理者ガイド』の「ログ・ストアのサイジング」を参照してください。

エラー・ストリーム

エラー・ストリームは、エラーと、エラーを発生させたレコードを収集します。

説明

エラー・ストリームは、エラー情報とエラーを発生させたデータを取得する手段を提供します。エラー・ストリームは、開発時にエラーをデバッグするのに役立

ちます。また、運用環境でプロジェクトをリアルタイムにモニタする機能も提供します。

1 つのプロジェクトで複数のエラー・ストリームを指定できます。

エラー・ストリームは、次を除いて他のユーザ定義のストリームと同じです。

- ソース・ストリームまたはソース・ウィンドウでエラーが発生した場合に、そのソース・ストリームまたはソース・ウィンドウからのレコードのみを受信する。受信するレコードは、エラーを発生させた、ソース・ストリームまたはソース・ウィンドウへの入力です。
- ユーザが変更できない、事前に定義されたスキーマを持つ。

スキーマ

カラム	データ型	説明
errorCode	integer	レポートされたエラーの数値コード
errorRecord	binary	エラーを発生させたレコード
errorMessage	string	エラーを説明する平文テキスト・メッセージ
errorStreamName	string	このエラーがレポートされたストリームの名前
sourceStreamName	string	エラーを発生させたレコードの送信元のストリームの名前
errorTime	bigdatetime	エラーが発生した時間。マイクロ秒単位のタイムスタンプ

エラー・コード

1. GENERIC_ERROR
2. FP_EXCEPTION
3. BADARGS
4. DIVIDE_BY_ZERO
5. OVERFLOW_ERR
6. UNDERFLOW_ERR
7. SYNTAX_ERR

制限事項
エラー・ストリームの構文によって、実行時に発生するエラーをトラップするメカニズムが提供されますが、以下の制限があります。

- レコードの演算時に発生するエラーのみが、エラー・ストリームに取得される。たとえば、変数とパラメータを初期化するために使用される式の評価など、サーバの起動時に発生する演算のエラーは、エラー・ストリームに取得さ

れません。接続エラーや演算以外のエラーなど他のエラーも、エラー・ストリームに取得されません。

- フレックス・ブロックの ON START TRANS ブロックや ON END TRANS ブロックのようにレコードをトリガすることのない演算の実行時に発生するエラーは、errorRecord フィールドに空のレコードが格納されたエラー・レコードを送信する。
- [1] 組み込み関数については、ストリーム名は、文字列リテラル定数である。この制限によって、組み込み関数の戻り値のレコード型がコンパイル時に決定できます。
- エラーをトリガするレコードは、提供されている組み込み関数を使用して取得される。レコードを直接に参照するための、ネイティブでネストされたレコードはサポートされません。
- レポートされる、エラーをトリガするレコードは、エラーが発生したストリームへの直近の入力である。このストリームは、ユーザ定義のストリームまたはコンパイラによって生成された中間ストリームのこともあります。[2] と [3] の組み込み関数を使用する場合、最初の引数は、中間ストリームに一致する必要があります (生成されている場合)。
- サブスクリプション・ユーティリティがエラー・レコードを自動的に復号 (バイナリから ASCII に変換) することはない。
- 出力アダプタがエラー・レコードを自動的に復号 (バイナリから ASCII に変換) することはない。
- 外部関数 (C と JAVA) で発生する算術エラーと変換エラーは処理されない。ユーザの責任で対応する必要があります。
- エラー・ストリームの動作は、デバッガ・フレームワークで保証されない。

エラー・ストリームのモニタリング

他のストリームのエラーと、エラーを発生させたイベントをモニタするために、エラー・ストリームを使用します。

プロセス

1. モニタするプロジェクトと特定のストリームを識別します。
2. 複数のエラー・ストリームを使用するかどうかを決定します。各エラー・ストリームに対して、アクセス性を決定します。
3. そのプロジェクト内でエラー・ストリームを作成します。
4. エラー・ストリームから情報 (エラー・レコードの一部またはすべての情報、エラー・レコードから集約または抽出された情報) を表示します。

---

[1] recordDataToRecord
[2] recordDataToString
[3] recordDataToRecord

例

次の例は、1 つの入力ストリームと 2 つの派生ストリームが構成されているプロジェクトで、3 つすべてのストリームをモニタする、ローカルでのみアクセス可能なエラー・ストリームを作成します。

```
CREATE ERROR STREAM AllErrors ON InputStream, DerivedStream1,
DerivedStream2;
```

レポートされるエラー・コードごとのエラー数を保持するには、次を追加します。

```
CREATE OUTPUT WINDOW errorHandlerAgg SCHEMA (errorNum integer, cnt
long)
PRIMARY KEY DEDUCED
AS
SELECT e.errorCode AS errorNum, COUNT(*) AS cnt
FROM AllErrors e
GROUP BY e.errorCode
;
```

次の例は、3 つの派生ストリームが構成されているプロジェクトで、出来高加重平均価格を計算する 3 番目の派生ストリームのみをモニタする、外部からアクセス可能なエラー・ストリームを作成します。

```
CREATE OUTPUT ERROR STREAM vwapErrors ON DerivedStream3;
```

エラーをトリガするレコードのフォーマットをバイナリから文字列に変換するには、次を追加します。

```
CREATE OUTPUT vwapMessages SCHEMA (errorNum integer, streamName
string, errorRecord string) AS
SELECT  e.errorcode AS errorNum,
        e.streamName AS streamName,
        recordDataToString(e.sourceStreamName, e.errorRecord) AS
errorRecord
FROM vwapErrors e;
```

エラーをトリガするレコードのフォーマットをバイナリからレコードに変換するには、次を追加します。

```
CREATE OUTPUT vwapMessages SCHEMA (errorNum integer, streamName
string, errorRecord string) AS
SELECT  e.errorcode AS errorNum,
        e.streamName AS streamName,
        recordDataToRecord(e.sourceStreamName, e.errorRecord) AS
errorRecord
FROM vwapErrors e;
```

第 6 章　文

CCL 文リファレンスは、構文、パラメータ説明、使用方法、例を提供します。

ADAPTER START 文

アダプタの起動時刻を制御します。

構文

```
ADAPTER START
GROUPS {groupName[NOSTART]},[,...]
...
;
```

使用法

アダプタ・グループは、その名前が、**ATTACH ADAPTER** 文の **GROUP** 句で使用されると、暗黙的に作成されます。各 **groupName** の指定順によって、アダプタ・グループの起動順が決定されます。順序付けされているグループのいずれにも割り当てられていないアダプタは、すべての順序付けされているグループが起動した後に起動されるグループに割り当てられます。デフォルトでは、1 つのグループ内のすべての出力アダプタが同時に起動され、その後、すべての入力アダプタが同時に起動されます。

ADAPTER START 文はオプションです。この文が指定されていない場合、すべての出力アダプタが同時に起動され、その後に、すべての入力アダプタが同時に起動されます。

NOSTART が指定されたアダプタは、他のアダプタが自動的に起動されても、起動されません。これらのアダプタは、外部 XMLRPC インタフェース (`esp_client.exe`) を介して起動できます。

ADAPTER START 文で次の動作が指定されていると、エラーが生成されます。

- 存在しないグループの参照。
- **ATTACH ADAPTER** 文で作成されたアダプタ起動グループの一部に対する不参照。
- 同じグループの繰り返し参照。

例

次の **ATTACH ADAPTER** 文は、2 つの名前付きアダプタ・グループ (`RunGroup1`、`NoRunGroup`) を作成します。各グループに 1 つのアダプタがあります。**ADAPTER**

START 文は、`RunGroup1` を起動します。**NOSTART** 構文は、プロジェクト・サーバに対して `NoRunGroup` を起動しないように指示します。

```
ATTACH INPUT ADAPTER csvInRun
 TYPE dsv_in
 TO TradeWindow
 GROUP RunGroup1
 PROPERTIES
   blockSize=1,
   dateFormat='%Y/%m/%d %H:%M:%S',
   delimiter=',',
   dir='$ProjectFolder/../data',
   expectStreamNameOpcode=false,
   fieldCount=0,
   file='stock-trades.csv',
   filePattern='*.csv',
   hasHeader=true,
   safeOps=false,
   skipDels=false,
   timestampFormat= '%Y/%m/%d %H:%M:%S';

ATTACH INPUT ADAPTER csvInNoRun
 TYPE dsv_in
 TO TradeWindow
 GROUP NoRunGroup
 PROPERTIES
   blockSize=1,
   dateFormat='%Y/%m/%d %H:%M:%S',
   delimiter=',',
   dir='$ProjectFolder/../data',
   expectStreamNameOpcode=false,
   fieldCount=0,
   file='stock-trades.csv',
   filePattern='*.csv',
   hasHeader=true,
   safeOps=false,
   skipDels=false,
   timestampFormat= '%Y/%m/%d %H:%M:%S';

ADAPTER START GROUPS NoRunGroup NOSTART, RunGroup1;
```

参照：

- *ATTACH ADAPTER 文* (70 ページ)

ATTACH ADAPTER 文

アダプタをストリームまたはウィンドウにアタッチするか、アダプタをグループに割り当てます。

構文

```
ATTACH { INPUT|OUTPUT } ADAPTER name
TYPE type
```

```
TO streamorwindow
[GROUP groupName]
[PROPERTIES {prop=value} [, ...]];
```

パラメータ

name	アダプタの名前
type	アダプタのタイプを指定
streamorwindow	アダプタのアタッチ先のストリームまたはウィンドウを指定

使用法

アダプタは、アダプタのタイプとアダプタを構成するプロパティのインライン定義を使用して定義されます。データ・ロケーションはサポートされません。**type** は、各アダプタに割り当てられている一意な ID です。各アダプタのタイプについては、『アダプタ・ガイド』を参照してください。

ATTACH ADAPTER 文は、**ADAPTER START** 文の前に指定する必要があります。

アダプタ・グループを作成する文はありません。アダプタをグループ化するには、**GROUP** 句にグループ名を指定します。このグループ化は、以降で、アダプタを規定された順序で起動するために **ADAPTER START** 文で使用されます。**ADAPTER START** 文のないグループは指定できません。

入力としてマークされているアダプタは、入力ストリームまたは入力ウィンドウのみにアタッチできます。出力としてマークされているアダプタは、入力または出力のストリームまたはウィンドウにアタッチできます。入力または出力のいずれのアダプタも、ローカル・ストリームまたはローカル・ウィンドウにアタッチできません。`cnxml` ファイルで入力アダプタとして定義されているアダプタは、出力アダプタとしてアタッチできません。`cnxml` ファイルで出力アダプタとして定義されているアダプタは、入力アダプタとしてアタッチできません。

ATTACH ADAPTER 文で有効なプロパティ名と値のペアは、アダプタのタイプによって異なります。プロパティ名では、大文字と小文字は区別されません。特定のアダプタによって要求されるプロパティに関連するすべての仕様は、アダプタの `cnxml` ファイルにあります。このファイルは、Sybase Event Stream Processor のインストール・フォルダに格納されており、プロパティの検証に使用されます。

アダプタの `cnxml` ファイルで名前が定義されているアダプタ・プロパティのみを指定できます。また、すべてのプロパティの値は、定義されているデータ型に一致する必要があります。同じプロパティを 2 回指定すると、コンパイラによってエラーが生成されます。

ATTACH ADAPTER 文内でプロパティ・セットを指定することもできます。プロパティ・セットとは、プロジェクト設定ファイルに保存されている再利用可能な一連のプロパティです。プロパティ・セットを指定する場合、すべての必須プロパ

ティが個々のプロパティとして設定されていることを確認する必要があります。プロパティ・セットは、**ATTACH ADAPTER** 文内で指定されている個々のプロパティに優先します。

例

```
ATTACH INPUT ADAPTER MacysInventory
TYPE dsv_in
TO InventoryInfo
PROPERTIES
dir='C:/Operations/Stock/Inventory/MacysInventory',
file='inventory.csv',
propertyset '<name>';
```

参照：

- *ADAPTER START 文* (69 ページ)

CREATE DELTA STREAM 文

insert、delete、update のすべての操作コード (opcode) を処理できるステートレス要素を定義します。

構文

```
CREATE [  LOCAL | OUTPUT ] DELTA STREAM name
[ schema_clause ]
primary_key_clause
[ local-declare-block ]
as_clause
Query;
```

コンポーネント

name	作成するデルタ・ストリームの名前。
schema_clause	新しいウィンドウ用のスキーマ定義。この句が指定されていないと、スキーマ定義はクエリから抽出されます。
primary_key_clause	プライマリ・キーを設定。詳細については、「PRIMARY KEY 句」を参照してください。
local-declare-block	(オプション) クエリ内でアクセスされる変数と関数の宣言。
as_clause	文へのクエリの取り込みを指定。
Query	文に実装されるクエリ。詳細については、「クエリ」を参照してください。

使用法
デルタ・ストリームは、すべての opcode を処理できるステートレス要素です。デルタ・ストリームは、計算、フィルタ、またはユニオンを実行する必要がある場合、ステータスを維持する必要がない場合に使用できます。

ただし、フィルタについては、デルタ・ストリームは受信した opcode を変更します。デルタ・ストリームは、通常、受信した opcode を転送します。フィルタ句を満たす insert opcode を持つ入力レコードの場合、出力にも insert opcode が格納されます。update opcode を持つ入力レコードについては、この入力レコードが条件を満たしても、元のレコードが条件を満たさない場合、出力には insert opcode が格納されます。

ただし、元のレコードが条件を満たす場合は、出力に update opcode が格納されます。入力レコードが delete opcode を持つ場合、フィルタ条件が満たされると、出力に delete opcode が格納されます。**CREATE DELTA STREAM** は、主に単純な射影を通して単純な変換を行う計算で使用されます。

制限事項
- デルタ・ストリームでは、**random()** など、繰り返し実行すると異なる値が生成される関数を使用できない。これは、デルタ・ストリームでは、レコードを削除する場合に、以前に同じレコードの挿入時に生成された結果と同じ結果を生成できる必要があるためです。
- デルタ・ストリームにサブスクライブする場合、デルタ・ストリームが生成する opcode は、安全な opcode として扱われる必要がある。これは、すべての insert/update が upsert (レコードが存在しない場合は insert、存在する場合は update) として扱われる必要があることを意味します。同様に、すべての delete は、レコードが存在する場合は delete として扱われ、存在しない場合は暗黙的に無視される必要があります。
- デルタ・ストリームを入力として使用するターゲット・ノードでは、実行する操作に制限はない。
- フレックス演算子を使用してデルタ・ストリームを定義した場合、SPLASH コードで出力できるのは、insert または delete のみである。upsert と update については、これらを正しく処理するためのステータスをデルタ・ストリームがサポートしないので、許可されません。update を実行するには、delete を発行し、その後に insert を発行します。
- デルタ・ストリームのクエリには、集約またはジョインを実行する句を指定できない。

例
次の例は、総コストを計算するデルタ・ストリームを作成します。

```
CREATE INPUT WINDOW Trades SCHEMA  (
  TradeId   long,
```

```
   Symbol   string,
   Price    money(4),
   Shares   integer
)
PRIMARY KEY (TradeId)
;

CREATE DELTA stream TradesWithCost
PRIMARY KEY DEDUCED
AS SELECT
        trd.TradeId,
        trd.Symbol,
        trd.Price,
        trd.Shares,
        trd.Price * trd.Shares TotalCost
FROM
        Trades trd
;
```

次の例は、総コストが 10,000 未満のレコードを除外するデルタ・ストリームを作成します。

```
CREATE DELTA stream LargeTrades
PRIMARY KEY DEDUCED
AS SELECT * FROM TradesWithCost twc WHERE twc.TotalCost >= 10000
;
```

参照：

- *DECLARE 文* (86 ページ)
- *PRIMARY KEY 句* (104 ページ)
- *SCHEMA 句* (105 ページ)
- *SELECT 句* (121 ページ)
- *AS 句* (97 ページ)
- *第 8 章、「クエリ」* (109 ページ)

CREATE ERROR STREAM 文

エラーと、エラーを発生させたイベントを収集するストリームを作成します。

構文

```
CREATE [LOCAL|OUTPUT] ERROR STREAM name ON source [, source ... ]
```

name は、新たに作成されるエラー・ストリームを識別する文字列です。

source は、以前に定義されたストリームまたはウィンドウを識別する文字列です。

使用法

エラー・ストリームは、指定されたストリームからエラー・データを収集します。各エラー・レコードには、エラー・コードとエラーを発生させた入力イベントが

格納されます。これらのレコードは、モニタリング目的で簡単に表示できます。また、これらのレコードを使用して、他のストリームからのレコードのように、ダウンストリームの処理ロジックをさらに起動することもできます。

運用環境では、エラー・ストリームはプロジェクト内の 1 つ以上のストリームをリアルタイムにモニタするために使用されます。開発環境では、プロジェクトをデバッグするときに、入力ストリームや派生ストリームをモニタするために使用できます。

エラー・ストリームのアクセス性は、デフォルトで LOCAL です。エラー・ストリームを外部のモニタリング・ツールまたはデバイスからアクセスできるようにするには、エラー・ストリームの作成時に OUTPUT を指定する必要があります。

1 つのプロジェクトで複数のエラー・ストリームを定義できます。

例

次の例は、プロジェクト内のすべてのストリームをモニタするために、1 つの入力ストリームと 2 つの派生ストリームを使用する、外部からアクセス可能な単一のエラー・ストリームを作成します。

```
CREATE OUTPUT ERROR STREAM AllErrors ON InputStream, DerivedStream1,
DerivedStream2
```

次の例は、プロジェクト内の入力ストリームと派生ストリームをモニタするために、2 つの入力ストリームを使用するエラー・ストリームと 3 つの派生ストリームを使用するエラー・ストリームの、2 つの異なるエラー・ストリームを作成します。これらのエラー・ストリームは、ローカルでのみアクセスできます。

```
CREATE ERROR STREAM InputErrors ON InputStream1, InputStream2
CREATE ERROR STREAM QueryErrors ON DerivedStream1, DerivedStream2,
DerivedStream3
```

CREATE FLEX 文

フレックス演算子は複数のステートフル要素とステートレス要素を取り込み、1 つのステートフル要素またはステートレス要素を生成します。SPLASH コードを使用して、カスタマイズ可能な処理ロジックを指定できます。

注意： フレックス演算子の名前は、スタジオでのラベル処理のためにのみ存在し、クエリで参照できません。代わりに、出力要素を参照します。

構文

```
CREATE FLEX procedureName
 IN input1 [KEEP keep_spec], ...
    OUT output_element
        BEGIN
            [DECLARE
```

```
                //variable and function declarations
            END;]
            ON input1 {
                //statements
                };
                [EVERY interval{
                 //periodically executing tasks
                };]
                [ON START TRANSACTION {
                    //tasks to be executed
                    //at the start of every transaction
                };]
                [ON END TRANSACTION {
                    //tasks to be executed
                    //at the end of each transaction
                };]
        END;

OUT output_element
output_element:
{{[OUTPUT/LOCAL] STREAM name schema_clause
        [OUTPUT/LOCAL] DELTA STREAM name schema_clause|PRIMARY
KEY{column1,column2,...)
        [OUTPUT/LOCAL] WINDOW name schema_clause}
       [PRIMARY KEY(column1,column2,...)][store_clause][keep_clause]
[aging_clause]
    }
}
```

コンポーネント

`procedureName`	作成するフレックス演算子の名前。
IN `input1`	フレックス演算子への入力は、**IN** 句で宣言される。入力には、ストリーム、デルタ・ストリーム、ウィンドウ、または他のフレックス演算子の出力を指定できます。
KEEP `keep_spec`	**KEEP** 句は、デルタ・ストリームまたはウィンドウの既存の入力要素の保持ポリシーを変更する。
OUT `output_element`	フレックス演算子の出力は、**OUT** 句で定義される。フレックス・ストリームで定義できる出力は、1 つのみです。**SCHEMA** 句は、すべての出力タイプで必須です。
`DECLARE ... END;`	(オプション) **DECLARE** ブロックですべての型の変数と関数を定義できる。レコード、ベクトル、辞書、イベント・キャッシュなどの複雑なデータ型も定義できます。 詳細については、『SPLASH プログラマーズ・ガイド』を参照してください。

ON `input1`	フレックス演算子のすべての入力に対して **ON** 入力句が宣言される必要がある。入力レコードが受信されるごとに、このブロックで指定されている SPLASH コードが実行されます。処理を必要としない入力要素に対しては、空の **ON** 入力句を使用します。
EVERY `interval`	(オプション) **EVERY** 期間句を使用すると、間隔が経過するごとに実行されるコード・ブロックを指定できる。間隔は、明示的に、または間隔型のパラメータを通して指定できます。
ON START TRANSACTION および **ON END TRANSACTION**	(オプション) START/END トランザクション・ブロック内で指定されている SPLASH 文は、各トランザクションのそれぞれ開始／終了時に実行される。 **START TRANSACTION** ブロックまたは **END TRANSACTION** ブロックは、他のブロックを使用することなく、個別に指定できます。

使用法

CREATE FLEX 文は、複数の入力要素を受け付けて 1 つの出力要素を生成するフレックス演算子を作成するために使用されます。 入力要素には、データ・モデルで定義されている既存のストリーム、デルタ・ストリーム、ウィンドウを指定できます。入力要素がデルタ・ストリームまたはウィンドウの場合、**KEEP** 句を指定して保持ポリシーを変更できます。 出力要素には、ストリーム、デルタ・ストリーム、またはウィンドウを指定でき、フレックス演算子によって生成される一意の名前が割り当てられます。すべての出力要素タイプで **SCHEMA** 句を指定する必要があります。出力要素がデルタ・ストリームまたはウィンドウの場合は、**PRIMARY KEY** を指定する必要があります。

ON 入力句は、特定の入力要素からの入力に対する処理ロジックを定義します。フレックス演算子の入力ごとに、**ON** 入力句を指定する必要があります。**ON START TRANSACTION** 句と **ON END TRANSACTION** 句はオプションで、各トランザクションのそれぞれ開始時と終了時に実行される処理ロジックを定義します。オプションの **EVERY** 間隔句は、定期的に実行されるロジックを定義します。

制限事項

- **KEEP** 句は、入力要素がウィンドウまたはデルタ・ストリームの場合に、フレックス演算子の入力に対して指定できる。
- **ON** 入力と **EVERY** 句内では、関数を宣言できない。
- イベント・キャッシュ・タイプは、ストリームに関連付けられているローカル **DECLARE** ブロックでのみ定義できる。
- フレックス・デルタ・ストリーム (出力がデルタ・ストリームであるフレックス・ストリーム) は、opcode が update または upsert のレコードを生成するため

には使用できない。これらの opcode のレコードを生成するには、フレックス・デルタ・ストリームの代わりにフレックス・ウィンドウを使用します。

- SPLASH 出力文は、フレックス演算子のローカル DECLARE ブロックで定義されている関数の本体内のみで使用できる。グローバル DECLARE ブロックまたは他の要素のローカル DECLARE ブロックで定義されている関数の本体内では使用できません。

例

次の例は、5 秒ごとの平均取引価格を計算します。

```
CREATE FLEX ComputeAveragePrice
 IN NASDAQ_Trades
 OUT OUTPUT WINDOW AverageTradePrice SCHEMA (Symbol string,
 AveragePrice money(4) ) PRIMARY KEY(Symbol)
 BEGIN
  DECLARE
   typedef [|money(4) TotalPrice; integer NumOfTrades] totalRec_t;
   dictionary(string,totalRec_t) averageDictionary;
 END;
 ON NASDAQ_Trades {
  totalRec_t rec := averageDictionary[NASDAQ_Trades.Symbol];
  if( isnull(rec) ) {
   averageDictionary[NASDAQ_Trades.Symbol] :=
   [|TotalPrice = NASDAQ_Trades.Price; NumOfTrades = 1];
  } else {
   // accumulate the total price and number of trades per input record
   averageDictionary[NASDAQ_Trades.Symbol] :=
   [|TotalPrice=rec.TotalPrice + NASDAQ_Trades.Price;
   NumOfTrades=rec.NumOfTrades + 1];
  }
 };
 EVERY 5 SECONDS {
  totalRec_t rec;
  for (sym in averageDictionary ) {
   rec := averageDictionary[sym];
   output setOpcode([Symbol=sym;|AveragePrice=(rec.TotalPrice/
rec.NumOfTrades);], upsert);
  }
 };
 END;
```

参照：

- *IN 句* (98 ページ)

CREATE LIBRARY 文

外部 C/C++ 関数と外部 Java 関数は、**CREATE LIBRARY** 文を使用して CCL プロジェクトで宣言した後に使用します。

構文

```
CREATE LIBRARY libraryName LANGUAGE {C|JAVA} FROM fileName(
returnType funcName (argType [argName],...);
...);
```

コンポーネント

libraryName	ユーザ指定のライブラリ名。
C、JAVA	ライブラリの言語を定義。名前は大文字と小文字が区別されません。
fileName	C/C++ 関数の場合、共有ライブラリのディレクトリ。カレント・ディレクトリに相対するパスを使用できます。 Java 関数の場合、.class サフィックスを除いたクラス・ファイルの名前です。文字列パラメータとして指定できます。Event Stream Processor サーバの起動時に -j オプションを使用して、クラス・ファイルのロケーションを提供できます。
funcName	宣言した関数の名前。
returnType、argType	それぞれ、関数の戻り値と関数の引数のデータ型。
argName	関数の引数の名前。

使用法

`libraryName.funcName` 表記を使用して、宣言した関数を呼び出します。

IMPORT 文を使用して、**CREATE LIBRARY** 文を別の CCL ファイルからメイン・プロジェクトにインポートします。

CREATE LIBRARY 文を使用して参照できる外部ライブラリは 1 つのみですが、**CREATE LIBRARY** 文を複数回使用して、その外部ライブラリを何回でも参照できます。

ライブラリは複数定義できます。このことは、ライブラリを宣言せずに使用できることを意味します。ただし、グローバルなユーザ定義関数が外部 C/C++ 関数または外部 Java 関数を使用する場合、ライブラリを宣言し、関数のシグネチャを指定した後に、グローバル DECLARE ブロックを宣言する必要があります。

注意： C/C++ の外部ライブラリ呼び出しは、すべてのデータ型を、具体的には、ブール、整数、長整数、浮動小数点数、通貨型 (n)、日付、bigdatetime、バイナリ、文字列をサポートします。

Java の外部ライブラリ呼び出しは、整数、長整数、倍精度浮動小数点数、文字列のデータ型のみをサポートします。

辞書、ベクトル、イベント・キャッシュ、レコード型などの複雑な型は、外部関数でサポートされていません。

例

C/C++ ライブラリの作成

```
CREATE LIBRARY MyCFunctions LANGUAGE C FROM '/opt/sybase/
MyFunctions.so' (
    integer MyFunc1 (integer, integer, float);
    string MyFunc2(string);
);
```

Java 関数の作成

```
CREATE LIBRARY MyJavaFunctions LANGUAGE JAVA FROM 'MyClass' (
    integer MyFunc1 (integer, integer, float);
    string MyFunc2(string);
);
```

参照：

- *IMPORT 文* (91 ページ)

CREATE LOG STORE 文

複数のステートフル要素をディスク上に保持します。

構文

```
CREATE [DEFAULT] LOG STORE storename
PROPERTIES
filename='filepath'
[sync={ true | false},]
[sweepamount=size,]
[reservepct=size,]
[ckcount=size,]
[maxfilesize=filesize];
```

パラメータ

filename	ログ・ストア・ファイルが書き込まれるフォルダの絶対パスまたは相対パス。相対パスの指定をおすすめします。

maxfilesize	ログ・ストア・ファイルの最大サイズ (MB 単位)。デフォルトは 8MB です。
sync	ストリームが更新されるたびにストリームと同期して永続化データを更新するかどうかを指定します。 true の値では、システムで受信確認されるすべてのレコードが永続化されますが、パフォーマンスが犠牲になります。false の値では、パフォーマンスは向上しますが、受信確認されても、まだ永続化されていないデータを失う可能性があります。デフォルトは false です。
reservepct	空き領域として保持するログの割合。デフォルトは 20% です。
sweepamount	単一のパスでクリーンアップできる、メガバイト単位のデータ量。デフォルトは、**maxfilesize** の 20% です。
ckcount	作成されるレコードの最大数。この数を超えると、中間メタデータが作成されます。デフォルトは 10,000 です。

コンポーネント

`storename`	ステートフル要素の **STORE** 句で参照可能な識別子。一意である必要があります。
`filepath`	ログ・ストア・フォルダへのパス。一重引用符で囲む必要があります。
`size`	整数。
`filesize`	MB 単位のサイズ。

使用法

ログ・ストアは、ディスクに最適化されたストアで、ディスク上で永続性を確保します。ログ・ストアに割り当てられたウィンドウのステータスは、リカバリ時にリストアされます。ログ・ストア・ウィンドウからデータを受け取るメモリ・ストア・ウィンドウのステータスは、可能な場合にかぎり、再計算されます。ログ・ストアは、メモリ・マップ・ファイルとして実装されます。 **filename** パラメータは必須ですが、**sync**、**sweepamount**、**reservepct**、**ckcount** はオプションです。これらのパラメータが指定されていない場合、ストアはデフォルト値を参照します。

パラメータは、**PARAMETERS** 句で指定します。順序は関係ありません。

メモリ・ストア用のパラメータはログ・ストアのパラメータに指定できませんし、その逆もまた真です。

DEFAULT を指定すると、ストアはモジュールまたはプロジェクトのデフォルト・ストアになります。このストアは、**STORE** 句でストアを明示的に指定しないステートフル要素に対して使用されます。プロジェクトまたはモジュールに対して

ストアが定義されていない場合、ステートフル要素を保持するために、デフォルトのメモリ・ストアが自動的に作成されます。

例

```
CREATE LOG STORE myStore
PROPERTIES
filename='myfile',
maxfiesize=16,
sweepamount=4,
ckcount=15000,
reservepct=20,
sync=false;
```

参照：

- *CREATE MEMORY STORE 文* (82 ページ)
- *ストア* (22 ページ)
- *永続性* (63 ページ)

CREATE MEMORY STORE 文

複数のステートフル要素をメモリに保持します。

構文

```
CREATE [DEFAULT] MEMORY STORE storename
[PROPERTIES
[INDEXTYPE={'tree'|'hash'},]
[INDEXSIZEHINT=size]]
```

パラメータ

INDEXTYPE	格納される要素のインデックス・メカニズムのタイプ。デフォルトは `'tree'` です。バイナリ・ツリーには、tree を使用します。バイナリ・ツリーは、メモリの使用量が予測可能で、処理速度が一定しています。ハッシュ・テーブルには hash を使用します。ハッシュ・テーブルは高速ですが、多くの場合、大量のメモリを消費します。
INDEXSIZEHINT	(オプション) hash を使用する場合、ハッシュ・テーブルの初期の要素数を決定する。値は 1024 の単位で指定します。この値が大きいほど多くのメモリが消費されますが、遅延時間のスパイクが発生する可能性は減ります。デフォルトは 8KB です。

コンポーネント

storename	ステートフル要素の **STORE** 句で参照可能な識別子。一意である必要があります。
'tree'	デフォルトのインデックス・メカニズム。
'hash'	代替のインデックス・メカニズム。

使用法

メモリ・ストアは主にメモリに常駐し、ディスク上での永続性は確保されません。**INDEXTYPE** パラメータはオプションです。このストアは、'tree' または 'hash' のインデックス・タイプをサポートします。インデックス・タイプとサイズのパラメータが指定されていない場合、ストアはデフォルト値を参照します。

パラメータを **PARAMETERS** 句で指定します。ただし、メモリ・ストアの場合はすべてのパラメータがオプションなので、この句もオプションです。プロパティはどのような順序で指定してもかまいません。

メモリ・ストア用のパラメータはログ・ストアに指定できませんし、その逆もまた真です。

DEFAULT を指定すると、ストアはモジュールまたはプロジェクトのデフォルト・ストアになります。このストアは、**STORE** 句でストアを明示的に指定しないステートフル要素に対して使用されます。プロジェクトまたはモジュールに対してストアが定義されていない場合、ステートフル要素を保持するために、デフォルトのメモリ・ストアが自動的に作成されます。

例

```
CREATE DEFAULT MEMORY STORE Store1 PROPERTIES INDEXTYPE='hash',
INDEXSIZEHINT=16;
```

参照：

- *CREATE LOG STORE 文* (80 ページ)
- *ストア* (22 ページ)
- *永続性* (63 ページ)

CREATE MODULE 文

LOAD MODULE 文を使用して CCL プロジェクトにロードできる、特別な機能を提供するモジュールを作成します。

構文

```
CREATE MODULE moduleName
IN input1 [,...]
OUT output1[,...]
BEGIN
    statements;
END;
```

コンポーネント

moduleName	モジュールの名前。
input1	入力ストリームまたは入力ウィンドウ。
output1	出力ストリームまたは出力ウィンドウ。

使用法

次を除くすべての CCL 文がモジュール内で有効です。

- **CREATE MODULE**
- **ATTACH ADAPTER**
- **ADAPTER START GROUPS**

moduleName は、文が存在するスコープ内のすべてのオブジェクト名を通して一意である必要があります。**IN** 句と **OUT** 句の名前は、**BEGIN-END** ブロックで定義されるストリームまたはウィンドウの名前に一致する必要があります。入力としてアクセスされるすべてのストリームまたはウィンドウは、**IN** 句内でリストされる必要があります。出力としてアクセスされるすべてのストリーム、ウィンドウ、デルタ・ストリーム (フレックス演算子によって作成されたものを含みます) は、**OUT** 句内でリストされる必要があります。それぞれ **IN** 句または **OUT** 句でリストされていない入力オブジェクトまたは出力オブジェクトがモジュール内に存在すると、コンパイラによってエラーが生成されます。

モジュール内で複数の **CREATE** 文 (**CREATE WINDOW** 文や **CREATE STREAM** 文など) を使用できますが、**CREATE STORE** 文は、モジュール外で使用できない特別な構文を使用します。モジュール内で使用される構文では、ストアのいずれのプロパティも指定できません。モジュール内の **CREATE STORE** 構文を次に示します。

```
CREATE [DEFAULT] {MEMORY|LOG} STORE store1-inmodule;
```

注意： **CREATE MODULE** 文のすべてのコンパイル・エラーは、致命的です。

制限事項

- **CREATE MODULE** 文は、モジュール定義内で使用できない。

例

次の例は、カラムの値に基づいてデータをフィルタする単純なモジュールを作成します。

```
CREATE MODULE filter_module
IN moduleIn
OUT moduleOut
BEGIN
    CREATE SCHEMA filter_schema (Value INTEGER);
    CREATE INPUT STREAM moduleIn SCHEMA filter_schema;
    CREATE OUTPUT STREAM moduleOut SCHEMA filterSchema AS SELECT *
FROM moduleIn WHERE moduleIn.Value > 10;
END;
```

参照：

- *CREATE LOG STORE 文* (80 ページ)
- *CREATE MEMORY STORE 文* (82 ページ)
- *IMPORT 文* (91 ページ)
- *LOAD MODULE 文* (92 ページ)

CREATE SCHEMA 文

以降に参照でき、モジュール内の 1 つ以上のクエリで再利用できる名前付きスキーマを定義します。

構文

```
CREATE SCHEMA name {(columnname type [,...])|
       INHERITS [FROM] schema_name [,...] [(columnname type [,...])]};
```

コンポーネント

name	ステートレス要素またはステートフル要素の定義時に参照される識別子。
columnname	カラムの一意な名前。
type	指定したカラムのデータ型。
schema_name	別のスキーマの名前。

使用法

CREATE SCHEMA 文は、ストリームやウィンドウなどのステートフル要素とステートレス要素によって参照可能な名前付きスキーマを定義します。インライ

ン・スキーマ定義として、または別のスキーマから定義を継承するようにスキーマを定義できます。

既存のスキーマ定義を継承するように設定し、さらにカラムを追加することによって、スキーマを拡張できます。指定した追加カラムは、継承したスキーマに追加されます。カラムを追加しない場合、継承したスキーマ定義は、指定した名前付きスキーマの正確なレプリカのままです。別の方法として、複数のスキーマ定義を継承してスキーマを拡張できます。

スキーマは、暗黙的に指定した順で連結されます。カラムも追加されます。これらのカラム名は一意である必要があり、重複している場合はエラーが発生します。

例

次の例は、`symbol_schema` と `trade_schema` の 2 つのスキーマを、`symbol_schema` から拡張して作成します。

```
CREATE SCHEMA symbol_schema (Symbol STRING);
CREATE SCHEMA trade_schema INHERITS FROM symbol_schema (Price
FLOAT);
```

参照：

- *スキーマ*(21 ページ)

DECLARE 文

DECLARE ブロック文は、CCL プロジェクトで使用される変数、パラメータ、typedef、関数を指定します。

構文

```
DECLARE
     [declaration;]
     ...
END;
```

使用法

CCL の DECLARE ブロックは、**DECLARE** 文、**END** 文、この 2 つの文の間のゼロ以上の宣言で構成されます。

DECLARE ブロック文は、変数、typedef、パラメータ、関数を定義するために使用できます。これらの宣言のそれぞれの構文を次に示します。

- 変数宣言は、SPLASH 構文を使用し、デフォルト値の指定が可能。
  ```
  datatypeName variableName [:=any_expression] [,...]
  ```
- typedef はデータ型の新しい名前を宣言。
  ```
  existingdatatypeName newdatatypeName
  ```

- パラメータ宣言は修飾子 **parameter** を使用し、デフォルト値の指定が可能。
  ```
  parameter datatypeName parameterName [:=constant_expression]
  ```

DECLARE ブロックは、ローカルにもグローバルにもできます。DECLARE ブロックを他のステートフル要素またはステートレス要素と一緒に使用すると、これらのブロックはローカル DECLARE ブロックになります。ローカル DECLARE ブロックは、それが使用されるステートフル要素またはステートレス要素内でのみアクセスできます。**DECLARE** ブロック文がモジュール内またはプロジェクト内で使用されると、これらのブロックはグローバル DECLARE ブロックになります。グローバル DECLARE ブロックは、プロジェクト内またはモジュール内のどこからでもアクセスできます。

DECLARE 文内の各宣言は、セミコロンで区切ります。

例

```
DECLARE
 ;
    integer x;   // defines a variable of type integer
   parameter string y; //declares a paramter (can specified only in a
global declare block)
END
```

参照：

- *CREATE DELTA STREAM 文* (72 ページ)
- *PRIMARY KEY 句* (104 ページ)
- *SCHEMA 句* (105 ページ)
- *SELECT 句* (121 ページ)
- *DECLARE ブロック* (51 ページ)

CREATE STREAM 文

イベントを外部ソースから受信する入力ストリームか、または 1 つ以上の入力に適用された継続クエリの結果である、イベントの派生ストリームのいずれかを作成します。

構文

```
CREATE [ LOCAL | OUTPUT ] STREAM name [ schema_clause ]
 [ local-declare-block ]
as_clause
;

CREATE INPUT STREAM name schema_clause
filter-expression-clause
;
```

コンポーネント

schema_clause	スキーマを指定。スキーマ句は、入力ストリームでは必須ですが、ローカル・ストリームと出力ストリームではオプションです。スキーマがローカル・ストリームと出力ストリームで指定されていない場合、クエリの仕様に基づいて、コンパイラによって自動的に推測されます。
filter-expression-clause	入力ストリームで指定可能。この句は、タプルをアダプタまたは外部のパブリッシャから受け付ける前にフィルタします。 カラムを *node name.column name* のフォーマットで指定します。入力ストリームの場合、ストリーム名をノード名に使用します。ローカル・ストリームと出力ストリームの場合、その名前か、**FROM** 句からのエイリアスを使用します。
local-declare-block	クエリ内の式でアクセス可能な変数と関数の宣言を可能にする。入力ストリームでは、local-declare-block を定義できません。
as_clause	文へのクエリの取り込みを指定。

使用法

CREATE STREAM 文は、ストリームと呼ばれるステートレス要素を明示的に作成します。ストリームは、入力、出力、またはローカルとして指定できます。入力ストリームには、必須のスキーマと、以降の処理が実行される前に不要なデータを削除するフィルタ式があります。ストリームは各受信イベントを処理し、出力をパブリッシュします。その後に、次のイベントを処理する準備を整えます。

出力ストリームとローカル・ストリームでは、スキーマ指定はオプションです。これらのストリームでは、ローカル DECLARE ブロックを宣言して、クエリの **SELECT** 句で使用できる変数と関数を定義できます。

例

次の例は、フィルタが指定されている入力ストリームを作成します。

```
CREATE INPUT STREAM InStr
SCHEMA (Col1 INTEGER, Col2 STRING)
WHERE InStr.Col2='abcd';
```

次の例は、出力ストリームを作成し、スキーマは **SELECT** 句によって暗黙的に決定されます。

```
CREATE OUTPUT STREAM OutStr
SELECT InStr.Col1, INStr.Col2
FROM InStr
WHERE INStr.Col1 > 1000;
```

参照：

- *SCHEMA 句* (105 ページ)

- *WHERE 句* (125 ページ)
- *DECLARE 文* (86 ページ)
- *SELECT 句* (121 ページ)
- *FROM 句：ANSI 構文* (112 ページ)
- *FROM 句：カンマ区切りの構文* (111 ページ)
- *AS 句* (97 ページ)

CREATE WINDOW 文

参照可能で、1 つ以上のダウンストリーム演算子によって使用可能な名前付きウィンドウを定義します。または、出力ウィンドウの場合は、結果をパブリッシュするために使用できる名前付きウィンドウを定義します。

構文

```
CREATE INPUT WINDOW name schema_clause
primary_key_clause
[store_clause]
[keep_clause];

CREATE [ LOCAL | OUTPUT ] WINDOW name schema_clause
{ PRIMARY KEY (column1, column2, ...) | PRIMARY KEY DEDUCED }
[store_clause]
[aging_clause]
[keep_clause]
[splash-block]
as_clause
;
```

コンポーネント

name	作成するウィンドウの名前。
schema_clause	入力ウィンドウでは必須、ローカル・ウィンドウと出力ウィンドウではオプション。スキーマ句がローカル・ウィンドウと出力ウィンドウで指定されていない場合、コンパイラによって自動的に推測されます。
primary_key_clause	プライマリ・キーを設定。
store_clause	(オプション) レコードのステータスを格納するために使用される物理メカニズムを指定。句が何も指定されていない場合は、プロジェクトまたはモジュールのデフォルトが適用されます。

keep_clause	(オプション) ウィンドウの保持ポリシーを指定。指定されていない場合、ウィンドウはデフォルトの KEEP ALL 保持ポリシーを使用します。
aging_clause	(オプション) データ・エイジング・ポリシーを指定。出力ウィンドウまたはローカル・ウィンドウでのみ使用されます。
splash-block	(オプション) クエリでアクセスできる変数と関数を宣言する SPLASH 文の選択。出力ウィンドウまたはローカル・ウィンドウでのみ使用されます。
as_clause	文へのクエリの取り込みを指定。

使用法

SCHEMA 句と **PRIMARY KEY** 句は、入力ウィンドウでは必須です。**SCHEMA** 句は、派生ウィンドウではオプションです。**SCHEMA** が定義されていないと、コンパイラは射影リストに基づいて暗黙的に決定します。派生ウィンドウの場合、プライマリ・キーは推測されるか、明示的に指定されます。これらの規則には、いくつかの例外があります。それらは、該当する箇所で説明されます。

CREATE WINDOW 文には、レコードの格納方法を決定する **STORE** 句と、格納するレコード数と期間を決定する **KEEP** 句も指定できます。入力、出力、またはローカルのいずれのウィンドウでも指定できます。ローカル・ウィンドウと出力ウィンドウでは、データのエイジング・ポリシーを指定する **AGING** 句を指定できます。

例

過去 10 分間に受信したポジション・レコードのみを含むローカル・ウィンドウを作成します。

```
CREATE WINDOW TradesAge
PRIMARY KEY DEDUCED
KEEP 10 MINUTES
AGES EVERY 5 SECONDS SET AgeColumn 5 TIMES
AS
SELECT Trades.*, 0 AgeColumn FROM Trades;
```

参照：

- *SCHEMA 句* (105 ページ)
- *PRIMARY KEY 句* (104 ページ)
- *STORE 句* (106 ページ)
- *AGING 句* (95 ページ)
- *KEEP 句* (100 ページ)
- *ユーザ定義の SPLASH 関数* (130 ページ)

- *SELECT 句* (121 ページ)
- *FROM 句：ANSI 構文* (112 ページ)
- *FROM 句：カンマ区切りの構文* (111 ページ)
- *AS 句* (97 ページ)

IMPORT 文

CCL ファイルから、ライブラリ、パラメータ、変数、スキーマ定義、関数定義、モジュール定義をプロジェクト、モジュール、または別の **IMPORT** ファイルにインポートします。

構文

```
IMPORT 'fileName';
```

コンポーネント

`fileName`	インポートする CCL テキスト・ファイルの絶対パスまたは相対パス。 相対パスは、**IMPORT** 文が定義されているファイルのファイル・ロケーションに相対です。

使用法

インポート・ファイルでは、次の CCL 文のみが有効です。ファイルに他の文があると、コンパイラによってエラー・メッセージが生成されます。

- **IMPORT**
- **CREATE MODULE**
- **DECLARE**
- **CREATE SCHEMA**
- **CREATE LIBRARY**

インポート・ファイルで使用される定義は、そのファイル内で定義されているか、ファイルによってインポートされる必要があります。これらの定義は、インポートされると、インポート先のスコープに属します。これらの定義は、**IMPORT** 文に続く文内でのみ使用できます。

インポート・ファイルは、**IMPORT** 文を使用することによって、他のインポート・ファイル内でネストできます。たとえば、ファイル A がファイル B をインポートし、プロジェクトがファイル A をインポートすると、プロジェクトは、ファイル A 内のすべての定義と、ファイル B 内のすべての定義にアクセスできます。

インポート循環は許可されず、コンパイラによって検出されます。たとえば、ファイル B がファイル A をインポートし、ファイル A がファイル B をインポートすると、コンパイラによって、ファイル A とファイル B との間に循環依存性があ

ることを示すエラー・メッセージが生成されます。単一のスコープ内で同じファイルを 2 回インポートすることも許可されません。これを行うと、エラー・メッセージが生成されます。

注意： インポート・ファイルをコンパイルできない場合、または **IMPORT** 文によって無効なファイル (不正なファイル・フォーマットまたはファイル・パス) のインポートが試行された場合には、プロジェクトを正しくコンパイルできません。

例

次の例は、2 つのスキーマをインポートし、使用します。

```
//Defines Schema1
//Imported using relative paths
IMPORT '../schemas/import1.ccl';

//Defines Schema2
//Imported using absolute paths
IMPORT '~/project/schemas/import2.ccl'; [For UNIX-based systems]
IMPORT 'C:/project/schemas/import2.ccl': [For Windows-based systems]

CREATE INPUT STREAM stream1 SCHEMA Schema1;
CREATE INPUT STREAM stream2 SCHEMA Schema2;
```

参照：

- *CREATE LIBRARY 文* (79 ページ)
- *モジュール性* (58 ページ)
- *CREATE MODULE 文* (84 ページ)
- *LOAD MODULE 文* (92 ページ)
- *DECLARE 文* (86 ページ)

LOAD MODULE 文

LOAD MODULE 文は、作成済みのモジュールをプロジェクトにロードします。

構文

```
LOAD MODULE modulename AS moduleIdentifier
    in-clause
    out-clause
    [parameters-clause]
    [stores-clause];
```

コンポーネント

moduleName	モジュールの名前。作成済みモジュールの名前に一致する必要があります。

module 識別子	識別子は、親スコープ内で一意である必要がある。
IN clause	モジュール内で定義されている入力ストリームまたは入力ウィンドウを、親スコープ内の作成済みストリームまたは作成済みウィンドウにバインドする。
OUT clause	モジュール内で定義されている 1 つ以上の出力ストリームを、一意な識別子を使用して親スコープに公開する。
PARAMETERS clause	モジュール内で定義されている 1 つ以上のパラメータを、ロード時に式にバインドするか、モジュール内のパラメータをメイン・プロジェクト内の別のパラメータにバインドする。パラメータにデフォルト値が定義されている場合、パラメータのバインドは必須ではありません。
STORES clause	モジュール内のストアを親スコープ内のストアにバインドする。

使用法

ロードされたモジュール内のすべてのストリームは、実行時にローカルにアクセスできます。つまり、サブスクライブ先、パブリッシュ元、またはクエリ先にはなれません。モジュールがサーバにロードされると、モジュール内のすべてのストリームとウィンドウと、出力を親スコープに公開することによって作成される出力ストリームと出力ウィンドウは、ローカルでアクセス可能であるかのように動作します。このため、モジュール内のストリームとウィンドウ、およびモジュールの公開された出力は、外部からクエリすることも、サブスクライブ先となることもできません。

LOAD MODULE は、以下をサポートします。

- **IN** 句
- **OUT** 句
- **PARAMETERS** 句
- **STORES** 句

注意： **LOAD MODULE** 文のすべてのコンパイル・エラーは、致命的です。

例

次の例は、生の株式取引情報を処理し、価格が 1.00 を超える取引のリストを出力するモジュールを使用します。モジュールは、**LOAD MODULE** 文を使用してメイン・プロジェクトにロードされます。

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
    CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
```

```
    Venue string,
    Symbol string,
    Price float,
    Shares integer
);

    CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
    CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema
    AS SELECT * FROM TradeData WHERE TradeData.Price > 1.00;
END;

CREATE INPUT STREAM NYSEData SCHEMA TradesSchema;

LOAD MODULE FilterByPrice AS FilterOver1 IN TradeData = NYSEData OUT
FilteredTradeData = NYSEPriceOver1Data;
```

参照：

- *モジュール性* (58 ページ)
- *IN 句* (98 ページ)
- *OUT 句* (101 ページ)
- *PARAMETERS 句* (102 ページ)
- *STORES 句* (107 ページ)

第 7 章　　句

文で使用されるさまざまな句の構文。

AGING 句

データ・エイジング・ポリシーを指定します。

構文

```
AGES EVERY agingTime SET agingField [maxAgingFieldValue TIMES] [FROM
agingTimeField]
```

コンポーネント

agingTime	期間。この期間が経過すると、データ経過期間プロセスが開始されます。期間を時間、分、秒、ミリ秒、またはマイクロ秒で指定します。単位を組み合わせても指定できます (たとえば、3 MINUTES 30 SECONDS)。 間隔パラメータを使用しても指定できます。
agingField	レコードのフィールドで、agingTime 期間が経過し、レコードに対していずれのアクティビティも発生しなかった場合ごとに 1 ずつ増分される。
maxAgingFieldValue	(オプション) agingField が増分される最大値。指定されていない場合、agingField は 1 回のみ増分されます。 間隔パラメータを使用しても指定できます。
agingTimeField	(オプション) エイジング・プロセスの起動時間を指定するフィールド。たとえば、agingTime カラムに指定した期間が経過すると、データ・エイジング・プロセスが開始されます。 指定しないと、内部ロー時間が使用されます。指定する場合は、フィールドに有効な起動時間を指定します。

使用法

データ・レコードが事前に定義された期間内に更新または削除されないと、それらは古いレコードとして扱われます。データ・レコードが古いレコードになると、通知が更新タプルとしてウィンドウのサブスクライバに送信されます。

注意： AGING 句は、ウィンドウなどのステートフル要素に対してのみ使用できます。

事前に定義された期間 (agingTime) が経過すると、レコード内の整数フィールド (agingField) が 1 回、または事前に定義された最大値 (maxAgingFieldValue) に到達するまで増分されます。エイジング・プロセスの起動時間は、レコードの agingTimeField を使用して指定されます。

起動時間が明示的に指定されていない場合、内部ロー時間が使用されます。エイジング・プロセスが起動されると、agingField がデフォルトで 0 に初期化され、事前に定義されている期間が経過するごとに 1 ずつ増分されます。エイジング・プロセスの開始後にレコードが更新されると、agingField が 0 にリセットされ、エイジング・プロセスが再起動されます。レコードが削除されると、エイジングの更新は生成されません。

挿入が受信されると、計数フィールドが 0 に設定され、挿入が渡され、エイジング・プロセスが起動されます。

エイジング・プロセスは、指定されている非アクティブ期間が経過した後に起動されます。データのエイジングが 5 秒に設定されていると、レコードに対して 5 秒間アクティビティがないと、計数フィールドの増分処理が開始されます。レコードは、更新または削除が発生しないと、非アクティブとして扱われます。

削除が受信されると、エイジング・プロセスは終了し、削除が渡されます。レコードの更新が受信されると、計数フィールドが 0 にリセットされます。

例

この例は、出力ウィンドウ `AgingWindow` を作成します。出力ウィンドウのエイジング・カラムは、10 秒ごとに 20 回更新されます。

```
CREATE  OUTPUT  WINDOW AgingWindow
  SCHEMA (
     AgeColumn integer,
     Symbol STRING,
     Ts bigdatetime )
  PRIMARY KEY (Symbol)
  AGES EVERY 10 SECONDS SET AgeColumn 20 TIMES
 AS
  SELECT 1 as AgeColumn,
  TradesWindow.Symbol AS Symbol,
  TradesWindow.Ts AS Ts
 FROM TradesWindow
;
```

参照：

- *CREATE WINDOW 文* (89 ページ)

AS 句

派生要素への CCL クエリの取り込みを指定します。

構文

```
[...]
    AS
        CCL Query
[...]
```

コンポーネント

AS	**AS** 句は、文の残りの部分への CCL クエリの取り込みを指定します。
CCL クエリ	CCL クエリの本体。

使用法

AS 句は、派生要素によって処理されるデータの型を決定する CCL クエリを提供するために、派生要素 (ストリーム、ウィンドウ、デルタ・ストリーム) 内で使用されます。このため、**AS** 句は、派生要素と共に使用される場合のみ有効です。

クエリを構築する方法の詳細については、「クエリ」を参照してください。

例

次の例は、派生ストリームで選択される情報を指定するために使用される **AS** 句を示します。

```
CREATE STREAM win1 SCHEMA ( col1 string )
    AS
        SELECT inputStream.col1
        FROM inputStream;
```

参照：

- *第 4 章、「CCL クエリの構築」*(39 ページ)
- *CREATE STREAM 文*(87 ページ)
- *CREATE WINDOW 文*(89 ページ)
- *第 8 章、「クエリ」*(109 ページ)

CASE 句

条件処理では、セット条件を評価して結果を決定します。

構文

```
CASE
       WHEN condition THEN expression [...] ELSE expression
END
```

コンポーネント

condition	ゼロかゼロでないかを評価する式。結果がゼロでない場合は、条件が true であることを示し、結果がゼロの場合は、条件が false であることを示します。
expression	評価された条件の結果。任意の有効な式または変数を指定できます。

使用法

CASE 句は順序に依存し、パラメータの **WHEN**、**THEN**、**ELSE** を必要とする条件式で構成されます。**WHEN** 条件は、特定のケースをフィルタし、条件セットが true かそれとも false かという評価によって結果を絞り込みます。true の場合、次の **THEN** 式が実行されます。false の場合は、後続の **WHEN** 条件がテストされます。

ELSE パラメータより前の条件がすべて false の場合、**ELSE** 式が実行されます。CASE 句は、キーワード **END** で終了します。

例

この例では、加重をフィルタし、各条件セットに対して数値を指定します。

```
CASE
WHEN weight<500 THEN 1
WHEN weight>1000 THEN 3
ELSE 2
END
```

IN 句

モジュールの入力を親スコープの入力にバインドするために **LOAD MODULE** 文内で使用されます。

構文

```
IN
        input1-inModule = input1-parentScope [,...]
```

コンポーネント

input1-inModule	モジュール内で定義されている入力ストリームまたは入力ウィンドウの名前。
input1-parentScope	親スコープ内のストリームまたはウィンドウの名前。モジュールの入力ストリームまたは入力ウィンドウをこのストリームにバインドします。

使用法

親スコープのストリームまたはウィンドウは、いずれのアクセス性のタイプでもかまいません。バインドされる入力ストリームまたは入力ウィンドウ間のスキーマには、互換性がある必要があります。スキーマに互換性があるとは、次の要件のいずれか 1 つが満たされることです。

- カラムは、数とデータ型が同じで、同じ順序である。
- 親スコープのストリームにはモジュールのストリームよりも多くのカラムがあり、先頭からのカラムは、データ型と順序が同じである。余分なカラムはモジュールによって無視され、プライマリ・キー・カラムにはなれません。
- 親モジュールのストリームはモジュールのストリームよりもカラムが少なく、先頭からのカラムは、データ型と順序が同じである。モジュール内の余分なカラムには、null 値が挿入されます。null 値のカラムはプライマリ・キーになれません。

注意： これらの各要件で、カラム名が一致する必要はありません。

入力を関連付ける場合、プライマリ・キーのない親レベルのオブジェクトは、プライマリ・キーを必要とするモジュールレベルのオブジェクトにバインドできません。たとえば、ストリームはウィンドウにバインドできません。

制限事項

- モジュールのすべての入力要素は、**IN** 句を使用してバインドされる必要があります。

例

この例は、モジュール (`modMarketIn1` と `modMarketIn2`) 内の入力ストリームが、親スコープ (`marketIn1` と `marketIn2`) の対応するストリームにバインドされることを示します。

```
LOAD MODULE filterModule AS filter1
IN modMarketIn1=marketIn1, modMarketIn2=marketIn2
OUT modMarketOut=marketOut;
```

参照：

- *モジュール性* (58 ページ)

- *LOAD MODULE 文* (92 ページ)
- *OUT 句* (101 ページ)
- *PARAMETERS 句* (102 ページ)
- *STORES 句* (107 ページ)

KEEP 句

保持するレコード数とその期間を指定します。

構文

```
KEEP { count_policy | time_policy | ALL }
count_policy: X ROW[S] SLACK Y
time_policy: interval_spec
```

コンポーネント

count_policy	ウィンドウ内のレコード数に基づくレコードの保持。
X	ウィンドウ内の最大ロー数を指定する整数リテラル、定数整数式、または整数パラメータ。
SLACK	ローに基づく保持で存在可能な最大スラックを指定。スラック値を大きくすると、ローを挿入するたびに別のローを削除する必要がなくなるのでパフォーマンスが向上します。
Y	ウィンドウ内で許可される最大スラックを指定する整数リテラル、定数整数式、または整数パラメータ。ウィンドウは、最大で X+Y ローを格納できます。デフォルト値は 1 です。
time_policy	レコードがウィンドウに保持される期間の長さに基づくレコードの保持。
interval_spec	ウィンドウ内のローの最大経過時間を指定する間隔リテラル、定数間隔式、または間隔パラメータ。マイクロ秒値または小さなミリ秒値は、「できるだけ早急に」削除することを意味します。
ALL	すべてのローを保持。

使用法

KEEP 句は、名前付きウィンドウまたは名前なしウィンドウの保持ポリシーを定義します。 ウィンドウの保持ポリシーには、時間基準のポリシー (ウィンドウがローを保持する期間) とカウント基準のポリシー (ウィンドウが保持できるローの最大数) があります。 ウィンドウ定義から **KEEP** 句を省略すると、デフォルト・ポリシーの KEEP ALL が適用されます。

KEEP 句の位置によって、名前なしウィンドウを作成するかどうかが決定されます。名前付きウィンドウは **CREATE WINDOW** 文を使用して作成され、**KEEP** 句は、この定義の一部として構成されます。ただし、文のクエリ部分に **KEEP** 句がある場合は、名前なしウィンドウが暗黙的に作成されます。これは、**KEEP** 句がクエリの **FROM** 句にアタッチされている場合にもあてはまります。

例
最初の **KEEP** 句は、名前付きウィンドウ RecentAvgVols の保持ポリシーを指定し、レコードを 10 分間保持します。**KEEP** 句の 2 つ目のインスタンスは、過去 5 分間の平均量を格納する名前なしウィンドウを作成します。

```
CREATE WINDOW RecentAvgVols PRIMARY KEY DEDUCED
KEEP 10 MINS
AS
SELECT Trades.Symbol, Avg(Trades.Volume) as Vol
FROM Trades KEEP 5 MINS
GROUP BY Trades.Symbol
```

参照：

- *ジョイン* (41 ページ)
- *CREATE DELTA STREAM 文* (72 ページ)
- *CREATE WINDOW 文* (89 ページ)
- *保持* (19 ページ)
- *永続性* (63 ページ)

OUT 句

モジュールの出力を親スコープに公開するために **LOAD MODULE** 文内で使用されます。

構文

```
OUT
        output1-inModule = output1-parentScope [,...]
```

コンポーネント

output1-inModule	モジュール内で定義されている出力の名前。
output1-parentScope	親スコープに出力を公開するために使用される名前。

使用法
LOAD MODULE 文で作成される公開出力ストリームは、ローカルでアクセスできます。つまり、出力アダプタを出力ストリームに直接アタッチできません。出力は、`output1-parentScope` 識別子を使用して親スコープに公開されます。`output`

マッピングは、親スコープから参照できるようにするために、モジュールの出力に一意な名前を割り当てます。

制限事項

- 1 つ以上の出力ストリームが親スコープに公開される必要がある。

例

この例は、モジュールの出力 (`modFilteredOut` と `marketAverageOut`) を、それぞれ `filteredOut` と `averageOut` の名前で公開します。

```
LOAD MODULE filterModule AS filter1
IN modMarketIn=marketIn1
OUT modFilteredOut=filteredOut, marketAverageOut=averageOut;
```

参照：

- *モジュール性* (58 ページ)
- *LOAD MODULE 文* (92 ページ)
- *IN 句* (98 ページ)
- *PARAMETERS 句* (102 ページ)
- *STORES 句* (107 ページ)

PARAMETERS 句

LOAD MODULE 文内で使用され、モジュール内のパラメータの値を、ロード時にバインドすることによって提供します。

構文

```
PARAMETERS
        parameter1-inModule = value-parentScope [,...]
```

コンポーネント

`parameter1-inModule`	モジュール内で定義されているパラメータの名前。
`value-parentScope`	バインド先の親スコープ内の値。この値は、親スコープ内で定義されている式または別のパラメータです。

使用法

パラメータのバインドとは、モジュール内のパラメータの値を、ロード時に提供する処理を意味します。これは、モジュールのインスタンスごとに固有のパラメータ値を提供できることを意味します。**LOAD MODULE** 文では、モジュール内のパラメータを次のものにバインドできます。

- 親スコープ内で宣言されている別のパラメータ。または、
- モジュールのロード時の式。

注意： パラメータまたは変数が関係する式は、パラメータのデフォルト値と変数の初期値を使用して、コンパイル時に評価されます。バインド式のパラメータまたは変数でデフォルト値が利用できない場合は、エラーが生成されます。

モジュール内で定義されているパラメータを実行時に直接バインドできません。これを行うと、サーバ警告が生成されます。モジュール・パラメータは、**LOAD MODULE** 文を介してのみバインドできます。

例

次の例は、モジュールのパラメータを別の値 (`minValue=2`) と別のパラメータ (`maxValue=serverMaxValue`) にマップします。

```
CREATE MODULE filterModule
IN filterIn
OUT filterOut
BEGIN
    CREATE SCHEMA filterSchema (Value Integer);
    DECLARE
        PARAMETER Integer minValue := 4;
        PARAMETER Integer maxValue;
    END;
    CREATE INPUT STREAM filterIn SCHEMA filterSchema;
    CREATE OUTPUT STREAM filterOut SCHEMA filterSchema AS SELECT *
FROM filterIn WHERE filterIn.Value > minValue and filterIn.Value <
maxValue;
END;

DECLARE
    PARAMETER Integer serverMaxValue;
END;

LOAD MODULE filterModule AS filter1
IN filterIn=marketIn
OUT filterOut=marketOut
PARAMETERS minValue=2, maxValue=serverMaxValue;
```

参照：

- *モジュール性* (58 ページ)
- *LOAD MODULE 文* (92 ページ)
- *IN 句* (98 ページ)
- *OUT 句* (101 ページ)
- *STORES 句* (107 ページ)
- *例：モジュール内のパラメータ* (61 ページ)

PRIMARY KEY 句

デルタ・ストリームまたはウィンドウのプライマリ・キーを指定します。

構文

```
PRIMARY KEY (column [,...]) | PRIMARY KEY DEDUCED
```

コンポーネント

column	要素のスキーマ内のカラムの名前。

使用法

プライマリ・キーはレコードを一意に識別します。ウィンドウとデルタ・ストリームで必須です。

プライマリ・キーは、通常、"strict" として扱われます。既存のレコードへの挿入、存在しないレコードの更新または削除など、一貫性ルールに違反するすべてのレコードは無視され、ログにレポートされます。

プライマリ・キーは、保持ポリシーがウィンドウで有効であると、"lax" として扱われます。**KEEP** 句によって発生するレコードの期限切れは、受信レコードとの間で矛盾を発生させせます。既存のレコードへの挿入は更新として扱われ、存在しないレコードの更新は挿入として扱われます。存在しないレコードの削除は、暗黙的に無視されます (safedelete として)。この動作は、関連する 2 つのレコードに保持期限ポリシーが適用されており、ターゲット・ウィンドウでこれより短い時間の保持期限が設定されている場合に発生することがあります。

使用法：明示的なプライマリ・キー

明示的に定義されたプライマリ・キーは、**PRIMARY KEY** 句を使用し、ウィンドウまたはデルタ・ストリームのスキーマの 1 つ以上のカラムを参照します。プライマリ・キーが指定されると、エンジンによって制約が強制され、不正な操作は不正レコードとしてフラグが設定され、実行時に無視されます。この問題を回避するには、プライマリ・キーを正しく定義します。

使用法：抽出されたプライマリ・キー

プライマリ・キーが **PRIMARY KEY DEDUCED** として指定されていると、コンパイラは自動的にプライマリ・キーを抽出します。プライマリ・キーが抽出されない場合、コンパイル・エラーが生成されます。

プライマリ・キーは次のように抽出されます。

- プライマリ・キーは、入力ウィンドウとフレックス演算子に対しては抽出されない。明示的に指定する必要があります。

- 単一ソース・クエリの場合、集約操作を除いて、プライマリ・キーはソースから抽出される。キー抽出が成功するには、ソースからのすべてのキー・カラムが一字一句そのままコピーされる必要があります。
- 集約操作の場合、プライマリ・キーは、GROUP BY 式を含む射影のカラムである。

注意： すべての **GROUP BY** 句が射影リスト内に存在する必要があります。同じ式が複数のカラムに存在する場合、**GROUP BY** 句を持つ最初のカラムがプライマリ・キーになります。

ジョインの場合、次の規則が適用されます。

- 左外部ジョインと右外部ジョインの場合、キーは外側から抽出される。たとえば、左ジョインの場合は左側、右ジョインの場合は右側です。プライマリ・キーの抽出が適切に動作するには、外側からのすべてのキー・カラムが、射影に存在する必要があります。
- 内部ジョインの場合、ジョインのカーディナリティに依存する。1 対多のカーディナリティの場合、キーは、多の側から抽出されます。多対多のカーディナリティの場合、抽出されるキーは、ジョインの両側からのキーの組み合わせになります。1 対 1 のカーディナリティの場合、キーはどちらかの側から抽出されます。キーとして選択される側は、正確に決定できません。すべての場合でキー抽出が成功するには、キーの候補となるカラムがソースから直接にコピーされる必要があります。
- 全外部ジョインの場合、ジョインの両側のキー・フィールドを引数として持つ coalesce() 関数のみを含むカラムが、キー・カラムとして抽出される。
- 複数ウィンドウのジョインの場合、これらの規則は遷移的に適用される。

参照：

- *CREATE DELTA STREAM 文* (72 ページ)
- *DECLARE 文* (86 ページ)
- *SCHEMA 句* (105 ページ)
- *SELECT 句* (121 ページ)
- *CREATE WINDOW 文* (89 ページ)

SCHEMA 句

新しいストリームとウィンドウのスキーマ定義を提供します。

構文

```
SCHEMA name | (column type [,...])
```

コンポーネント

name	スキーマの名前。指定されたモジュール内で一意である必要があります。
column	カラムの名前。
type	カラムのエントリのデータ型。

使用法

SCHEMA 句は、ストリームまたはウィンドウのカラムとデータ型 (インライン・スキーマ) を定義するか、定義済みの名前付きスキーマを参照します。別の CCL ファイルからインポートされたスキーマを参照することもあります。

SCHEMA 句は、入力ストリーム、入力ウィンドウ、フレックス演算子で必須です。他のすべての場合では、オプションです。指定されていない場合、射影リストのカラムによって暗黙的に決定されます。

UNION の場合、スキーマが明示的に指定されていないと、**UNION** の最初の **SELECT** 文から暗黙的に決定されます。

参照：

- *CREATE DELTA STREAM 文* (72 ページ)
- *DECLARE 文* (86 ページ)
- *PRIMARY KEY 句* (104 ページ)
- *SELECT 句* (121 ページ)
- *CREATE STREAM 文* (87 ページ)
- *CREATE WINDOW 文* (89 ページ)
- *UNION 演算子* (123 ページ)

STORE 句

任意のステートフル要素定義内で、要素のストアを割り当てます。

構文

```
STORE storename
```

参照：

- *永続性* (63 ページ)
- *CREATE WINDOW 文* (89 ページ)

STORES 句

モジュール内のストアを親スコープのストアにバインドするために **LOAD MODULE** 文内で使用されます。

構文

```
STORES
        store1-inModule = store1-parentScope [,...]
```

コンポーネント

store-inModule	モジュール内で定義されているストアの名前。
store1-parentScope	親スコープ内のストアの名前。モジュールのストアをこのストアにバインドします。

使用法

ストアがバインドされていない場合、コンパイル・エラーが発生します。ストアを指定せずにウィンドウを作成し、デフォルトのストアを作成していない場合、デフォルトのパーサ生成のメモリ・ストアが、モジュール用に一時的に作成されます。モジュールをロードすると、このパーサ生成のストアが親スコープのデフォルトのメモリ・ストアに割り当てられます。デフォルトのメモリ・ストアが親スコープに存在しない場合、モジュール内のパーサ生成のメモリ・ストアが、親スコープ内で生成されたパーサ生成のメモリ・ストアに割り当てられます。

注意： モジュールは、ストア依存関係ループに関連する可能性があります。すべての依存関係ループは無効であるので、モジュール内に依存関係ループのインスタンスがあると、モジュールはコンパイルできません。

制限事項

- 同じタイプのストアのみをバインドできます。たとえば、ログ・ストアを別のログ・ストアと、メモリ・ストアを別のメモリ・ストアとバインドできます。

例

次の例は、モジュール内のストアを親スコープ内のストアにマップします。

```
CREATE MODULE filterModule
IN filterIn
OUT filterOut
BEGIN
    CREATE MEMORY STORE filterStore;
    CREATE SCHEMA filterSchema (ID Integer, Value Integer);
    CREATE INPUT WINDOW filterIn SCHEMA filterSchema PRIMARY KEY ID
STORE filterStore;
    CREATE OUTPUT WINDOW filterOut SCHEMA filterSchema PRIMARY KEY
DEDUCED STORE filterSTore AS SELECT * FROM filterIn WHERE
```

```
filterIn.Value > 10;
END;

CREATE MEMORY STORE mainStore;
CREATE SCHEMA filterSchema (ID Integer, Value Integer);

LOAD MODULE filterModule AS filter1
IN filterIn=marketIn
OUT filterOUT=marketOut
STORES filterStore=mainStore;
```

参照：

- *モジュール性* (58 ページ)
- *LOAD MODULE 文* (92 ページ)
- *IN 句* (98 ページ)
- *OUT 句* (101 ページ)
- *PARAMETERS 句* (102 ページ)

第 8 章　クエリ

句と演算子を使用してクエリを構築し、機能を指定します。ここでは、クエリ、クエリ句、演算子のリファレンスを提供します。

構文

```
select_clause
from_clause
[matching_clause]
[where_clause]
[groupFilter_clause]
[groupBy_clause]
[groupOrder_clause]
[having_clause]
```

コンポーネント

select_clause	ソースからカラムを選択する。詳細については、以下と、「SELECT 句」を参照してください。
from_clause	データを抽出するソースを選択する。詳細については、以下と、「FROM 句」を参照してください。
matching_clause	パターン一致に使用。詳細については、「MATCHING 句」と「パターン一致」を参照してください。
where_clause	フィルタを実施。詳細については、「WHERE 句」と「フィルタ」を参照してください。
groupFilter_clause	集約時に受信データをフィルタリングする。詳細については、「GROUP FILTER 句」と「集約」を参照してください。
groupBy_clause	集約操作を使用するローのコレクションを指定する。詳細については、「GROUP BY 句」と「集約」を参照してください。
groupOrder_clause	集約前にグループのデータを並べ替える。詳細については、「GROUP ORDER BY 句」と「集約」を参照してください。
having_clause	集約時に抽出コンポーネントによって出力されるデータをフィルタする。詳細については、「HAVING 句」と「集約」を参照してください。

使用法
クエリは、「CCL クエリの構築」の章で説明するように、上述の句を使用してさまざまな機能を実行できます。ただし、基本構造は、クエリを開始するときと同じ構造のままです。以下の例では、どのクエリでも使用される **SELECT** 句と **FROM** 句の使用方法を説明します。

SELECT 句は、**AS** 句の直後に使用されます。**SELECT** 句の目的は、ソースまたは式からどのカラムが使用するかを決定することです。

SELECT 句に続いて、**FROM** 句で、クエリが使用するソースを指定します。**FROM** 句の後には、クエリされるデータに対して、フィルタ、ユニオン、ジョイン、パターン一致、集約を使用するために利用可能な句を実装します。

例
この例では、取引総額、出来高、証券コードごとの VWAP を 5 分間隔で取得します。

```
[...]
SELECT
    q.Symbol,
    (trunc(q.TradeTime) + (((q.TradeTime - trunc(q.TradeTime))/
300)*300)) FiveMinuteBucket,
    sum(q.Shares * q.Price)/sum(q.Shares) Vwap,
    count(*) TotalTrades,
    sum(q.Shares) TotalVolume
FROM
    QTrades q
[...]
```

参照：

- *第 4 章、「CCL クエリの構築」*(39 ページ)
- *FROM 句：カンマ区切りの構文*(111 ページ)
- *FROM 句：ANSI 構文*(112 ページ)
- *GROUP BY 句*(114 ページ)
- *GROUP FILTER 句*(115 ページ)
- *GROUP ORDER BY 句*(116 ページ)
- *HAVING 句*(117 ページ)
- *MATCHING 句*(118 ページ)
- *ON 句：ジョインの構文*(121 ページ)
- *SELECT 句*(121 ページ)
- *UNION 演算子*(123 ページ)
- *WHERE 句*(125 ページ)
- *AS 句*(97 ページ)

FROM 句

カンマ区切りの構文または ANSI 構文を使用して **FROM** 句を記述します。

参照：

- *FROM 句：カンマ区切りの構文* (111 ページ)
- *FROM 句：ANSI 構文* (112 ページ)

FROM 句：カンマ区切りの構文

WHERE 句と組み合わせて、さらに代替のカンマ区切りの構文を使用して、クエリ内の 2 つのデータ・ソースをジョインします。

構文

```
FROM { stream [ [AS] alias ] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] } [, …]
```

コンポーネント

stream	データ・ストリームの名前
alias	ストリームまたはウィンドウのエイリアス
keep_clause	ローがウィンドウで維持される方式を指定するポリシー
window_name	ウィンドウの名前

使用法

MATCHING 句を使用する単一ソース・クエリ、内部ジョイン、クエリのカンマ区切りの構文と一緒に **FROM** 句を使用します。この構文は、クエリで 1 つ以上のデータ・ソースを指定します。 クエリの他の句内のすべてのカラムまたはデータソースの参照は、この句で指定されているデータ・ソースの 1 つを指し示している必要があります。

カンマ区切りの **FROM** 句には、内部ジョインを使用して結合される複数のデータ・ソースを指定できます。複数のソースは、カンマで区切って指定します。カンマ区切りの構文を使用する場合に必須な **WHERE** 句は、ジョインの選択条件を作成します。

MATCHING 句と一緒に **FROM** 句のカンマ区切りの構文を使用して、特定のパターンをモニタする必要のあるデータ・ソースを指定します。データ・ソースのリストには、データ・ストリームのみを指定でき、**MATCHING** 句で指定されているすべてのデータソースを指定する必要があります。これ以外のデータソースはいずれも指定できません。

ストリームまたはウィンドウの名前を省略するためにエイリアスを使用します。これは、特に同じデータ・ストリームまたはデータ・ウィンドウが **FROM** 句内で複数回使用される場合に、インスタンス間を区別するために必要です。

参照：

- *キー・フィールド・ルール* (43 ページ)
- *ジョイン* (41 ページ)
- *ジョインの例：カンマ区切りの構文* (46 ページ)
- *FROM 句* (111 ページ)

FROM 句：ANSI 構文

外部ジョインまたは内部ジョインの構文を使用してクエリ内の 2 つのデータ・ソースをジョインします。

構文

```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
[ RIGHT | LEFT | FULL ] JOIN
{ stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
nested_join
```

コンポーネント

stream	データ・ストリームの名前
alias	ストリームまたはウィンドウのエイリアス
keep_clause	ローがウィンドウで維持される方式を指定するポリシー
window_name	ウィンドウの名前
nested_join	ネストしたジョイン。以下を参照してください。

nested_join

```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
[ RIGHT | LEFT | FULL ] JOIN
{ stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
[ on_clause ]
```

コンポーネント

stream	データ・ストリームの名前

alias	ストリームまたはウィンドウのエイリアス
keep_clause	ローがウィンドウで維持される方式を指定するポリシー
window_name	ウィンドウの名前
nested_join	ネストしたジョイン。詳細については、「ジョイン」を参照してください。
on_clause	ジョイン条件

使用法

外部ジョインの場合は、**ON** 句を使用してジョイン条件を指定します。これは、内部ジョインではオプションです。

次の **FROM** のタイプを使用して、内部ジョイン、左ジョイン、右ジョイン、フル・ジョインを作成できます。

JOIN	両方のデータ・ソースの共通部分からのすべての可能な組み合わせのローをパブリッシュする (選択条件が指定されている場合、選択条件によって制限されます)。
RIGHT JOIN	両方のデータ・ソースの共通部分からのすべての可能な組み合わせのローをパブリッシュする (選択条件が指定されている場合、選択条件によって制限されます)。右側のデータ・ソースからの他のすべてのローもパブリッシュされます。左側のデータ・ソースの一致しないカラムに対しては、null 値がパブリッシュされます。
LEFT JOIN	両方のデータ・ソースの共通部分からのすべての可能な組み合わせのローをパブリッシュする (選択条件が指定されている場合、選択条件によって制限されます)。左側のデータ・ソースからの他のすべてのローもパブリッシュされます。右側のデータ・ソースの一致しないカラムに対しては、null 値がパブリッシュされます。
FULL JOIN	両方のデータ・ソースの共通部分からのすべての可能な組み合わせのローをパブリッシュする (選択条件が指定されている場合、選択条件によって制限されます)。両方のデータ・ソースからの他のすべてのローもパブリッシュされます。いずれかのデータ・ソースの一致しないカラムに対しては、null 値がパブリッシュされます。

この構文で使用されるデータ・ソースには、データ・ストリーム式、名前付きウィンドウまたは名前なしウィンドウの式、クエリを指定できます。**FROM** 句のこのタイプのデータソースにはエイリアスを使用できます。

FROM 句 (ANSI 構文) のジョイン・タイプは、2 つのデータソースに制限されます。さらにデータソースを指定するには、データソースの 1 つとしてネストしたジョインを使用します。ネストしたジョインを使用する場合、オプションをカッコで

囲み、それ自身の **ON** 句を指定できます。ネストしたジョインで **ON** 句を使用する場合に適用される規則は、ネストしたジョインを含むジョインで **ON** 句を使用する場合に適用される規則と同じです。

制限事項

- クエリの他の句内のすべてのカラムまたはデータソースの参照は、この句で指定されているデータ・ソースの 1 つを指し示している必要があります。
- 左外部ジョインの場合、データ・ストリームは左側のみに指定できます。同様に、右外部ジョインの場合、データ・ストリームは右側のみに指定できます。
- 全外部ジョインでは、ウィンドウをデータ・ストリームにジョインできません。

参照：

- *キー・フィールド・ルール* (43 ページ)
- *ジョイン* (41 ページ)
- *ジョインの例：ANSI 構文* (44 ページ)
- *FROM 句* (111 ページ)

GROUP BY 句

集約操作を実行する式を指定します。

構文

```
GROUP BY expression1 [, expression2 ...]
```

コンポーネント

expression	定数を使用する式で、入力ウィンドウまたは入力ストリームからの 1 つ以上の式を指定できます。ただし、式は集合関数を使用できません。

使用法

1 つ以上の結果ローを、出力の 1 つのローに組み合わせます。 **GROUP BY** 句は、クエリ結果が集合関数を含む場合に、集約操作の実行対象となる式を指定するために使用されます。

GROUP BY 句がクエリで使用される場合、関連付けられているウィンドウには、プライマリ・キーがコンパイラによって抽出されている必要があります。複数のカラムに同じ式がある場合、最初のカラムが **GROUP BY** 式と一致していない場合に使用されます。

注意： **GROUP BY** 句のすべての式が、少なくとも 1 つの **SELECT** カラム式に存在する必要があります。

GROUP BY 句では、実際の式を使用します。式のエイリアスを使用すると、プロジェクトがコンパイルできません。たとえば、Symbol のようなエイリアスではなく、T.Symbol を使用します。

例

GROUP BY 句は、T.Symbol に従ってローを集約します。

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)
PRIMARY KEY DEDUCED
KEEP ALL
AS
SELECT T.Symbol, max(T.Price) MaxPrice
FROM Trades T
GROUP FILTER rank() < 10
GROUP BY T.Symbol
GROUP ORDER BY T.Volume DESC
HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

参照：

- *集合関数* (136 ページ)
- *集約操作* (48 ページ)
- *GROUP FILTER 句* (115 ページ)
- *GROUP ORDER BY 句* (116 ページ)

GROUP FILTER 句

グループのデータをフィルタしてから、集約操作を実行します。

構文

```
GROUP FILTER expression
```

コンポーネント

expression	**min()** や **max()** などの集合関数を使用しないブール式。式は、ソース・ストリームまたはソース・ウィンドウからのカラムを使用できます。

使用法

GROUP FILTER 句は、集約操作がローに適用される前にデータをフィルタします。**GROUP FILTER** 句は、**GROUP BY** 句と一緒に使用されます。**GROUP FILTER** 句が **GROUP ORDER BY** 句と一緒に使用されると、**GROUP FILTER** 句が実行された後に **GROUP ORDER BY** 句が実行されます。

GROUP FILTER 句内の式は、一般に **rank()** などの関数に基づくフィルタを使用します。これらの関数は、集約で使用されるローを制限します。**rank()** 関数は、グ

ループ内の個々のレコードにランクを割り当てます。**rank()** が有意になるのは、**GROUP ORDER BY** 句と一緒に使用される場合のみです。

例

GROUP FILTER 句は、選択されたローを除外し、ランクが 10 未満のローのみを維持します。

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)
PRIMARY KEY DEDUCED
KEEP ALL
AS
SELECT T.Symbol, max(T.Price) MaxPrice
FROM Trades T
GROUP FILTER rank() < 10
GROUP BY T.Symbol
GROUP ORDER BY T.Volume DESC
HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

参照：

- *集合関数* (136 ページ)
- *集約操作* (48 ページ)
- *GROUP BY 句* (114 ページ)
- *GROUP ORDER BY 句* (116 ページ)

GROUP ORDER BY 句

グループのデータを並び替えてから、**GROUP FILTER** 句を適用し、データを集約します。

構文

```
GROUP ORDER BY column [ASC[ENDING]|DESC[ENDING]] [, ...]
```

コンポーネント

column	ソース・ストリームまたはソース・ウィンドウの任意のカラム。並び替えは、複数のカラムを使用して実行できます。

使用法

GROUP ORDER BY 句は、**GROUP BY** 句と一緒に使用されます。ローの並び替えは、ストリームまたはウィンドウの 1 つ以上のカラムを使用して実行できます。**GROUP ORDER BY** はグループのデータの並びを変更してから、集約操作 (と **GROUP FILTER**) を適用します。

ASC と DESC のキーワードを使用して、カラム・データをそれぞれ昇順または降順に並べ替えます。キーワードを指定しない場合、デフォルト値は ASC (昇順) です。

GROUP FILTER 句と一緒に使用されると、**GROUP ORDER BY** 句が実行されてから、**GROUP FILTER** 句が実行されます。**GROUP ORDER BY** 句は、この句で指定された並び替え条件に基づいて各グループのレコードを並び替えます。

例

GROUP ORDER BY 句は、選択されたローを T.Volume の降順に並び替えます。

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)
PRIMARY KEY DEDUCED
KEEP ALL
AS
SELECT T.Symbol, max(T.Price) MaxPrice
FROM Trades T
GROUP FILTER rank() < 10
GROUP BY T.Symbol
GROUP ORDER BY T.Volume DESC
HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

参照：

- *集合関数* (136 ページ)
- *集約操作* (48 ページ)
- *GROUP FILTER 句* (115 ページ)
- *GROUP BY 句* (114 ページ)

HAVING 句

グループ化の句によってグループ化されたローをフィルタします。

構文

```
HAVING expression
```

コンポーネント

expression	任意のブール式。集合関数と、カラムに対する単純なフィルタも指定できます。

使用法

HAVING 句は意味的に **WHERE** 句に似ていますが、**GROUP BY** 句を指定するクエリのみで使用できます。**HAVING** 句は、ローが **GROUP BY** 句によって処理された後に、ローをフィルタします。**WHERE** 句と異なり、**HAVING** 句では式内に集合関数

を使用できます。この機能は、グループ化された結果ローの一部を削除することです。

例

HAVING 句は、ローが **GROUP FILTER** 句、**GROUP BY** 句、**GROUP ORDER** 句によってグループ化された後に、ローをフィルタします。

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)
PRIMARY KEY DEDUCED
KEEP ALL
AS
SELECT T.Symbol, max(T.Price) MaxPrice
FROM Trades T
GROUP FILTER rank() < 10
GROUP BY T.Symbol
GROUP ORDER BY T.Volume DESC
HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

参照：

- *集合関数* (136 ページ)
- *集約操作* (48 ページ)
- *GROUP BY 句* (114 ページ)
- *GROUP FILTER 句* (115 ページ)
- *GROUP ORDER BY 句* (116 ページ)
- *式* (35 ページ)

MATCHING 句

この句は、クエリ内でパターン一致を行うため使用されます。この句を使用することによって、1 つ以上のソース全体でイベントを検出できます。

注意： **ON** 句のこの形式は、**JOIN** 構文の **ON** 句とは異なります。両方の形式は同時に指定できません。

構文

```
MATCHING [interval:pattern]
ON { {source.column = source.column [=...]}|
                    {source.column = constant }|
                    {getOpcode() = opcode_constant} [AND...]
            }

pattern:[!]{event | (event)} [&&| || |,}event]
```

コンポーネント

`MATCHING`	MATCHING 句を識別。
`interval:pattern`	`interval` は間隔を指定し、`pattern` は一致するパターンを指定。
`source.column`	ソース入力とカラムの名前。
`getOpcode()`	パターンの opcode 条件の取得。
`opcode_constant`	opcode を指定。
`pattern`	識別するパターン。イベント演算子によって連結されたイベントを指定します。
`event`	パターンで比較されるイベント。

使用法

MATCHING 句は、**SELECT** 文の **FROM** 句の直後に続きます。**FROM** 句には、パターン一致の入力として使用される派生要素を指定します。

MATCHING 句が指定されている **SELECT** 文には、フィルタリングまたは集約の条件は指定できません。

MATCHING 句には、間隔とパターン仕様を指定する必要があります。

間隔には、パターンを検出する期間を指定します。間隔は、マイクロ秒の精度をサポートしており、間隔定数 (間隔のデータ型を参照してください) またはパラメータのいずれかで表されます。

パターン仕様は、指定された間隔内で発生すべき、または発生すべきでない、パターン一致情報を満たすイベントまたはイベントのグループを示します。パターン仕様が複数のイベントで構成される場合、イベントまたはイベントのグループは、次の表にリストされている演算子で連結されている必要があります。

演算子	演算子名	説明
!	Not 演算子	パターン・コンポーネントの否定条件を指定します。パターン条件は、パターン・コンポーネントが指定された期間に発生しない場合に満たされます。これは否定条件なので、指定された期間が経過すると、パターン一致は満たされたと判断されます。
&&	連結 (論理 AND) 演算子	連結演算子によって結び付けられているパターン・コンポーネントの両方が発生すると、一致条件が満たされたことになります。発生の順序は関係しません。

演算子	演算子名	説明
||	分離 (論理 OR) 演算子	分離演算子によって結び付けられているパターン・コンポーネントの 1 つまたは両方が発生すると、一致条件が満たされたことになる。分離一致によって生成される各出力ローは、分離演算子のメンバの 1 つの一致を示し、他のメンバには null 値を示します。これは、分離演算子の複数のメンバがイベントを生成した場合にも、真となります。
,	後続演算子	この演算子で結び付けられているパターン・コンポーネントの両方が、リストされている順序で発生すると、一致条件が満たされたことになる。

パターン一致の可能性についてパターン・コンポーネントが分析される順序のデフォルトは、表にリストされている演算子の順序に従います。演算子とパターン・コンポーネントの優先順位が最も高いのが、Not 演算子の場合です。連結、分離、後続の各演算子で結び付けられているイベントについては、この順序で優先順位が低くなります。デフォルトの優先順位は、パターン・コンポーネントをカッコで囲むことによって上書きできます。

Not 演算子のパターン一致は、指定した時間間隔が経過した後でのみ一致しているとみなされるので、Not 演算子と後続演算子を一緒に使用する場合は、Not 演算子を最後のコンポーネントにする必要があります。これは、Not 演算子で一致するイベントは、時間間隔が経過するまで、パターン・ルール・エンジンによって評価されないためです。

FROM 句に複数の派生要素が指定されている **SELECT** 文の **MATCHING** 句には、オプションで、パターン一致条件の細分性をさらに高める 1 つ以上の等価式を定義する **ON** サブ句を指定できます。

等価式は、入力レコードのカラム値またはその opcode と比較するために使用されます。等価式の左側には、完全修飾されたカラム名か、関数 [4] のいずれかを指定できます。右側には、完全修飾されたカラム名、定数値、またはパラメータを指定できます。

左側に関数 [5] を指定する場合は、右側には該当する opcode を指定する定数を指定する必要があります。有効な opcode 値は、insert、update、delete です。

参照：

- *パターン一致* (47 ページ)

[4] getOpcode()
[5] getOpcode()

ON 句：ジョインの構文

JOIN 用語を使用して、構文のジョイン条件を指定します。

構文

```
ON source1.columnA = source2.columnB [AND...]
```

コンポーネント

`source`	**FROM** 句内のソースの名前。
`column`	特定のソースからのカラムの名前。複数カラムの比較を指定する場合は、`AND` を使用します。`OR` 式はサポートされません。

使用法

ON 句のこの形式は、外部ジョインと内部ジョインで必須です。1 つのデータ・ソースのカラムと別のデータ・ソースのカラムを比較する、1 つ以上の単純比較で構成される必要があります。

`source1` と `source2` は、**FROM** 句のソース (ストリーム、ウィンドウ、デルタ・ストリーム) を参照します。**FROM** 句でエイリアスが使用されている場合、実際のソース名ではなく、エイリアスを使用します。

制限事項

- ジョイン条件は、ジョインの 2 つのデータ・ソースのカラム間の比較に制限されます。比較にリテラル値は指定できません。また、同じデータ・ソースの 2 つのカラムは比較できません。

参照：

- *ジョイン* (41 ページ)
- *ジョインの例：ANSI 構文* (44 ページ)

SELECT 句

クエリの射影リストを指定します。

構文

```
SELECT { expression[AS column]}[,...]
```

コンポーネント

expression	対応する送信先カラムと同じデータ型の値に評価する式。
column	クエリ送信先のカラムの名前。

使用法

各 select リスト項目内の式には、リテラル、**FROM** 句で参照されているソースからのカラム名、演算子、スカラ関数、カッコを指定できます。ワイルド・カード文字 (*) は、**FROM** 句で参照される基になるソースからのすべてのカラムを選択します。AS カラム参照は、送信先のカラム名にマップされる必要があります。

射影内のすべての項目は、AS 拡張を使用して、項目を送信先カラムにマップする必要があります。または、すべてがマップされないようにする必要があります。この場合、割り当ては、左から右に実行されます。場合によっては、クエリに基づいて、送信先用のスキーマが自動的に作成されることがあります。式については、AS 拡張を使用してカラムを指定します。

クエリ内の **SELECT** 句は、1 つ以上の項目の select リストを指定します。**FROM** 句でリストされているデータソースからのローは、**WHERE** 句によってフィルタされた後に (指定されている場合)、**SELECT** 句に渡されます。リスト内の式の結果は、他の句によって処理されます (指定されている場合)。クエリは通常、入力として処理済みの select リスト結果を使用します。

次の規則が select リストに適用されます。

- 各 select リスト項目内の式には、リテラル、**FROM** 句でリストされているデータソースの 1 つからのカラム名、演算子、スカラ関数とさまざまな関数、カッコを指定できます。クエリの select リストの式には、集合関数も指定できます。または、"すべて選択" (ワイルドカード) 文字 (*) を使用して式を指定できます。これは、文の **FROM** 句にリストされているすべてのデータソースからのすべてのカラム値を、左から右にリストすることに等価です。また、指定されたデータ・ソース (データソースは **FROM** 句でリストされたデータ・ソースの 1 つの名前またはエイリアスです) からのすべてのカラム値を意味する data-source.* を使用することと等価です。
- これらの規則は、ワイルドカード文字が指定されていないすべての式に適用される。
  - 各リスト項目は、送信先内のカラムを示す AS 出力カラム参照サブ句を指定できる。このカラムに、select リスト項目がパブリッシュされます。AS サブ句を使用する場合は、select リストのすべての項目に対して使用するか、いずれにも使用しない必要があります。

参照：

- *CREATE DELTA STREAM 文* (72 ページ)
- *DECLARE 文* (86 ページ)
- *PRIMARY KEY 句* (104 ページ)
- *SCHEMA 句* (105 ページ)
- *第 4 章、「CCL クエリの構築」* (39 ページ)
- *FROM 句* (111 ページ)
- *AS 句* (97 ページ)
- *スカラ関数* (168 ページ)
- *演算子* (31 ページ)

UNION 演算子

2 つ以上の **SELECT** 句の結果を 1 つのストリームまたはウィンドウに結合します。

構文

```
{select_clause} UNION {select_clause} [ UNION ...]
```

コンポーネント

select_clause	**SELECT** 句。

使用法

ユニオン演算は、ストリーム、デルタ・ストリーム、またはウィンドウを生成できます。

- ウィンドウを生成するユニオンへの入力がストリームの場合、集約操作を実施する必要があります。
- ユニオンが 2 つの **SELECT** 句をジョインするとき、2 つの **SELECT** 句で選択されるカラムのスキーマは一致する必要があります。
- 特定のキー値を持つレコードが複数の入力ノードで生成されないことを確認してください。そうでない場合、重複ローまたは無効な更新が発生することがあります。
- 互換性を得るには、ユニオンが行われるすべてのノードのスキーマが同じデータ型を持つ必要があります。ただし、スキーマのカラム名は異なってもかまいません。この場合、スキーマの推測には、最初の **SELECT** 句のカラム名が使用されます。
- **SELECT** 文がソースからの直接コピーではない場合、中間ノードが作成されます。コンパイラは、デルタ・ストリームまたはストリームを作成しようとしますが、集約または **KEEP** 句ではウィンドウを生成する必要があります。

- **DECLARE** ブロックは、ユニオン演算では使用できません。
- ユニオン演算で作成されるノードは、ターゲットがウィンドウの場合、**KEEP** 句と **AGING** 句を含むことができます。

制限事項

- ユニオンへの入力は、ストリーム、デルタ・ストリーム、ウィンドウの任意の組み合わせとすることが可能。
- ユニオンのデルタ・ストリームへの入力は、デルタ・ストリームまたはウィンドウとすることができるが、ストリームは不可。
- ユニオンのウィンドウへの入力は、ストリーム、デルタ・ストリーム、ウィンドウの任意の組み合わせとすることが可能 (ストリームを含むクエリに **GROUP BY** が含まれる場合)。
- ユニオンのストリームまたはデルタ・ストリームは、その元となるクエリのいずれかで指定された **GROUP BY** 句を含むことは不可。

例

次の例では、ユニオン演算を使用して、出力ストリームを生成します。

```
CREATE SCHEMA MySchema (a0 integer, a1 STRING, a2 string);
CREATE SCHEMA MySchema2 (a0 integer, a1 STRING, a2 string);

CREATE INPUT STREAM InputStream1 SCHEMA MySchema;
CREATE INPUT STREAM InputStream2 SCHEMA MySchema2;
CREATE INPUT STREAM InputStream3 SCHEMA MySchema2;

CREATE OUTPUT STREAM UnionStream1 AS SELECT * FROM InputStream1
UNION
SELECT * FROM InputStream2;
```

ユニオン演算を使用して、出力ウィンドウを生成します。

```
CREATE OUTPUT WINDOW UnionWindow1
PRIMARY KEY DEDUCED
AS
    SELECT in1.a0, min(in1.a1) a1, min(in1.a2) a2
    FROM InputStream1 in1 GROUP BY in1.a0
    UNION
    SELECT in2.a0, min(in2.a1) a1, min(in2.a2) a2
    FROM InputStream2 in2 GROUP BY in2.a0;
```

注意： ソースがストリームでターゲットがウィンドウなので、集約は必要に応じて指定されます。

次の例では、ユニオン演算を使用して、デルタ・ストリームを生成します。

```
CREATE DELTA STREAM Union1 PRIMARY KEY DEDUCED
AS
    SELECT * FROM Stream1
    UNION
   SELECT a.col1, a.col2, a.col3 FROM DeltaStream1 a WHERE a.col1 >
10
```

```
    UNION
    SELECT a.a, sum(a.b), max(a.c) FROM Window2 GROUP BY a.a
```

参照：

- *union* (40 ページ)
- *例：ストリームまたはウィンドウからのデータのマージ*(41 ページ)

WHERE 句

データのローをフィルタするための選択条件、ジョイン条件、更新条件、または削除条件を指定します。

構文

```
WHERE condition | filterexpression
```

コンポーネント

condition	コンテキストに応じて、選択、更新、削除、またはジョインの条件を表すブール式。
filterexpression	ストリームからのカラムに基づくブール式。

使用法

WHERE 句は、複数の CCL 文のローとカラムを、同じ構文で、しかも異なる使用法とコンテキストで、フィルタします。**WHERE** 句は以下の機能を提供します。

- **QUERY** 要素のデータ・ソースからの入力をフィルタするための選択条件を指定する。
- **FROM** 句でジョイン条件を提供する。

選択条件

WHERE 句は **FROM** 句と組み合わされて使用されると、選択条件として機能します。

この句のブール式は、クエリのデータ・ソースに到着するローをフィルタする条件を作成します。フィルタされたローのみが、**SELECT** 句に渡されます。**WHERE** 句のフィルタ処理は、**GROUP BY** 句と集約 (指定されている場合) の前に実行されます。このため、集合関数または集約結果に基づく結果のフィルタ処理は組み込めません。集約後フィルタ処理には、**HAVING** 句を使用できます。

選択条件には、リテラル、**FROM** 句にリストされているクエリのデータ・ソースからのカラム参照、演算子、スカラ関数、パラメータ、カッコを記述できます。

クエリでは、選択条件内のカラム参照は、クエリの 1 つのデータ・ソースのカラムを参照する必要があります。

ジョイン条件
FROM 句のカンマ区切り構文フォームと組み合わせて使用されると、**WHERE** 句はカンマ区切りジョイン用の 1 つ以上のジョイン条件を作成します。カンマ区切りジョインでの **WHERE** 句の使用はオプションです。ジョイン条件が指定されていないと、すべてのデータ・ソースからのすべてのローが選択されます。**WHERE** 句が指定されている場合、その構文は、ANSI ジョイン構文の **ON** 句に似ています。

ジョイン条件には、ジョインの条件を指定する任意の有効なブール式を記述できます。**WHERE** 句のこのフォームのすべてのカラム参照は、**FROM** 句で指定されたデータ・ソースを参照する必要があります。

フィルタ式
フィルタ式は、入力ストリームでのみサポートされています。

フィルタ式でカラムを使用する場合、`nodeName.columnName` 表記を使用します。ここで、`nodeName` は入力ストリームの名前です。

制限事項

- **WHERE** 句では、集合関数を使用できません。
- **WHERE** 句を **MATCHING** 句と組み合わせて使用することはできません。
- **JOIN** キーワードを使用するジョインは、ジョイン条件を指定するために **WHERE** 句は使用しません (ただし、選択条件フォームでは使用できます)。

例
次の例は、選択条件として **WHERE** 句を使用します。

```
CREATE INPUT WINDOW QTrades SCHEMA (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
PRIMARY KEY (Id);

CREATE OUTPUT WINDOW QTradesComputeSelected
PRIMARY KEY DEDUCED
AS SELECT
    trd.*
FROM
    QTrades trd
WHERE
    trd.Symbol IN ('DELL','CSCO','SAP')
;
```

次の例は、ジョイン条件として **WHERE** 句を使用します。

```
CREATE INPUT WINDOW QTrades SCHEMA (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
PRIMARY KEY (Id);

CREATE OUTPUT WINDOW RecentQTrades
PRIMARY KEY DEDUCED
AS
    SELECT q.Symbol, nth(0, q.Price) Price, nth(0, q.Shares) Shares
FROM
    QTrades q
GROUP BY q.Symbol
GROUP ORDER BY q.ROWID DESC
;

CREATE INPUT WINDOW Positions
SCHEMA (BookId STRING, Symbol STRING, SharesHeld INTEGER)
PRIMARY KEY (BookId, Symbol)
;

CREATE OUTPUT WINDOW PositionValue
PRIMARY KEY (BookId, Symbol)
AS SELECT
    pos.BookId,
    pos.Symbol,
    pos.SharesHeld,
    pos.SharesHeld * q.Price Value
FROM
    Positions pos, RecentQTrades q WHERE pos.Symbol = q.Symbol
;
```

次の例は、フィルタ式として **WHERE** 句を使用します。

```
CREATE INPUT STREAM LSETradesFiltered SCHEMA (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
WHERE LSETradesFiltered.Symbol IN ('SAP', 'CSCO', 'DELL')
;
```

参照：

- *フィルタリング*(39 ページ)
- *式*(35 ページ)

第 9 章　関数

関数とは、特定のタスクを実行する自己完結型の再利用可能なコード・ブロックです。

Sybase Event Stream Processor では、次の関数がサポートされます。

- 組み込み関数 - 集合関数、スカラ関数など
- ユーザ定義の SPLASH 関数
- ユーザ定義の外部関数

組み込み関数とはこのソフトウェアに付属する関数で、一般的な算術演算、集約、データ型変換、セキュリティの関数が含まれます。

演算の評価順序
関数内の演算は右から左に評価されます。これは、変数が別の演算に依存し、関数の実行前に変数を渡しておかなければ、その演算で予期しない結果が生じる可能性がある場合に重要です。例を示します。

```
integer a := 1;
integer b := 2;
max( a + b, ++a );
```

組み込み関数 **max()** は、カンマで区切られた値リストの最大値を返します。++a が最初に評価されるので 4 が返されます。そのため、予想されていた `max(3, 2)` ではなく `max(4, 2)` が実行されます。

参照：

- *集合関数* (136 ページ)
- *その他の関数* (236 ページ)
- *スカラ関数* (168 ページ)
- *ユーザ定義の SPLASH 関数* (130 ページ)
- *ユーザ定義の外部関数* (130 ページ)

ユーザ定義の SPLASH 関数

SPLASH プログラミング言語を使用して、ユーザ定義関数を **CREATE STREAM** 文、**CREATE WINDOW** 文、**CREATE FLEX** 文に記述し、グローバルまたはローカルの DECLARE ブロックで使用できます。

構文

```
DECLARE
    returnType funcName (argType argName,...) {

        //function body

        return value;
    }
END;
```

使用法
関数名の大文字と小文字は区別されます。

モジュール・レベルまたはプロジェクト・レベルで定義されている関数は、そのモジュール内またはプロジェクト内の式の任意の場所で使用できます。ただし、ストリーム、ウィンドウ、**FLEX** 演算子内で定義された関数は、それらの要素のスコープ内でのみアクセスできます。

複数の関数を定義できます。関数を宣言する必要はありません。たとえば、関数 f2 は、定義されていない関数 f1 を参照できます。

ユーザ定義の外部関数

CCL プロジェクトでは、**CREATE LIBRARY** 文を使用して、C/C++ または Java で記述されたユーザ定義関数を呼び出せます。

C/C++ 関数を共有ライブラリ、Linux と UNIX の `.so` ファイル、Windows の `.dll` からロードします。Java 関数は、`.class` ファイルまたは `.jar` ファイルのいずれかからロードします。

CREATE LIBRARY 文を使用して、CCL で外部関数を宣言します。1 回宣言すると、組み込み関数を使用する任意の場所でその関数を使用できます。

注意： C/C++ の外部ライブラリ呼び出しは、すべてのデータ型を、具体的には、ブール、整数、長整数、浮動小数点数、通貨型 (n)、日付、bigdatetime、バイナリをサポートします。

Java の外部ライブラリ呼び出しは、整数、長整数、倍精度浮動小数点数、文字列のデータ型のみをサポートします。

辞書、ベクトル、イベント・キャッシュ、レコード型などの複雑な型は、外部関数でサポートされていません。

外部 C/C++ 関数の要件

外部 C/C++ 関数は、データ型、引数と戻り値、出力の各要件に従うことによって、Sybase Event Stream Processor のインタフェースに準拠する必要があります。

構文

Event Stream Processor のインタフェースに、関数シグネチャを記述します。

```
int32_t funcName (int numargs,
    DataValue::DataValue * top,
    DataValue::DataValue * nextArgs,
    std::vector<void *> & arena)
```

データ型の要件

Event Stream Processor は、関数の各引数を DataValue として渡し、戻り値を DataValue として受信することを想定します。DataValue は、Event Stream Processor でサポートされるすべてのデータ型を含む構造体で、`$ESP_HOME¥include` にある `DataValue.hpp` で定義されています。DataValue 構造体の定義は、次のとおりです。

```
struct DataValue {
    union {
        bool booleanv;
        int16_t    int16v;
        int32_t    int23v;
        int64_t    int64v;
        interval_t    intervalv;
        money_t    moneyv;
        double    doublev;
        time_t    datev;
        timestampval_t    timestampv;
        const char * stringv;
        hirestime_t    bigdatetimev;
        binary_t    binaryv;
        void * objectv;
    }
    bool null;
}
```

ブール型のフラグ `null` が true に設定されている場合、引数の値は null です (引数には値がありません)。`binary_t` は、次のように定義される 2 つの public メンバ変数を持つクラスです。

- `const uint8 t * _data;`

この変数は、バッファ内のデータの最初のバイトを指し示します。

- `byte_size_t _used;`

  この変数は、バッファ内の使用されるデータの長さを定義します。

注意： メモリを `_data` に割り当てます。`malloc` または `calloc` を使用し、`new` は使用しません。

`moneyv` は、任意の精度の通貨型引数の汎用的なプレースホルダです。個別の通貨型引数に対しては精度を指定する必要があります。

引数と戻り値の要件

Event Stream Processor の内部処理エンジンは、スタックの最上部を特別なロケーションに維持するバイトコード・スタック・マシンなので、Event Stream Processor が関数の引数を次の 2 つに分割できるようにする必要があります。

- 型 `DataValue` のスタックの最上部へのポインタ。スタックの最上部は、複数の引数が関数に渡された場合には最後の引数を指し示し、1 つの引数のみが渡された場合には最初の引数を指し示します。インタフェースの最初の引数は、渡される引数の数を示します。
- 型 `DataValue` の引数の残りの部分へのポインタ。このポインタは、複数の引数が関数に渡された場合には最初の引数を指し示します。関数に 1 つの引数のみが渡される場合の動作は定義されていません。

注意： 関数の戻り値をスタックの最上部に書き込みます。

関数が `malloc` または `calloc` を呼び出すことによってメモリを割り当てると、Event Stream Processor は、メモリをアリーナに追加することによって、レコードを処理した後にメモリを解放できます。アリーナは関数への最後の引数で、型 `void *` のベクトルとして定義されます。`new` を使用してアリーナに割り当てられたメモリにポインタは追加できません。これを行うとメモリが破壊され、リカバリ不能なエラーが発生する可能性があります。

出力要件

関数が、関数処理の正常な完了を示すために、エラー・コードを返すことを確認します。戻り値は、`int32_t` 型です。0 の値は、エラーがないことを示し、他の任意の値はエラーを示します。エラーが発生すると、Event Stream Processor は現在のレコードを拒否します。

参照：

- *例：外部 C/C++ 関数の使用* (133 ページ)

例：外部 C/C++ 関数の使用

Event Stream Processor インタフェースまでの距離を計算する C/C++ 関数を記述します。関数をコンパイルして共有ライブラリに格納後、**CREATE LIBRARY** 文を使用してこの関数を宣言し、CCL プロジェクトで必要に応じて呼び出します。

前提条件

Event Stream Processor のインタフェースに対して C/C++ 関数を記述するための構文と要件を習得している必要があります。

手順

1. Event Stream Processor のインタフェースに準拠していることを確認しながら、関数を記述します。

   たとえば、次の関数は、距離を計算します。

```
#include math.h
double distance(int numvals, double * vals){
    double sum = 0.0;
    for (int i=0; i<numvals; i++){
        sum += vals[i]*vals[i];
    }
return sqrt(sum);
}
```

   Event Stream Processor のインタフェースに準拠するために、関数を次のように記述します。

```
#include math.h
#include <vector>
#include "DataValue.hpp"

extern "C"
int32_t distance(int numargs, DataTypes::DataValue * top,
        DataTypes::DataValue * nextArgs,
        std::vector<void *>& arena){
    double sum = 0.0;
    if (numargs <= 0){
        top->null = false;
        top->val.double = 0.0;
        return 0;
    }
    if(top->null)return 0;

    double dist = top->val.doublev * top->val.doublev;

    for(int i=numargs-2; i>=0; i--){
        if((nextArgs+i)->null){
            top->null = true;
            break;
        }
```

```
        dist +=(nextArgs + i)->val.doublev *
        (nextArgs + i)->val.doublev;
    }
    top->val.doublev = sqrt(dist);
    top->null = false;
    return 0;
}
```

注意： 戻り値 (top) が null であるかどうかを明示的に設定します。

`extern` 宣言は、関数と同じ名前がライブラリにあり、名前は C++ 関数名ではないことを確実にします。

2. 関数をコンパイルして共有ライブラリに格納します。
   たとえば、gcc コンパイラを使用すると、次のコマンドは、`distance.so` の名前の共有ライブラリを作成します。

```
gcc -fPIC -shared -m64 -I.. -c -o distance.o distance.cpp
gcc -fPIC -shared -m64 distance.o -o distance.so
```

3. **CREATE LIBRARY** 文を使用して、CCL プロジェクトで関数を宣言します。

```
CREATE LIBRARY DistanceLib LANGUAGE C FROM 'distance.so'(
    float distance(float arg1, float arg2, float arg3);
);
```

注意： 共有ライブラリ (`.dll` ファイル) を検索する場合、Windows はアプリケーションのパスを確認します。.dll ファイルがそのディレクトリで見つからない場合、PATH 環境変数で指定されているディレクトリを順に検索します。

関数の名前が、ライブラリにある関数の名前と一致することを確認します。

4. `DistanceLib.distance(arg1, arg2, arg3)` を使用して、距離関数をプロジェクトで呼び出します。

参照：
- *外部 C/C++ 関数の要件* (131 ページ)

例：Java 関数の使用

距離を計算する Java 関数を記述します。関数を `.class` ファイルまたは `.jar` ファイルとしてコンパイルした後に、**CREATE LIBRARY** 文を使用してこの関数を宣言し、CCL プロジェクトで必要に応じて呼び出します。最後に、ライブラリを Event Stream Processor にリンクします。

注意： Java 1.6 ランタイム環境が、Sybase Event Stream Processor に同梱されています。関数で別の Java バージョンが必要な場合は、環境変数 ESP_JAVA_HOME を該当する Java 仮想マシン共有ライブラリのロケーションに設定します。これは通常、Linux、UNIX、または Solaris では `libjvm.so`、Windows では `jvm.dll` です。

たとえば、Linux、UNIX、または Solaris のマシンで変数をシェルに設定するには、次を使用します。

```
export ESP_JAVA_HOME=/user/bin/java/jre/lib/libjvm.so
```

---

1. 関数を記述します。

   すべての関数をクラス内の public 静的メソッドとして定義します。たとえば、次の関数は、距離を計算します。

   ```
   public class Distance {
       public static double distance(double arg1, double arg2,
       double arg3) {
           double sum = 0;
           sum += arg1 * arg1;
           sum += arg2 * arg2;
           sum += arg3 * arg3;

           return Math.sqrt(sum);
       }
   }
   ```

   注意： 外部 Java 関数に null を渡したり、返したりはできません。

2. 関数をコンパイルして共有ライブラリに格納します。

   ```
   javac -d /home/sybase/user/java/lib Distance.java
   ```

   また、Java クラスのアーカイブ (`.jar` ファイル) を作成し、CCL プロジェクトで関数を宣言するときにそれらを参照できます。

3. **CREATE LIBRARY** 文を使用して、CCL プロジェクトで関数とライブラリを宣言します。

   ```
   CREATE LIBRARY DistanceLib LANGUAGE JAVA FROM 'Distance' (
       double distance(double arg1, double arg2, double arg3);
   );
   ```

   注意： `'Distance'` はクラスの名前です。クラスがパッケージ内で定義されている場合、クラス名をディレクトリで置き換え、名前を付加します。

   ライブラリ内の関数シグネチャが、.class ファイル内の関数と同じ名前、引数のデータ型、戻り値のデータ型であることを確認します。

4. `DistanceLib.distance(arg1, arg2, arg3)` を使用して、関数をプロジェクトで呼び出します。

5. Java ライブラリを Event Stream Processor サーバにリンクします。

   Event Stream Processor には、Java ランタイム環境が組み込まれています。Java 関数をアプリケーションにリンクするには、-j オプションを使用してサーバを起動します。

   .class ファイルの場合は、ファイルのディレクトリのみを指定します。

```
sp -j /home/sybase/user/java/lib
```

クラスが、たとえば `/home/sybase/user/java` にある .jar ファイルの内部に存在する場合、ファイルのディレクトリとファイル名を指定します。

```
sp -j /home/sybase/user/java/Distance.jar
```

複数のパスは、Linux/UNIX の場合は ":" を、Windows の場合は ";" を使用して区切ります。

集合関数

集合関数は複数のレコードを対象にして、値のグループから 1 つの値を計算します。

集合関数は、複数のローで構成される入力に基づいて、グループごとに 1 つの結果ローを返します。グループは、**SELECT** 文の **GROUP BY** 句を使用して形成されます。 **GROUP BY** 句は、**GROUP FILTER** 句、**GROUP ORDER BY** 句と一緒に使用できます。

単純な集合関数 (`sum()`、`min()`、`max()`、`avg()`、`count()` など) は、select リスト内と、**SELECT** 文の **HAVING** 内でのみ使用できます。 これらの関数は、データベースに含まれるローのグループのデータを集約します。

注意： 集合関数は、カラムまたは式に適用できますが、ネストすることはできません。つまり、集合関数は、別の集合関数を含む式に対して適用できません。
`count()` を除くすべての集合関数は、集約計算を行うときに null 値を無視します。ただし、関数に渡されるすべての入力が null で構成される場合は、null 値を返します。

参照：

- *GROUP BY 句* (114 ページ)
- *GROUP FILTER 句* (115 ページ)
- *GROUP ORDER BY 句* (116 ページ)
- *HAVING 句* (117 ページ)
- *集約操作* (48 ページ)

any()

集合。値のグループの任意の値を返します。

構文

```
any ( expression )
```

パラメータ

expression	任意のデータ型

使用法

値のグループから、null 値以外の任意の値を返します。すべての値が null の場合、関数は null を返します。関数は引数として任意のデータ型を受け取り、同じデータ型を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, any(xinput.d0) AS d0, any(xinput.d1) AS d1,
any(xinput.d2) AS d2, any(xinput.d3) AS d3, any(xinput.d4) AS d4,
any(xinput.d5) AS d5, any(xinput.d6) AS d6, any(xinput.d7) AS d7
FROM xinput
GROUP BY xinput.a;
```

avg()

集合。ローのセットの平均値を計算します。

構文

```
avg ( numeric-expression )
```

パラメータ

numeric-expression	平均値が計算される数値式。式は、ブール以外のすべての型を受け入れます。

使用法

平均値は、正規分布であることを想定する、次の式に従って計算されます。

```
s=(1/N)*SUM(x(i))
```

avg 関数は、null 値を受け取ると 0 を生成します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, avg(xinput.d0) AS d0, avg(xinput.d1) AS d1,
avg(xinput.d2) AS d2, avg(xinput.d3) AS d3
FROM xinput
GROUP BY xinput.a;
```

corr()

集合。一連の数値ペアの相関係数を返します。

構文

```
corr ( dependent-expression, independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除くすべての数値データ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除くすべての数値データ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。

dependent-expression および **independent-expression** は両方とも数値です。関数は、**dependent-expression** または **independent-expression** が NULL のペアを排除した後、(**dependent-expression**, **independent-expression**) のセットに適用されます。計算は、次の式で実行されます。

$$\frac{n\sum xy - \sum x \sum y}{\sqrt{n\sum x^2 - (\sum x)^2}\sqrt{n\sum y^2 - (\sum y)^2}}$$

ここで、x は **independent-expression** を表し、y は **dependent-expression** を表します。row_count、sum_x、sum_y、sum_xx、sum_yy、sum_xy の各総和計算を実行する必要があります。

例

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, corr(xinput.d0, xinput.d1) AS d0d1,
corr(xinput.d1, xinput.d0) AS d1d0,
corr(xinput.d2, xinput.d3) AS d2d3,
corr(xinput.d3, xinput.d2) AS d3d2,
corr(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

covar_pop()

集合。一連の数値ペアの母共分散を返します。

構文

```
covar_pop ( dependent-expression, independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。
independent-expression	結果に影響を与える変数。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。**dependent-expression** および **independent-expression** は両方とも数値です。関数は、**dependent-expression** または **independent-expression** が NULL であるペアをすべて除外した後で、(**dependent-expression, independent-expression**) のペアのセットに適用されます。計算は、次の式で実行されます。

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

ここで、*x* は **dependent-expression** を、*y* は **independent-expression** を、*n* は *x* と *y* のいずれも null でない *(x,y)* ペアの個数を表します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
```

```
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, covar_pop(xinput.d0,xinput.d1) AS d0,
covar_pop(xinput.d1,xinput.d2) AS d1,
covar_pop(xinput.d2,xinput.d3) AS d2,
covar_pop(xinput.d3,xinput.d4) AS d3,
covar_pop(xinput.d4,xinput.d5) AS d4,
covar_pop(xinput.d5,xinput.d6) AS d5,
covar_pop(xinput.d6,xinput.d0) AS d6
FROM xinput
GROUP BY xinput.a;
```

covar_samp()

集合。一連の数値ペアの標本共分散を返します。

構文

```
covar_samp ( dependent-expression, independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。
independent-expression	結果に影響を与える変数。

使用法

この関数は、引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数が空のセットに適用される場合は、NULL を返します。**dependent-expression** および **independent-expression** は両方とも数値です。関数は、**dependent-expression** または **independent-expression** が NULL であるペアをすべて除外した後で、(**dependent-expression**, **independent-expression**) のペアのセットに適用されます。

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / (n -1)
```

ここで、*x* は **dependent-expression** を、*y* は **independent-expression** を、*n* は *x* と *y* のいずれも null でない *(x,y)* ペアの個数を表します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)
```

```
CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, covar_samp(xinput.d0,xinput.d1) AS d0,
covar_samp(xinput.d1,xinput.d2) AS d1,
covar_samp(xinput.d2,xinput.d3) AS d2,
covar_samp(xinput.d3,xinput.d4) AS d3,
covar_samp(xinput.d4,xinput.d5) AS d4,
covar_samp(xinput.d5,xinput.d6) AS d5,
covar_samp(xinput.d6,xinput.d0) AS d6
FROM xinput
GROUP BY xinput.a;
```

count()

集合。グループ内のローの数を返します。null 値を含めることも、含めないこともできます。

構文

```
count ( * | expression )
```

パラメータ

expression	カラム。**expression** が null 値でない場合に、各グループのローの数を返します。

使用法

グループ内のローの数を返します。null 値を含めることも、含めないこともできます。* 構文が使用された場合には、グループ内のローの数を返します。または、**expression** 引数が使用された場合には、null 以外のローの数を返します。**expression** は計数されるカラムまたは別の **expression** です。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 INTEGER, d2 INTEGER, d3 INTEGER,
d4 INTEGER, d5 INTEGER, d6 INTEGER, d7 INTEGER)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, count(xinput.d0) AS d0,
count(xinput.d1) AS d1, count(xinput.d2) AS d2,
count(xinput.d3) AS d3, count(xinput.d4) AS d4,
```

```
count(xinput.d5) AS d5, count(xinput.d6) AS d6,
count(xinput.d7) AS d7
FROM xinput
GROUP BY xinput.a;
```

count(distinct)

集合。グループの一意なローの数を返します。

構文

```
count ( distinct expression )
```

パラメータ

distinct expression	バイナリを除く、任意のデータ型のカラム。

使用法

各グループの null 以外の一意なローの数を返します。重複するローはカウントされません。**distinct expression** はカウントされるカラムまたは別の **distinct expression** です。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW InAnaFunc
SCHEMA (ID INTEGER, Symbol STRING, Money_1 MONEY(5), Interval_2
INTERVAL, Money_2 MONEY(8))
PRIMARY KEY (ID)

CREATE  OUTPUT  WINDOW OutAnaFunc
SCHEMA (Symbol STRING, Money5_result INTEGER, Interval_Result
INTEGER, Money8_Result INTEGER)
 PRIMARY KEY DEDUCED

SELECT InAnaFunc.Symbol AS Symbol,
count(distinct InAnaFunc.Money_1) AS Money5_result,
count(distinct InAnaFunc.Interval_2) AS Interval_Result,
count(distinct InAnaFunc.Money_2) AS Money8_Result
FROM InAnaFunc
GROUP BY InAnaFunc.Symbol;
```

exp_weighted_avg()

集合。指数加重平均を計算します。

構文

```
exp_weighted_avg ( expression, period-expression )
```

パラメータ

expression	加重値を計算する数値式。
period-expression	平均を計算する間隔を指定する数値式。

使用法
指数移動平均 (EMA) 関数は、指数関数的に減少する加重係数を値に適用します。古いデータ・ポイントごとの加重は指数関数的に減少し、古い観測値を考慮しながら、最近の観測値の重要性を増加させます。

加重が減少する度合いは、0 ～ 1 間の数値である一定の補整定数 α として表されます。α はパーセンテージとしても表現できます。たとえば、10% の補整定数は、α=0.1 に等価です。または、α は N 期間としても表せます。次に例を示します。

$$\alpha = \frac{2}{N+1}$$

N=19 は α=0.1 に等価です。

期間 t の観測値は Y_t で指定され、任意の期間 t の EMA 値は S_t で指定されます。S_1 は未定義です。S_2 は、多くの方法で初期化できます。最も一般的には、S_2 を Y_1 に設定します。また、S_2 を最初の 4 つまたは 5 つの観測値の平均値に設定する方法もあります。生成される移動平均に対する S_2 初期化の影響度は、α によって異なります。小さい α 値は、S_2 の選択の重要性を大きい α 値に比べて相対的により大きくします。これは、α が古い観測値を考慮する程度が急速に減少するからです。

このタイプの移動平均は、最近の価格変動に対して、単純な移動平均よりもすばやく反応します。12 日と 26 日の EMA は、最も一般的に使用される短期間平均値で、移動平均収束拡散 (MACD) やパーセンテージ・プライス・オシレータ (PPO) などの指標を作成するために使用されます。一般的に、50 日と 200 日の EMA が、長期傾向のシグナルとして使用されます。

例
この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE OUTPUT WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT, d5
```

```
FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, exp_weighted_avg(xinput.d0,3) AS d0,
exp_weighted_avg(xinput.d1,3) AS d1, exp_weighted_avg(xinput.d2,3)
AS d2, exp_weighted_avg(xinput.d3,3) AS d3,
exp_weighted_avg(xinput.d5,3)as d5, exp_weighted_avg(xinput.d6,3)
AS d6
FROM xinput
GROUP BY xinput.a;
```

first()

集合。値のグループの最初の値を返します。

構文

```
first ( expression, index )
```

パラメータ

expression	関数は、引数と同じデータ型を返す。
index	(オプション) インデックスは、null 値と整数のデータ型を受け入れる。引数と同じデータ型を返します。

使用法

値のグループの最初の値を返します。任意のデータ型の **expression** 引数と、オプションで整数の **index** 引数を受け取り、 **expression** と同じデータ型を返します。関数は、指定された式で計算を実行し、null 値を含む最初の値を返します。

引数が純正なカラム名の場合、スカラとして使用します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW InAnaFunc
SCHEMA (ID INTEGER, Symbol STRING, Money_1 MONEY(5),
Interval_2 INTERVAL, Money_2
MONEY(8))
PRIMARY KEY (ID)

CREATE  OUTPUT  WINDOW OutAnaFunc
SCHEMA (Symbol STRING, Money_result MONEY(5),
Interval_Result INTERVAL, MoneyLast_Result MONEY(5),
Intervallast_Result INTERVAL, Moneyfirst MONEY(8),
Moneylast MONEY(8))
 PRIMARY KEY DEDUCED

SELECT InAnaFunc.Symbol AS Symbol, first(InAnaFunc.Money_1,1)
AS Money_result, first(InAnaFunc.Interval_2,0) AS Interval_Result,
last(InAnaFunc.Money_1,0) AS MoneyLast_Result,
last(InAnaFunc.Interval_2,0) AS Intervallast_Result,
```

```
first(InAnaFunc.Money_2,0) AS Moneyfirst, last(InAnaFunc.Money_2,0)
AS Moneylast
FROM InAnaFunc
GROUP BY InAnaFunc.Symbol;
```

first_value()

集合。値のグループの最初の値を返します。first() のエイリアスです。

last()

集合。値のグループの最後の値を返します。

構文

```
last ( expression, index )
```

パラメータ

expression	関数は、引数と同じデータ型を返す。
index	(オプション) インデックスは、null 値と整数のデータ型を受け入れる。引数と同じデータ型を返します。

使用法

指定された式で計算を実行し、null 値を含む最初の値を返します。

引数が純正なカラム名の場合、スカラとして使用します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW InAnaFunc
SCHEMA (ID INT32, Symbol STRING, Money_1 MONEY(5), Interval_2
INTERVAL, Money_2 MONEY(8))
PRIMARY KEY (ID)

CREATE  OUTPUT  WINDOW OutAnaFunc
SCHEMA (Symbol STRING, Money_result MONEY(5), Interval_Result
INTERVAL, MoneyLast_Result MONEY(5), Intervallast_Result INTERVAL,
Moneyfirst MONEY(8), Moneylast MONEY(8))
 PRIMARY KEY DEDUCED

SELECT InAnaFunc.Symbol AS Symbol,
first(InAnaFunc.Money_1,1) AS Money_result,
first(InAnaFunc.Interval_2,0) AS Interval_Result,
last(InAnaFunc.Money_1,0) AS MoneyLast_Result,
last(InAnaFunc.Interval_2,0) AS Intervallast_Result,
first(InAnaFunc.Money_2,0) AS Moneyfirst,
last(InAnaFunc.Money_2,0) AS Moneylast
FROM InAnaFunc
GROUP BY InAnaFunc.Symbol;
```

last_value()

集合。値のグループの最後の値を返します。last() のエイリアスです。

lwm_avg()

集合。値のグループの線形加重移動平均を返します。

構文

```
lwm_avg ( numeric-expression )
```

パラメータ

numeric-expression	式には、整数、長整数、浮動小数点数、通貨、タイムスタンプ、間隔の型を指定できる。

使用法

関数は引数として、ブール値を除く任意のデータ型を受け取り、同じデータ型を返します。関数は、直近に受信したデータほど大きな重要度を付加します。null 値は含まれません。

算術加重平均は、データ・ポイントごとに異なる加重を付加する乗算係数を使用する平均値です。技術的な分析の分野では、加重移動平均 (WMA) は、算術的に減少する、特別な意味の加重を持ちます。n 日の WMA では、最後の日が n の加重、最後の日の前日が $n-1$ の加重を持ち、以下同様に、加重がゼロに向かって小さくなります。

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \cdots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \cdots + 2 + 1}$$

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT, d5
TIMESTAMP, d6 INTERVAL)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, lwm_avg(xinput.d0) AS d0, lwm_avg(xinput.d1)
AS d1, lwm_avg(xinput.d2) AS d2, lwm_avg(xinput.d3) AS d3,
lwm_avg(xinput.d5) as d5, lwm_avg(xinput.d6) AS d6
```

```
FROM xinput
GROUP BY xinput.a;
```

max()

集合。値のグループの、null 以外の最大値を返します。

構文

```
max (expression)
```

パラメータ

expression	任意のデータ型

使用法

戻り値は、入力のデータ型に基づきます。すべての値が null の場合、関数は null を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0max INTEGER, d0min INTEGER, d1max LONG,
d1min LONG, d2max MONEY(4), d2min MONEY(4),
d3max FLOAT, d3min FLOAT, d4max BOOLEAN, d4min BOOLEAN,
d5max TIMESTAMP, d5min TIMESTAMP, d6max INTERVAL, d6min INTERVAL,
d7max BIGDATETIME, d7min BIGDATETIME)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, max(xinput.d0) AS d0max,
min(xinput.d0) AS d0min, max(xinput.d1) AS d1max,
min(xinput.d1) AS d1min, max(xinput.d2) AS d2max,
min(xinput.d2) AS d2min, max(xinput.d3) AS d3max,
min(xinput.d3) AS d3min, max(xinput.d4) AS d4max,
min(xinput.d4) AS d4min, max(xinput.d5) AS d5max,
min(xinput.d5) AS d5min, max(xinput.d6) AS d6max,
min(xinput.d6) AS d6min, max(xinput.d7) AS d7max,
min(xinput.d7) AS d7min
FROM xinput
GROUP BY xinput.a;
```

meandeviation()

集合。複数のローを対象にして、指定された式の平均絶対偏差を返します。絶対偏差は、すべての値を平均した値からの偏差の絶対値の平均値です。

構文

```
meandeviation ( numeric-expression )
```

パラメータ

numeric-expression	標本ベースの標準偏差がローのセットに対して計算される式 (通常はカラム名)。

使用法

この関数は、引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。平均偏差は、正規分布であることを想定する、次の式に従って計算されます。

```
s = (1/N) * SUM ( ABS(x[i] -  MEAN(x) ) )
```

この平均偏差では、**numeric-expression** が null のローは除外されます。グループにローが含まれていない場合は、null を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, meandeviation(xinput.d0) AS d0,
meandeviation(xinput.d1) AS d1, meandeviation(xinput.d2) AS d2,
meandeviation(xinput.d3) AS d3, meandeviation(xinput.d4) AS d4,
meandeviation(xinput.d5) AS d5, meandeviation(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

median()

集合。複数のローを対象にして、指定された式の中間値を返します。

構文

```
median ( column )
```

パラメータ

column	バイナリを除く任意のデータ型を受け入れるカラム名。

使用法

関数は、カラムと同じデータ型を返します。

中央値は、標本分布、母集団分布、または確率分布の上半分と下半分を分割する数値として説明されます。数値の有限リストの中央値は、すべての観測値を最小値から最大値まで順序で並び替えたときの、中央の値です。観測値が偶数個存在する場合、単一の中央値は存在しません。この場合、中央値は、通常、2 つの中央の値の平均値として定義されます。

median 関数の動作は、データ型によって異なります。

- 整数 – 結果は、2 つの中央の値の平均値を算出し、最も近い整数に丸めた値。
- 通貨 – 結果は、2 つの中央の値の平均値。
- 文字列 – 結果は、2 つの中央の文字の最初の文字。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, median(xinput.d0) AS d0,
median(xinput.d1) AS d1, median(xinput.d2) AS d2,
median(xinput.d3) AS d3, median(xinput.d4) AS d4,
median(xinput.d5) AS d5, median(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

min()

集合。値のグループの、null 以外の最小値を返します。

構文

```
min ( expression )
```

パラメータ

expression	任意のデータ型

使用法

戻り値は、入力のデータ型に基づきます。すべての値が null の場合、関数は null を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0max INTEGER, d0min INTEGER, d1max LONG,
d1min LONG, d2max MONEY(4), d2min MONEY(4), d3max FLOAT,
d3min FLOAT, d4max BOOLEAN, d4min BOOLEAN, d5max TIMESTAMP,
d5min TIMESTAMP, d6max INTERVAL, d6min INTERVAL,
d7max BIGDATETIME, d7min BIGDATETIME)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, max(xinput.d0) AS d0max,
min(xinput.d0) AS d0min, max(xinput.d1) AS d1max,
min(xinput.d1) AS d1min, max(xinput.d2) AS d2max,
min(xinput.d2) AS d2min, max(xinput.d3) AS d3max,
min(xinput.d3) AS d3min, max(xinput.d4) AS d4max,
min(xinput.d4) AS d4min, max(xinput.d5) AS d5max,
min(xinput.d5) AS d5min, max(xinput.d6) AS d6max,
min(xinput.d6) AS d6min, max(xinput.d7) AS d7max,
min(xinput.d7) AS d7min
FROM xinput
GROUP BY xinput.a;
```

nth()

集合。値のグループの n 番目の値を返します。最初の引数によって、返す値が決定されます。

構文

```
nth ( number, expression )
```

パラメータ

number	整数。
expression	任意のデータ型の値のグループ。

使用法
関数は、**expression** 引数と同じデータ型を返します。

例
この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, nth(0,xinput.d0) AS d0,
nth(1,xinput.d1) AS d1, nth(2,xinput.d2) AS d2,
nth(3,xinput.d3) AS d3, nth(4,xinput.d4) AS d4,
nth(5,xinput.d5) AS d5, nth(6,xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

recent()

集合。値のグループの、null 以外の最新の値を返します。

構文

```
recent ( expression )
```

パラメータ

expression	任意のデータ型の値のグループ。

使用法
関数は、式で使用されているのと同じデータ型を返します。

例
この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
 PRIMARY KEY DEDUCED
```

```
SELECT xinput.a AS a, recent(xinput.d0) AS d0,
recent(xinput.d1) AS d1, recent(xinput.d2) AS d2,
recent(xinput.d3) AS d3, recent(xinput.d4) AS d4,
recent(xinput.d5) AS d5, recent(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

regr_avgx()

集合。回帰線の独立変数の平均を計算します。

構文

```
regr_avgx ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、整数、長整数、浮動小数点数、タイムスタンプ、間隔、通貨のデータ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、整数、長整数、浮動小数点数、タイムスタンプ、間隔、通貨のデータ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を除外した後で、次の計算が実行されます。ここで、式の *y* は **dependent-expression** を表します。

```
avg( y )
```

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d4d5 FLOAT, d5d4 FLOAT, d1d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_avgx(xinput.d0, xinput.d1) AS d0d1,
regr_avgx(xinput.d1, xinput.d0) AS d1d0, regr_avgx(xinput.d2,
xinput.d3) AS d2d3, regr_avgx(xinput.d3, xinput.d2) AS d3d2,
regr_avgx(xinput.d4, xinput.d5) AS d4d5, regr_avgx(xinput.d5,
xinput.d4) AS d5d4, regr_avgx(xinput.d1, xinput.d6) AS d1d6
```

```
FROM xinput
GROUP BY xinput.a;
```

regr_avgy()

集合。回帰線の従変数の平均を計算します。

構文

```
regr_avgy ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、整数、長整数、浮動小数点数、タイムスタンプ、間隔、通貨のデータ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、整数、長整数、浮動小数点数、タイムスタンプ、間隔、通貨のデータ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を除外した後で、次の計算が実行されます。ここで、式の *x* は **independent-expression** を表します。

```
avg( x )
```

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d4d5 FLOAT, d5d4 FLOAT, d1d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_avgy(xinput.d0, xinput.d1) AS d0d1,
regr_avgy(xinput.d1, xinput.d0) AS d1d0, regr_avgy(xinput.d2,
xinput.d3) AS d2d3, regr_avgy(xinput.d3, xinput.d2) AS d3d2,
regr_avgy(xinput.d4, xinput.d5) AS d4d5,
regr_avgy(xinput.d5, xinput.d4) AS d5d4,
regr_avgy(xinput.d1, xinput.d6) AS d1d6
FROM xinput
GROUP BY xinput.a;
```

regr_count()

集合。回帰線適合のために使用された非 NULL 値のペアの数を示す整数を返します。

構文

```
regr_count ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、整数、長整数、浮動小数点数、タイムスタンプ、間隔、通貨のデータ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、整数、長整数、浮動小数点数、タイムスタンプ、間隔、通貨のデータ型を受け入れます。

使用法

この関数は、null 以外のローのすべてのセットを計数し、長整数を返します。いずれか、または両方が null のローは除外されます。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 LONG, d2d3 LONG, d4d5 LONG, d6d7 LONG)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_count(xinput.d0, xinput.d1) AS d0d1,
regr_count(xinput.d2, xinput.d3) AS d2d3,
regr_count(xinput.d4, xinput.d5) AS d4d5,
regr_count(xinput.d6, xinput.d7) AS d6d7
FROM xinput
GROUP BY xinput.a;
```

regr_intercept()

集合。従変数と独立変数が最も適合する線形回帰線の y 切片を計算します。

構文

```
regr_intercept ( dependent-expression, independent-
expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を削除した後に、この計算が行われます。ここで、*x* は独立変数を表し、*y* は従属変数を表します。

```
avg( x ) - regr_slope( x, y ) * avg( y )
```

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_intercept(xinput.d0, xinput.d1) AS d0d1,
regr_intercept(xinput.d1, xinput.d0) AS d1d0,
regr_intercept(xinput.d2, xinput.d3) AS d2d3,
regr_intercept(xinput.d3, xinput.d2) AS d3d2,
regr_intercept(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

regr_r2()

集合。回帰線の決定係数 (R-squared または適合度統計とも呼ばれます) を計算します。

構文

```
regr_r2 ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、次の式を使用し、データを 1 回参照して同時に計算されます。ここで、*x* は独立変数を表し、*y* は従属変数を表します。

```
covarPOP = ((_sum_xy * count) - (sum_x * sum_y)) * ((_sum_xy * count)
- (sum_x * sum_y))
xVarPop = (sum_xx  * count) - (sum_x * sum_x)
yVarPop = (sum_yy * count) - (sum_y * sum_y)
result = covarPOP / (xvarPop * yVarPop)
```

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_r2(xinput.d0, xinput.d1) AS d0d1,
regr_r2(xinput.d1, xinput.d0) AS d1d0,
regr_r2(xinput.d2, xinput.d3) AS d2d3,
regr_r2(xinput.d3, xinput.d2) AS d3d2,
regr_r2(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

regr_slope()

集合。NULL 以外のペアに調整された線形回帰直線の傾きを計算します。

構文

```
regr_slope ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

パラメータ

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を削除した後に、この計算が行われます。ここで、*x* は独立変数を表し、*y* は従属変数を表します。

```
covar_pop( x, y ) / var_pop( y )
```

例

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
  PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_slope(xinput.d0, xinput.d1) AS d0d1,
regr_slope(xinput.d1, xinput.d0) AS d1d0,
regr_slope(xinput.d2, xinput.d3) AS d2d3,
regr_slope(xinput.d3, xinput.d2) AS d3d2,
regr_slope(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

regr_sxx()

集合。線形回帰モデルに使用される独立した式の平方値の合計を返します。回帰モデルの統計的な有効性を評価するときに使用できます。

構文

```
regr_sxx ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を削除した後に、この計算が行われます。ここで、*x* は独立変数を表し、*y* は従属変数を表します。

```
regr_count( x, y ) * var_pop( x )
```

例

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_sxx(xinput.d0, xinput.d1) AS d0d1,
regr_sxx(xinput.d1, xinput.d0) AS d1d0,
regr_sxx(xinput.d2, xinput.d3) AS d2d3,
regr_sxx(xinput.d3, xinput.d2) AS d3d2,
regr_sxx(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

regr_sxy()

集合。従変数および独立変数の積和を返します。回帰モデルの統計的な有効性を評価するときに使用できます。

構文

```
regr_sxy ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。
independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を削除した後に、この計算が行われます。ここで、*x* は従属変数を表し、*y* は独立変数を表します。

```
regr_count( x, y ) * covar_pop( x, y )
```

例

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_sxy(xinput.d0, xinput.d1) AS d0d1,
regr_sxy(xinput.d1, xinput.d0) AS d1d0,
regr_sxy(xinput.d2, xinput.d3) AS d2d3,
regr_sxy(xinput.d3, xinput.d2) AS d3d2,
regr_sxy(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

regr_syy()

集合。回帰モデルの統計的な有効性を表す値を返します。

構文

```
regr_syy ( dependent-expression , independent-expression )
```

パラメータ

dependent-expression	独立した変数の影響を受ける変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

independent-expression	結果に影響を与える変数。式は、タイムスタンプ、bigdatetime、間隔を除く数値データ型を受け入れます。

使用法

この関数は引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。関数は、空のセットに適用されると null を返します。関数は、いずれかの変数が null である **dependent-expression** と **independent-expression** のペアのすべてが削除された後に、このペアのセットに適用されます。関数は、データを 1 回参照して同時に計算されます。null 値を削除した後に、この計算が行われます。ここで、*x* は従属変数を表し、*y* は独立変数を表します。

```
regr_count( x, y ) * var_pop( y )
```

例

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0d1 FLOAT, d1d0 FLOAT, d2d3 FLOAT, d3d2 FLOAT,
d1d4 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, regr_syy(xinput.d0, xinput.d1) AS d0d1,
regr_syy(xinput.d1, xinput.d0) AS d1d0,
regr_syy(xinput.d2, xinput.d3) AS d2d3,
regr_syy(xinput.d3, xinput.d2) AS d3d2,
regr_syy(xinput.d1, xinput.d4) AS d1d4
FROM xinput
GROUP BY xinput.a;
```

stddev()

集合。標本の標準偏差を計算します。stddev_samp() のエイリアスです。

stddeviation()

集合。複数のローを対象にして、指定された式の標準偏差を返します。stddev_samp() のエイリアスです。

stddev_pop()

集合。数値式からなる母集団の標準偏差を浮動小数点数として計算します。

構文

```
stddev_pop ( numeric-expression )
```

パラメータ

numeric-expression	その母集団ベースの標準偏差がローのセットに対して計算される式 (通常はカラム名)。

使用法
この関数は、引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。母集団ベースの標準偏差は、次の式に従って計算されます。

```
s = [ (1/N) * SUM( xi - MEAN( x ) )2 ]1/2
```

この標準偏差には、数値式が null のローは含まれません。関数は、ローを含まないグループに対して null を返します。

例
この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, 6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, stddev_pop(xinput.d0) AS d0,
stddev_pop(xinput.d1) AS d1, stddev_pop(xinput.d2) AS d2,
stddev_pop(xinput.d3) AS d3, stddev_pop(xinput.d4)
AS d4, stddev_pop(xinput.d5) AS d5, stddev_pop(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

stddev_samp()

集合。数値式からなる標本の標準偏差を浮動小数点数として計算します。

構文

```
stddev_samp ( numeric-expression )
```

パラメータ

numeric-expression	標本ベースの標準偏差がローのセットに対して計算される式 (通常はカラム名)。

使用法

この関数は、引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。標準偏差は、正規分布であることを想定する、次の式に従って計算されます。

```
s = [ (1/( N - 1 )) * SUM( xi - mean( x ) )² ]½
```

この標準偏差には、数値式が null のローは含まれません。関数は、0 個または 1 個のローを含むグループに対して null を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, stddev_samp(xinput.d0) AS d0,
stddev_samp(xinput.d1) AS d1, stddev_samp(xinput.d2) AS d2,
stddev_samp(xinput.d3) AS d3, stddev_samp(xinput.d4) AS d4,
stddev_samp(xinput.d5) AS d5, stddev_samp(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

sum()

集合。各グループのローの、指定された式の合計値を返します。

構文

```
sum ( expression )
```

パラメータ

expression	合計値を計算する対象のオブジェクト。式は、ブール以外のすべての型を受け入れます。

使用法

一般的に、**sum** は、カラムに対して実行されます。関数は、式と同じデータ型を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d5 TIMESTAMP, d6 INTERVAL)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, sum(xinput.d0) AS d0, sum(xinput.d1) AS d1,
sum(xinput.d2) AS d2, sum(xinput.d3) AS d3, sum(xinput.d5) AS d5,
sum(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

valueinserted()

集合。対象のグループに最後に挿入されたローに基づいて、グループから値 (null 値の場合もあります) を返します。

構文

```
valueinserted ( expression )
```

パラメータ

expression	式はすべてのデータ型を受け入れる。

使用法

関数は引数として任意のデータ型を受け取り、式と同じデータ型を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE  INPUT  WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
PRIMARY KEY (id)

CREATE  OUTPUT  WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL, d7 BIGDATETIME)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, valueinserted(xinput.d0) AS d0,
valueinserted(xinput.d1) AS d1, valueinserted(xinput.d2) AS d2,
valueinserted(xinput.d3) AS d3, valueinserted(xinput.d4) AS d4,
valueinserted(xinput.d5) AS d5, valueinserted(xinput.d6) AS d6,
valueinserted(xinput.d7) AS d7
FROM xinput
GROUP BY xinput.a;
```

var_pop()

集合。1 つの数値式で構成される母集団の統計分散を浮動小数点数として計算します。

構文

```
var_pop ( numeric-expression )
```

パラメータ

numeric-expression	ローのセット。**expression** は、通常、カラム名です。

使用法

この関数は、引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。数値式 (*x*) の母集団ベースの分散 (*s2*) は、次の式に従って計算されます。

```
s2 = (1/N) * SUM( xi - mean( x ) )2
```

この偏差には、numeric-expression が null のローは含まれません。関数は、ローを含まないグループに対して null を返します。

例

この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE INPUT WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE OUTPUT WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, var_pop(xinput.d0) AS d0,
var_pop(xinput.d1) AS d1, var_pop(xinput.d2) AS d2,
var_pop(xinput.d3) AS d3, var_pop(xinput.d4) AS d4,
var_pop(xinput.d5) AS d5, var_pop(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

var_samp()

集合。数値式からなる標本の統計分散を浮動小数点数として計算します。

構文

```
var_samp ( numeric-expression )
```

パラメータ

numeric-expression	ローのセット。**expression** は、通常、カラム名です。

使用法
この関数は、引数を浮動小数点数に変換し、計算を倍精度浮動小数点で実行して、浮動小数点数を返します。数値式 (*x*) の分散 (*s2*) は、正規分布であることを想定する、次の式に従って計算されます。

```
s2 = (1/( N - 1 )) * SUM( xi - mean( x ) )2
```

この偏差には、numeric-expression が null のローは含まれません。関数は、0 個または 1 個のローを含むグループに対して null を返します。

例
この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE INPUT WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d4 BOOLEAN, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE OUTPUT WINDOW aggregate
SCHEMA (a INTEGER, d0 FLOAT, d1 FLOAT, d2 FLOAT, d3 FLOAT,
d4 FLOAT, d5 FLOAT, d6 FLOAT)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, var_samp(xinput.d0) AS d0,
var_samp(xinput.d1) AS d1, var_samp(xinput.d2) AS d2,
var_samp(xinput.d3) AS d3, var_samp(xinput.d4) AS d4,
var_samp(xinput.d5) AS d5, var_samp(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

vwap()

集合。**vwap** 関数は、一連のトランザクションの出来高加重平均価格を計算します。

構文

```
vwap ( price, quantity )
```

パラメータ

price	一連のトランザクション・レコードの価格を含むカラムの名前。
quantity	一連のトランザクション・レコード内で指定した価格で取引された単位数を含むカラムの名前。

注意： これらの両方のパラメータでカラム名を含む式を指定できますが、カラム名を含む必要があります。

使用法
出来高加重平均価格 (VWAP) は、一定期間に株式が取引された平均価格を測定したものです。各取引に対して、株式ごとの支払い価格と取引された株式数を乗算することによって値を決定します。次に、これらのすべての値の総和を算出し、取引されたすべての株式の総数で除算します。

vwap 関数は引数として、支払われた価格と取引された株式の数を受け取ります。入力ストリームまたは入力ウィンドウから提供される取引イベントを使用して、**vwap** 関数は VWAP を計算し、株式が取引された平均価格を追跡します。

例
次の例は、名前付きウィンドウ Trades に存在する一連の取引レコードを通して、各証券コードの VWAP を計算します。

```
CREATE INPUT WINDOW Trades
SCHEMA (Id integer, TradeTime date, Symbol string, Price float,
Shares integer)
PRIMARY KEY(Id) ;

CREATE OUTPUT WINDOW VWAP
PRIMARY KEY DEDUCED
AS SELECT trd.Symbol, vwap(trd.Price, trd.Shares) vwap FROM Trades
trd
GROUP BY trd.Symbol;
```

weighted_avg()

集合。算術 (または線形) 加重平均を計算します。

構文

```
weighted_avg ( expression )
```

パラメータ

expression	整数、長整数、浮動小数点数、通貨、タイムスタンプ、間隔のデータ型を受け入れる数値式。

使用法
算術加重平均では、データ・ポイントごとに異なる加重を付加する乗算係数が使用されます。技術的な分析の分野では、加重移動平均 (WMA) は、算術的に減少する、特別な意味の加重を持ちます。n 日の WMA では、最後の日が n の加重、最後の日の前日が $n-1$ の加重を持ち、以下同様に、加重がゼロに向かって小さくなり

ます。

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \cdots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \cdots + 2 + 1}$$

例
この例は、関数を CCL コードに組み込む方法を示しています。

```
CREATE INPUT WINDOW xinput
SCHEMA (id INTEGER, a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4),
d3 FLOAT, d5 TIMESTAMP, d6 INTERVAL)
PRIMARY KEY (id)

CREATE OUTPUT WINDOW aggregate
SCHEMA (a INTEGER, d0 INTEGER, d1 LONG, d2 MONEY(4), d3 FLOAT,
d5 TIMESTAMP, d6 INTERVAL)
 PRIMARY KEY DEDUCED

SELECT xinput.a AS a, weighted_avg(xinput.d0) AS d0,
weighted_avg(xinput.d1) AS d1, weighted_avg(xinput.d2) AS d2,
weighted_avg(xinput.d3) AS d3, weighted_avg(xinput.d5) AS d5,
weighted_avg(xinput.d6) AS d6
FROM xinput
GROUP BY xinput.a;
```

xmlagg()

集約。複数の XML 値を単一の値に集約します。

構文

```
xmlagg ( value )
```

パラメータ

value	文字列として表される XML 値。

使用法
この関数は、集約ストリーム内、またはイベント・キャッシュと組み合わせてのみ使用でき、文字列を返します。

例

```
xmlagg ( xmlparse (stringCol) )
```

スカラ関数

スカラ関数は、スカラ引数のリストを受け取り、単一のスカラ値を返します。

次のタイプのスカラ関数が利用できます。

- 数値関数
- 文字列関数
- 変換関数
- XML 関数
- 日付と時刻の関数

スカラ関数は、引数として 1 つ以上の式値を受け取り、クエリによって処理されたデータのローごとに単一の結果値を返します。これらの関数は、ほとんどの式で使用できます。また、**SELECT** 句と **WHERE** 句で頻繁に使用されます。

参照：

- *SELECT 句* (121 ページ)
- *WHERE 句* (125 ページ)

数値関数

数値関数は、数値と一緒に使用されます。一部の数値関数は、間隔や bigdatetime の値とも一緒に使用できます。数値関数の例には、`round（）`と `sqrt（）`があります。

acos()

スカラ。指定された値の逆余弦を返します。

構文

```
acos ( value )
```

パラメータ

value	-1 と 1 の間の浮動小数点数。

使用法

関数は浮動小数点数を返します。-1 ～ 1 の範囲外の値が指定されると、関数は null を返します。

例

`acos(0.0)` は、1.570796 を返します。

asin()

スカラ。指定された値の逆正弦を返します。

構文

```
asin ( value )
```

パラメータ

value	-1 と 1 の間の浮動小数点数。

使用法

関数は浮動小数点数を返します。-1 ～ 1 の範囲外の値が指定されると、関数は null を返します。

例

`asin(1.0)` は、1.570796 を返します。

atan()

スカラ。指定された値の逆正接を返します。

構文

```
atan ( value )
```

パラメータ

value	浮動小数点数。

使用法

関数は浮動小数点数を返します。

例

`arctan(1.0)` は、0.785398 を返します。

atan2()

スカラ。指定された 2 つの値の商の逆正接を返します。

構文

```
atan2 ( value1, value2 )
```

パラメータ

value1	浮動小数点数。

value2	浮動小数点数。

使用法
標準的な逆正接関数の範囲内で、指定された値の商の逆正接を返します。

- **value2** > 0 の場合、atan2 (value1, value2) は atan (value1/value2) の値を返す。
- **value1** >= 0 かつ **value2** < 0 の場合、atan2 (value1, value2) は atan (value1/value2) + pi() の値を返す。
- **value1** < 0 かつ **value2** < 0 の場合、atan2 (value1, value2) は atan (value1/value2) - pi() の値を返す。
- **value1** > 0 かつ **value2** = 0 の場合、atan2 (value1, value2) は pi()/2 の値を返す。
- **value1** < 0 かつ **value2** = 0 の場合、tan2 (value1, value2) は -pi()/2 の値を返す。
- **value1** = **value2** = 0 の場合、atan2 (value1, value2) は 0 を返す。

例
atan2 (1, 2) は、atan (0.5) の値である 0.463647609 を返します。

avgof()

スカラ。複数の式の平均値を返します。null パラメータは無視します。

構文

```
avgof ( expression, [,...] )
```

パラメータ

expression	少なくとも 1 つの引数が必要で、すべての引数は同じデータ型である必要あり。

使用法
すべてのパラメータが null の場合、関数は null を返します。 関数は、浮動小数点数、整数、長整数、間隔、通貨型、日付／時刻型の各データ型を受け入れます。

関数は、引数と同じデータ型を返しますが、式が数値型 (整数、浮動小数点数、または長整数) の場合は浮動小数点数を返します。

例
avgof (1, 2, NULL, 3, NULL) は、2.0 を返します。

bitand()

スカラ。2 つの式に対して AND ビット処理演算を実行した結果を返します。

構文

```
bitand ( expression1, expression2 )
```

パラメータ

expression1	整数または長整数に単純化する式 (**expression2** と同じデータ型である必要あり)。
expression2	整数または長整数に単純化する式 (**expression1** と同じデータ型である必要あり)。

使用法

関数は、2 つの式を引数に取り、ビットの各ペアに対して AND 論理演算を実行します。両ビットが 1 の場合は、ペアの結果は 1 です。それ以外の場合は、ペアの結果は 0 です。両方の引数は同じデータ型 (整数または長整数) である必要があり、関数は引数と同じデータ型を返します。

例

`bitand（5, 3）`は 1 を返します。または、バイナリ値で`bitand ( 101, 011)`は 001 を返します。ユーザはバイナリ値を直接指定できません。

bitclear()

スカラ。特定のビットをゼロに設定した後の式の値を返します。

構文

```
bitclear ( expression, bit )
```

パラメータ

expression	整数または長整数の初期値。
bit	最下位ビットを 0 として始めて、クリアするビット。

使用法

bit 引数は整数である必要があります。関数は、最初の **expression** 引数と同じデータ型を返します。

例

`bitclear（13, 0）`は 12 を返します。または、バイナリ値で`bitclear（1101, 0）`は 1100 を返します。ユーザはバイナリ値を直接指定できません。

bitflag()

スカラ。指定されたビットを除くすべてのビットがゼロに設定された値を返します。

構文

```
bitflag ( bit )
```

パラメータ

bit	どのビットを設定するかを示す整数。最下位ビットを 0 として始めます。

使用法

関数は整数を返します。

例

`bitflag( 3 )` は 8 すなわちバイナリの 1000 を返します。

bitflaglong()

スカラ。指定されたビットを除くすべてのビットがゼロに設定された値を返します。

構文

```
bitflaglong ( bit )
```

パラメータ

bit	どのビットを設定するかを示す整数。最下位ビットを 0 として始めます。

使用法

関数は長整数を返します。

例

`bitflaglong（35）` は 34359738368 すなわちバイナリの 100000000000000000000000000000000000 を返します。

bitmask()

スカラ。指定された範囲のビットを除くすべてのビットが 0 に設定された値を返します。

構文

```
bitmask ( first, last )
```

パラメータ

first	最下位ビットを 0 として始めて、設定する最初のビット。
last	最下位ビットを 0 として始めて、設定する最後のビット。

使用法
両方の引数は整数である必要があります。関数は整数を返します。引数の順序は重要ではありません。つまり、`bitmask（1，3）`は、 `bitmask（3，1）`と同じ結果が得られます。

例
`bitmask（1，3）`は 14 すなわちバイナリの 1110 を返します。

`bitmask（3，0）`は 15 すなわちバイナリの 1111 を返します。

bitmasklong()
スカラ。指定された範囲のビットを除くすべてのビットが 0 に設定された値を返します。

構文
```
bitmasklong ( first, last )
```

パラメータ

first	最下位ビットを 0 として始めて、設定する最初のビット。
last	最下位ビットを 0 として始めて、設定する最後のビット。

使用法
両方の引数は整数である必要があります。関数は長整数を返します。

例
`bitmasklong（33，35）`は 60129542144 すなわちバイナリの 111000000000000000000000000000000000 を返します。

bitnot()
スカラ。すべてのビットが反転された、式の値を返します。

構文
```
bitnot ( expression )
```

パラメータ

expression	整数または長整数。

使用法
ビット処理演算を実行した後の式の値を返します。0 であったビットは 1 になり、1 であったビットは 0 になります。関数は、引数と同じデータ型を返します。

例
`bitnot（7）`は -8 を返します。または、バイナリ値で `bitnot（111）` は 11111111111111111111111111111000 を返します。ユーザはバイナリ値を直接指定できません。

bitor()

スカラ。2 つの式に対して OR ビット処理演算を実行した結果を返します。

構文

```
bitor ( expression1, expression2 )
```

パラメータ

expression1	整数または長整数に単純化する式 (**expression2** と同じである必要あり)。
expression2	整数または長整数に単純化する式 (**expression1** と同じである必要あり)。

使用法
関数は、2 つのビット・パターンを引数に取り、ビットの各ペアに対して OR 論理演算を実行して同じ長さの別のビット・パターンを生成します。最初のビット・パターンのビットまたは 2 番目のビット・パターンのビットが 1 の場合または両ビット・パターンのビットが 1 の場合は、ペアの結果は 1 です。それ以外の場合は、ペアの結果は 0 です。関数は引数と同じデータ型を返します。

例
`bitor（5, 3）`は 7 を返します。または、バイナリ値で `bitor（0101, 0011）` は 0111 を返します。ユーザはバイナリ値を直接指定できません。

bitset()

スカラ。特定のビットを 1 に設定した後の式の値を返します。

構文

```
bitset ( expression, bit )
```

パラメータ

expression	整数または長整数の初期値。
bit	最下位ビットを 0 として始めて、設定するビット。

使用法

bit 引数は、整数である必要があります。関数は、最初の **expression** 引数と同じデータ型を返します。

例

`bitset（2，3）`は 10 を返します。または、バイナリ値で `bitset（0010，3）`は 1010 を返します。ユーザはバイナリ値を直接指定できません。

bitshiftleft()

スカラ。ビットを、特定の位置数だけ左にシフトさせた後の式の値を返します。

構文

```
bitshiftleft ( expression, count )
```

パラメータ

expression	整数または長整数の初期値。整数または長整数を指定できます。
count	シフトする位置数。この数と同じ数の右端のビットが 0 に設定されます。整数である必要があります。

使用法

左へのシフトによってあふれたビットは破棄され、右側にゼロが設定されます。**expression** 引数は整数または長整数とすることができますが、**count** 引数は整数である必要があります。関数は、最初の **expression** 引数と同じデータ型を返します。

例

`bitshiftleft（10，2）`は 40 を返します。または、バイナリ値で `bitshiftleft（1010，2）`は 101000 を返します。ユーザはバイナリ値を直接指定できません。

bitshiftright()

スカラ。ビットを、特定の位置数だけ右にシフトさせた後の式の値を返します。

構文

```
bitshiftright ( expression, count )
```

パラメータ

expression	整数または長整数の初期値。整数または長整数を指定できます。
count	シフトする位置数。この数と同じ数の左端のビットが 0 に設定されます。整数である必要があります。

使用法
右へのシフトによってあふれたビットは破棄され、左側にゼロが設定されます。関数は、最初の **expression** 引数と同じデータ型を返します。

例
`bitshiftright（3, 1）`は 1 を返します。または、バイナリ値で `bitshiftright（0011, 1）`は 0001 を返します。ユーザはバイナリ値を直接指定できません。

bittest()

スカラ。バイナリ値内の特定ビットの値を返します。

構文

```
bittest ( expression, bit )
```

パラメータ

expression	整数または長整数の初期値。
bit	返すビット。他のすべてのビットはゼロに設定されます。

使用法
bit 引数は整数である必要があります。関数は、**expression** 引数のデータ型と同じデータ型を返します。

例
`bittest（15, 3）`は 8 を返します。または、バイナリ値で `bittest( 1111, 3）`は 1000 を返します。ユーザはバイナリ値を直接指定できません。

bittoggle()

スカラ。特定のビットの値を反転させた後の式の値を返します。

構文

```
bittoggle ( expression, bit )
```

パラメータ

expression	整数または長整数の初期値
bit	切り換えるビット

使用法
expression 引数は整数または長整数とすることができますが、**bit** 引数は整数である必要があります。 関数は、**expression** 引数のデータ型と同じデータ型を返します。

例
`bittoggle（7, 3）`は 15 を返します。または、バイナリ値で `bittoggle（0111, 3）`は 1111 を返します。ユーザはバイナリ値を直接指定できません。

bitxor()
スカラ。2 つの式に対して排他 OR (XOR) ビット処理演算を実行した結果を返します。

構文
```
bitxor ( expression1, expression2 )
```

パラメータ

expression1	整数または長整数に単純化する式 (**expression2** と同じデータ型である必要あり)。
expression2	整数または長整数に単純化する式 (**expression1** と同じデータ型である必要あり)。

使用法
関数は、2 つの式の対応するビットの各ペアの対して XOR 論理演算を実行します。2 つのビットが異なる場合は、ビットのペアの結果は 1 です。2 つのビットが同じ場合は、結果は 0 です。同じ式に対して **bitxor()** を使用すると、0 が得られます。関数は、引数と同じデータ型を返します。

例
`bitxor（3, 3）`は、0 を返します。

`bitxor（10, 15）`は 5 を返します。または、バイナリ値で `bitxor（1010, 1111）`は 0101 を返します。ユーザはバイナリ値を直接指定できません。

cbrt()

スカラ。数値の立方根を返します。

構文

```
cbrt ( value )
```

パラメータ

value	数値データ型

使用法

関数は浮動小数点数を返します。引数が無効な場合、サーバで、`Floating-point exception` エラーが記録されます。

例

`cbrt（1000.00）`は、10.0 を返します。

ceil()

スカラ。数値を最も近い整数に切り上げます。

構文

```
ceil ( value )
```

パラメータ

value	浮動小数点数または通貨型

使用法

関数は、引数と同じデータ型を返します。

例

`ceil（100.20）`は、101.0 を返します。

compare()

スカラ。2 つの値のどちらの方が大きいかを判断します。

構文

```
compare ( value1, value2 )
```

パラメータ

value1	任意のデータ型

value2	任意のデータ型

使用法
関数は整数 (1、-1、または 0) を返します。最初の値の方が大きい場合は、関数は 1 を返します。2 番目の値の方が大きい場合は、-1 を返します。両方の値が等しい場合は、0 を返します。

例
`compare ( (asin(0.5), (acos(0.5) )` は、-1 を返します。

cos()
スカラ。指定された値の余弦をラジアンで返します。

構文
```
cos ( value )
```

パラメータ

value	浮動小数点数。

使用法
関数は浮動小数点数を返します。

例
`cos (0.5)`は、0.87758 を返します。

cosd()
スカラ。指定された値の余弦を度数で返します。

構文
```
cosd ( value )
```

パラメータ

value	浮動小数点数。

使用法
関数は浮動小数点数を返します。

例
`cosd (90.0)`は、-0.448073616 を返します。

cosh()

スカラ。指定された値の双曲線余弦をラジアンで返します。

構文

```
cosh ( value )
```

パラメータ

value	浮動小数点数。

使用法

関数は浮動小数点数を返します。

例

`cosh (0.5)`は、1.12762597 を返します。

distance()

スカラ。2 次元または 3 次元の 2 点間の距離を表す値を返します。

構文

```
distance ( point1x, point1y, [point1z], point2x, point2y,
[point2z] )
```

パラメータ

point1x	x 軸上の最初の点の位置を表す値に評価される式。
point1y	y 軸上の最初の点の位置を表す値に評価される式。
point1z	z 軸上の最初の点の位置を表す値に評価される式。
point2x	x 軸上の 2 番目の点の位置を表す値に評価される式。
point2y	y 軸上の 2 番目の点の位置を表す値に評価される式。
point2z	z 軸上の 2 番目の点の位置を表す値に評価される式。

使用法

2 次元または 3 次元のいずれかの 2 点間の距離を表す数値を返します。すべての引数は同じ数値型である必要があり、関数は同じデータ型を返します。

例

`distance (7.5, 6.5, 10.5, 10.5)`は、5.0 を返します。

`distance ( 1.2, 3.4, 5.6, 7.8, 9.10, 11.12 )` は、10.320872 を返します。

distancesquared()

スカラ。2 次元または 3 次元のいずれかの 2 点間の距離の平方を表す数値を返します。

構文

```
distancesquared ( point1x, point1y, [point1z], point2x, point2y,
[point2z] )
```

パラメータ

point1x	x 軸上の最初の点の位置を表す値に評価される式。
point1y	y 軸上の最初の点の位置を表す値に評価される式。
point1z	z 軸上の最初の点の位置を表す値に評価される式。
point2x	x 軸上の 2 番目の点の位置を表す値に評価される式。
point2y	y 軸上の 2 番目の点の位置を表す値に評価される式。
point2z	z 軸上の 2 番目の点の位置を表す値に評価される式。

使用法

2 次元または 3 次元のいずれかの 2 点間の距離の平方を表す数値を返します。すべての引数は同じ数値型である必要があり、関数は同じデータ型を返します。

例

`distancesquared (7.5, 6.5, 10.5, 10.5)` は、25.0 を返します。

`distancesquared ( 1.2, 3.4, 5.6, 7.8, 9.10, 11.12 )` は、106.502400 を返します。

floor()

スカラ。値を切り捨てます。

構文

```
floor ( value )
```

パラメータ

value	浮動小数点数または通貨型。

使用法
指定された数値を最も近い整数に切り捨てます。関数は、引数として浮動小数点数または通貨型を受け取り、引数と同じデータ型を返します。

例
`floor（100.20）`は、100.0 を返します。

`floor（ 1.56）`は、1.0 を返します。

isnull()

スカラ。式が null であるかどうかを判断します。

構文
```
isnull ( expression )
```

パラメータ

expression	任意のデータ型の式。

使用法
式が null であるかどうかを判断します。引数のデータ型に制限はありません。関数は、整数を返します。引数が null の場合は 1 を、その他の場合は 0 を返します。

例
`isnull（'examplestring'）`は、0 を返します。

length()

スカラ。指定されたバイナリ値のバイト数を返します。

構文
```
length ( binary )
```

パラメータ

binary	バイナリ値。

使用法
指定されたバイナリ値を構成するバイト数を返します。関数はバイナリ値を引数として受け取り、整数を返します。バイナリ値が null であれば、関数は null を返します。

例
`length（hex_binary（'0xaa1234'））`は、3 を返します。

`length (hex_binary ('aa'))` は、1 を返します。

ln()

スカラ。指定された数値の自然対数を返します。

構文

```
ln ( value )
```

パラメータ

value	浮動小数点数。

使用法

数値の自然対数を返します。引数が無効な (たとえば、0 未満) 場合、サーバで「浮動小数点数例外」エラーが記録されます。 関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例

`ln (2.718281828)` は、1.0 を返します。

log2()

スカラ。指定された値の、2 を底とする対数を返します。

構文

```
log2 ( value )
```

パラメータ

value	0 以上の浮動小数点数に評価される式。

使用法

指定された値の、2 を底とする対数を返します。関数は、引数として浮動小数点数を想定しますが、関数の実行時に整数が浮動小数点数に拡張されます。関数は浮動小数点数を返します。

例

`log2 (8.0)` は、3.0 を返します。

log10()

スカラ。指定された値の、10 を底とする対数を返します。

構文

```
log10 ( value )
```

パラメータ

value	0 以上の浮動小数点数に評価される式。

使用法

指定された値の、10 を底とする対数を返します。関数は、引数として浮動小数点数を想定しますが、関数の実行時に整数が浮動小数点数に拡張されます。関数は浮動小数点数を返します。

例

`log（100.0）`は、2.0 を返します。

logx()

スカラ。指定された値の、指定された底の対数を返します。

構文

```
logx ( base, value )
```

パラメータ

base	1 より大きい浮動小数点数に評価される式。
value	0 以上の浮動小数点数に評価される式。

使用法

指定された値の、指定された底の対数を返します。関数は、引数として浮動小数点数を想定しますが、関数の実行時に整数が浮動小数点数に拡張されます。関数は浮動小数点数を返します。

例

`logx（2.0, 8.0）`は、3.0 を返します。

maxof()

スカラ。式のリストから最大値を返します。

構文

```
maxof ( expression [,...] )
```

パラメータ

expression	少なくとも 1 つの引数が必要で、すべての引数は同じデータ型である必要あり。

使用法
式のリストから最大値を返します。null 値は無視されます。 すべての引数が null の場合、関数は null を返します。 引数のデータ型に制限はありませんが、すべての引数が同じデータ型である必要があります。関数は、引数と同じデータ型を返します。

例
`maxof（1.34, 3.35, 10.93, NULL）`は、10.93 を返します。

minof()
スカラ。式のリストから最小値を返します。

構文
```
minof ( expression [,...] )
```

パラメータ

expression	少なくとも 1 つの引数が必要で、すべての引数は同じデータ型である必要あり。

使用法
式のリストから最小値を返します。null 値は無視されます。すべての引数が null の場合、関数は null を返します。 引数のデータ型に制限はありませんが、すべての引数が同じデータ型である必要があります。関数は、引数と同じデータ型を返します。

例
`min（ 0.61, NULL, 2.34, 1.32）`は、0.61 を返します。

nextval()
スカラ。前回の呼び出しで返された値より大きい値を返します。最初の呼び出し時は、1 を返します。

構文
```
nextval()
```

使用法
関数を初めて呼び出すと 1 が返され、それ以降呼び出されるたびに、前回の呼び出しで返された値より大きい値が返されます。値の増分は 1 とはかぎりません。増分は、1 より大きいことがあります。 **nextval()** は、単一の文で複数回呼び出された場合でも、呼び出されるごとに新しい値を返します。関数は、引数を受け取らず、長整数を返します。

例

`nextval()` を初めて呼び出すと、1 が返されます。`nextval()` を 2 回目に呼び出すと、たとえば 14 が返されます。

pi()

スカラ。定数 pi の近似値を返します。

構文

```
pi()
```

使用法

定数 pi の近似値を返します。関数は引数を受け取らず、浮動小数点数を返します。

例

`pi()` は、3.141593 を返します。

power()

スカラ。指定された底を指定された指数で累乗した値を返します。

構文

```
power ( base, exponent )
```

パラメータ

base	任意の数値型。
exponent	**base** を累乗する数字を指定する浮動小数点数。

使用法

指定された底を指定された指数で累乗した値を返します。関数は、**base** 引数で数値型を受け取りますが、**exponent** は浮動小数点数である必要があります。関数は、**base** 引数と同じデータ型を返します。

例

`power（2.0, 3.0）` は、8.0 を返します。

random()

スカラ。0 以上かつ 1 未満の乱数値を返します。

構文

```
random()
```

使用法
0 以上かつ 1 未満の乱数値を返します。関数は引数を受け取らず、浮動小数点数を返します。

例
`random()` は呼び出し時に乱数 (たとえば、0.54) を返します。

round()
スカラ。指定された桁数に丸められた数値を返します。

構文
```
round ( value, digits )
```

パラメータ

value	丸める必要がある値を表す浮動小数点数。
digits	値を丸める、小数点以下の桁数。

使用法
指定された桁数に丸められた数値を返します。値は、**digits** 引数で指定された、小数点以下の桁数に丸められます。関数は、標準的な丸めルールに従います。両方の引数は浮動小数点数である必要があります。関数は、浮動小数点数を返します。

例
`round (66.778, 1)` は、66.8 を返します。

sign()
スカラ。指定された値の正負を判断します。

構文
```
sign ( value )
```

パラメータ

value	符号を持つことが可能な任意の型 (整数、浮動小数点数、長整数、間隔、通貨)。

使用法
指定された値の正負を判断します。関数は、値が正の場合は 1 を、値が負の場合は -1 を、それ以外の場合は 0 を返します。引数には、符号を持つ任意の型を指定でき、関数は整数を返します。

例

`sign ( cosd(45.0) )`は、1 を返します。

sin()

スカラ。指定された値の正弦を返します。

構文

```
sin ( value )
```

パラメータ

value	浮動小数点数。

使用法

指定された値の正弦をラジアンで返します。関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例

`sin ( pi() )`は、0 を返します。

sinh()

スカラ。指定された値の双曲線正弦を返します。

構文

```
sinh ( value )
```

パラメータ

value	浮動小数点数。

使用法

指定された値の双曲線正弦をラジアンで返します。関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例

`sinh（0.5）`は、0.521095305 を返します。

sqrt()

スカラ。指定された数値の平方根を返します。

構文

```
sqrt ( value )
```

パラメータ

value	通貨型または数値型。

使用法
指定された数値の平方根を返します。関数は、引数として数値型または通貨型を受け取り、浮動小数点数を返します。引数が無効な場合、関数は、「浮動小数点数例外」エラーを返します。

例
`sqrt （100.0）`は、10.0 を返します。

tan()

スカラ。指定された値の正接を返します。

構文

```
tan ( value )
```

パラメータ

value	浮動小数点数。

使用法
指定された値の正接をラジアンで返します。関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例
`tan （0.0）`は、0 を返します。

tand()

スカラ。指定された値の正接を度数で返します。

構文

```
tand ( value )
```

パラメータ

value	浮動小数点数。

使用法
指定された値の正接を度数で返します。関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例

`tand（45.0）`は、1.61977519 を返します。

tanh()

スカラ。指定された値の双曲線正接を返します。

構文

```
tanh ( value )
```

パラメータ

value	浮動小数点数。

使用法

指定された値の双曲線正接を返します。関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例

`tanh（0.5）`は、0.462117157 を返します。

文字列関数

文字列関数は、STRING 値と一緒に使用され、通常、STRING 値を返します。文字列関数の例には、`left（）`、`rtrim（）`、`replace（）`があります。

int32()

スカラ。指定された文字列を整数に変換します。

構文

```
int32 ( string )
```

パラメータ

string	オプションのマイナス記号から始まり、数字のみを含む文字列。

使用法

指定された文字列を整数に変換します。関数は、引数として文字列を受け取り、整数を返します。無効な文字列が指定されると、関数は null を返します。

例

`int32（'1935'）`は、1935 を返します。

left()

スカラ。指定された文字列の先頭から、指定された数だけ文字を返します。

構文

```
left ( string, count )
```

パラメータ

string	文字列。
count	返される文字数。

使用法

指定された文字列の先頭から、指定された数だけ文字を返します。関数は、**count** 引数として文字列と整数を受け取ります。関数は文字列を返します。**count** が負の整数の場合、関数は null を返します。**count** が 0 の場合、関数は空の文字列を返します。

-U サーバ・オプションが指定されていると、関数は、UTF-8 文字列を処理します。

例

`left ('examplestring', 7)` は 'example' を返します。

like()

スカラ。指定された文字列が、指定されたパターン文字列に一致するかどうかを判断します。

構文

```
like ( string, pattern )
```

パラメータ

string	文字列。
pattern	文字のパターンである文字列。ワイルドカードを指定できます。

使用法

文字列がパターン文字列に一致するかどうかを判断します。関数は、文字列がパターンに一致する場合は 1 を、そうでない場合は 0 を返します。 pattern 引数にはワイルドカードを指定できます。"_" は任意の 1 文字に一致します。"%" は 0 個以上の任意の文字に一致します。 関数は、引数として 2 つの文字列を受け取り、整数を返します。

注意： SQL では、sourceString like patternString のような infix 表記も使用できます。

例
`like ('MSFT', 'M%T')` は、1 を返します。

lower()

スカラ。指定された文字列のすべての文字を小文字にした新しい文字列を返します。

構文
```
lower ( string )
```

パラメータ

string	文字列。

使用法
指定された文字列のすべての文字を小文字にした文字列を返します。関数は、引数として文字列を受け取り、文字列を返します。

例
`upper ('This Is A Test')` は、'this is a test' を返します。

ltrim()

スカラ。文字列の左側からスペースを削除します。

構文
```
ltrim ( string )
```

パラメータ

string	文字列。

使用法
文字列の左側からスペースを削除します。関数は、引数として文字列を受け取り、文字列を返します。

例
`ltrim (' examplestring')` は、'examplestring' を返します。

patindex()

スカラ。ソース文字列内で、パターンが n 番目に出現する位置を判断します。

構文

```
patindex ( string, pattern, number [, position] [,
constant_string] )
```

パラメータ

string	ソース文字列。
pattern	検索するパターンを示す文字列。
number	検索するパターンの出現回数。
position	(オプション) 検索の開始位置 (0 から始まるインデックス)。デフォルトは 0。
constant_string	(オプション) pattern 引数をパターンではなく文字列定数として処理すべきかどうかを示すブール値。デフォルトは false です。

使用法

ソース文字列内で、パターンが n 番目に出現する位置を判断します。パターンにはワイルドカードを指定できます。 "_" は任意の 1 文字に一致します。"%" は 0 個以上の任意の文字に一致します。 パターンが文字列で検出された回数が n 未満の場合、関数は -1 を返します。

関数は、**string** 引数と **pattern** 引数で文字列を、**number** 引数と **position** 引数で整数を受け取ります。**constant_string** 引数はブール値です。関数は、指定された文字列内でパターンが n 番目に出現する位置を示す整数を返します。

number がゼロ以下の場合、関数は null を返します。**position** が 0 未満の場合、関数は、文字列の先頭から検索を開始します。position が **string** 引数の長さより大きい場合、**patindex()** は -1 を返します。

-U サーバ・オプションが指定されていると、関数は、UTF-8 文字列を処理します。

例

`patindex('longlonglongstring', 'long', 2)` は、4 を返します。

`patindex('longstring', 'long', 2)` は、-1 を返します。

`patindex('String', __n, 1)` は、2 を返します。

`patindex('String', %n, 1)` は、0 を返します。

`patindex('String', __n, 1, false)` は、2 を返します。

`patindex('String', __n, 1, true)` は、-1 を返します。

`patindex('String', S, 1, 0, false)` は、0 を返します。

`patindex('Stringi', i, 2, 2, true)` は、6 を返します。

real()

スカラ。指定された文字列を浮動小数点数に変換します。

構文

```
real ( string )
```

パラメータ

string	有効な文字列は、数字の並びで、オプションで小数点文字も指定可能。入力には、最初の文字としてマイナス記号をオプションで指定できます。また、"e" または "E" で始まり、符号と数字の並びで構成される指数部もオプションで指定できます。

使用法

指定された文字列を浮動小数点数に変換します。関数は、引数として文字列を受け取り、浮動小数点数を返します。無効な文字列が指定されると、関数は null を返します。

例

`real ('43.4745')` は、43.4745 を返します。

regexp_firstsearch()

スカラ。指定された文字列で見つかった POSIX の正規表現パターンの最初の出現箇所を返します。

構文

```
regexp_firstsearch ( string, regex )
```

パラメータ

string	文字列。
regex	POSIX の正規表現パターン。このパターンは、Perl 構文に限定されます。

使用法

指定された文字列で見つかった POSIX の正規表現パターンの最初の出現箇所を返します。文字列にパターンに一致する部分が見つからない場合、または指定した

パターンが有効な正規表現でない場合は、関数は null を返します。パターンには、1 つまたは複数の部分式をそれぞれカッコで囲んで指定できます。 文字列にパターンに一致する部分が見つかった場合、関数は、最初の部分式で指定したパターンの部分のみを返します。関数は文字列を返します。

-U サーバ・オプションが指定されていると、関数は、UTF-8 文字列を処理します。

例

`regexp_firstsearch('aaadogaaa', '[b-z]*')` は、'dog' を返します。

`regexp_firstsearch('h', '[i-z]*')` は、null を返します。

`regexp_firstsearch('aaaaabaaaabbbaaa', '[b-z]*')` は、'b' を返します。

regexp_replace()

スカラ。指定された文字列で POSIX の正規表現パターンに一致する最初の出現箇所を見つけて 2 番目に指定した文字列で置換し、これによって生成された文字列を返します。

構文

```
regexp_replace ( string, regex, replacement )
```

パラメータ

string	文字列。
regex	POSIX の正規表現パターン。このパターンは、Perl 構文に限定されます。
replacement	regex に一致する文字列部分を置換する文字列。

使用法

指定された文字列を、POSIX の正規表現パターンに一致する最初の出現箇所を 2 番目に指定した文字列で置換して返します。パターンに一致する部分が文字列で見つからない場合、関数は、置換を行わずに文字列をそのまま返します。 **regex** が有効な正規表現でない場合は、関数は null を返します。

-U サーバ・オプションが指定されていると、関数は、UTF-8 文字列を処理します。

例

`regexp_replace('aaadogaaa', '[b-z]*', 'cat')` は、'aaacataaa' を返します。

`regexp_replace('aaadogaaa', '[b-z]*', '')` は、'aaaaaa' を返します。

`regexp_replace('aaa', '[a-z]*', 'dog')` は、'dog' を返します。

`regexp_replace('aaa', '[b-z]*', 'dog')` は、'aaa' を返します。

regexp_search()

スカラ。POSIX の正規表現パターンに一致する部分が文字列にあるかどうかを判断します。

構文

```
regexp_search ( string, regex )
```

パラメータ

string	文字列。
regex	POSIX の正規表現パターン。このパターンは、Perl 構文に限定されます。

使用法

POSIX の正規表現パターンに一致する部分が文字列にあるかどうかを判断します。関数は、文字列にパターンがあるかどうかによって、ブール式 (TRUE または FALSE) を返します。

-U サーバ・オプションが指定されていると、関数は、UTF-8 文字列を処理します。

例

`regexp_search('aaadogaaa', '[b-z]*')` は、TRUE を返します。

`regexp_search('h', '[i-z]*')` は、FALSE を返します。

replace()

スカラ。最初の文字列での 2 番目の文字列のすべての出現を 3 番目の文字列で置換した新しい文字列を返します。

構文

```
replace ( target, substring, repstring )
```

パラメータ

target	文字列。
substring	置換対象の文字列。
repstring	文字を置換する文字列。

使用法
最初の文字列での 2 番目の文字列のすべての出現を 3 番目の文字列で置換した新しい文字列を返します。関数は、3 つの文字列引数を受け取り、文字列を返します。

例
`replace ('NewAmsterdam', 'New', 'Old')` は、'OldAmsterdam' を返します。

right()
スカラ。文字列の右端の文字を返します。

構文
```
right ( string, number )
```

パラメータ

string	文字列。
number	文字列から返される文字数。

使用法
文字列の右端の文字を返します。関数は、引数として文字列と整数を受け取り、文字列を返します。

例
`right ('examplestring', 6)` は、'string' を返します。

rtrim()
スカラ。文字列の右からスペースを削除します。

構文
```
rtrim ( string )
```

パラメータ

string	文字列。

使用法
文字列の右側からスペースを削除します。関数は、引数として文字列を受け取り、文字列を返します。

例

`rtrim ('examplestring ')` は、'examplestring' を返します。

string()

スカラ。指定された任意の型の値を等価の文字列に変換します。

構文

```
string ( value )
```

パラメータ

value	バイナリまたは文字列以外の任意のデータ型の引数。

使用法

指定された値を等価の文字列式に変換します。引数には、バイナリまたは文字列を除く任意のデータ型を指定できます。関数は文字列を返します。

例

`string (1935)` は、'1935' を返します。

substr()

スカラ。指定された文字列から、開始位置と文字数に基づいて作成した部分文字列を返します。

構文

```
substr ( string, position, number )
```

パラメータ

string	文字列。
position	部分文字列の開始位置。文字列の先頭文字またはスペースの位置が 0 になります。
number	部分文字列の文字数。

使用法

指定された文字列から、開始位置と文字数に基づいて作成した部分文字列を返します。最初の引数は文字列である必要があります。**position** 引数と **number** 引数は整数である必要があります。関数は文字列を返します。

例

`substr ('thissubstring', 4, 3)` は、'sub' を返します。

trim()

スカラ。指定された文字列から、後続スペースと先行スペースを削除して返します。

構文

```
trim ( string )
```

パラメータ

string	文字列。UTF-8 文字列で動作します。

使用法

指定された文字列から、後続スペースと先行スペースを削除して返します。関数は、引数として文字列を受け取り、文字列を返します。関数は、指定された文字列に **ltrim()** と **rtrim()** を適用した場合と同じ値を返します。

例

`trim (' examplestring ')` は、'examplestring' を返します。

`trim(' ')` は、'' を返します。

`trim('a')` は、'a' を返します。

trunc()

スカラ。日付の時刻部分を 00:00:00 に切り捨て、新しい日付値を返します。

構文

```
trunc ( datevalue )
```

パラメータ

datevalue	日付または bigdatetime。

使用法

日付値の時刻部分を 00:00:00 に切り捨て、新しい日付値を返します。関数は、引数として日付または bigdatetime を受け取り、引数と同じデータ型を返します。

例

`trunc (undate ('2001:05:23 12:34:64'))` は、2001:05:23 00:00:00 を返します。

upper()

スカラ。指定された文字列のすべての文字を大文字にした文字列を返します。

構文

```
upper ( string )
```

パラメータ

string	文字列。

使用法

指定された文字列のすべての文字を大文字にした文字列を返します。関数の引数は文字列です。関数は文字列を返します。

例

`upper ('This Is A Test')` は、'THIS IS A TEST' を返します。

変換関数

変換関数は、さまざまなデータ型のデータ値を関数名で指定されたデータ型に変換します。

ascii()

スカラ。特定の文字の Unicode のコード・ポイントを返します。-U サーバ・オプションが指定されている場合は、UTF-8 のコード・ポイントを返します。

構文

```
ascii ( character )
```

パラメータ

character	文字列。

使用法

空または null の場合は、関数は null を返します。 そうでない場合は、関数はコード・ポイントを整数として返します。

例

`ascii ('D')` は、68 を返します。

最初の文字だけが変換されるため、`ascii ('Dog')` も 68 を返します。

base64_binary()

スカラ。base64 でエンコードされた、指定された文字列のバイナリ値を返します。

構文

```
base64_binary ( string )
```

パラメータ

string	base64 でエンコードされた文字列。有効な文字は、a ～ z、A ～ Z、0 ～ 9、/、+ です。

使用法

関数は、base64 でエンコードされた文字列をバイナリ型に変換します。文字列長は、4 で割ったときに 1 が余らないようにしてください。 4 で割ったときに 1 が余ると、エンコードが無効になります。場合によっては、文字列長が 4 で割り切れるようにするために、埋め込み文字 '=' を 1 つまたは 2 つ使用してください。

例

`base64_binary ( 'bGVhc3VyZS4=' )` は、6C6561737572652E を返します。

`base64_binary ( 'ZQ==' )` は、65 を返します。

base64_string()

スカラ。指定されたバイナリ値の、base64 でエンコードされた文字列を返します。

構文

```
base64_string ( binary )
```

パラメータ

binary	バイナリ値。

使用法

関数は、バイナリ値をエンコードして、base64 でエンコードされた文字列を生成します。文字列長が 4 で割り切れるようにするために、埋め込み文字 '=' を 1 つまたは 2 つ使用します。関数は文字列を返します。

例

`base64_string ( hex_binary ('64') )` は、ZQ== を返します。

`base64_string ( hex_binary ('6C6561737572652E') )` は、bGVhc3VyZS4= を返します。

cast()

スカラ。あるデータ型の値を別のデータ型に変換し、オーバフローやトランケーションを可能にします。

構文

```
cast ( type, number )
```

パラメータ

type	バイナリと文字列を除く任意のデータ型。
number	指定された新しいデータ型にキャスト可能なデータ型。

使用法

type 引数は、数値型、通貨型、または日付／時刻型である必要があります。バイナリ型または文字列型を除く任意の型の式をキャストできます。

大きな型から小さな型にキャストすると、オーバフローが発生することがあります。 小数点を持つ型 (浮動小数点数または通貨など) から小数点を持たない型 (整数など) にキャストすると、小数点以下が切り捨てられます。 オーバフローもトランケーションも可能です。整数から長整数へ変換する場合など、暗黙的なキャストが許可されない場合に、この関数を使用して強制的にキャストします。

精度が異なる値を比較するとき、片方を別の方の型にキャストすると、2 つの値が互換性を持つようになります。たとえば、精度が異なる通貨値は、共通の型にキャストしないと比較できません。

精度が異なる通貨値をキャストする方法は、2 つの値を比較する方法によって異なります。

- money(2) 型の 100.55D2 が money(3) 型の 100.545D3 より大きいとする比較を行うと、これらの値は内部では小数点のない値として表されるので、結果は false。なぜなら、10055 は 100545 より大きいとはならないからです。この例では、いずれかの値をキャストすると、true の結果が得られます。10055 を 100545 にキャストすると、比較する式は 100550>100545 となるので、true です。100545 を 10055 にキャストすると、比較する式は 10055>10054 となるので、これも true です。
- 100.55D2 が 100.556D3 に等しいとして比較すると、結果は false。この例では、どちらの値をキャストするかによって結果が変わります。10055 を 100556 にキャストすると、比較する式は 100550=100556 となるので、false です。100556 を 10055 にキャストすると、比較する式は 10055=10055 となるので、true です。

誤った比較結果を避け、精度を維持するには、精度の低い値を精度の高い値にキャストすることをおすすめします。

例

`cast ( integer, 1.23)` は、1 を返します。

char()

スカラ。1 つまたは複数の Unicode のコード・ポイントに対応する文字を返します。-U サーバ・オプションが指定されている場合は、UTF-8 のコード・ポイントを返します。

構文

```
char ( expression [,...] )
```

パラメータ

expression	1 つまたは複数の Unicode のコード・ポイント。引数は整数である必要があります。

使用法

無効なコード・ポイント、0、または null を指定すると、null が返されます。関数は文字列を返します。

例

`char (68)` は 'D' を返します。

`char(68, 68, 68)` は 'DDD' を返します。

concat()

スカラ。指定された 2 つのバイナリ値を連結して生成されたバイナリ値を返します。

構文

```
concat ( binary1, binary2 )
```

パラメータ

binary1	バイナリ値
binary2	バイナリ値

使用法

関数はバイナリ値を返します。いずれかの引数が null の場合、関数は null を返します。

例

`concat ( hex_binary ('aabbcc'), hex_binary ('ddeeff') )` は、AABBCCDDEEFF を返します。

`concat ( hex_binary ('ddeeff'), hex_binary ('aabbcc') )` は、DDEEFFAABBCC を返します。

extract()

スカラ。指定されたバイナリ値の一部を抽出して返します。

構文

```
extract ( binary, startByte, numberOfBytes )
```

パラメータ

binary	バイナリ値。
startByte	抽出の開始位置を表す整数。
numberOfBytes	抽出の長さを表す整数。

使用法

バイナリ値を、**startByte** 引数で表される位置から指定された長さだけ抽出します。関数は、引数として、バイナリ値と 2 つの整数 (**startByte** と **numberOfBytes**) を受け取り、バイナリ値を返します。

たとえば、バイナリ値がバイト abcde で構成されている場合、`extract( bytes, 2,3 )` は cde を生成します。長さがバイナリ値の末尾を越える場合、バイナリ値の残りが返されます。 前の例では、`extract( bytes,2,4 )` は cde を返します。

例

`extract ( hex_binary ('a1b2c3e4'),1,2)` は、B2C3 を返します。

`extract ( hex_binary ('a1b2c3e4'),3,1)` は、E4 を返します。

`extract ( hex_binary ('a1b2c3e4'),0,4)` は、A1B2C3E4 を返します。

fromnetbinary()

スカラ。ネットワーク・バイト順序のバイナリを、ホスト・バイト順序の整数に変換します。

構文

```
fromnetbinary ( binary )
```

パラメータ

binary	ネットワーク・バイト順序のバイナリ。

使用法
ネットワーク・バイト順序のバイナリを受け取って、ホスト・バイト順序の整数に変換します。正または負の値に対して動作します。関数はバイナリ値を引数として受け取り、整数を返します。バイナリ値が 4 バイト長を超える場合、関数はエラーを返します。

例
`fromnetbinary（FFFFFFF6）`は、-10 を返します。

`fromnetbinary（0012ADE4）` は、1224164 を返します。

hex_binary()

スカラ。16 進文字列をバイナリ型に変換します。

構文

```
hex_binary ( string )
```

パラメータ

string	16 進文字列。先頭に "0x" または "0X" があってもなくてもかまいません。

使用法
16 進文字列を受け取って、バイナリ型に変換します。16 進文字列で有効な文字は、a ～ f、A ～ F、0 ～ 9 です。文字列には、偶数個の文字がある必要があります。関数は文字列を引数として受け取り、バイナリ値を返します。

例
`hex_binary（'0xAA1B223F'）`は AA1B223F を返します。

`hex_binary（'0xaa'）`は AA を返します。

hex_string()

スカラ。バイナリ値を 16 進文字列に変換します。

構文

```
hex_string ( binary )
```

パラメータ

binary	バイナリ値。

使用法
バイナリ値を 16 進文字列に変換します。関数はバイナリ値を引数として受け取り、16 進文字列を表す文字列をすべて大文字で、先頭に "0x" を付加せずに返します。

例
`hex_string ( hex_binary ('0xaa') )` は AA を返します。

`hex_string ( hex_binary ('0xaa1234') )` は AA1234 を返します。

msecToTime()
スカラ。指定されたミリ秒数を bigdatetime 値に変換します。

構文
```
msecToTime ( milliseconds )
```

パラメータ

milliseconds	Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのミリ秒数を表す長整数。

使用法
指定されたミリ秒数を bigdatetime 値に変換します。関数は、引数として長整数を受け取り、bigdatetime 値を返します。

例
`msecToTime (3661001)` は 1970-01-01 01:01:01.001 を返します。

secToTime()
スカラ。指定された秒数を bigdatetime 値に変換します。

構文
```
secToTime ( seconds )
```

パラメータ

seconds	Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からの秒数を表す長整数。

使用法
指定された秒数を bigdatetime 値に変換します。関数は、引数として長整数を受け取り、bigdatetime 値を返します。

例
`secToTime（3661）`は 1970-01-01 01:01:010.000000 を返します。

timeToMsec()

スカラ。bigdatetime 値を Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのミリ秒数に変換します。

構文
```
timeToMsec ( time )
```

パラメータ

time	bigdatetime。

使用法
bigdatetime 値を Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのミリ秒数に変換します。この関数は、bigdatetime を引数として受け取り、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのミリ秒数を表す長整数を返します。bigdatetime のマイクロ秒の部分は切り捨てられます。

例
`timeToMsec ( unbigdatetime('1970-01-01 01:01:01:002100')）`は、3661002 を返します。

timeToUsec()

スカラ。bigdatetime 値を Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数に変換します。

構文
```
timeToUsec ( time )
```

パラメータ

time	bigdatetime。

使用法
bigdatetime 値を Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数に変換します。この関数は、bigdatetime を引数として受け取り、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表す長整数を返します。

例
`timeToUsec ( unbigdatetime ('1970-01-01 01:01:01.000001'))` は、3661000001 を返します。

timeToSec()

スカラ。bigdatetime 値を Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からの秒数に変換します。

構文
```
timeToSec ( time )
```

パラメータ

time	bigdatetime。

使用法
bigdatetime 値を Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からの秒数に変換します。この関数は、bigdatetime を引数として受け取り、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からの秒数を表す長整数を返します。bigdatetime のミリ秒またはマイクロ秒の部分は切り捨てられます。

例
`timeToSec ( unbigdatetime('1970-01-01 01:01:01:000000') )` は、3661 を返します。

to_binary()

スカラ。指定された値をバイナリ値に変換します。

構文
```
to_binary ( value )
```

パラメータ

value	キャスト元の文字列またはバイナリの型の値。

使用法
指定された文字列をバイナリ値に変換します。関数は文字列を引数として受け取り、バイナリ値を返します。この関数は引数としてバイナリ値も受け取れますが、この場合、同じバイナリ値を返します。

例
`to_binary('0123456789abcdef')` は、0x30313233343536373839616263646566 と等価のバイナリ値を返します。

to_binary('Hello there!') は、0x48656c6c6f20746865726521 と等価のバイナリ値を返します。

to_string(to_binary('Good morning.')) は、文字列 'Good morning.' をバイナリ値にキャストし、さらに文字列のデータ型に戻して文字列 'Good morning.' を返します。

to_bigdatetime()

スカラ。指定された値を bigdatetime に変換します。

構文

```
to_bigdatetime ( value )
to_bigdatetime ( value, format )
```

パラメータ

value	文字列、浮動小数点数、長整数、または bigdatetime。文字列は、format 引数で指定されたフォーマットに一致している必要があります。数値は、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表します。
format	フォーマット文字列。値が文字列の場合のみ、有効です。bigdatetime のフォーマット・コードの 1 つである必要があります。詳細については、「日付／時刻のフォーマット・コード」を参照してください。

使用法
指定された値を bigdatetime に変換します。関数は、引数として浮動小数点数、長整数、または文字列 (と関連するフォーマット文字列) を受け取り、bigdatetime を返します。この関数は引数として bigdatetime も受け取れますが、この場合、同じ bigdatetime を返します。

例
`to_bigdatetime(3600000000)` は 1970-01-01 01:00:00.000000 を返します。

`to_bigdatetime('02/19/2010 10:15', '%m/%d/%Y %H:%M')` は 2010-02-19 10:15:00.000000 を返します。

`to_bigdatetime('07/19/2010 10:15 -07.00', 'MM/DD/YYYY HH:MI TZH:TZM')` は 2010-07-19 03:15:00.000000 を返します。

to_boolean()

スカラ。指定された値をブール値に変換します。

構文

```
to_boolean ( value )
```

パラメータ

value	文字列またはブール値。

使用法

指定された文字列をブール値に変換します。関数は文字列を引数として受け取り、ブール値を返します。この関数は引数としてブール値も受け取れますが、この場合、同じブール値を返します。

文字列 "True"、"Yes"、"On" (大文字、小文字は関係ありません)、または数値の "1" が引数として指定されると、TRUE を返します。null が引数として指定されると、null を返します。他のすべての文字列が引数として指定されると、FALSE を返します。

例

`to_boolean ('1')` は、TRUE を返します。

`to_boolean ('FALSE')` は、FALSE を返します。

`to_boolean ('example')` は、FALSE を返します。

to_date()

スカラ。指定された値を日付に変換します。

構文

```
to_date ( value )
to_date ( value, format )
```

パラメータ

value	文字列、浮動小数点数、長整数、または日付。文字列は、format 引数で指定されたフォーマットに一致している必要があります。数値は、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表します。

format	フォーマット文字列。値が文字列の場合のみ、有効です。日付のフォーマット・コードの 1 つである必要があります。詳細については、「日付／時刻のフォーマット・コード」を参照してください。

使用法
指定された値を日付に変換します。関数は、引数として浮動小数点数、長整数、または文字列 (と関連するフォーマット文字列) を受け取り、日付を返します。この関数は引数として日付も受け取れますが、この場合、同じ日付を返します。

例
`to_date('02/19/2010 10:15', '%m/%d/%Y %H:%M')` は 2010-02-19 10:15:00 を返します。

`to_date('07/19/2010 10:15 -07.00', 'MM/DD/YYYY HH:MI TZH:TZM')` は 2010-07-19 03:15:00 を返します。

to_float()

スカラ。指定された値を浮動小数点数に変換します。

構文
```
to_float ( value )
```

パラメータ

value	文字列、間隔、日付／時刻型、数値型、または通貨型。

使用法
指定された値を浮動小数点数に変換します。この関数は、文字列、間隔、日付／時刻型、数値型、または通貨型を引数として受け取り、浮動小数点数を返します。この関数は引数として浮動小数点数も受け取れますが、この場合、同じ浮動小数点数値を返します。

文字列は、浮動小数点数リテラルのフォーマットに基づいて変換されます。引数が間隔の場合、マイクロ秒単位の数を表す値を返します。引数が日付／時刻型の場合、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表す値を返します。Epoch 時間よりも過去の日付／時刻型は、負の値に変換されます。

例
`to_float ('100.0')` は、100.0 を返します。

to_integer()

スカラ。指定された値を整数に変換します。

構文

```
to_integer ( value )
```

パラメータ

value	整数にキャストするブール、通貨、文字列、日付、または任意の数値の型の値。

使用法

指定された値を整数に変換します。関数は、引数として文字列、日付、または任意の数値の型を受け取り、整数を返します。この関数は引数として整数も受け取れますが、この場合、同じ整数を返します。

引数として数値が指定されると、その値の整数部分を返します。整数として有効な範囲以外の値、または文字列値に数値以外の文字が含まれている場合は、null を返します。 引数が日付型の場合、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からの秒数を表す値を返します。Epoch 時間よりも過去の時間は、負の値に変換されます。

例

`to_integer ('1')` は、1 を返します。

to_interval()

スカラ。指定された値を間隔に変換します。

構文

```
to_interval ( value )
```

パラメータ

value	マイクロ秒単位の数を表す文字列、長整数、浮動小数点数、間隔。文字列は、間隔リテラルのフォーマットに従う必要があります。

使用法

指定された値を間隔に変換します。関数は、引数として文字列、長整数、浮動小数点数を受け取り、間隔を返します。この関数は引数として間隔も受け取れますが、この場合、同じ間隔を返します。

例

`to_interval( '1234' )` は、1234 を返します。

to_long()

スカラ。指定された値を長整数に変換します。

構文

```
to_long ( value )
```

パラメータ

value	文字列、間隔、日付／時刻型、数値型、または通貨型。

使用法

指定された文字列を長整数に変換します。この関数は、文字列、間隔、日付／時刻型、数値型、または通貨型を引数として受け取り、長整数を返します。この関数は引数として長整数も受け取れますが、この場合、同じ長整数を返します。

引数として数値型が指定されると、その値の整数部分を返します。数値以外の文字を含む文字列、または長整数の有効な範囲以外の値を含む文字列が指定されると、null を返します。間隔が指定されると、マイクロ秒単位の数を返します。引数が日付／時刻型の場合、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表す値を返します。Epoch 時間よりも過去の時間は、負の値に変換されます。

例

`to_long ('23')` は、23 を返します。

to_money()

スカラ。指定された値を、指定された精度で、適切な通貨型に変換します。

構文

```
to_money ( value, scale )
```

パラメータ

value	文字列または数値型。文字列は、数字と小数点のみを含むことができます。
scale	1 ～ 15 の整数。

使用法

指定された値を、指定された精度で、通貨型に変換します。関数は、引数として文字列、または数値型を受け取り、通貨を返します。

例

`to_money (12.361, 2)` は、12.36 を返します。

to_xml()

スカラ。指定された値を XML に変換します。

構文

```
to_xml ( value )
```

パラメータ

value	文字列、または XML 型オブジェクト。

使用法

指定された値を XML に変換します。関数は、引数として文字列を受け取り、文字列を返します。引数として XML 型オブジェクトも受け取れますが、この場合、同じオブジェクトを返します。この関数は **xmlparse()** と同じですが、XML 入力も処理できます。

例

`xmlserialize ( to_xml ('<t/>') )` は、'<t/>' を返します。この文字列は、XML に変換され、次に文字列に戻されます。

tonetbinary()

スカラ。ホスト・バイト順序の整数を、ネットワーク・バイト順序の 4 バイトのバイナリに変換します。

構文

```
tonetbinary ( integer )
```

パラメータ

integer	ホスト・バイト順序の整数。

使用法

ホスト・バイト順序の整数を受け取って、ネットワーク・バイト順序の 4 バイトのバイナリに変換します。正または負の値に対して動作します。

例

`tonetbinary (1224164)` は、0012ADE4 を返します。

`tonetbinary (-1224164)` は、FFED521C を返します。

to_string()

スカラ。指定された値を文字列に変換します。

構文

```
to_string ( value [, format] [, timezone] )
```

パラメータ

value	任意のデータ型の値。
format	(オプション) フォーマット文字列。値が日付／時刻または数値の型の場合のみ有効です。
timezone	(オプション) タイム・ゾーン。値が日付／時刻型の場合のみ有効です。何も指定されていない場合は、UTC タイム・ゾーンが使用されます。

使用法

指定された値を文字列に変換します。引数のデータ型に制限はありません。関数は、文字列を返します。この関数は引数として文字列も受け取れますが、この場合、同じ文字列を返します。この関数は値を次のように変換します。

- 整数または長整数の場合、出力文字列のフォーマットを指定するために、オプションでフォーマット文字列を指定可能。フォーマット文字列は、fprintf の ISO 標準に従います。整数式のデフォルトは '%d' で、長整数式のデフォルトは '%lld' です。
- 日付／時刻型の場合、出力文字列のフォーマットを指定可能。文字列は、有効なタイムスタンプ・フォーマット・コードである必要があります。
- オプションのタイム・ゾーン引数は、日付／時刻型の場合のみ使用可能。この文字列は、有効なタイム・ゾーン文字列である必要があります。タイム・ゾーンが指定されていない場合、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。
- XML 値を文字列に変換する場合は、**xmlserialize()** と同じ動作。
- バイナリ値の場合、返される値に印刷不可文字が含まれていることがある。これは、関数が、変換ではなく、バイナリから文字列への単純なキャストを実行するために発生します。バイナリ値の 16 進文字列表現を変換するには、hex_string() 関数を使用します。

浮動小数点数値の場合、文字列としての浮動小数点数の出力フォーマットを指定するオプションのフォーマット文字列を指定できます。フォーマット文字列には、以下の文字列を指定できます。

. または D	指定した位置に小数点を配置して返す。1 つの小数点のみ指定できます。指定されていない場合、出力には値の代わりにシャープ記号が含まれます。

9	出力で、値の 1 つの数字に置き換わる。フォーマット文字列に指定されている 9 の数の桁数で、値が返されます。 値が正の場合、先行するスペースが、値の左側に付加されます。値が負の場合、先行するマイナス記号が、値の左側に付加されます。 整数部分よりも多く指定されている 9 はスペースに置き換えられ、小数部分よりも多く指定されている 9 はゼロで置き換えられます。整数部分よりも 9 の数が少ない場合、値はシャープ記号に置き換えられ、小数部分よりも 9 の数が少ない場合、丸めが発生します。
0	整数部分では、値の単一の桁またはゼロ (ゼロの位置に数字がない場合) で置き換わる。小数部分では、9 として扱われます。 値が正の場合、先行するスペースが値の左側に付加されます。値が負の場合、先行するマイナス記号が、値の左側に付加されます。
EEEE	科学的記数法で値を返す。このフォーマットの出力では、常に、小数点の前に 1 桁の数値があります。小数点と 9 を組み合わせて、精度を指定します。小数点の左側の 9 は無視されます。 フォーマット文字列の最後に配置する必要があります。
S	値の正負に応じて、先行または後続のマイナス記号 (-) またはプラス記号 (+) を返す。フォーマット文字列の先頭または最後にのみ指定できます。 通常の単一の先行するスペースを削除しますが、9、ゼロ、またはカンマが多すぎる場合の先行するスペースは削除されません。
$	先行するドル記号を値の前に返す。フォーマット文字列の任意の場所に配置できます。
.	指定した位置にカンマを返す。カンマの左側に数字がない場合、カンマはスペースに置き換えられます。 複数のカンマを指定できますが、フォーマットの最初の文字として、または小数点の右側には、カンマを指定できません。
FM	出力からスペースを削除する。

例

`to_string (45642)` は、'45642' を返します。

`to_string (1234.567,'999')` は、'####' を返します。

`to_string (1234.567,'9999D999')` は、'1234.567' を返します。

`to_string (1234.567,'.99999999EEEE')` は、'1.23456700E+03' を返します。

to_timestamp()

スカラ。指定された値をタイムスタンプに変換します。

構文

```
to_timestamp ( value )
to_timestamp ( value, format )
```

パラメータ

value	文字列、浮動小数点数、または長整数。文字列は、format 引数で指定されたフォーマットに一致している必要があります。数値は、Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表します。
format	フォーマット文字列。値が文字列の場合のみ、有効です。タイムスタンプのフォーマット・コードの 1 つである必要があります。詳細については、「日付／時刻のフォーマット・コード」を参照してください。

使用法

指定された値をタイムスタンプに変換します。関数は、引数として浮動小数点数、長整数、または文字列 (と関連するフォーマット文字列) を受け取り、タイムスタンプを返します。この関数は引数としてタイムスタンプも受け取れますが、この場合、同じタイムスタンプを返します。

例

`to_timestamp('02/19/2010 10:15', '%m/%d/%Y %H:%M')` は 2010-02-19 10:15:00.000 を返します。

`to_timestamp('07/19/2010 10:15 -07.00', 'MM/DD/YYYY HH:MI TZH:TZM')` は 2010-07-19 03:15:00.000 を返します。

usecToTime()

スカラ。指定されたマイクロ秒数を bigdatetime 値に変換します。

構文

```
usecToTime ( microseconds )
```

パラメータ

microseconds	Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数を表す長整数。

使用法
指定されたマイクロ秒数を bigdatetime 値に変換します。関数は、引数として長整数を受け取り、bigdatetime 値を返します。

例
`usecToTime（3661000001）`は 1970-01-01 01:01:01.000001 を返します。

XML 関数

XML データを正しく処理するために設計された特別なスカラ関数があります。

xmlconcat()

スカラ。複数の XML 値を単一の値に連結します。

構文

```
xmlconcat ( value, value [,value ...] )
```

パラメータ

value	XML 値。

使用法
複数の XML 値を単一の値に連結します。この関数は、2 つ以上の XML 値を受け取り、1 つの XML 値を返します。

例

```
xmlconcat ( xmlparse(stringCol), xmlparse('<t/>')))
```

xmlelement()

スカラ。属性と XML 式を含む、新しい XML データ要素を作成します。

構文

```
xmlelement ( name, [xmlattributes (string AS name, ..., string AS
name),]
[ XML value, ..., XML value] )
```

パラメータ

string	属性の名前と値のペア。例： 'attrValue' AS attrName は、結果の XML 要素内に作成される attrName = "attrValue" 属性となります。
name	新しい要素の名前。命名規則に準拠する必要があります。
XML value	子要素を表す XML 値。

使用法
属性と XML 式を含む、新しい XML データ要素を作成します。この関数は、属性の名前と値を引数として受け取り、XML 値を返します。

例
`xmlelement (top, xmlattributes('data' as attr1), xmlparse('<t/>'))` は、top という名前の新しい XML 要素を返します。この XML 要素には、'data' 属性と <t/> 子要素が含まれます。

xmlparse()

スカラ。文字列を XML 値に変換します。

構文
```
xmlparse ( string )
```

パラメータ

value	文字列として表される XML 値。

使用法
文字列を XML 値に変換します。この関数は引数として文字列を受け取り、XML 値を返します。XML のデータ型はないので、この関数から返される値は、入力として XML 値を想定する他の関数 (**xmlserialize()** など) への入力としてのみ使用できます。

例
`xmlserialize ( xmlparse ('<t/>') )` は、'<t/>' を返します。この文字列は、XML 値に変換され、次に文字列に戻されます。

xmlserialize()

スカラ。XML 値を文字列に変換します。

構文
```
xmlserialize ( value )
```

パラメータ

value	XML 値。

使用法
XML 値を文字列に変換します。この関数は引数として XML 値を受け取り、文字列を返します。

例

`xmlserialize ( xmlparse ('<t/>') )` は、'<t/>' を返します。この文字列は、XML 値に変換され、次に文字列に戻されます。

日付と時刻の関数

日付と時刻の関数は、タイム・ゾーン・パラメータと日付フォーマット・コード設定を設定し、カレンダを定義します。

business()

スカラ。指定したオフセットに基づいて、日付値から翌営業日を判断します。

構文

```
business ( calendarfile, datevalue, offset )
```

パラメータ

calendarfile	カレンダ・ファイルのファイル・パスを表す文字列。
datevalue	日付／時刻型。
offset	負または正の整数 (0 にしないでください)。

使用法

関数は、**datevalue** 引数と同じデータ型を返します。

offset 引数は任意の負または正の整数とすることができますが、ゼロにすることはできません。 offset がゼロの場合、関数は null を返し、エラー・メッセージが記録されます。負の整数の場合は、前の営業日が返されます。

例

`business('/aleri/cals/us.cal',v.TradeTime, 1)` は、カレンダである us.cal 内の、TradeTime (取引時刻) 日より後の翌営業日を返します。

businessday()

スカラ。日付値が営業日 (週末でも休日でもない日) に該当するかどうかを判断します。

注意： この関数の名前は、大文字、小文字が区別されません。Event Stream Processor は **businessday()** と **businessDay()** を同じ関数と見なします。

構文

```
businessday ( calendarfile, datevalue )
```

パラメータ

calendar	カレンダ・ファイルのファイル・パスを表す文字列
datevalue	日付／時刻型

使用法
関数は、日付が営業日に該当する場合は 1 (true) を、そうでない場合は 0 (false) を返します。関数は整数を返します。

例
`businessDay('/aleri/cals/us.cal',v.TradeTime)` は、v.TradeTime の日付部分が営業日に該当する場合は 1 を、そうでない場合は 0 を返します。

date()

スカラ。日付値を、YYYYMMDD という形式の整数に変換します。

構文

```
date ( datevalue )
```

パラメータ

datevalue	日付

使用法
関数は整数を返します。

例
`date (undate ('1991-04-01 12:43:32'))` は、19910401 を返します。

dateceiling()

スカラ。指定された date-time、multiple、date_part の引数に基づいて新しい date-time を計算し、date_part より下位の部分をゼロに設定します。次に、結果の date_part 部分が、入力したタイムスタンプより大きいか等しい、multiple の最小倍数に切り上げられます。

構文

```
dateceiling ( date_part, expression [, multiple] )
```

パラメータ

date_part	希望する粒度を識別するキーワード。有効なキーワードについては、後述します。

expression	評価する値を含む date-time 式。
multiple	倍数演算で使用される date_part の値。指定する場合は、ゼロ以外の正の整数値である必要があります。指定しない場合または null の場合、値は 1 と想定されます。

有効な date_part のキーワードと multiple

キーワード	キーワードの意味	**multiple**
`yy` または `year`	年	任意の正の整数
`qq` または `quarter`	四半期	任意の正の整数
`mm` または `month`	月	任意の正の整数
`wk` または `week`	週	任意の正の整数
`dd` または `day`	日	任意の正の整数
`hh` または `hour`	時間	1、2、3、4、6、8、12、24
`mi` または `minute`	分	1、2、3、4、5、6、10、12、15、20、30、60
`ss` または `second`	秒	1、2、3、4、5、6、10、12、15、20、30、60
`ms` または `millisecond`	ミリ秒	1、2、4、5、8、10、20、25、40、50、100、125、200、250、500、1000

使用法

この関数は、date_part 部分が multiple の倍数で、入力したタイムスタンプよりも大きいか等しい最小のタイムスタンプ値を決定し、date_part より粒度の小さい date_part 部分をすべてゼロに設定します。

date_part はキーワードです。expression は、日付／時刻 (またはタイムスタンプ) データ型に評価されるか、またはこのデータ型に暗黙的に変換できる式です。multiple は、date_part の数値で、この倍数が上限操作で使用されます。 たとえば、10 分間隔に基づいて日付の上限を求めるには、date_part に MINUTE または MI を使用し、multiple として 10 を使用します。

既知のエラー：

- 必須引数の値が null と評価されると、サーバは `invalid argument` エラーを生成する。
- multiple 引数の値が指定された date_part 引数の有効な範囲内にない場合、サーバは `invalid argument` エラーを生成する。たとえば、date_part mi が指定されている場合は、multiple の値を 60 未満にする必要があります。

標準と互換性
Sybase 拡張。

例

```
dateceiling( 'MINUTE', to_timestamp('2010-05-04T12:00:01.123',
'YYYY-MM-DDTHH24:MI:SS.FF'))
returns '2010-05-04 12:01:00.000'
```

datefloor()

スカラ。指定された date-time、multiple、date_part の引数に基づいて新しい date-time を計算し、date_part より下位の部分をゼロに設定します。結果の date_part 部分が、入力したタイムスタンプより小さいか等しい、multiple の最大倍数に切り上げられます。

構文

```
datefloor ( date_part, expression [, multiple] )
```

パラメータ

date_part	希望する粒度を識別するキーワード。有効なキーワードについては、後述します。
expression	評価する値を含む date-time 式。
multiple	倍数演算で使用される date_part の値。指定する場合は、ゼロ以外の正の整数値である必要があります。指定しない場合または null の場合、値は 1 と想定されます。

有効な date_part のキーワードと multiple

キーワード	キーワードの意味	multiple
`yy` または `year`	年	任意の正の整数
`qq` または `quarter`	四半期	任意の正の整数
mm または `month`	月	任意の正の整数

キーワード	キーワードの意味	multiple
wk または week	週	任意の正の整数
dd または day	日	任意の正の整数
hh または hour	時間	1、2、3、4、6、8、12、24
mi または minute	分	1、2、3、4、5、6、10、12、15、20、30、60
ss または second	秒	1、2、3、4、5、6、10、12、15、20、30、60
ms または millisecond	ミリ秒	1、2、4、5、8、10、20、25、40、50、100、125、200、250、500、1000

使用法

この関数は、date_part で指定される粒度より小さい日時値部分をすべてゼロに設定します。 date_part はキーワードです。expression は、日付／時刻 (またはタイムスタンプ) データ型に評価されるか、またはこのデータ型に暗黙的に変換できる式です。multiple は、date_part の数値で、この倍数が下限操作で使用されます。たとえば、10 分間隔に基づいて日付の下限を求めるには、date_part に MINUTE または MI を使用し、multiple として 10 を使用します。

既知のエラー：

- 必須引数の値が null と評価されると、サーバは「invalid argument」エラーを生成する。
- multiple 引数の値が指定された datepart 引数の有効な範囲内にない場合、サーバは「invalid argument」エラーを生成する。たとえば、date_part mi が指定されている場合は、multiple の値を 60 未満にする必要があります。

標準と互換性

Sybase 拡張。

例

```
datefloor( 'MINUTE', to_timestamp('2010-05-04T12:00:01.123', 'YYYY-
MM-DDTHH24:MI:SS.FF'))
returns '2010-05-04 12:00:00.000'
```

datename()

スカラ。日付値を文字列に変換します。

構文

```
datename ( datevalue )
```

パラメータ

datevalue	日付または bigdatetime。

使用法

日付値を、'YYYY-MM-DD' の形式の文字列に変換します。関数は、引数として日付または bigdatetime を受け取り、文字列を返します。

例

`datename (undate ('2010-03-03 12:34:34'))` は、'20100303' を返します。

datepart()

スカラ。日付の一部を表す整数を返します。

構文

```
datepart ( portion, datevalue )
```

パラメータ

portion	以下の文字列のいずれか。 • 文字列が yy または yyyy の場合は、年。 • 文字列が mm または m の場合は、月。 • 文字列が dy または y の場合は、年内の日。 • 文字列が dd または d の場合は、月内の日。 • 文字列が dw の場合は、曜日。 • 文字列が hh の場合は、時。 • 文字列が mi または n の場合は、分。 • 文字列が ss または s の場合は、秒。
datevalue	日付または bigdatetime。

使用法

日付の一部を表す整数を返します。関数が返すことのできる部分は、年、月、年内の日、月内の日、曜日、時、分、秒です。関数は、**portion** 引数として文字列

を、**datevalue** 引数として日付または bigdatetime を受け取ります。関数は整数を返します。

例

`datepart ( 'ss', undate ('2010-03-03 12:34:34'))` は、34 を返します。

dateround()

スカラ。指定された date-time、multiple、date_part の引数に基づいて新しい date-time を計算し、date_part より下位の部分をゼロに設定します。次に、結果の date_part 部分が、入力したタイムスタンプに最も近い、multiple の倍数に丸められます。

構文

```
dateround ( date_part, expression [, multiple] )
```

パラメータ

date_part	希望する粒度を識別するキーワード。有効なキーワードについては、後述します。
expression	評価する値を含む date-time 式。
multiple	倍数演算で使用される date_part の値。指定する場合は、ゼロ以外の正の整数値である必要があります。指定しない場合または null の場合、値は 1 と想定されます。

有効な date_part のキーワードと multiple

キーワード	キーワードの意味	multiple
`yy` または `year`	年	任意の正の整数
`qq` または `quarter`	四半期	任意の正の整数
`mm` または `month`	月	任意の正の整数
`wk` または `week`	週	任意の正の整数
`dd` または `day`	日	任意の正の整数
`hh` または `hour`	時間	1、2、3、4、6、8、12、24
`mi` または `minute`	分	1、2、3、4、5、6、10、12、15、20、30、60

キーワード	キーワードの意味	multiple
ss または second	秒	1、2、3、4、5、6、10、12、15、20、30、60
ms または millisecond	ミリ秒	1、2、4、5、8、10、20、25、40、50、100、125、200、250、500、1000

使用法

この関数は、入力したタイムスタンプ値について、date_part 部分を最も近い値に、または date_part 部分を最も近い multiple の倍数に丸め、date_part またはその倍数より粒度の小さい date_part 部分をすべてゼロに設定します。たとえば、最も近い時間 (正時) に丸める場合、分の部分が判定され、分の部分が 30 以上の場合、時間の部分は 1 増分されます。分と日付のその他の下位の部分はゼロになります。

date_part はキーワードです。expression は、日付／時刻 (またはタイムスタンプ) データ型に評価されるか、またはこのデータ型に暗黙的に変換できる式です。multiple は、date_part の数値で、この倍数が丸め操作で使用されます。たとえば、最も近い 10 分の間隔に丸めるには、date_part に MINUTE または MI を使用し、multiple として 10 を使用します。

既知のエラー：

- 必須引数の値が null と評価されると、サーバは「invalid argument」エラーを生成する。
- multiple 引数の値が指定された datepart 引数の有効な範囲内にない場合、サーバは「invalid argument」エラーを生成する。たとえば、date_part mi が指定されている場合は、multiple の値を 60 未満にする必要があります。

例

```
dateround( 'MINUTE', to_timestamp('2010-05-04T12:00:01.123', 'YYYY-MM-DDTHH24:MI:SS.FF'))
returns '2010-05-04 12:00:00.000'
```

dayofmonth()

スカラ。指定された bigdatetime 値から抽出された月内の日を表す整数を返します。

構文

```
dayofmonth ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) 有効なタイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法
指定された bigdatetime 値から抽出された月内の日を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例
`dayofmonth ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、3 を返します。

dayofweek()

スカラ。指定された bigdatetime 値から抽出された曜日を表す整数 (日曜日が 1) を返します。

構文

```
dayofweek ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) 有効なタイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法
指定された bigdatetime 値から抽出された曜日を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。日曜日が 1 で表され、以降の曜日は 1 ずつ加算された整数で表されます。

例
`dayofweek ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、4 を返します。

dayofyear()

スカラ。指定された bigdatetime 値から抽出された年内の日を表す整数を返します。

構文

```
dayofyear ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) 有効なタイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法

指定された bigdatetime 値から抽出された年内の日を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例

`dayofyear ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、62 を返します。

hour()

スカラ。指定された bigdatetime 値から抽出された時間を表す整数を返します。

構文

```
hour ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) 有効なタイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法

指定された bigdatetime 値から抽出された時間を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例

`hour ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、12 を返します。

intdate()

スカラ。1970-01-01 00:00:00 UTC (Epoch 時間) からの秒数を表す整数を日付に変換します。

注意： この関数の名前は、大文字、小文字が区別されません。Event Stream Processor は **intdate()** と **intDate()** の両方をサポートし、同じ関数と見なします。

構文

```
intdate ( number )
```

パラメータ

number	1970-01-01 00:00:00 UTC (Epoch 時間) からの秒数を表す整数。

使用法

1970-01-01 00:00:00 UTC (Epoch 時間) からの秒数を表す値を日付に変換します。 関数は、引数として整数を受け取り、日付を返します。

例

`intDate(1)` は、日付 1970-01-01 00:00:01 を返します。

makebigdatetime()

スカラ。指定された値から bigdatetime 値を作成します。

構文

```
makebigdatetime ( year, month, day, hour, minute, second, microsecond
[ ,timezone ] )
```

パラメータ

year	0001 ～ 9999 の値に評価される式。1970 ～ 2099 の範囲以外の値は、うるう年と夏時間が適切に考慮されていないので、不正確な値になることがあります。
month	月を指定する値に評価される式。0 ～ 12 が 1 ～ 12 月を示し、0 と 1 の両方が 1 月を表します。12 を超える値は、以降の年に繰り越され、負の値は指定された年の 1 月から逆算された月を示します。

day	月内の日を指定する値に評価される式。0 と 1 は、両方とも月の最初の日を表します。指定された月の有効な日よりも大きい値は、以降の月に繰り越され、負の値は指定された月の最初の日から逆算された日を示します。
hour	日内の時間を指定する値に評価される式。23 を超える値は、以降の日に繰り越され、負の値は指定された日の午前 0 時から逆算された時間を示します。
minute	分を指定する値に評価される式。59 を超える値は、以降の時間に繰り越され、負の値は指定された時間から逆算された分を示します。
second	秒を指定する値に評価される式。59 を超える値は、以降の分に繰り越され、負の値は指定された分から逆算された秒を示します。
microsecond	マイクロ秒を指定する値に評価される式。999999 を超える値は、以降の秒に繰り越され、負の値は指定された秒から逆算されたマイクロ秒を示します。
timezone	(オプション) タイム・ゾーンを表す文字列。省略された場合、エンジンはローカル・タイム・ゾーンを想定します。有効なタイム・ゾーン文字列の詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法

指定された値から bigdatetime 値を作成します。関数は引数として整数値 (と、タイム・ゾーンを表すオプションの文字列) を受け取り、bigdatetime を返します。いずれかの引数が NULL であれば、関数から NULL が返されます。

例

`to_string (makebigdatetime (2010, 3, 3, 12, 34, 34, 59111))` は '2010-03-03 12:34:34:059111' を返します。

microsecond()

スカラ。指定された bigdatetime 値から抽出されたマイクロ秒を表す整数を返します。

構文

```
microsecond ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。

timezone	(オプション) タイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法
指定された bigdatetime 値から抽出されたマイクロ秒を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例
`microsecond ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、059111 を返します。

minute()

スカラ。指定された bigdatetime 値から抽出された分を表す整数を返します。

構文

```
minute ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) 有効なタイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法
指定された bigdatetime 値から抽出された分を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例
`minute ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、34 を返します。

month()

スカラ。指定された bigdatetime 値から抽出された月を表す整数を返します。

構文

```
month ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) 有効なタイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法

指定された bigdatetime 値から抽出された月を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例

`month ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、3 を返します。

second()

スカラ。指定された bigdatetime 値から抽出された秒を表す整数を返します。

構文

```
second ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) タイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法

指定された bigdatetime 値から抽出された秒を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。いずれかの引数が null であれば、関数から null が返されます。

例

`second ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、34 を返します。

sysdate()

スカラ。日付値として現在のシステム日付を返します。

構文

```
sysdate ()
```

使用法

日付値として現在のシステム日付を返します。関数は引数を受け取らず、日付を返します。

例

2010 年 3 月 3 日 12:34:34 に `sysdate()` を実行すると、2010-03-03 12:34:34 が返されます。

systimestamp()

スカラ。タイムスタンプ値として現在のシステム日付を返します。

構文

```
systimestamp ()
```

使用法

タイムスタンプ値として現在のシステム日付を返します。関数は引数を受け取らず、タイムスタンプを返します。

例

2010 年 3 月 3 日 12:34:34:059 に `systimestamp()` を実行すると、2010-03-03 12:34:34:059 が返されます。

unbigdatetime()

スカラ。指定された文字列を bigdatetime 値に変換します。

構文

```
unbigdatetime ( string )
```

パラメータ

string	bigdatetime 値を表す文字列。

使用法

指定された文字列を bigdatetime 値に変換します。関数は、引数として文字列を受け取り、bigdatetime 値を返します。

例

`unbigdatetime ('2003-06-14 13:15:00:232323')` は、2003-06-14 13:15:00:232323 を返します。

undate()

スカラ。指定された文字列を日付値に変換します。

構文

```
undate ( string )
```

パラメータ

string	日付値を表す文字列。

使用法

指定された文字列を日付値に変換します。関数は、引数として文字列を受け取り、日付値を返します。

例

`undate ('2003-06-14 13:15:00')` は、2003-06-14 13:15:0 を返します。

weekendday()

スカラ。指定された日付／時刻型が週末に該当するかどうかを判断します。

注意： この関数の名前は、大文字、小文字が区別されません。Event Stream Processor は **weekendday()** と **weekendDay()** の両方をサポートし、同じ関数と見なします。

構文

```
weekendday ( calendarfile, datevalue )
```

パラメータ

calendar	カレンダ・ファイルのファイル・パスを表す文字列。
datevalue	日付／時刻型。

使用法

日付／時刻型値が週末に該当するかどうかを判断します。関数は、日付／時刻型が週末に該当する場合は 1 (true) を、そうでない場合は 0 (false) を返します。カレンダのパスを表す文字列を受け取り、**datevalue** として日付／時刻型を受け取ります。関数は整数を返します。

例
`weekendDay('/aleri/cals/us.cal',v.TradeTime)` は、v.TradeTime の日付部分が週末に該当する場合は 1 を、そうでない場合は 0 を返します。

year()

スカラ。指定された bigdatetime 値から抽出された年を表す整数を返します。

構文
```
year ( bigdatetime [ ,timezone ] )
```

パラメータ

bigdatetime	bigdatetime 値。
timezone	(オプション) タイム・ゾーンを表す文字列。何も指定されていない場合は、UTC が使用されます。詳細については、「タイム・ゾーン」と「タイム・ゾーンのリスト」を参照してください。

使用法
指定された bigdatetime 値から抽出された年を表す整数を返します。関数は引数として bigdatetime 値 (オプションでタイム・ゾーンを表す文字列) を受け取り、整数を返します。

例
`year ((unbigdatetime ('2010-03-03 12:34:34:059111'))` は、2010 を返します。

その他の関数

集合型でもスカラ型でもないすべての関数のリストのリファレンスです。

cacheSize()

イベント・キャッシュの現在のバケットのサイズを返します。

構文
```
cacheSize ( )
```

使用法
イベント・キャッシュの現在のバケットのサイズを返します。関数は引数を受け取らず、整数を返します。

例

この例は、証券コードごとに上位 3 つの異なる価格を取得します。このタスクを達成するために、例では、getCache()、cacheSize()、deleteCache() の関数を使用します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;
CREATE FLEX flexOp
    IN QTrades
    OUT OUTPUT WINDOW QTradesStats SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
       DECLARE
          typedef [integer Id;| date TradeTime; string Venue;
             string Symbol; float Price;
             integer Shares] QTradesRecType;
         eventCache(QTrades[Symbol], manual, Price asc) tradesCache;
          typeof(QTrades) insertIntoCache( typeof(QTrades) qTrades )
          {
             integer counter := 0;
             typeof (QTrades) rec;
             long cacheSz := cacheSize(tradesCache);
             while (counter < cacheSz) {
                rec := getCache( tradesCache, counter );
                if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                   deleteCache(tradesCache, counter);
                   insertCache( tradesCache, qTrades );
                   return rec;
                   break;
                } else if( qTrades.Price < rec.Price) {
                   break;
                }
                counter++;
            }
            if(cacheSz < 3) {
               insertCache(tradesCache, qTrades);
               return qTrades;
            } else {
               rec := getCache(tradesCache, 0);
               deleteCache(tradesCache, 0);
               insertCache(tradesCache, qTrades);
               return rec;
            }
```

```
            return null;
         }
      END;

      ON QTrades {
         keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
         typeof(QTrades) rec := insertIntoCache( QTrades );
         if(rec.Id) {
            if(rec.Id <> QTrades.Id) {
               output setOpcode(rec, delete);
            }
            output setOpcode(QTrades, upsert);
         }
     };
    END;
```

coalesce()

その他。式のリストから、最初の null 以外の式を返します。

構文

```
coalesce ( expression [,...] )
```

パラメータ

expression	すべての式は同じデータ型である必要あり。

使用法

式のリストから、最初の null 以外の式を返します。引数のデータ型に制限はありませんが、すべての引数が同じデータ型である必要があります。 関数は、引数と同じデータ型を返します。

例

`coalesce (NULL, NULL, 'examplestring', 'teststring', NULL)` は、'examplestring' を返します。

dateint()

日付を 1970-01-01 00:00:00 UTC (Epoch 時間) からの秒数を表す整数に変換します。

注意： この関数の名前は、大文字、小文字が区別されません。Event Stream Processor は **dateint()** と **dateInt()** の両方をサポートし、同じ関数と見なします。

構文

```
dateint ( datevalue )
```

パラメータ

datevalue	日付。

使用法
日付を 1970-01-01 00:00:00 UTC (Epoch 時間) からの秒数を表す整数に変換します。関数は、引数として日付を受け取り、整数を返します。

例
`dateint (undate ('1970:01:01 00:01:01'))` は、61 を返します。

deleteCache()

イベント・キャッシュ内の、インデックスで指定される特定の場所のローを削除します。

構文

```
deleteCache (index )
```

パラメータ

index	整数で表される、イベント・キャッシュのロー・インデックス。

使用法
イベント・キャッシュ内の、インデックスで指定される特定の場所のローを削除します。このインデックスは 0 から始まります。関数は、引数として整数を受け取り、ローを削除します。出力は生成しません。無効なインデックス・パラメータを指定すると、不正なレコードが生成されます。

例
この例は、証券コードごとに上位 3 つの異なる価格を取得します。このタスクを達成するために、例では、getCache()、cacheSize()、deleteCache() の関数を使用します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;
```

```
CREATE FLEX flexOp
    IN QTrades
    OUT OUTPUT WINDOW QTradesStats SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
       DECLARE
          typedef [integer Id;| date TradeTime; string Venue;
             string Symbol; float Price;
             integer Shares] QTradesRecType;
         eventCache(QTrades[Symbol], manual, Price asc) tradesCache;
          typeof(QTrades) insertIntoCache( typeof(QTrades) qTrades )
          {
             integer counter := 0;
             typeof (QTrades) rec;
             long cacheSz := cacheSize(tradesCache);
             while (counter < cacheSz) {
                rec := getCache( tradesCache, counter );
                if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                   deleteCache(tradesCache, counter);
                   insertCache( tradesCache, qTrades );
                   return rec;
                   break;
                } else if( qTrades.Price < rec.Price) {
                   break;
                }
                counter++;
            }
            if(cacheSz < 3) {
               insertCache(tradesCache, qTrades);
               return qTrades;
            } else {
               rec := getCache(tradesCache, 0);
               deleteCache(tradesCache, 0);
               insertCache(tradesCache, qTrades);
               return rec;
            }
            return null;
         }
      END;

      ON QTrades {
         keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
         typeof(QTrades) rec := insertIntoCache( QTrades );
         if(rec.Id) {
            if(rec.Id <> QTrades.Id) {
               output setOpcode(rec, delete);
            }
            output setOpcode(QTrades, upsert);
         }
     };
    END;
```

exp()

e (自然対数の底) を、指定された数字だけ累乗した値を返します。

構文

```
exp ( value )
```

パラメータ

value	浮動小数点数。

使用法

e (自然対数の底である 2.78128) を、指定された数字だけ累乗した値を返します。引数が無効な場合、サーバで、浮動小数点数例外エラーが記録されます。

例

`exp（2.0）`は、7.3890 を返します。

firstnonnull()

その他。式のリストから、最初の null 以外の式を返します。

構文

```
firstnonnull ( expression [,...] )
```

パラメータ

expression	すべての式は同じデータ型である必要あり。

使用法

式のリストから、最初の null 以外の式を返します。引数のデータ型に制限はありませんが、すべての引数が同じデータ型である必要があります。 関数は、引数と同じデータ型を返します。この関数の動作は、**coalesce()** とまったく同じです。

例

`firstnonnull（NULL, NULL, 'examplestring', 'teststring', NULL）`は、'examplestring' を返します。

get*columnbyindex()

インデックスで識別されるカラムの値を返します。

構文

```
getbinarycolumnbyindex ( record, colname )
getstringcolumnbyindex ( record, colname )
```

```
getlongcolumnbyindex ( record, colname )
getintegercolumnbyindex ( record, colname )
getdatecolumnbyindex ( record, colname )
gettimestampcolumnbyindex ( record, colname )
getbigdatetimecolumnbyindex ( record, colname )
getintervalcolumnbyindex ( record, colname )
getbooleancolumnbyindex ( record, colname )
getfloatcolumnbyindex ( record, colname )
```

パラメータ

name	ストリームまたはウィンドウの名前。
colindex	カラムのインデックス値に対応する整数。インデックスは 0 から始まります。

使用法

インデックスで識別されるカラムの値を返します。この関数は、**name** 引数の値として文字列を、**colindex** 引数の値として整数を受け取ります。 関数は、関数名に指定されているのと同じデータ型 (たとえば、**getstringcolumnbyindex()** では文字列) を返します。

colname 引数が null に評価されるか、指定されたカラムが関連するウィンドウまたはストリームに存在しない場合、関数は null を返し、エラー・メッセージを生成します。

例

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCHEMA (a int, b string)
PRIMARY KEY (a) MEMORY STORE "memstore";
```

iwin1 に渡された入力が (1, 'hello') の場合、`getstringcolumnbyindex ( iwin1, 1 )` は 'hello' を返します。

get*columnbyname()

実行時に評価される式によって識別されるカラムの値を返します。

構文

```
getbinarycolumnbyname ( name, colname )
getstringcolumnbyname ( name, colname )
getlongcolumnbyname ( name, colname )
getintegercolumnbyname ( name, colname )
getfloatcolumnbyname ( name, colname )
getdatecolumnbyname ( name, colname )
gettimestampcolumnbyname ( name, colname )
getbigdatetimecolumnbyname ( name, colname )
getintervalcolumnbyname ( name, colname )
getbooleancolumnbyname ( name, colname )
```

パラメータ

name	ストリームまたはウィンドウの名前。
colname	ストリームまたはウィンドウ内の、関数と同じデータ型を持つカラムの名前に評価される式。 **getstringcolumnbyname()** の **colname** 引数は、たとえば、文字列を含みます。

使用法
実行時に評価される式によって識別されるカラムの値を返します。この関数は、**name** の値として文字列を受け取ります。 **colname** 引数のデータ型は、関数の型に対応します。たとえば、**getstringcolumnbyname()** では文字列です。関数は、**colname** と同じデータ型 (関数名に指定されている型) を返します。

colname 引数が null に評価されるか、指定されたカラムが関連するウィンドウまたはストリームに存在しない場合、関数は null を返し、エラー・メッセージを生成します。

例

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCHEMA (a int, b string)
PRIMARY KEY (a) MEMORY STORE "memstore";
```

iwin1 に渡された入力が (1, 'hello') の場合、`getstringcolumnbyname ( iwin1, a )` は 'hello' を返します。

getCache()

イベント・キャッシュの現在のバケットから、指定されたインデックスのローを返します。

構文

```
getCache (index )
```

パラメータ

index	整数で表される、イベント・キャッシュのロー・インデックス。

使用法
イベント・キャッシュの現在のバケットから、指定されたインデックスのローを返します。このインデックスは 0 から始まります。関数は、引数として整数を受け取り、ローを返します。無効なインデックス・パラメータを指定すると、不正なレコードが生成されます。

例

この例は、証券コードごとに上位 3 つの異なる価格を取得します。このタスクを達成するために、例では、getCache()、cacheSize()、deleteCache() の関数を使用します。

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;
CREATE FLEX flexOp
    IN QTrades
    OUT OUTPUT WINDOW QTradesStats SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
       DECLARE
          typedef [integer Id;| date TradeTime; string Venue;
             string Symbol; float Price;
             integer Shares] QTradesRecType;
         eventCache(QTrades[Symbol], manual, Price asc) tradesCache;
          typeof(QTrades) insertIntoCache( typeof(QTrades) qTrades )
          {
             integer counter := 0;
             typeof (QTrades) rec;
             long cacheSz := cacheSize(tradesCache);
             while (counter < cacheSz) {
                rec := getCache( tradesCache, counter );
                if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                   deleteCache(tradesCache, counter);
                   insertCache( tradesCache, qTrades );
                   return rec;
                   break;
                } else if( qTrades.Price < rec.Price) {
                   break;
                }
                counter++;
            }
            if(cacheSz < 3) {
               insertCache(tradesCache, qTrades);
               return qTrades;
            } else {
               rec := getCache(tradesCache, 0);
               deleteCache(tradesCache, 0);
               insertCache(tradesCache, qTrades);
               return rec;
            }
```

```
            return null;
         }
      END;

      ON QTrades {
         keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
         typeof(QTrades) rec := insertIntoCache( QTrades );
         if(rec.Id) {
            if(rec.Id <> QTrades.Id) {
               output setOpcode(rec, delete);
            }
            output setOpcode(QTrades, upsert);
         }
    };
   END;
```

getData()

この関数はデータベース・クエリを取得し、外部データベースのテーブルからローを取得して、レコードのベクトル形式でそれらを返します。

構文

```
getData(vector, service, query, expr1, ... exprn)
```

パラメータ

vector	選択されたレコードを返すために使用するベクトルの名前
service	データベース・クエリを実行するために使用するサービスの名前の文字列
query	データベースのクエリを表す文字列
expr	クエリと一緒にデータベースに渡す追加のパラメータ。任意の基本データ型 (通貨、整数、文字列など) を指定できます。

使用法

関数によって返されるレコードを格納するベクトルの名前を最初の引数として指定します。関数は、選択されたレコードが格納されている、指定された名前のベクトルを返します。

データベースをクエリするときに使用するサービスを、2 つ目の引数として指定します。データベース・クエリを実行するために使用できるサービスは、service.xml ファイルで定義されています。このファイルと、定義されているサービスについては、『管理者ガイド』を参照してください。

データベースに対して実行するクエリを、3 つ目の引数として指定します。クエリは、該当するサービスが service.xml ファイルで定義されているかぎり、いずれのデータベース・クエリ言語 (SQL など) でもかまいません。クエリと一緒にデータベースに渡す追加のパラメータを 4 つ目以降の引数として指定します。

注意： 追加のパラメータを渡す場合、パラメータごとに "?" 文字でマークされるプレースホルダをクエリ文に指定する必要があります。

例

`getData(v, 'MyService', 'SELECT col1, col2 FROM myTable WHERE id= ?', 'myId');` テーブル "myTable" からサービス "MyService" を使用してレコードを取得します。"id" が値 "myId" に等しいすべてのローの最初の 2 つのカラムを選択し、ベクトル "v" に格納して返します。

getmoneycolumnbyindex()

インデックスで識別されるカラムの値を返します。

構文

```
getmoneycolumnbyindex ( name, colindex, scale )
```

パラメータ

name	ストリームまたはウィンドウの名前。
colname	カラムのインデックス値に対応する整数。インデックスは 0 から始まります。
scale	1 ～ 15 の整数。

使用法

インデックスで識別されるカラムの値を返します。この関数は、**name** 引数の値として文字列を、**colindex** 引数と **scale** 引数の値として整数を受け取ります。 関数は、指定された精度の通貨型を返します。

colname 引数が null に評価されるか、指定されたカラムが関連するウィンドウまたはストリームに存在しない場合、関数は null を返し、エラー・メッセージを生成します。

例

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCHEMA (a money(1), b money(3))
PRIMARY KEY (a) MEMORY STORE "memstore";
```

iwin1 に渡された入力が (1.2, 1.23) の場合、`getmoneycolumnbyindex ( iwin1, 1, 3 )` は 1.123 を返します。

getmoneycolumnbyname()

実行時に通貨型として評価される式によって識別されるカラムの値を返します。

構文

```
getmoneycolumnbyname ( name, colname, scale )
```

パラメータ

name	ストリームまたはウィンドウの名前。
colname	ストリームまたはウィンドウ内の、通貨型を持つカラムの名前に評価される式。
scale	1 〜 15 の整数。

使用法

実行時に評価される式によって識別されるカラムの値を返します。この関数は、**name** 引数と **colname** 引数の値として文字列を受け取ります。また、通貨型の精度を表す整数を受け取ります。 関数は、指定された精度の通貨型を返します。

colname 引数が null に評価されるか、指定されたカラムが関連するウィンドウまたはストリームに存在しない場合、関数は null を返し、エラー・メッセージを生成します。

例

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCEHMA (a money(1), b money(3))
PRIMARY KEY (a) MEMORY STORE "memstore";
```

iwin1 に渡された入力が (1.2, 1.23) の場合、`getmoneycolumnbyname ( iwin1, b, 3 )` は 1.123 を返します。

getrowid()

その他。ウィンドウの指定されたローのシーケンス番号を返します。

構文

```
getrowid ( row )
```

パラメータ

row	ウィンドウ内のロー。

使用法
ウィンドウの指定されたローのシーケンス番号を返します。この関数は、引数としてローを受け取り、ウィンドウのローのシーケンス番号を返します。このシーケンス番号は rowid と呼ばれ、ローが挿入されるときに一意に割り当てられます。

例

```
CREATE MEMORY STORE "memstore";

CREATE INPUT WINDOW iwin1 SCHEMA (a money(1), b money(3))
PRIMARY KEY (a) MEMORY STORE "memstore";

CREATE INPUT WINDOW iwin2 SCHEMA (a money(1), b money(3))
PRIMARY KEY (a) MEMORY STORE "memstore";
```

これらのウィンドウでは、`getrowid (iwin1)` は 0 を、`getrowid (iwin2)` は 1 を返します。

now()

現在のシステム日付を bigdatetime 値として返します。

構文

```
now ()
```

使用法
現在のシステム日付を bigdatetime 値として返します。関数は引数を受け取らず、bigdatetime 値を返します。この関数は、**sysbigdatetime()** と同じ機能を提供します。

例
2010 年 3 月 3 日 12:34:34:059111 に `now()` を実行すると、2010-03-03 12:34:34:059111 が返されます。

rank()

その他。現在のグループでのローの位置を返します (**GROUP HAVING** 式内でのみ使用されます)。

構文

```
rank()
```

使用法
現在のグループ内のローの位置を返します。位置は、0 で始まります。この関数は、**GROUP HAVING** 式内で使用する場合のみ役立ちます。 関数は引数を受け取らず、整数を返します。

例

`rank() > 3` は、グループ内の先頭の 3 つのローには 1 を、その他のすべてのローには 0 を返します。

recordDataToRecord

指定されたソース・ストリームのスキーマに基づいて、バイナリの errorRecord 値を RECORD データ型値に変換します。

構文

```
recordDataToRecord (string sourceStreamName, binary errorRecord)
```

パラメータ

sourceStreamName は、エラー・レコードの発生元のストリームの名前を提供する文字列です。戻り値の型をチェックできるようにするために、名前が格納される変数ではなく、実際の名前を指定する必要があります。この引数が既存のストリームを示していない場合、**recordDataToRecord** は、不正な引数が指定されたことを示すエラー・フラグを設定して、null を返します。

errorRecord は、エラーをトリガしたレコードを提供するバイナリです。これは、常に、エラー・ストリームの errorRecord フィールドである必要があります。

注意： 任意のバイナリ文字列または一致しないスキーマ (ストリーム) 名を渡すと、不確定な動作が発生することがあります。不確定な動作は、レコードにガーベジが残るだけといった場合から、サーバがクラッシュする場合に至るまでの広範囲にわたります。この組み込み関数への引数は、常に、同じエラー・ストリームの *sourceStreamName* フィールドと *errorRecord* フィールドである必要があります。

---

recordDataToString

バイナリの errorRecord 値を文字列フォーマットに変換します。

構文

```
recordDataToString (string sourceStreamName, binary errorRecord)
```

パラメータ

sourceStreamName は、エラー・レコードの発生元のストリームの名前を提供する文字列です。これは、常に、エラー・ストリームの sourceStreamName フィールドである必要があります。別のストリーム (エラー・ストリームなど) の名前を指定すると、スキーマ不一致によって致命的なエラーが発生することがあります。こ

の引数が既存のストリームを示していない場合、**recordDataToString** は、不正な引数が指定されたことを示すエラー・フラグを設定して、null を返します。

errorRecord は、エラーをトリガしたレコードを提供するバイナリです。これは、常に、エラー・ストリームの errorRecord フィールドで、スキーマは、常に、レコードに一致する必要があります。

注意： 任意のバイナリ文字列または一致しないスキーマ (ストリーム) 名を渡すと、不確定な動作が発生することがあります。不確定な動作は、レコードにガーベジが残るだけといった場合から、サーバがクラッシュする場合に至るまでの広範囲にわたります。この組み込み関数への引数は、常に、同じエラー・ストリームの *sourceStreamName* フィールドと *errorRecord* フィールドである必要があります。

---

sind()

指定された値の正弦を度数で返します。

構文

```
sind ( value )
```

パラメータ

value	浮動小数点数。

使用法

指定された値の正弦を度数で返します。関数は、引数として浮動小数点数を受け取り、浮動小数点数を返します。

例

`sind(45.0)` は、0.850903525 を返します。

sysbigdatetime()

現在のシステム日付を bigdatetime 値として返します。

構文

```
sysbigdatetime ()
```

使用法

現在のシステム日付を bigdatetime 値として返します。関数は引数を受け取らず、bigdatetime 値を返します。この関数は、**now()** と同じ機能を提供します。

例

2010 年 3 月 3 日 12:34:34:059111 に `sysbigdatetime()` を実行すると、2010-03-03 12:34:34:059111 が返されます。

totimezone()

日付を、指定されたタイム・ゾーンから、指定された別のタイム・ゾーンに変換します。

構文

```
totimezone ( datevalue, fromzone, tozone )
```

パラメータ

datevalue	日付または bigdatetime。
fromzone	法定タイム・ゾーンを表す文字列。
tozone	法定タイム・ゾーンを表す文字列。

使用法

日付を、指定されたタイム・ゾーンから、新しいタイム・ゾーンに変換します。最初の引数は変換する日付、2 番目の引数は元のタイム・ゾーン、3 番目の引数は新しいタイム・ゾーンです。タイム・ゾーン値は、業界標準の TZ データベースから取得されます。 最初の引数は日付である必要があります。2 番目と 3 番目の引数は、法定タイム・ゾーンを表す文字列である必要があります。関数は日付を返します。

例

`totimezone(v.TradeTime, 'GMT', 'EDT')` は、各 TradeTime (取引時刻) の時刻部分を、グリニッジ標準時から米国東部夏時間に変換します。

付録 A　キーワードのリスト

CCL の予約語では、大文字と小文字は区別されません。キーワードは、CCL オブジェクトの識別子として使用できません。

CCL で提供されているキーワードのリスト：

adapter	age(s)	all	and	as	asc
attach	auto	begin	break	case	cast
connection	continue	count	create	day(s)	declare
deduced	default	delete	delta	desc	distinct
dumpfile	dynamic	else	end	eventCache	every
exit	external	false	fby	filter	first
flex	for	foreign	foreignJava	from	full
group	groups	having	hour(s)	hr	if
import	in	inherits	inner	input	insert
into	is	join	keep	key	last
language	left	library	like	load	local
log	max	memory	micros	microsecond(s)	millis
millisecond(s)	min	minute(s)	module	money	name
new	nostart	not	nth	NULL	on
or	order	out	outfile	output	parameter(s)
pattern	primary	properties	rank	records	retain
return	right	row(s)	safedelete	schema	sec
second(s)	select	set	setRange	slack	start
static	store(s)	stream	sum	sync	switch
then	times	to	top	transaction	true
type	typedef	typeof	union	update	upsert

values	when	where	while	window	within
xmlattributes	xmlelement				

参照：

- *大文字と小文字の区別* (25 ページ)

付録 B　日付と時刻のプログラミング

タイム・ゾーン・パラメータ、日付フォーマット・コード設定を設定し、カレンダを定義します。

タイム・ゾーン

タイム・ゾーンは、一般的にローカル時間と呼ばれる同じ標準時間を使用する地域を表します。

ほとんどの場合、隣接したタイム・ゾーンとは、1 時間の時差があります。慣例によって、すべてのタイム・ゾーンは、ローカル時間を GMT/UTC からのオフセットとして計算します。グリニッジ標準時 (GMT) は歴史上の用語で、元々、英国の王立グリニッジ天文台での平均太陽時を意味していました。GMT は、原子時計を基準にする協定世界時 (UTC) に置き換えられています。すべての Sybase Event Stream Processor の使用目的で、GMT と UTC は等価です。政治的または地理的な実用性によって、タイム・ゾーン特性が時間経過と共に変化することがあります。たとえば、夏時間の開始日と終了日は年によって異なることがあり、新しい国が誕生すると新しいタイム・ゾーンが導入されることがあります。

内部的には、Event Stream Processor は日付／時刻型の情報を、データ型に応じて 1970 年 1 月 1 日午前 0 時 UTC からの秒数、ミリ秒数、またはマイクロ秒数として常に格納します。タイム・ゾーン指示子が使用されていない場合は、UTC 時間が適用されます。

夏時間
夏時間は、そのタイム・ゾーンで夏時間が使用されており、指定されたタイムスタンプが夏時間の期間内にあるときに考慮されます。夏時間の開始日と終了日は、C++ ライブラリに格納されています。

特定のタイム・ゾーンが指定されており、そのタイム・ゾーンで夏時間が使用されている場合、Event Stream Processor はこれらの日付を考慮して日付／時刻のデータ型を調整します。たとえば、次の例では、米国太平洋標準時 (PST) は夏時間の期間内にあるので、タイムスタンプが調整されます。

```
to_timestamp('2002-06-18 13:52:00.123456 PST','YYYY-MM-DD
HH24:MI:SS.ff TZD')
```

標準時から夏時間への移行とその逆の移行
夏時間への移行時と夏時間からの移行時には、特定の時間がありません。たとえば、米国の場合、標準時から夏時間への移行時に、時計は 01:59 から 03:00 に変化

し、02:00 がありません。逆に、夏時間から標準時への移行時には、1 晩に 01:00 〜 01:59 が 2 回発生します。これは、夏時間の終了時に時間が 2:00 から 1:00 に変化するためです。

ただし、これらの未定義の時間に受信データの入力が発生する可能性があり、エンジンは何らかの方法で対処する必要があります。夏時間への移行時に、Event Stream Processor は 02:59 PST を 01:59 PST として解釈します。標準時に戻るときには、Event Stream Processor は 02:00 PDT を 01:00 PST として解釈します。

タイム・ゾーンのデフォルトの変更

特定の日付と時刻の関数でオプションのタイム・ゾーン・パラメータの値が指定されない場合、Event Stream Processor では UTC を使用します。

Sybase CEP の対応する関数は、パラメータが指定されなかったときには、サーバのローカル・タイム・ゾーンをデフォルトで使用します。タイム・ゾーンが定義されていない CEP プロジェクトを移行する場合は、Event Stream Processor に変換されるときに UTC が使用されます。サーバのローカル・タイム・ゾーンを継続して使用するには、そのタイム・ゾーンを、以下の関数のタイム・ゾーン・パラメータで明示的に設定します。

Sybase CEP の関数	Event Stream Processor の関数
dayofmonth	dayofmonth
dayofweek	dayofweek
dayofyear	dayofyear
hour	hour
maketimestamp	makebigdatetime
microsecond	microsecond
minute	minute
month	month
second	second
to_string	to_string
year	year

タイム・ゾーンのリスト

Event Stream Processor は標準タイム・ゾーンとそれらの省略形をサポートします。

以下の表は、Event Stream Processor で使用されるタイム・ゾーンのリストで、業界標準の Olson タイム・ゾーン (別名 TZ) データベースから取得されています。

ACT	AET	AGT
ART	AST	Africa/Abidjan
Africa/Accra	Africa/Addis_Ababa	Africa/Algiers
Africa/Asmera	Africa/Bamako	Africa/Bangui
Africa/Banjul	Africa/Bissau	Africa/Blantyre
Africa/Brazzaville	Africa/Bujumbura	Africa/Cairo
Africa/Casablanca	Africa/Ceuta	Africa/Conakry
Africa/Dakar	Africa/Dar_es_Salaam	Africa/Djibouti
Africa/Douala	Africa/El_Aaiun	Africa/Freetown
Africa/Gaborone	Africa/Harare	Africa/Johannesburg
Africa/Kampala	Africa/Khartoum	Africa/Kigali
Africa/Kinshasa	Africa/Lagos	Africa/Libreville
Africa/Lome	Africa/Luanda	Africa/Lubumbashi
Africa/Lusaka	Africa/Malabo	Africa/Maputo
Africa/Maseru	Africa/Mbabane	Africa/Mogadishu
Africa/Monrovia	Africa/Nairobi	Africa/Ndjamena
Africa/Niamey	Africa/Nouakchott	Africa/Ouagadougou
Africa/Porto-Novo	Africa/Sao_Tome	Africa/Timbuktu
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/Danmarkshavn
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/Petersburg
America/Indiana/Vevay	America/Indiana/Vincennes	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/Moncton	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/Scoresbysund	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/Azores	Atlantic/Bermuda
Atlantic/Canary	Atlantic/Cape_Verde	Atlantic/Faeroe
Atlantic/Jan_Mayen	Atlantic/Madeira	Atlantic/Reykjavik
Atlantic/South_Georgia	Atlantic/St_Helena	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	BET	BST
Brazil/Acre	Brazil/DeNoronha	Brazil/East
Brazil/West	CAT	CET
CNT	CST	CST6CDT
CTT	Canada/Atlantic	Canada/Central
Canada/East-Saskatchewan	Canada/Eastern	Canada/Mountain
Canada/Newfoundland	Canada/Pacific	Canada/Saskatchewan
Canada/Yukon	Chile/Continental	Chile/EasterIsland
Cuba	EAT	ECT
EET	EST	EST5EDT
Egypt	Eire	Etc/GMT
Etc/GMT+0	Etc/GMT+1	Etc/GMT+10
Etc/GMT+11	Etc/GMT+12	Etc/GMT+2
Etc/GMT+3	Etc/GMT+4	Etc/GMT+5
Etc/GMT+6	Etc/GMT+7	Etc/GMT+8
Etc/GMT+0	Etc/GMT-0	Etc/GMT-1
Etc/GMT-10	Etc/GMT-11	Etc/GMT-12
Etc/GMT-13	Etc/GMT-14	Etc/GMT-2
Etc/GMT-3	Etc/GMT-4	Etc/GMT-5
Etc/GMT-6	Etc/GMT-7	Etc/GMT-8
Etc/GMT-9	Etc/GMT0	Etc/Greenwich
Etc/UCT	Etc/UTC	Etc/Universal

Etc/Zulu	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
Factory	GB	GB-Eire
GMT	GMT+0	GMT-0
GMT0	Greenwich	HST
Hongkong	IET	IST
Iceland	Indian/Antananarivo	Indian/Chagos
Indian/Christmas	Indian/Cocos	Indian/Comoro
Indian/Kerguelen	Indian/Mahe	Indian/Maldives
Indian/Mauritius	Indian/Mayotte	Indian/Reunion
Iran	Israel	JST
Jamaica	Japan	Kwajalein

Libya	MET	MIT
MST	MST7MDT	Mexico/BajaNorte
Mexico/BajaSur	Mexico/General	Mideast/Riyadh87
Mideast/Riyadh88	Mideast/Riyadh89	NET
NST	NZ	NZ-CHAT
Navajo	PLT	PNT
PRC	PRT	PST
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	ROC	ROK
SST	Singapore	SystemV/AST4
SystemV/AST4ADT	SystemV/CST6	SystemV/CST6CDT
SystemV/EST5	SystemV/EST5EDT	SystemV/HST10
SystemV/MST7	SystemV/MST7MDT	SystemV/PST8
SystemV/PST8PDT	SystemV/YST9	SystemV/YST9YDT
Turkey	UCT	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/Pacific-New	US/Samoa
UTC	Universal	VST
W-SU	WET	Zulu

日付／時刻のフォーマット・コード

日付／時刻型 (日付、タイムスタンプ、bigdatetime) のフォーマットを指定するのに使用できる有効なコンポーネントのリスト。

日付／時刻型のフォーマットは、Event Stream Processor のフォーマット・コードまたは C++ の strftime() 関数で提供されるタイムスタンプ変換コードのサブセットのいずれかを使用して指定する必要があります。 多くの有効なコードがありますが、有効な日付／時刻型指定では、特定の時間単位を指定するコード (たとえば年を指定するコード) は 1 回しか出現できません。

注意： 年、月、日、時、分、または秒のすべての指定は、コードで指定されているよりも少ない桁数にも対応します。たとえば、DD は 2 桁と 1 桁の両方の日エントリに対応します。

Event Stream Processor の時間フォーマット・コード

カラム・コード	説明	入力	出力
MM	月 (01 ～ 12。1 月 = 01)。	Y	Y
YYYY	4 桁の年。	Y	Y
YYY	年の下 3 桁。	Y	Y
YY	年の下 2 桁。	Y	Y
Y	年の下 1 桁。	Y	Y
Q	四半期 (1、2、3、4。1 ～ 3 月 = 1)。	N	Y
MON	月の省略名 (JAN、FEB、...、DEC)。	Y	Y

カラム・コード	説明	入力	出力
MONTH	9 文字になるようにブランクが追加された月名 (JANUARY、FEBRUARY、...、DECEMBER)。	Y	Y
RM	ローマ数字で表す月 (I 〜 XII。1 月 = I)。	Y	Y
WW	年内の何週目であるかを示す数 (1 〜 53)。第 1 週は年の最初の日に始まり、7 日ごとに次の週が始まります。	N	Y
W	月内の何週目であるかを示す数 (1 〜 5)。第 1 週は月の最初の日に始まり、7 日ごとに次の週が始まります。	N	Y
D	曜日 (1 〜 7。日曜日 = 1)。	N	Y
DD	日 (1 〜 31)。	Y	Y
DDD	年内の何日目であるかを示す数 (1 〜 366)。	N	Y
DAY	曜日名 (SUNDAY、MONDAY、...、SATURDAY)。	Y	Y
DY	曜日の省略名 (SUN、MON、...、SAT)。	Y	Y
HH	時間 (1 〜 12)。	Y	Y
HH12	時間 (1 〜 12)。	Y	Y
HH24	時間 (0 〜 23)。	Y	Y
AM	午前／午後の識別子 (AM/PM)。	Y	Y
PM	午前／午後の識別子 (AM/PM)。	Y	Y
MI	分 (0 〜 59)。	Y	Y
SS	秒 (0 〜 59)。	Y	Y
SSSSS	午前 0 時以降の秒数 (0 〜 86399)。	Y	Y
SE	Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からの秒数。このフォーマットは、FF フォーマットまたはタイム・ゾーン・コードの TZD、TZR、TZH、TZM との組み合わせで用いられた場合に、自身によってのみ使用されます。	Y	Y
MIC	Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのマイクロ秒数。	Y	Y

カラム・コード	説明	入力	出力
FF	秒の小数部分 (0 〜 999999)。出力で使用されると、FF はマイクロ秒を示す 6 桁の数を生成します。FFFF は、12 桁の数を生成し、マイクロ秒を示す 6 桁を 2 回繰り返します (多くの場合、これは望ましい出力ではありません)。入力で使用されると、FF は数値以外の文字が検出されるまですべての桁を収集します。ただし、使用されるのは最初の 6 桁のみで、残りは破棄されます。	Y	Y
FF[1 〜 9]	秒の小数部分。出力の場合、指定された桁の数が生成され、必要に応じて、丸め、または後続ゼロの追加が行われます。	N	Y
MS	Epoch 時間 (1970 年 1 月 1 日午前 0 時 UTC) からのミリ秒数。入力の場合、このフォーマット・コードは、FF (マイクロ秒) とタイム・ゾーン・コードの TZD、TZR、TZH、TZM との組み合わせでのみ使用できます。他のすべてのフォーマット・コードとの組み合わせは、エラーになります。さらに、MS と FF を一緒に使用する場合、MS コードを FF コードより先に指定する必要があります。たとえば、MS.FF と指定します。	Y	Y
FM	フィル・モードの指定。ゼロとブランクを抑制するかどうかを指定します (デフォルトは抑制なし)。	Y	Y
FX	厳密モードの指定。大文字／小文字と句読表記を正確に一致させるかどうかを指定します (デフォルトは、正確な一致なし)。	Y	Y
RR	2 桁のみを使用して、20 世紀の日付を 21 世紀の日付として格納できるようにします。	Y	N
RRRR	4 桁の年数。4 桁または 2 桁の入力を受け付けます。2 桁の場合、RR と同じ結果を返します。	Y	N
TZD	PST などのような省略形のタイム・ゾーン指示子。	Y	Y
TZH	タイム・ゾーンの GMT との時間の差。たとえば、-5 は GMT より 5 時間早いタイム・ゾーンを示します。	N	Y
TZM	タイム・ゾーンの GMT との時間と分の差。たとえば、-5:30 は GMT より 5 時間 30 分早いタイム・ゾーンを示します。	N	Y
TZR	タイム・ゾーンの地域名。たとえば、PST については、US/Pacific。	N	Y

Strftime() タイムスタンプ変換コード

Event Stream Processor の時間フォーマット・コードを使用する代わりに、C++ の strftime() 関数コードのサブセットを使用して、出力のタイムスタンプ・フォーマットを指定できます。以下の規則が適用されます。

- パーセント記号 (%) を含むすべてのタイムスタンプ・フォーマット指定が、strftime() コードとして扱われる。
- 文字列は、Event Stream Processor のフォーマット・コードまたは strftime() コードのいずれかのみを指定できる。
- strftime() コードには、Microsoft Windows 上でのみ有効なものと、UNIX 系オペレーティング・システム上でのみ有効なものがある。strftime() の実装ごとに、コード解釈が若干異なることもあります。エラーを回避するために、ESP サーバと ESP スタジオを同じプラットフォーム上に配置し、互換性のある strftime() 実装を使用します。また、提供されているコードがプラットフォームの要件を満たしていることを確認する必要があります。
- strftime() を使用して指定されたフォーマットのすべてのタイム・ゾーンは、ローカル・タイム・ゾーンであると想定される。
- strftime() コードは、日付／時刻型入力を指定するためには使用できず、日付／時刻型出力のみで使用できる。

Event Stream Processor は、以下の strftime() コードをサポートします。

Strftime() コード	**説明**
%a	曜日名の省略形。たとえば、"Mon"。
%A	完全な曜日名。たとえば、"Monday"。
%b	月名の省略形。たとえば、"Feb"。
%B	完全な月名。たとえば、"February"。
%c	完全な日付と時刻の文字列。このコードの出力フォーマットは、Microsoft Windows または UNIX 系オペレーティング・システムのいずれを使用しているかによって異なります。Microsoft Windows での出力例：08/26/08 20:00:00。UNIX 系オペレーティング・システムでの出力例：Tue Aug 26 20:00:00 2008。
%d	01 ～ 31 の範囲の 2 桁の 10 進整数で表される、月内の日。
%H	00 ～ 23 の範囲の 2 桁の 10 進整数で表される時間。
%I	01 ～ 12 の範囲の 2 桁の 10 進整数で表される時間。

Strftime() コード	説明
%j	001 〜 366 の範囲の 3 桁の 10 進整数で表される、年内の日。
%m	01 〜 12 の範囲の 2 桁の 10 進整数で表される月。
%M	00 〜 59 の範囲の 2 桁の 10 進整数で表される分。
%p	AM または PM と等価なロケールの文字列。
%S	00 〜 61 の範囲の 2 桁の 10 進整数で表される秒。
%U	年内の第何週であるかを示す数。00 〜 53 の範囲の 2 桁の 10 進整数で表され、週は日曜日から始まるとみなされます。
%w	曜日を示す数。0 〜 6 の範囲の 1 桁の 10 進整数で表され、0 は日曜日を示します。
%W	年内の第何週であるかを示す数。00 〜 53 の範囲の 2 桁の 10 進整数で表され、週は月曜日から始まるとみなされます。
%x	完全な日付文字列 (時刻なし)。このコードの出力フォーマットは、Microsoft Windows または UNIX 系オペレーティング・システムのいずれを使用しているかによって異なります。Microsoft Windows での出力例：08/26/08。UNIX 系オペレーティング・システムでの出力例： Tue Aug 26 2008。
%X	完全な時刻文字列 (日付なし)。
%y	世紀部分のない年。00 〜 99 の範囲の 2 桁の 10 進整数で表されます。
%Y	世紀部分のある年。4 桁の 10 進整数で表されます。
%%	% に置換される。

カレンダ・ファイル

指定された期間の休日や週末を示すテキスト・ファイル。

構文

```
weekendStart <integer>
weekendEnd <integer>
holiday yyyy-mm-dd
holiday yyyy-mm-dd
...
```

コンポーネント

weekendStart	曜日を表す整数。ここで、月曜日 =0、火曜日 =1、...、土曜日 =5、日曜日 =6。
weekendEnd	曜日を表す整数。ここで、月曜日 =0、火曜日 =1、...、土曜日 =5、日曜日 =6。
holiday	yyyy-mm-dd の形式で表される日付。カレンダ・ファイルに指定できる休日の数に制限はありません。

使用法

カレンダ・ファイルは、週末の開始日と終了日、その年の休日が記述されたテキスト・ファイルです。'#' 文字で始まる行は無視され、詳細説明やコメントを記述するのに使用できます。

カレンダ・ファイルは、Event Stream Processor によって、必要なときにロードされ、キャッシュされます。カレンダ・ファイルに変更が発生した場合は、コマンド refresh_calendars を送信して、キャッシュされているカレンダ・データを更新する必要があります。

例

次に、法定休日のカレンダ・ファイルの例を示します。

```
# Sybase calendar data for US 1983
weekendStart 5
weekendEnd 6
holiday 1983-02-21
holiday 1983-04-01
holiday 1983-05-30
holiday 1983-07-04
holiday 1983-09-05
holiday 1983-11-24
holiday 1983-12-26
```

索引

A

acos() 168
ADAPTER START 文 69
AGING 句 95
ANSI 構文 44
any() 136
API
　サポートされている言語 9
AS 句 97
ascii() 200
asin() 169
atan() 169
atan2() 169
ATTACH ADAPTER 文 70
avg() 137
avgof() 170

B

base64_binary() 201
base64_string() 201
bigdatetime
　フォーマット・コード 264
bitand() 171
bitclear() 171
bitflag() 172
bitflaglong() 172
bitmask() 172
bitmasklong() 173
bitnot() 173
bitor() 174
bitset() 174
bitshiftleft() 175
bitshiftright() 175
bittest() 176
bittoggle() 176
bitxor() 177
business() 220
businessday() 220

C

cacheSize() 236
CASE 句 98
cast() 202
cbrt() 178
CCL
　概要 10
　言語コンポーネント 25
　高度な手法の概要 51
　要素の順序 13
CCL キーワード 253
CCL 関数 129
CCL 文
　リファレンス 69
ceil() 178
char() 203
coalesce() 238
compare() 178
concat() 203
cos() 179
cosd() 179
cosh() 180
count() 141
count(distinct) 142
covar_pop() 139
covar_samp() 140
CREATE DELTA STREAM 文 72
CREATE FLEX 文 75
CREATE LIBRARY 文 79
CREATE LOG STORE 文 80
CREATE MEMORY STORE 文 82
CREATE MODULE 文 84
CREATE SCHEMA 文 21, 85
CREATE STREAM 文 87
CREATE WINDOW 文 89

D

date() 221
dateceiling() 221
datefloor() 223
dateint() 238
datename() 225
datepart() 225
dateround() 226

dayofmonth() 227
dayofweek() 228
dayofyear() 229
DECLARE ブロック
    DECLARE 文 86
    グローバル 51
    ローカル 51
    概要 51
    宣言 56
DECLARE 文 86
deleteCache() 239
distance() 180
distancesquared() 181
DST 255

E

Event Stream Processor
    コンポーネント 7
exp_weighted_avg() 142
exp() 241
extract() 204

F

first_value()
    次を参照： first()
first() 144, 145
firstnonnull() 241
floor() 181
FROM 句 111
    ANSI 構文 112
    カンマ区切りの構文 111
fromnetbinary() 204

G

get*columnbyindex() 241
get*columnbyname() 242
getbigdatetimecolumnbyindex() 241
getbigdatetimecolumnbyname() 242
getbinarycolumnbyindex() 241
getbinarycolumnbyname() 242
getbooleancolumnbyindex() 241
getbooleancolumnbyname() 242
getCache() 243
getData 関数 245
getdatecolumnbyindex() 241
getdatecolumnbyname() 242
getfloatcolumnbyindex() 241
getfloatcolumnbyname() 242
getintegercolumnbyindex() 241
getintegercolumnbyname() 242
getintervalcolumnbyindex() 241
getintervalcolumnbyname() 242
getlongcolumnbyindex() 241
getlongcolumnbyname() 242
getmoneycolumnbyindex() 246
getmoneycolumnbyname() 247
getrowid() 247
getstringcolumnbyindex() 241
getstringcolumnbyname() 242
gettimestampcolumnbyindex() 241
gettimestampcolumnbyname() 242
GROUP BY 句 114
    rank() 248
GROUP FILTER 句 115
    rank() 248
GROUP ORDER BY 句 116
    rank() 248
GUI オーサリング
    次を参照： ビジュアル・オーサリング

H

HAVING 句 117
    rank() 248
hex_binary() 205
hex_string() 205
hour() 229

I

IMPORT 文 91
IN 句 98
int32() 190
intdate() 230
isnull() 182

K

KEEP 句 100
    保持ポリシー 19

L

last_value()
　次を参照： last()
last() 145, 146
left() 191
length() 182
like() 191
ln() 183
LOAD MODULE 文 92, 98, 101, 102, 107
log10() 183
log2() 183
logx() 184
lower() 192
ltrim() 192
lwm_avg() 146

M

makebigdatetime() 230
MATCHING 句 47
max() 147
maxof() 184
meandeviation() 148
median() 148
microsecond() 231
min() 149
minof() 185
minute() 232
month() 232
msecToTime() 206

N

nextval() 185
now() 248
nth() 150

O

ON 句
　ジョイン構文 118, 121
opcode
　挿入、更新、削除のイベント 6
　定義 6
OUT 句 101

P

PARAMETERS 句 102
　パラメータのバインド 61
patindex() 193
pi() 186
POSIX の正規表現関数
　regexp_firstsearch() 194
　regexp_replace() 195
　regexp_search() 196
power() 186
PRIMARY KEY 句 104

R

random() 186
rank() 115, 248
real() 194
recent() 151
recordDataToRecord 249
recordDataToString 249
regexp_firstsearch() 194
regexp_replace() 195
regexp_search() 196
regr_avgx() 152
regr_avgy() 153
regr_count() 154
regr_intercept() 154
regr_r2() 155
regr_slope() 156
regr_sxx() 157
regr_sxy() 158
regr_syy() 159
replace() 196
right() 197
round() 187
rtrim() 197

S

SCHEMA 句 21, 105
SDK
　サポートされている言語 9
second() 233
secToTime() 206
SELECT 句 121
sign() 187
sin() 188
sind() 250
sinh() 188
SPLASH
　概要 11

SPLASH 関数
　宣言 130
sqrt() 188
stddev_samp() 160, 161
stddev()
　次を参照： stddev_samp()
stddeviation()
　次を参照： stddev_samp()
STORE 句 106
STORES 句 107
string() 198
substr() 198
sum() 162
sysbigdatetime() 250
sysdate() 234
systimestamp() 234

T

tan() 189
tanh() 190
timeToMsec() 207
timeToSec() 208
timeToUsec() 207
to_bigdatetime() 209
to_binary() 208
to_boolean() 210
to_date() 210
to_float() 211
to_integer() 212
to_interval() 212
to_long() 213
to_money() 213
to_string() 215
to_timestamp() 217
to_xml() 214
tonetbinary() 214
totimezone() 251
trim() 199
trunc() 199
typedef 52

U

unbigdatetime() 234
undate() 235
union 40, 123
UNION 演算子 40, 41, 123
upper() 200
usecToTime() 217

V

valueinserted() 163
var_pop() 164
var_samp() 164
vwap() 165

W

weekendday() 235
weighted_avg() 166
WHERE 句 39, 125

X

XML 関数
　xmlagg() 167
　xmlconcat() 218
　xmlelement() 218
　xmlparse() 219
　xmlserialize() 219
xmlagg() 167
xmlconcat() 218
xmlelement() 218
xmlparse() 219
xmlserialize() 219

Y

year() 236

あ

アダプタ 23
　カスタム 9
　概要 8
　出力アダプタ 24
　入力アダプタ 23

い

イベント
　更新 6
　削除 6

挿入 6
例 2
イベント・キャッシュ関数
cacheSize() 236
deleteCache() 239
getCache() 243
getrowid() 247
イベント・ストリーム
概要 2
インポート
CCL ファイル 91
IMPORT 文 91
スキーマ定義 91
パラメータ 91
関数定義 91
変数 91

う

ウィンドウ 14, 16
スキーマ 6, 21
ローカル 16, 17, 89
暗黙的 18
永続性 63
概要 5
構造 6, 21
出力 16, 17, 89
入力 16, 17, 89
名前なし 17, 18
名前付き 17, 89
ウィンドウ・アクセス関数
cacheSize() 236
deleteCache() 239
getCache() 243
getrowid() 247

え

エラー・ストリーム 64, 74

か

カウント基準の保持 19
カスタム・アダプタ
概要 9
カラム
BIGROWTIME 15
ROWID 15
ROWTIME 15
カラム・アクセス関数
get*columnbyindex() 241
get*columnbyname() 242
getbigdatetimecolumnbyindex() 241
getbigdatetimecolumnbyname() 242
getbinarycolumnbyindex() 241
getbinarycolumnbyname() 242
getbooleancolumnbyindex() 241
getbooleancolumnbyname() 242
getdatecolumnbyindex() 241
getdatecolumnbyname() 242
getfloatcolumnbyindex() 241
getfloatcolumnbyname() 242
getintegercolumnbyindex() 241
getintegercolumnbyname() 242
getintervalcolumnbyindex() 241
getintervalcolumnbyname() 242
getlongcolumnbyindex() 241
getlongcolumnbyname() 242
getmoneycolumnbyindex() 246
getmoneycolumnbyname() 247
getstringcolumnbyindex() 241
getstringcolumnbyname() 242
gettimestampcolumnbyindex() 241
gettimestampcolumnbyname() 242
カラム／ウィンドウ・アクセス関数
cacheSize() 236
deleteCache() 239
get*columnbyindex() 241
get*columnbyname() 242
getbigdatetimecolumnbyindex() 241
getbigdatetimecolumnbyname() 242
getbinarycolumnbyindex() 241
getbinarycolumnbyname() 242
getbooleancolumnbyindex() 241
getbooleancolumnbyname() 242
getCache() 243
getdatecolumnbyindex() 241
getdatecolumnbyname() 242
getfloatcolumnbyindex() 241
getfloatcolumnbyname() 242
getintegercolumnbyindex() 241
getintegercolumnbyname() 242
getintervalcolumnbyindex() 241
getintervalcolumnbyname() 242
getlongcolumnbyindex() 241
getlongcolumnbyname() 242

getmoneycolumnbyindex() 246
getmoneycolumnbyname() 247
getrowid() 247
getstringcolumnbyindex() 241
getstringcolumnbyname() 242
gettimestampcolumnbyindex() 241
gettimestampcolumnbyname() 242
カレンダ 268
カレンダ関数 268
business() 220
businessday() 220
weekendday() 235
カンマ区切りの構文 46

き

キー・フィールド・ルール 43
キーワード 253

く

クエリ
FROM 句 111
GROUP BY 句 48, 114
GROUP FILTER 句 48, 115
GROUP ORDER BY 句 48, 116
HAVING 句 48
KEEP 句 100
MATCHING 句 47, 118
ON 句 121
SELECT 121
UNION 演算子 40, 123
WHERE 句 39, 125
基本的な構文 109
クエリの結合 40
クエリの構築
クエリの組み合わせ 39
データのフィルタリング 39
データの集約 39
パターン一致ルールの使用 39
複数データソースのジョイン 39
グループ・フィルタリング関数
rank() 248

し

ジョイン
ANSI 構文 44
カーディナリティ 41
キー・フィールド・ルール 43
ストリームとウィンドウの簡単な左ジョインの例 44
ストリームとウィンドウの複雑なジョインの例 44
タイプ 41
簡単な左ジョインの例 44
簡単な全外部ジョインの例 44
簡単な内部ジョインの例 44
複雑なジョインの例 44, 46
例 41

す

スカラ
acos() 168
ascii() 200
asin 169
atan() 169
atan2() 169
avgof() 170
base64_binary() 201
base64_string() 201
bitand() 171
bitclear() 171
bitflag() 172
bitflaglong() 172
bitmask() 172
bitmasklong() 173
bitnot() 173
bitor() 174
bitset() 174
bitshiftleft() 175
bitshiftright() 175
bittest() 176
bittoggle() 176
bitxor() 177
business() 220
businessday() 220
cast() 202
cbrt() 178
ceil() 178
char 203
compare() 178
concat() 203
cos() 179
cosd() 179

cosh() 180
date() 221
dateceiling() 221
datefloor() 223
dateint() 238
datename() 225
datepart() 225
dateround() 226
dayofmonth() 227
dayofweek() 228
dayofyear() 229
distance() 180
distancesquared() 181
exp() 241
extract() 204
floor() 181
fromnetbinary() 204
hex_binary() 205
hex_string() 205
hour() 229
int32() 190
intdate() 230
isnull() 182
left() 191
length() 182
like() 191
ln() 183
log10() 183
log2() 183
logx() 184
lower() 192
ltrim() 192
makebigdatetime() 230
maxof() 184
microsecond() 231
minof() 185
minute() 232
month() 232
msecToTime() 206
nextval() 185
now() 248
patindex() 193
pi() 186
power() 186
random() 186
real() 194
regexp_firstsearch() 194
regexp_replace() 195
regexp_search() 196
replace() 196
right() 197
round() 187
second() 233
secToTime() 206
sign() 187
sin() 188
sind() 250
sinh() 188
sqrt 188
string() 198
substr() 198
sysbigdatetime() 250
sysdate() 234
systimestamp() 234
tan() 189
tanh() 190
timeToMsec() 207
timeToSec() 208
timeToUsec() 207
to_bigdatetime() 209
to_binary() 208
to_boolean() 210
to_date() 210
to_float() 211
to_integer() 212
to_interval() 212
to_long() 213
to_money() 213
to_string() 215
to_timestamp() 217
to_xml() 214
tonetbinary() 214
totimezone() 251
trim() 199
trunc() 199
unbigdatetime() 234
undate() 235
usecToTime() 217
weekendday() 235
xmlconcat() 218
xmlelement() 218
xmlparse() 219
xmlserialize() 219
year() 236
スカラ関数 168
rtrim() 197
upper() 200

スキーマ 21
　概要 6
スコープ
　モジュール 58
スタジオ
　概要 10
ステートフル要素 17
ステートレス要素
　デルタ・ストリーム 72
ストア
　メモリ・ストア 22
　ログ・ストア 22, 63
ストリーム 14, 16, 66
　エラー 64, 74
　スキーマ 6, 21
　ローカル 16, 87
　概要 5
　構造 6, 21
　出力 16, 87
　入力 16, 87
スラック
　カウント基準の保持 19
　パフォーマンス 19

せ

セット関数
　avgof() 170
　coalesce() 238
　firstnonnull() 241
　maxof() 184
　minof() 185

そ

その他の関数 236

た

タイム・ゾーン 255, 257
タイムスタンプ
　フォーマット・コード 264

て

データのフィルタリング 39
データフロー・プログラミング
　概要 3
　例 3
データベース
　Sybase Event Stream Processor との比較 2
データ型
　Event Stream Processor でサポートされるデータ型 26
データ経過期間
　AGING 句 95
テキスト・オーサリング
　概要 10
デルタ・ストリーム 14, 20, 72

は

バイナリ関数
　base64_binary() 201
　base64_string() 201
　bitand() 171
　bitclear() 171
　bitflag() 172
　bitflaglong() 172
　bitmask() 172
　bitmasklong() 173
　bitnot() 173
　bitor() 174
　bitset() 174
　bitshiftleft() 175
　bitshiftright() 175
　bittest() 176
　bittoggle() 176
　bitxor() 177
　concat() 203
　extract() 204
　fromnetbinary() 204
　hex_binary() 205
　hex_string() 205
　length() 182
　tonetbinary() 214
パターン一致 47
パフォーマンス
　カウント基準の保持 19
　スラック値 19
パラメータ 53
　モジュール内 61
　実行時のパラメータの初期化 53

パラメータの宣言
　パラメータ 53

ひ

ビジュアル・オーサリング
　概要 10
ビット処理関数
　bitand() 171
　bitclear() 171
　bitflag() 172
　bitflaglong() 172
　bitmask() 172
　bitmasklong() 173
　bitnot() 173
　bitor() 174
　bitset() 174
　bitshiftleft() 175
　bitshiftright() 175
　bittest() 176
　bittoggle() 176
　bitxor() 177

ふ

ファイル
　カレンダ 268
フィルタ
　WHERE 句 125
フォーマット・コード
　bigdatetime 264
　タイムスタンプ 264
　日付 264
　日付／時刻 264
フレックス・ストリーム 75
フレックス演算子 57
　CREATE FLEX 文 75
プロジェクト
　開発タスク・フロー 13
　概要 4
プロジェクトの基本コンポーネント
　クエリ 109

め

メモリ・ストア 22
　CREATE MEMORY STORE 文 82

も

モジュール
　パラメータ 61
　ロード 59, 61, 92
　規則 58
　作成 59, 61, 84
　使用 59, 61
モジュール性 59, 61, 92, 98, 101, 102, 107
　CREATE MODULE 文 84
　概要 58
モニタリング 66

ろ

ローカル 16
ログ・ストア
　CREATE LOG STORE 文 80
　CREATE MEMORY STORE 文 82
　リカバリ後のステータス 63
　ログ・ストアのループ 22
　機能 63
　最適化手法 64