User Guide

by *M Gaffiero*

gaffie@users.sourceforge.net

Pequel ETL

2.4-6

Table of Contents
Pequel ETL

OVERVIEW — WHAT IS PEQUEL?

Pequel is a comprehensive system for high performance data file processing and transformation. It features a simple, user-friendly event driven scripting interface that transparently generates, builds and executes highly efficient data-processing programs. By using the ***Pequel*** scripting language, the user can create and maintain complex data transformation processes quickly, easily, and accurately. Incidentally, the name *pequel* is derived from *perl'ish sequel*.

The ***Pequel*** system can be used by both technical (programmers) and non-technical end users. For non-technicasl users the ***Pequel*** scripting language is simple to learn and ***Pequel*** will transperantly generate, build and execute the transformation process. For developers the generated transformation program can be examined and extended, though this is rarely necessary as the scripting language contains constructs that are powerfull enough to handle even the most complex tranformation process. A Perl module ***Pequel.pm*** is provided for developers which will allow the creation of ***Pequel*** processes within Perl programs.

The ***Pequel*** scripting language is both simple and powerfull. It is event driven with each event defining a specific stage in the overall transformation process. Each event section is filled in systematically by a list of *items*. These items can be *condition statements*, *field names*, *property settings*, *aggregation statements*, *calculation statements*, and so on. A full and comprehensive array of *aggregates* and *macros* are available. Perl statements and regular expressions can be embedded within ***Pequel*** statements.

Pequel generates highly efficient Perl and C code. The generated code is as efficient as hand-written code. The emphasis in the generated code is performance — to process maximum records in minimum time. The generated code can be dumped into a program file and executed independently of ***Pequel***.

The ***Pequel*** scipt is self-documenting via ***pequeldoc***. ***Pequel*** will automatically generate the Pequel Script Programmer's Reference Manual in pdf format. This manual contains detailed and summarised information about the script, and includes cross-reference information. It will also contain an optional listing of the generated program.

Pequel is installed as a Perl module.

Pequel currently supports the following incoming data stream formats: variable length delimited, CVS, fixed length, Apache CLF, and anything else that Perl *pack/unpack* can handle.

Pequel has a multitude of uses:

Selecting Columns
Use ***Pequel*** to output selected columns from an input data stream.

Selecting Records
Output selected records based on filtering conditional statements. Full Perl regular expressions are available.

Deriving New Columns
Derive new columns using simple to complex expressions. Perform calculations on input fields to generate new (derived) fields, using Perl expressions. Calculations can be performed on both numeric fields (mathematical) and string fields (such as concatenation, substr, etc).

Grouping and Aggregating Data
Records with similar characteristics can be grouped together. Calculate aggregations, such as max, min, mean, sum, and count, on grouped record sets.

In-Memory Sort-less Aggregation
Grouping can be performed in memory on unsorted input data using the `hash` option.

Statistics
Pequel provides a comprehensive array of statistical aggregate functions.

Data Cleansing
Pequel can be effectively used for checking and resolving invalid data.

Data Frequency/Quality Analysis
TBD

Data Conversion
Convert data using any of the built-in macros and Perl regular expressions. Perform any kind of data conversion. These include, converting from one data type to another, reformatting, case change, splitting a field into two or more fields, combining two or more fields into one field, converting date fields from one date format to another, padding, etc.

Distributed Data Processing
Data can be distributed based on conditions to multiple ***Pequel*** processes.

Combining Data
Data output from multiple ***Pequel*** processes can be combined into the incoming data stream.

Merging Data
Data from any number of external files or other ***Pequel*** processes can be merged via the ***Pequel*** *tables* facility.

Piped Data Processing
The output from one ***Pequel*** process can be piped into a second ***Pequel*** process simply by specifying the first script name as the *input_file* property for the second script.

Array Fields
Pequel supports *array* fields and provides a comprehensive set of array *macros* to manipulate or generate array fields.

Database Connectivity
Direct access to database (Oracle, Sqlite, etc) tables via the ***Pequel*** *table* facility. Pequel will generate low level database API code. Currently supported databases are Oracle (via OCI), and Sqlite.

USAGE

pequel scriptfile.pql **<** *file_in* **>** *file_out*
Execute ***pequel*** with *scriptfile.pql* script to process *file_in* data file, resulting in *file_out*. The *scriptfile.pql* will contain the transformation instructions.

pequel -c scriptfile.pql
Check the syntax of the pequel script *scriptfile.pql*.

pequel -viewcode scriptfile.pql
Generate and display the code for the pequel script *scriptfile.pql*.

pequel -dumpcode scriptfile.pql
Generate the pequel code for the script *scriptfile.pql* and save generated code in the file *scriptname.pql.2.code*.

pequel -v
Display version informatio for ***Pequel***.

pequel -usage
Display Pequel usage command summary.

pequel -pequeldoc pdf -detail scriptfile.pql
Generate the Script Reference document in pdf format for the Pequel script *scriptfile.pql*. The document will include a section showing the generated code (***-detail***).

QUICK START

Create Pequel Script

Use your prefered text editor to create a pequel script *myscript.pql*. Syntax highlighting is available for ***vim*** with the ***pequel.vim*** syntax file (in *vim/sytnax*) — copy the ***pequel.vim*** file into the *syntax* directory of the *vim* installation.

All that is required is to fill in, at least, the ***output section***, or specify ***transfer*** option. The ***transfer*** option will have the effect of copying all input field values to the output. This is effectively a *straight through* process — the resulting output is identical to the input.

```
options
    transfer

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION

output section
```

Check The Pequel Script

Do a syntax check on the script by using the Pequel `-c` option. This should return the words *`myscript.pql`* `Syntax OK`.

```
pequel -c myscript.pql
myscript.pql Syntax OK
```

Dump and View The Generated Perl Program

Optionally, the generated Perl program can be *dumped* and viewed. The program will be dumped in a file with the same name and path as the script with a *'.2.code'* suffix.

```
pequel -dumpcode myscript.pql
Processing pequel script 'myscript.pql'..................
->myscript.pql.2.code
```

Run The Pequel Script

If syntax check is ok, run the script — the *sample.data* data file in the *examples* directory can be used:

`pequel myscript.pql` < *inputdata* > *outputdata*

TUTORIAL

Select A Subset Of Records

We next do something *usefull* to transform the input data. Create a filter to output a subset of records, consisting of records which have `LOCATION` starting with `10`. The filter example uses a Perl regular expression to match the `LOCATION` field content with the Perl regular expression `=~ /^10/`. This is specified in the ***filter*** section. Check and run the updated script as instructed above:

```
options
    transfer

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION

filter
    LOCATION =~ /^10/
```

Create New Derived Fields

Create additional, derived fields based on the other input fields. In our example, two new fields are added `COST_VALUE` and `SALES_VALUE`. Derived fields must be specified in the input section *after* the last input field. The derived field name is followed by the `=>` operator, and a calculation expression. Derived fields will also be output when the ***transfer*** options is specified.

```
options
    transfer

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION,
    COST_VALUE => COST_PRICE * QUANTITY,
    SALES_VALUE => SALES_PRICE * QUANTITY

filter
    LOCATION =~ /^10/

output section
```

Select Which Fields To Output

In the above examples, the output record has the same (field) format as the input record, plus the additional derived fields. In the following example we select which fields to output, and their order, on the output record. To do this we need to remove the ***transfer*** option, and create the ***output section***. The output fields `PRODUCT`, `LOCATION`, `DESCRIPTION`, `QUANTITY`, `COST_VALUE`, and `SALES_VALUE` are specified to create a new output format. In this example, all the output field names have the same name as the input fields.

```
options

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION,
```

```
    COST_VALUE => COST_PRICE * QUANTITY,
    SALES_VALUE => SALES_PRICE * QUANTITY

filter
    LOCATION =~ /^10/

output section
    string PRODUCT      PRODUCT,
    string LOCATION     LOCATION,
    string DESCRIPTION  DESCRIPTION,
    numeric QUANTITY    QUANTITY,
    decimal COST_VALUE  COST_VALUE,
    decimal SALES_VALUE SALES_VALUE
```

Group Records For Analysis

Records with similar characteristics can be grouped together, and aggregations can then be performed on the grouped records' data. The following example groups the records by `LOCATION`, and *sums* the `COST_VALUE` and `SALES_VALUE` fields within each group. Grouping is activated by creating a ***group by*** section. Input data must also be sorted on the grouping field(s). If the data is not pre-sorted then this needs to be done in the script by creating a ***sort by*** section. Alternatively, by specifying the ***hash*** option, the input data need not be sorted.

```
options

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION,
    COST_VALUE => COST_PRICE * QUANTITY,
    SALES_VALUE => SALES_PRICE * QUANTITY

filter
    LOCATION =~ /^10/

sort by
    LOCATION

group by
    LOCATION

output section
    string LOCATION     LOCATION,
    string PRODUCT      PRODUCT,
    string DESCRIPTION  DESCRIPTION,
    numeric QUANTITY    QUANTITY,
    decimal COST_VALUE  sum COST_VALUE,
    decimal SALES_VALUE sum SALES_VALUE
```

Select A Subset Of Grouped Records

A subset of groups can be select by creating a ***having*** section. The ***having*** section is similar to the ***filter*** section, but instead is applied to the aggregated group of records. In this example we will output only records for locations which have a total `SALES_VALUE` of `1000` or more. Note that `SALES_VALUE` in the ***having*** section refers to the output field (`sum SALES_VALUE`) and not the input field with same name (`SALES_PRICE * QUANTITY`). The ***having*** section gives preference to output fields when interpreting field names.

```
options

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION,
    COST_VALUE => COST_PRICE * QUANTITY,
    SALES_VALUE => SALES_PRICE * QUANTITY

filter
```

```
    LOCATION =~ /^10/

sort by
    LOCATION

group by
    LOCATION

output section
    string LOCATION     LOCATION,
    string PRODUCT      PRODUCT,
    string DESCRIPTION  DESCRIPTION,
    numeric QUANTITY    QUANTITY,
    decimal COST_VALUE  sum COST_VALUE,
    decimal SALES_VALUE sum SALES_VALUE

having
    SALES_VALUE >= 1000
```

Aggregation Based On Conditions

Output fields can be aggregated conditionally. That is, the aggregation will only occur for records, within the group, that evaluate the condition to *true*. This is done by adding a `where` clause to the aggregate function. In this example we create three new output fields `SALES_VALUE_RETAIL`, `SALES_VALUE_WSALE` and `SALES_VALUE_OTHER`. These fields will contain the sales value for records within the group which have sales code equal to ‘R‘, ‘W‘, and other codes, respectively.

```
options

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION,
    COST_VALUE => COST_PRICE * QUANTITY,
    SALES_VALUE => SALES_PRICE * QUANTITY

filter
    LOCATION =~ /^10/

sort by
    LOCATION

group by
    LOCATION

output section
    string LOCATION            LOCATION,
    string PRODUCT             PRODUCT,
    string DESCRIPTION         DESCRIPTION,
    numeric QUANTITY           QUANTITY,
    decimal COST_VALUE         sum COST_VALUE,
    decimal SALES_VALUE        sum SALES_VALUE,
    decimal SALES_VALUE_RETAIL sum SALES_VALUE where SALES_CODE eq ’R’,
    decimal SALES_VALUE_WSALE  sum SALES_VALUE where SALES_CODE eq ’W’,
    decimal SALES_VALUE_OTHER  sum SALES_VALUE where SALES_CODE ne ’R’ and SALES_CODE ne ’W’
```

Derived Fields Based On Output Fields

An output derived field, the calculation of which is based on *output* fields, can be created by declaring an output field with the = *calulation expression*.

```
options

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION,
    COST_VALUE => COST_PRICE * QUANTITY,
    SALES_VALUE => SALES_PRICE * QUANTITY
```

```
filter
    LOCATION =~ /^10/

sort by
    LOCATION

group by
    LOCATION

output section
    string LOCATION            LOCATION,
    string PRODUCT             PRODUCT,
    string DESCRIPTION         DESCRIPTION,
    numeric QUANTITY           QUANTITY,
    numeric TOTAL_QUANTITY     sum QUANTITY,
    decimal COST_VALUE         sum COST_VALUE,
    decimal SALES_VALUE        sum SALES_VALUE,
    decimal SALES_VALUE_RETAIL sum SALES_VALUE where SALES_CODE eq 'R',
    decimal SALES_VALUE_WSALE  sum SALES_VALUE where SALES_CODE eq 'W',
    decimal SALES_VALUE_OTHER  sum SALES_VALUE where SALES_CODE ne 'R' and SALES_CODE ne 'W',
    decimal AVG_SALES_VALUE    = SALES_VALUE / TOTAL_QUANTITY
```

Note

In order to protect against a divide by zero exception, the AVG_SALES_VALUE field would actually be better declared as follows. This form uses a Perl *alternation* ?: operator. If TOTAL_QUANTITY is zero, it will set AVG_SALES_VALUE to zero, otherwise it will set AVG_SALES_VALUE to SALES_VALUE / TOTAL_QUANTITY. Thus, the division will only be performed on non-zero TOTAL_QUANTITY.

```
decimal AVG_SALES_VALUE    = TOTAL_QUANTITY == 0 ? 0.0 : SALES_VALUE / TOTAL_QUANTITY
```

Create Intermediate (Transparent) Output Fields

In the previous example, supposing that the TOTAL_QUANTITY field was not required in the output, it could be made *transparent* by declaring it with an *underdash* (_) prefix. Transparent output fields are usefull for creating intermediate fields required for calculations.

```
numeric _TOTAL_QUANTITY    sum QUANTITY,
decimal AVG_SALES_VALUE    = SALES_VALUE / _TOTAL_QUANTITY
```

Cleaning Data

Data can be cleaned in a variety of ways, and invalid records placed in a *reject* file. The following example determines the validity of a record by a) the length of certain fields, and b) the content of field QUANTITY. The PRODUCT and LOCATION fields must be at least 8 and 2 characters long, respectively; the QUANTITY field must contain only numeric digits, decimal point and minus sign. The rejected records will be placed in the reject file called *scriptname*.reject

```
options
    transfer

input section
    PRODUCT,
    COST_PRICE,
    DESCRIPTION,
    SALES_CODE,
    SALES_PRICE,
    QUANTITY,
    SALES_DATE,
    LOCATION

reject
    length(PRODUCT) < 8 || length(LOCATION) < 2,
    QUANTITY !~ /^[0-9\.\-]+$/
```

Converting Data

Any sort of data conversion can be performed. These include, converting from one data type to another, reformatting, case change, splitting a field into two or more fields, combining two or more fields into one field, converting date fields from one date format to another, padding, etc. The following script demonstrates these data conversions.

```
options

input section
```

```
        PRODUCT,
        COST_PRICE,
        DESCRIPTION,
        SALES_CODE,
        SALES_PRICE,
        QUANTITY,
        SALES_DATE,
        LOCATION

    output section
        string PRODUCT_U     = &uc(PRODUCT), // Convert case to upper
        string DESCRIPTION_U = &uc(DESCRIPTION), // Convert case to upper
        string PCODE_1       = &substr(PRODUCT,0,2), // Split field
        string PCODE_2       = &substr(PRODUCT,2,4), //  ""
        string ANALYSIS_1    = SALES_CODE . sprintf("%08d", COST_PRICE), // Combine fields
        string S_QUANTITY    = sprintf("%08d", QUANTITY) // Reformat/Convert field
        string NEW_PRODUCT   = PCODE_2 . PCODE_1 . &substr(PRODUCT,6) // Reformat
        decimal SALES_PRICE  SALES_PRICE // no change
        decimal SALES_CODE   SALES_CODE  // no change
        string LOCATION      LOCATION // no change
```

Using Date Fields

TBC

Counting Records

TBC

Extracting *n* Distinct Values For A Field

TBC

Tabulating Data

TBC

Statistical Analysis

TBC

Declaring And Using Tables For Value Lookup

TBC

Using External Tables

TBC

Using Date Fields

TBC

Create A Summary Report

TBC

Using Array Fields

TBC

Database Tables: oracle

TBC

Database Tables: sqlite

TBC

Merg Database Tables

TBC

View The Generated Perl Code

To view the generated Perl code use the Pequel `-viewcode` option:

```
pequel -viewcode scriptname.pql | more
```

Dump The Generated Perl Code

To dump the generated Perl code use the Pequel `-dumpcode` option. This will save the generated Perl program in the file with the name *script_name**.2.code***. So, if your script is called *myscript.pql* the resulting generated Perl program will be saved in the the file *myscript.pql.2.code*, in the same path:

```
pequel -dumpcode scriptname.pql
```

Produce The Script Specification Document

Use the Pequel `-pequeldoc pdf` option to produce a presentation script specification for the Pequel script. The generated pdf document will be saved in a file with the same name as the script but with the file extension changed from *pql* to *pdf*.

```
pequel scriptname.pql -pequeldoc pdf
```

Use the `-detail` option to include the generated code in the document.

```
pequel scriptname.pql -pequeldoc pdf -detail
```

Display Summary Information For Script

This options will display the parsed details from the script in a summarised format.

```
pequel scriptname.pql -list
```

COMMAND LINE OPTIONS

—prefix, —prefix_path
Prefix for filenames directory path

—verbose, —ver
Display progress counter

—noverbose, —silent, —quite
Do not display progress counter

—input_file, —is, —if, —i
Input data filename

—usage
Display command usage description

—output_file, —os, —of, —o
Output data filename

—script_name, —script, —s, —pql
Script filename

—header
Write header record to output.

—pequeldoc, —doc
Generate pod / pdf pequel script Reference Guide.

—viewcode, —vc
Display the generated Perl code for pequel script

—dumpcode, —dc, —diag
Dump the generated Perl code for pequel script

—syntax_check, —c, —check
Check the pequel script for syntax errors

—version, —v
Display Pequel Version information

—table_info, —ti
Display Table information for all tables declared in the pequel script

cpp_cmd, cpp_args
Override the default *cpp* command name and any additional agruments required.

PEQUEL LANGUAGE REFERENCE

A Pequel script is divided into sections. Each section begins with a section name, which appears on a line on its own, followed by a list of items. Each *item* line must be terminated by a newline comma (or both). In order to split an *item* line into mutiple lines (for better readability) use the line continuation character \.

Pequel is *event* driven. Each section within an Pequel script describes an event. For example, the ***input section*** is activated whenever an input record is read; the ***output section*** is activated whenever an aggregation is performed.

The sections must appear in the order described below. A minimal script must contain ***input section*** and ***output section***, or, ***input section*** and ***transfer*** option. All other sections are optional, and need only appear in the Pequel script if they contain statements.

The main sections are ***input section*** and ***output section***. The ***input section*** defines the format, in fields, of the input data stream. It can also define new calculated (derived) fields. The ***output section*** defines the format of the output data stream. The ***output section*** is required in order to perform aggregation. The ***output section*** will consist of input fields, aggregations based on grouping the input records, and new calculated fields.

Input sorting can be specified with the ***sort by*** section. Break processing (grouping) can be specified with the ***group by*** section. Input filtering is specified with the ***filter*** section. Groups of records can be filtered with the ***having*** section.

A powerfull feature of Pequel is its built-in tables feature. Tables, consisting of key and value pairs. Tables are used to perform merge and joins on multiple input datasources. They can also be used to access external data for cross referencing, and value lookups.

Pequel also handles a number of date field formats. The ***&date()*** macro provides access to date fields.

Field Names

Field names are case-sensitive and must begin with an alphabetical or **'_'** character, and can contain a mix of alphabetical, numerical and **'_'**s characters. Field names are declared in the *input section* and the *output section*. The same field name can only be declared once within each section type. Field names in the *output section* can have the same name as an *input section* field name. Field names within the *output section* that begin with a **'_'** character are *intermediate* fields — these fields can be referenced in calculation expressions but will not appear in the output stream.

Statements

A ***Pequel*** statement can contain a mix of Perl code, including regular expressions, field names, Pequel-Macro calls, and Pequel-Table calls. The ***Pequel*** compiler will first parse the statement for ***Pequel*** field names, macros and table names, and translate these into Perl code.

Comments

Any text following and including the # symbol or `//` is considered as comment text. If the *cpp* preprocessor is available then comments are limited to C style comments with (`//` and `/* ... */`) — the # becomes a macro directive.

Statement Line Continuation

Each item within a section must appear on a single line. In order to break up an item statement (for better readability) us the line continuation character \.

Pre Processor

If your system provides the ***cpp*** preprocessor, your Pequel script may include any C/C++ style macros and defines.

Section Types

The following table describes the different *section types*.

options
Specify properties.

description
This section contains free-format text to describe the function of the script.

use package
Specify any external Perl package modules.

input section
The items within this section consist of input data stream field names followed by any derived field definitions.

field preprocess
Specify any input field pre-processing which will occur before the field is referenced by any derived field.

filter
The *filter* section specifies one or more condition item statements which will be used to match incoming data records and filter out any records that do not match **all** the condition item statements.

reject
The *reject* section specifies one or more condition item statements which will be used to match incoming data records and filter out any records that do not match **any** of the condition item statements.

divert input record
If the input record matches any of the condition item statements then *divert* the record to the specified ***Pequel*** process or file.

copy input record
If the input record matches any of the condition item statements then *copy* the record to the specified ***Pequel*** process or file.

display message on input
If the input record matches any of the condition item statements then display the specified message to *stderr*.

display message on input abort
If the input record matches any of the condition item statements then display the specified message to *stderr* then exit the process.

sort by
The *sort by* section contains a list of input field items with optional type and sort order specifications. These fields specify the sort ordering for the input data stream.

group by
The *group by* section contains a list of input field items with optional type specification. These fields specify the grouping requirements for the input data stream.

dedup on

The *dedup on* section contains a list of input field items with optional type specification. Consecutive incoming records that contain the same values within these fields will be de-duped, i.e. only one record for each duplicate set will be processed.

output section
The *output section* contains a list of output field definitions.

field postprocess
Specify any output field post-processing.

having
The *having* section specifies one or more condition item statements which will be used to match output data records and filter out any records that do not match all the condition item statements.

divert output record
If the output record matches any of the condition item statements then *divert* the record to the specified ***Pequel*** process or file.

copy output record
If the output record matches any of the condition item statements then *copy* the record to the specified ***Pequel*** process or file.

display message on output
If the output record matches any of the condition item statements then display the specified message to *stderr*.

display message on output abort
If the output record matches any of the condition item statements then display the specified message to *stderr* then exit the process.

init table
Initialise local tables.

load table
Load and initialise external tables.

load table pequel
Load table from output of external ***Pequel*** script.

OPTIONS SECTION

This section is used to declare various options described in detail below. Options define the overall character of the data transformation.

Format
options
<option> [**(**<arg>**)**] [, ...]

Example

```
options
    input_delimiter(\s+), # one or more space(s) delimit input fields.
    verbose(100000), # print progress on every 100000'th input record.
    optimize,
    varnames,
    default_date_type(DD/MM/YY),
    nonulls,
    diag
```

verbose

Set the verbose option to display progress information to STDERR during the transform run. Requires one parameter. This will instruct Pequel to display a counter message on specified number of records read from input.

silent
Supress all processing messages to *stderr*.

prefix
Specify a prefix path. The prefix will be used with all external file names unless the name starts with a '/'.

input_delimiter
Specify the character that is used to delimit columns in the input data stream. This is usually the pipe | character, but can be any character including the space character. For multiple spaces use `\s+`, and for multiple tabs use `\t+`. This input delimiter will default to the pipe character if *input_delimiter* is not specified.

output_delimiter
Specify the character that will delimit columns in the output. The output delimiter will default to the input delimiter if not specified. Refer to *input_delimiter* above for more information regarding types of delimiters.

discard_header
If the input data stream contains an initial header record then this option must be specified in order to discard this record from the processing.

input_file
Specify the file name as a parameter. If specified, the input data will be read from this file; otherwise it will be read from STDIN. If the *input_file* option contains a ***Pequel*** script name (anyting ending in *.pql*) then the output from executin this input script will be chained to produce the input data stream.

output_file
Specify the file name as a parameter. If specified, the output will be written to this file (the file will be overwritten!); otherwise it will be sent to STDOUT.

transfer
Copy the input record to output. The input record is copied as is, including calculated fields, to the output record. Fields specified in the ***output section*** are placed after the input fields. The `transfer` option is not available when ***group by*** us in use.

hash
Use hash processing mode. Hash mode is only available when break processing is activated with 'group by'. In hash mode input data need not be sorted. Because this mode of processing is memory intensive, it should only be used when generating a small number of groups. The optional 'numeric' modifier can be specified to sort the output numerically; if not specified, a string sort is done.

header
If specified then an initial header record will by written to output. This header record contains the output field names. By default a header record will be output if neither header nor noheader is specified.

noheader
Specify this option to suppress writing of header record.

addpipe
Specify this option to add an extra delimiter character after the last field. This is the default action if neither addpipe nor noaddpipe is specified.

noaddpipe

Specify this option to suppress adding an extra delimiter character after the last field.

optimize
If specified the generated Perl code will be optimized to run more efficiently. This optimisation is done by grouping similar `where` conditions into `if-else` blocks. Thus if a number of where clauses contain the same condition, these statements will be grouped under one if condition. The *optimize* option should only be used by users with some knowledge of Perl.

nooptimize
Specify this option to prevent code from being optimised. This is the default setting.

nulls
If specified, numeric and decimal values with a zero/null value will be output as null character. This is the default setting.

nonulls
If specified, numeric and decimal values with a zero/null value will be output as `0`.

reject_file
Use this option to specify a file name to contain the rejected records. These are records that are rejected by the filter specified in the reject section. If no reject file option is specified then the default reject file name is the script file name with `.reject` appended.

dumpcode
Set this option to save the generated code in scriptname.2.code files. The scriptname.2.code file contains the generated perl code. This latter contains the actual Perl program that will process the input data stream. This generated Perl program can be executed independatly of Pequel.

default_date_type
Specify a default date type. Currently supported date types are: `YYYYMMDD`, `YYMMDD`, `DDMMYY`, `DDMMMYY`, `DDMMYYYY`, `DD/MM/YY`, `DD/MM/YYYY`, and US date formats: `MMDDYY`, `MMDDYYYY`, `MM/DD/YY`, `MM/DD/YYYY`. The `DDMMMYY` format refers to dates such as `21JAN02`.

default_list_delimiter
Specify the default list delimiter for *array fields* created by `values_all` and `values_uniq` aggregates. Any delimiter specified as a parameter to the aggregate function will override this.

rmctrlm *v3*
If the input file is in DOS format, specify 'rmctrlm' option to remove the Ctrl-M at end of line.

input_record_limit *v3*
Specify number of records to process from input file. Processing will stop after the number of records as specified have been read.

suppress_output *v3*
Use this option when ***summary section*** is used to prevent output of raw results.

pequeldoc
Generate PDF for Programmer's Reference Manual for the Pequel script. The next three options are also required.

doc_title
Specify the title that will appear on the pequeldoc generated manual.

doc_email
Specify the user's email that will appear on the pequeldoc generated manual.

doc_version
Specify the Pequel script version number that will appear on the pequeldoc generated manual.

gzcat_cmd, gzcat_args
Override the default *gzcat* command name and any additional agruments required.

cat_cmd, cat_args
Override the default *cat* command name and any additional agruments required.

sort_cmd, sort_args
Override the default *sort* command name and any additional agruments required.

pack_output, output_pack_fmt
The output data stream can be packed using the format specified in the *output_pack_fmt*. These properties can also be used to produce *fixed format* and *binary* output. The default format is *A3/Z** repeated for each output field. Please refer to the Perl *perlpacktut* manual for a detailed desctiption of formats.

unpack_input, input_pack_fmt
The packed input data stream can be unpacked using the format specified in the *unput_pack_fmt*. These properties can also be used to input *fixed format* and *binary* input. The default format is *A3/Z** repeated for each input field. Please refer to the Perl *perlpacktut* manual for a detailed desctiption of formats.

INLINE OPTIONS

The following options require that the Inline::C Perl module and a C compiler system is installed on your system.

use_inline
The ***use_inline*** option will instruct Pequel to generate (and compile/link) ***C*** code — replacing the input file identifier inside the main ***while*** loop by a ***readsplit()*** function call. The ***readsplit*** function is implemented in ***C***.

input_delimiter_extra
Specify one or more extra field delimiter characters. These may be one of any quote character, ***'***, ***"***, ***`***, and optionally, one of and bracket character, ***{***, ***[***, ***(***. For example, this option can be used to parse input Apache log files in CLF format:

```
 options
     input_delimiter_extra("[)  // Apache CLF log quoted fields and bracketed timestamp
```

inline_clean_after_build
Tells Inline to clean up the current build area if the build was successful. Sometimes you want to DISABLE this for debugging. Default is 1.

inline_clean_build_area
Tells Inline to clean up the old build areas within the entire Inline DIRECTORY. Default is 0.

inline_print_info
Tells Inline to print various information about the source code. Default is 0.

inline_build_noisy
Tells ILSMs that they should dump build messages to the terminal rather than be silent about all the build details.

inline_build_timers
Tells ILSMs to print timing information about how long each build phase took. Usually requires Time::HiRes

inline_force_build
Makes Inline build (compile) the source code every time the program is run. The default is 0.

inline_directory
The DIRECTORY config option is the directory that Inline uses to both build and install an extension.
Normally Inline will search in a bunch of known places for a directory called '.Inline/'. Failing that, it will create a directory called '_Inline/'
If you want to specify your own directory, use this configuration option.
Note that you must create the DIRECTORY directory yourself. Inline will not do it for you.

inline_CC
Specify which compiler to use.

inline_OPTIMIZE
This controls the MakeMaker OPTIMIZE setting. By setting this value to '-g', you can turn on debugging support for your Inline extensions. This will allow you to be able to set breakpoints in your C code using a debugger like gdb.

inline_CCFLAGS
Specify extra compiler flags.

inline_LIBS
Specifies external libraries that should be linked into your code.

inline_INC
Specifies an include path to use. Corresponds to the MakeMaker parameter.

inline_LDDLFLAGS
Specify which linker flags to use.
NOTE: These flags will completely override the existing flags, instead of just adding to them. So if you need to use those too, you must respecify them here.

inline_MAKE
Specify the name of the 'make' utility to use.

USE PACKAGE SECTION

Use this section to specify Perl packages to use. This section is optional.

Format
use package
<Perl package name> [, ...]

Examples

```
use package
    Benchmark,
    EasyDate
```

INIT TABLE SECTION

Use ***init table*** to initialise tables in the Pequel script. This will consist of a list of table name followed by key value (or value list) pairs. The key must not contain any spaces. In order to avoid clutter in the script, use load table as described above. To look up a table key/value use the ***%table name(key)*** syntax. Table column values are accessed by using the ***%table name(key)*****-**=>*n* syntax, when n refers to a column number starting from '1'. The column specification is not required for single value tables. All entries within a table should have the same number of values, empty values can be declared with a null quoted value (''). This section is optional.

Format

init table
<table> <key> <value> [, <value>...]

Example

```
init table
// Table-Name Key-Value Field->1             Field-2  Field-3
   LOCINFO    NSW       'New South Wales'    '2061'   '02'
   LOCINFO    WA        'Western Australia'  '5008'   '07'
   LOCINFO    SA        'South Australia'    '8078'   '08'

input section
   LOCATION,
   LDESCRIPT => %LOCINFO(LOCATION)->1 . " in postcode " . %LOCINFO(LOCATION)->2
```

LOAD TABLE SECTION

Use this section to declare tables that are to be initialised from an external data file. If the table is in `.tbl` format (key|value) then only the table name (without the `.tbl`) need be specified. The filename can consist of the full path name. Compressed files (ending in .gz, .z, .Z, .zip) will be handled properly. If key column is not specified then this is set to 1 by default; if the value column is not specified then this is set to 2 by default. Column numbers are 1 base. To look up a table key/value use the ***%table name(****key****)*** syntax. If the table name is prefixed with the _ character, this table will be loaded at runtime instead of compile time. Thus the table contents will not appear in the generated code. This is useful if the table contains more than a few hundred entries, as it will not clutter up the generated code.

persistant option
The ***persistant*** option will make the table disk-based instead of memory-based. Use this option for tables that are too big to fit in available memory. The disk-based table snapshot file will have the name `_TABLE_name.dat`, where `name` is the table name. When the `persistant` option is used, the table is generated only once, the first time it is used. Thereafter it will be loaded from the snaphot file. This is alot quicker and therefore usefull for large tables. In order to re-generate the table, the snapshot file must be manually deleted. In order to use the `persistant` option the Perl DB_File module must be available. The effect of `persistant` is to `tie` the table's associative array with a DBM database (Berkeley DB). Note that using `persistant` tables will downgrade the overall performance of the script.

Format
load table [**persistant**]
<table> [<filename> [<key_col> [<val_col>]]] [, ...]

Examples

```
load table
    POSTCODES
    MONTH_NAMES /data/tables/month_names.tbl
    POCODES pocodes.gz 1 2
    ZIPSAMPLE zipsample.txt 3 21
```

INPUT SECTION

This section defines the format of the input data stream. Any calculated fields must be placed after the last input field. The calculation expression must begin with => and consists of (almost) any valid Perl statement, and can include input field names. All macros are also available to calculation expressions. The input section must appear before all the sections described below. Each input field name must be unique.

Format
input section
<input field name> [=> <calculation expression>] [, ...]

Example

```
input section
    ACL,
    AAL,
    ZIP,
    CALLDATE,
    CALLS,
    DURATION,
```

```
    REVENUE,
    DISCOUNT,
    KINSHIP_KEY,
    INV => REVENUE + DISCOUNT,
    MONTH_CALLDATE => &month(CALLDATE),
    GROUP => MONTH_CALLDATE <= 6 ? 1 : 2,
    POSTCODE => %POSTCODES(AAL),
    IN_SAMPLE => exists %ZIPSAMPLE(ZIP),
    IN_SAMPLE_2 => exists %ZIPSAMPLE(ZIP) ? 'yes': 'no'
```

FIELD PREPROCESS SECTION

Use this section to perform addition formatting/processing on input fields. These statements will be performed right after the input record is read and before calculating the input derived fields.

FIELD POSTPROCESS SECTION

Use this section to perform addition formatting/processing on output fields. These statements will be performed after the aggregations and just prior to the output of the aggregated record.

SORT BY SECTION

Use this section to sort the input data by field(s). One or more sort fields can be specified. This section must appear after the ***input section*** and before the ***group by*** and ***output sections***. The ***numeric*** option is used to specify a *numeric* sort, and the ***desc*** option is used to specify a *descending* sort order. The standard Unix *sort* command is used to perform the sort. The ***numeric*** option is translated to the -n Unix *sort* option; the ***desc*** option is translated to the -r Unix *sort* option. If the input data is pre sorted then the ***sort by*** section is not required (even if break processing is activated with a ***group by*** section declaration). The ***sort by*** section is not required when the ***hash*** option is specified.

Format

sort by

<field name> [**numeric**] [**desc**] [, ...]

Examples

```
sort by
    ACL,
    AAL numeric desc
```

REJECT SECTION

Specify one or more filter expressions. Filter expression can consist of any valid Perl statement, and must evaluate to Boolean true or false (0 is false, anything else is true). It can contain input field names and macros. Each input record is evaluated against the filter(s). Records that evaluate to true on any one filter will be rejected and written to the reject file. The reject file is named scriptname.reject unless specified in the ***reject_file*** option.

Format

reject

<filter expression> [, ...]

Examples

```
reject
    !exists %ZIPSAMPLE(ZIP)
    INV < 200
```

FILTER SECTION

Specify one or more filter expressions. Filter expression can consist of any valid Perl statement, and must evaluate to Boolean true or false. It can contain input field names and macros. Each input record is evaluated against the filter(s). Only records that evaluate to true on all filter statements will be processed; that is, records that evaluate to false on any one filter statement will be discarded.

Format

filter
<filter expression> [, ...]

Examples

```
filter
    exists %ZIPSAMPLE(ZIP)
    ACL =~ /^356/
    ZIP eq '52101' or ZIP eq '52102'
```

GROUP BY SECTION

Use this section to activate break processing. Break processing is required to be able to use the aggregates in the output section. One or more fields can be specified - the input data must be sorted on the group by fields, unless the ***hash*** option is used. A break will occur when any of the group field values changes. The ***group by*** section must appear after the ***sort by*** section and before the ***output section***. The ***numeric*** option will cause leading zeros to be stripped from the input field. Group by on *calculated* input fields is usefull when the ***hash*** option is in use because the input does not need to be pre-sorted.

Format
group by
<input field name> [**numeric** | **decimal** | **string**] [, ...]

Examples

```
group by
    AAL,
    ACL numeric
```

DEDUP ON SECTION

OUTPUT SECTION

This is where the output data stream format is specified. At least one output field must be defined here (unless the ***transfer*** option is specified). Each output field definition must end with a comma or new line (or both). Each field definition must begin with a type (`numeric`, `decimal`, `string`, `date`). The output field name can be the same as an input field name, unless the output field is a calculated field. Each output field name must be unique. This name will appear in the header record (if the ***header*** option is set). The aggregate expression must consist of at least the input field name.

The aggregates `sum`, `min`, `max`, `avg`, `first`, `last`, `distinct`, `values_all`, and `values_uniq` must be followed by an input field name. The aggregates `count` and `flag` must be followed by the * character. The aggregate `serial` must be followed by a number (indicating the serial number start).

A prefix of _ in the output field name causes that field to be *transparent*; these fields will not be output, their use is mainly for intermediate calculations. <input field name> can be any field declared in the input section, including calculated fields. This section is required unless the ***transfer*** option is specified.

Format
output section
<type> <output field name> <output expression> [, ...]

<type>
numeric, **decimal**, **string**, **date** [**(**<datefmt>**)**]

<output field name>
Each output field name must be unique. Output field name can be the same as the input field name, unless the output field is a calculated field. A _ prefix denotes a *transparent* field. Transparent fields will not be output, they are used for intermediate caclulations.

<datefmt>
YYYYMMDD, **YYMMDD**, **DDMMYY**, **DDMMMYY**, **DDMMYYYY**, **DD/MM/YY**, **DD/MM/YYYY**, **MMDDYY**, **MMDDYYYY**, **MM/DD/YY**, **MM/DD/YYYY**

<output expression>

<input field name>
|
<aggregate> <input field name> [**where** <condition expression>]
|
serial <start num> [**where** <condition expression>]
|
count * [**where** <condition expression>]
|
flag * [**where** <condition expression>]
|
= <calculation expression> [**where** <condition expression>]

<aggregate>
sum | **maximum** | **max** | **minimum** | **min** | **avg** | **mean** | **first** | **last** | **distinct**
| **sum_distinct** | **avg_distinct** | **count_distinct**
| **median** | **variance** | **stddev** | **range** | **mode**
| **values_all** [**(**<delim>**)**] | **values_uniq** [**(**<delim>**)**]

<input field name>
Any field specified in the input section.

<calculation expression>
Any valid Perl expression, including input and output field names, and Pequel macros. This expression can consist of numeric calculations, using arithmetic operators (+, *, -, etc) and functions (`abs`, `int`, `rand`, `sqrt`, etc.), string calculations, using string operators (eg. `.` for concatenation) and functions (`uc`, `lc`, `substr`, `length`, etc.).

<condition expresion>
Any valid Perl expression, including input and output field names, and Pequel macros, that evaluates to true (non-zero) or false (zero).

Aggregates

sum <*input field*>
Accumulate the total for all values in the group. Output type must be ***numeric***, ***decimal*** or ***date***.

sum_distinct <*input field*>
Accumulate the total for *distinct* values only in the group. Output type must be ***numeric***, ***decimal*** or ***date***.

maximum | max <*input field*>
Output the maximum value in the group. Output type must be ***numeric***, ***decimal*** or ***date***.

minimum | min <*input field*>
Output the minimum value in the group. Output type must be ***numeric***, ***decimal*** or ***date***.

avg | mean <*input field*>
Output the average value in the group. Output type must be ***numeric***, ***decimal*** or ***date***.

avg_distinct <*input field*>
Output the average value for *distinct* values only in the group. Output type must be ***numeric***, ***decimal*** or ***date***.

first <*input field*>
Output the first value in the group.

last <*input field*>
Output the last value in the group.

count_distinct* | *distinct <*input field*>
Output the count of unique values in the group. Output type must be ***numeric***.

median <*input field*>
The median is the middle of a distribution: half the scores are above the median and half are below the median. When there is an odd number of values, the median is simply the middle number. When there is an even number of values, the median is the mean of the two middle numbers. Output type must be ***numeric***.

variance <*input field*>
Variance is calculated as follows: (*sum_squares* / *count*) - (*mean* ** 2), where *sum_squares* is each value in the distribution squared (** 2); *count* is the number of values in the distribution; *mean* is discussed above. Output type must be ***numeric***.

stddev <*input field*>
Stddev is calculated as the square-root of *variance*. Output type must be ***numeric***.

range <*input field*>
The range is the maximum value minus the minimum value in a distribution. Output type must be ***numeric***.

mode <*input field*>
The mode is the most frequently occuring score in a distribution and is used as a measure of central tendency. A distribution may have more than one mode, in which case a space delimited list is returned. Any output type is valid.

values_all <*input field*>
Output the list of all values in the group. The specified delimiter delimits the list. If not specified then the ***default_list_delimiter*** specified in options is used.

values_uniq <*input field*>
Output the list of unique values in the group. The specified delimiter delimits the list. If not specified then the ***default_list_delimiter*** specified in options is used.

serial <*n*>
Output the next serial number starting from *n*. The serial number will be incremented by one for each successive output record. Output type must be ***numeric***.

count *
Output the count of records in the group. Output type must be ***numeric***.

flag *
Output *1* or *0* depending on the result of the where condition clause. If no where clause is specified then the output value is set to *1*. The output will be set to *1* if the where condition evaluates to true at least once for all records within the group. Output type must be ***numeric***.

corr <*input field*>
New in v2.5. Returns the coefficient of correlation of a set of number pairs.

covar_pop <*input field*>
New in v2.5. Returns the population covariance of a set of number pairs.

covar_samp <*input field*>
New in v2.5. Returns the sample covariance of a set of number pairs.

cume_dist <*input field*>
New in v2.5. Calculates the cumulative distribution of a value in a group of values.

dense_rank <*input field*>
New in v2.5. Computes the rank of a row in an ordered group of rows.

rank <*input field*>
New in v2.5. Calculates the rank of a value in a group of values.

= <calculation expression>
Calculation expression follows. Use this to create output fields that are based on some calculation expression. The calculation expression can consist of any valid Perl statement, and can contain input field names, output field names and macros.

Examples

```
output section
    numeric AAL            AAL
    string  _HELLO         = 'HELLO'
    string  _WORLD         = 'WORLD'
    string  HELLO_WORLD    = _HELLO . ' ' . _WORLD
    decimal _REVENUE       sum REVENUE
    decimal _DISCOUNT      sum DISCOUNT
    decimal INVOICE        = _REVENUE + _DISCOUNT
```

HAVING SECTION

The ***having*** section is applied after the grouping performed by ***group by***, for filtering groups based on the aggregate values. Break processing must be activated using the ***group by*** section. The ***having*** section must appear after the ***output section***. Specify one or more filter expressions. Filter expression can consist of any valid Perl statement, and must evaluate to Boolean true or false. It can contain input field names, output field names and macros. Only groups that evaluate to true on all filter statements will be output; that is, groups that evaluate to false on any one filter statement will be discarded. Each filter statement must end with a comma and/or new line.

Format
having
<filter expression> [, ...]

Examples

```
having
    SAMPLE == 1
    MONTH_1_COUNT > 2 and MONTH_2_COUNT > 2
```

SUMMARY SECTION

This section contains any perl code and will be executed once after all input records have been processed. Input, output field names, and macros can be used here. This section is mostly relevant when ***group by*** is omitted, so that a `group all` is in effect. The ***suppress_output*** option should also be used. If the script contains a ***group by*** section and more than one group of records is produced, only the last group's values will appear in the summary section.

Format
summary section
< Perl code >

Examples

```
summary section
    print "*** Summary Report ***";
    print "Total number of Products:   ", sprintf("%12d", COUNT_PRODUCTS);
    print "Total number of Locations:  ", sprintf("%12d", COUNT_LOCATIONS);
    print "*** End of report ***";
```

GENERATED PROGRAM OUTLINE

- Open Input Stream
- Load/Connect Tables
- Read Next Input Record
- Output Aggregated Record If Grouping Key Changes
- Calculate Derived Input Fields
- Perform Aggregations
- *Process Outline:*

```
open input stream
load tables
while (read_input_record)
    split input record into fields
    pre-process input fields
    if (grouping_key not equals previous_grouping_key) then
        post-process output fields
        print aggregated record
        initialize aggregate record buffer
        set previous_grouping_key
     end if
     calculate derived input fields
     perform aggregations
end while
post-process output fields
print (last) aggregated record
close input stream
close output stream
```

ARRAY FIELDS

TBC

DATABASE CONNECTIVITY

TBC

Connecting To Oracle Databases

TBC

Connecting To Sqlite Databases

TBC

Connecting To Mysql Databases

TBC

MACROS

Macros are in the format ***&***<macro_name>***(***<arg_list>***)***.

&lookup

Tables that were built using the ***init table*** and ***load table*** sections are accessed with the ***&lookup()*** macro. This macro requires the key as a parameter and will return the matching value. Use the Perl *exists()* function to check for just the existence of a key in table, disregarding the value.

Format
&lookup(<table>, <key>**)**

&lookup(<table>, <key>**)->**<field>

Examples

```
input section
    GROUP => MONTH_CALLDATE <= 6 ? 1 : 2,
    POSTCODE => &lookup(POSTCODES, AAL),
    IN_SAMPLE => exists &lookup(ZIPSAMPLE, ZIP),
    IN_SAMPLE_2 => exists &lookup(ZIPSAMPLE, ZIP) ? 'yes': 'no'
    STREET => &lookup(POSTCODES, AAL)->STREET_NAME
```

&date

Use the ***&date()*** macro to indicate field value is a date. This is required when using date fields in arithmetic calculations and expressions. The ***&date()*** macro actually converts a date value into YYYYMMDD format. The second, optional, argument contains the date format specification. If the format specification is omitted then the ***default_datetype*** option specification is used. The format specification describes the positions and lengths of the day (D), month (M), and year (Y) parts, and any optional delimiters. Day and month data must be two digit zero front padded. The MMM month format indicates abbreviated three character month name (JAN, FEB, MAR, etc). The delimiter can be any special character such as /, -, :, etc. Pequel built-in date types include: DD/MM/YYYY, DD/MM/YY, DDMMYY, DDMMYYYY, DDMMMYY, YYYYMMDD, YYMMDD, MM/DD/YYYY, MM/DD/YY, MMDDYYYY, MMDDYY.

Format
&date(<date> [**,** <datefmt>]**)**

Examples

```
filter
    &date(SALES_DATE) >= &date(01/01/2002),
    &date(SALES_DATE) <= 20023101
```

&d &m &y

Returns the day, month and year portion for *date* field, respectively. The ***&m*** macro will return the abbreviated month name (JAN, FEB, etc) if the date format contains MMM, otherwise the numeric month number is returned.

Format
&d(<date> [**,** <datefmt>]**)**
&m(<date> [**,** <datefmt>]**)**
&y(<date> [**,** <datefmt>]**)**

Examples

```
input section
    DAY_TODAY  =>  &d(&today())
    MOMTH_TODAY  =>  &m(&today())
    YEAR_TODAY  =>  &y(&today())
```

&today

Returns the current date.

Format
&today()

Examples

```
input section
    TODAY  =>  &today()
```

&months_since

Returns the number of months between the current date and the date specified in the argument. An optional second argument containing the date format specification may be specified.

Format
&months_since(<field> [**,** <date_format>]**)**

Examples

```
input section
    MONTHS_IN_USE  =>  &months_since(PURCHASE_DATE)
```

&add_months

New in v2.5. The ***add_months*** macro returns the first argument date *field* plus *n* months. The argument *n* can be any integer. If *field* is the last day of the month or if the resulting month has fewer days than the day component of *field*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *field*.

Format
&add_months(<field> <n>**)**

Examples

```
input section
    NEXT_MONTH  =>  &add_months(PURCHASE_DATE, 1)
```

&months_between

New in v2.5. The ***months_between*** macro returns the absolute number of months between the two dates *field-1* and *field-2*.

Format
&months_between(<field-1>, <field-2>**)**

Examples

```
input section
    MONTHS_PURCHASE  =>  &months_between(EARLIEST_PURCHASE_DATE, LATEST_PURCHASE_DATE)
```

&last_day

The ***last_day*** macro returns the last *day* number for the month in the date *field*.

Format
&last_day(<field>**)**

Examples

```
input section
    LAST_DAY  =>  &last_day(PURCHASE_DATE)
```

&date_last_day

The ***date_last_day*** macro returns the *date* for the last day for the month in the date *field*.

Format
&date_last_day(<field>**)**

Examples

```
input section
    LAST_DAY_DATE  =>  &date_last_day(PURCHASE_DATE)
```

&date_next_day

The ***date_next_day*** macro returns the *date* for the next day for the month in the date *field*. If the date *field* is the last day in the month the the returned date will be the first day for the following month.

Format

&date_next_day(<field>**)**

Examples

```
input section
    NEXT_DAY_DATE  =>  &next_day(PURCHASE_DATE)
```

&day_number

The ***day_number*** macro returns the day number within the year for the *date*.

Format

&day_number(<field>**)**

Examples

```
input section
    DAY_NUMBER  =>  &day_number(PURCHASE_DATE)
```

&month

Initialise the ***&month*** table using the ***init _MONTH*** section. Then use the ***&month()*** macro to return the month number for a date.

Format

&month(<date> [**,** <datefmt>]**)**

Examples

```
input section
    MONTH_CALLDATE => &month(CALLDATE)
```

&period

Initialise the ***&period*** table using the ***init _PERIOD*** section. Then use the ***&period*** macro to return the month number for a date.

Format

&period(<date> [**,** <datefmt>]**)**

Examples

```
input section
    PERIOD_CALLDATE => &period(CALLDATE)
```

&select

Similar to a *switch* statement. Parameters consist of a list of expression-value pairs, followed by one default value. Each expression is evaluated in turn and the first to evaluate to true will return its associated valued, otherwise the default value is returned.

Format

&select(<expr>, <value> [[, <expr>, <value>] [,...]], <default value>**)**

Examples

```
input section
    HOUSEHOLD_TYPE => &select(KINSHIP==5, 1, KINSHIP==6, 2, 0)
```

&map

The ***map*** macro will process (lookup) each element within the array field *field*, looking up each element in *table* and setting that element to the looked up value. Returns an array of results. Non-existing key values will be mapped to null.

Format

&map(<table>, <field> [, ...] **)**

Examples

```
input section
    LEISURE_INTEREST => &map(LI_RECODE, LEISURE_INTEREST_IN)
```

&to_array

New in v2. The ***to_array*** macro will convert a field value into an array list by splitting the field value on the list-delimiter.

Format

&to_array(<field>**)**

Examples

```
output section
    string LEISURE_INTEREST      values_uniq &to_array(LEISURE_INTEREST_IN)
```

&arr_size

New in v2. The ***arr_size*** macro will return the total number of elements in the array *field*, or combined arrays if more than one array *field* is specified.

Format

&arr_size(<field> [**,** <field>, ...]**)**

Examples

```
output section
    numeric COUNT_PHONES      &arr_size(PHONE_LIST_1, PHONE_LIST_2)
```

&arr_sort

New in v2. The ***arr_sort*** macro will sort the elements within the array field *field*.

Format

&arr_sort(<field>**)**

Examples

```
input section
    LEISURE_INTEREST => &arr_sort(&map(LI_RECODE, LEISURE_INTEREST_IN))
```

&arr_reverse

New in v2. The ***arr_reverse*** macro will return the elements in array *field* in reverse order.

Format

&arr_reverse(<field>**)**

Examples

```
input section
    LEISURE_INTEREST => &arr_reverse(&map(LI_RECODE, LEISURE_INTEREST_IN))
```

&arr_first

Returns the first element in an array field.

Format

&arr_first(<field> [**,** <field>, ...]**)**

Examples

```
input section
    FIRST_MONTH => &arr_first(&to_array(MONTH_LIST))
```

&arr_last

Returns the last element in an array field.

Format

&arr_last(<field> [**,** <field>, ...]**)**

Examples

```
input section
    LAST_MONTH => &arr_last(&to_array(MONTH_LIST))
```

&arr_min

Returns the element with the minimum (numeric) value in an array field.

Format

&arr_min(<field> [**,** <field>, ...]**)**

Examples

```
input section
    EARLIEST_MONTH => &arr_min(&to_array(MONTH_LIST))
```

&arr_max

Returns the element with the maximum (numeric) value in an array field.

Format

&arr_max(<field> [**,** <field>, ...]**)**

Examples

```
input section
    LATEST_MONTH => &arr_max(&to_array(MONTH_LIST))
```

&arr_avg

Returns the average value for all elements in an array field.

Format

&arr_avg(<field> [**,** <field>, ...]**)**

Examples

```
input section
    AVG_PRICE => &arr_avg(&to_array(PRICE_LIST))
```

&arr_sum

Returns the total value for all elements in an array field.

Format

&arr_sum(<field> [**,** <field>, ...]**)**

Examples

```
input section
    SUM_PRICE => &arr_sum(PRICE_1, PRICE_2, PRICE_3)
```

&arr_median

New in v2.5.

Format
&arr_median(<field> [**,** <field>, ...]**)**

Examples

&arr_variance
New in v2.5.

Format
&arr_variance(<field> [**,** <field>, ...]**)**

Examples

&arr_stddev
New in v2.5.

Format
&arr_stddev(<field> [**,** <field>, ...]**)**

Examples

&arr_range
New in v2.5.

Format
&arr_range(<field> [**,** <field>, ...]**)**

Examples

&arr_mode
New in v2.5.

Format
&arr_mode(<field> [**,** <field>, ...]**)**

Examples

&arr_values_uniq
Returns the unique values for elements in the array field(s) argument.

Format
&arr_values_uniq(<field> [**,** <field>, ...]**)**

Examples

```
input section
    UNIQ_LEISURE_INTEREST => &arr_values_uniq(LEISURE_INTEREST_1, LEISURE_INTEREST_2)
```

&arr_shift
New in v2. The ***arr_shift*** macro takes the first element of the array and returns it, removing the first element and shortening the array *field* by one element, moving everything down one place.

Format
&arr_shift(<field>**)**

Examples

```
input section
    FIRST_LEISURE_INTEREST => &arr_shift(LEISURE_INTEREST)
```

&arr_push

New in v2. The ***arr_push*** macro adds *value* or values to the end of an array *field* and increases the length of the array by the number of elements added, then return the new array.

Format
&arr_push(<field>, <value> [,...]**)**

Examples

```
input section
    LEISURE_INTEREST => &arr_push(ANOTHER_INTEREST)
```

&arr_pop

New in v2. The ***arr_pop*** macro returns the last element of an array, deleting this last element from *field*, thus shortening the array *field* by one element.

Format
&arr_pop(<field>**)**

Examples

```
input section
    LAST_LEISURE_INTEREST => &arr_pop(LEISURE_INTEREST)
```

&arr_lookup

The ***arr_lookup*** macro returns `1` (true) if the 1st parameter value exists in the array 2nd parameter, else returns `0` (false).

Format
&arr_lookup(<value, array-field>**)**

Examples

```
input section
    LAST_LEISURE_INTEREST => &arr_lookup(14, &to_array(SOURCE_LIST))
```

&extract_init

The ***extract_init*** macro returns the 1st character of each word in the contents of the parameter. *field* can be any valid expression. An example of usage for this macro is to extract the initials from a full name field.

Format
&extract_init(<field>**)**

Examples

```
input section
    NAME_INITIALS => &extract_init(FORENAME . ' ' . MIDDLE_NAMES)
```

&remove_numeric

This macro will remove all numeric characters from the field specified in argument.

Format
&remove_numeric(<field>**)**

Examples

```
input section
    CLEAN_NAME  =>  &remove_numeric(NAME)
```

&remove_special

This macro will remove all special characters from the field specified in argument. Special characters consist of **!@#$%^*(){}[]:;\?/+**<>.

Format
&remove_special(<field>**)**

Examples

```
input section
    CLEAN_NAME  =>  &remove_special(NAME)
```

&remove_spaces

This macro will remove all space characters from the field specified in argument.

Format
&remove_spaces(<field>**)**

Examples

```
input section
    CLEAN_NAME  =>  &remove_spaces(NAME)
```

&match*, *&match_any

These macros are identical and will return true (1) if the *field* content matches any of the *match list* items, else returns false (0).

Format
&match(<field>, <match list>**)**

Examples

```
input section
    EAST_COAST => &match(STATE, QLD, NSW, VIC) ? 'yes' : 'no';
```

&remove_non_numeric*, *&extract_numeric*, *&to_number

These macros are identical and will remove all non-numeric characters from the field specified in argument.

Format
&extract_numeric(<field>**)**

Examples

```
input section
    CLEAN_SERIAL  =>  &extract_numeric(SERIAL)
```

&length

New in v2. The ***length*** macro will return the length in characters of a field (string) value.

Format
&length(<field>**)**

Examples

```
input section
    NAME_FIELD_LENGTH => &length(NAME)
```

&substr

New in v2. The ***substr*** macro extracts a substring of length *len* out of *field* and returns it. If *offset* is negative, counts from the end of the string.

Format

&substr(<field>, <offset>, <len>**)**

Examples

```
input section
    LINK_TYPE => &substr(LINK, 0, 3)
```

&index

New in v2. The ***index*** macro returns the position of *substr* in *field* at or after *offset*. If the substring is not found, returns **-1**.

Format
&index(<field>, <substr>, <offset>**)**

&rindex

New in v2. The ***rindex*** macro returns the postion of the last *substr* in *field* at or before *offset*.

Format
&rindex(<field>, <substr>, <offset>**)**

Examples

&lc

New in v2. The ***lc*** macro returns the lower case version of *field*.

Format
&lc(<field>**)**

Examples

```
input section
            FIRST_NAME,
    MIDDLE_NAME,
    LAST_NAME,
    NAME_FORMATTED =>  &uc_first(&lc(FIRST_NAME)) \
                       . ' ' .  &uc_first(&lc(MIDDLE_NAME)) \
                       . ' ' .  &uc_first(&lc(LAST_NAME))
```

&lc_first

New in v2. The ***lc_first*** macro returns *field* with the first character lower case.

Format
&lc_first(<field>**)**

Examples

&uc

New in v2. The ***uc*** macro returns the upper case version of *field*.

Format
&uc(<field>**)**

Examples

```
input section
            FIRST_NAME,
    MIDDLE_NAME,
    LAST_NAME,
    NAME_FORMATTED =>  &uc(FIRST_NAME) \
                       . ' ' .  &uc(MIDDLE_NAME) \
                       . ' ' .  &uc(LAST_NAME)
```

&uc_first

New in v2. The ***uc_first*** macro returns *field* with the first character upper case.

Format
&uc_first(<field>**)**

Examples

```
input section
            FIRST_NAME,
    MIDDLE_NAME,
    LAST_NAME,
    NAME_FORMATTED =>  &uc_first(&lc(FIRST_NAME)) \
                       . ' ' .  &uc_first(&lc(MIDDLE_NAME)) \
                       . ' ' .  &uc_first(&lc(LAST_NAME))
```

&clip_str

New in v2. The ***clip_str*** macro returns *field* with all *leading* and *trailing* spaces removed.

Format
&clip_str(<field>**)**

Examples

```
input section
            FIRST_NAME,
    MIDDLE_NAME,
    LAST_NAME,
    NAME_FORMATTED =>  &uc_first(&lc(&clip_str(FIRST_NAME))) \
                       . ' ' .  &uc_first(&lc(MIDDLE_NAME)) \
                       . ' ' .  &uc_first(&lc(LAST_NAME))
```

&left_clip_str

New in v2. The ***left_clip_str*** macro returns *field* with all *leading* spaces removed.

Format
&left_clip_str(<field>**)**

Examples

```
input section
            FIRST_NAME,
    MIDDLE_NAME,
    LAST_NAME,
    NAME_FORMATTED =>  &uc_first(&lc(&left_clip_str(FIRST_NAME))) \
                       . ' ' .  &uc_first(&lc(MIDDLE_NAME)) \
                       . ' ' .  &uc_first(&lc(LAST_NAME))
```

&right_clip_str

New in v2. The ***right_clip_str*** macro returns *field* with all *trailing* spaces removed.

Format
&right_clip_str(<field>**)**

Examples

```
input section
            FIRST_NAME,
    MIDDLE_NAME,
    LAST_NAME,
    NAME_FORMATTED =>  &uc_first(&lc(&right_clip_str(FIRST_NAME))) \
                       . ' ' .  &uc_first(&lc(MIDDLE_NAME)) \
                       . ' ' .  &uc_first(&lc(LAST_NAME))
```

&left_pad_str

New in v2. The ***left_pad_str*** macro returns *field* padded with the specified pad character on the left, and up to *len* maximum length.

Format

&left_pad_str(<field>, <pad-char>, <len>**)**

Examples

```
input section
           FMT_AMOUNT  => &left_pad_str(AMOUNT, '*', 16)
```

&right_pad_str

New in v2. The ***right_pad*** macro returns *field* padded with the specified pad character on the right, and up to *len* maximum length.

Format
&right_pad_str(<field>, <pad-char>, <len>**)**

Examples

```
input section
           FMT_NAME  => &right_pad_str(NAME, ' ', 32)
```

&trim

New in v2. The ***trim*** macro returns *field* with the specified leading and trailing *trim-char* character(s) removed. If *trim-char* is not specified, then the default value is space character.

Format
&trim(<field> [**,** <trim-char(s)>]**)**

Examples

```
input section
           SERIAL  => &trim(RAW_SERIAL, 0)
```

&trim_leading

New in v2. The ***trim_leading*** macro returns *field* with the specified leading *trim-char* character(s) removed. If *trim-char* is not specified, then the default value is space character.

Format
&trim_leading(<field> [**,** <trim-char(s)>]**)**

Examples

```
input section
           SERIAL  => &trim_leading(RAW_SERIAL, 0)
```

&trim_trailing

New in v2. The ***trim_trailing*** macro returns *field* with the specified trailing *trim-char* character(s) removed. If *trim-char* is not specified, then the default value is space character.

Format
&trim_trailing(<field> [**,** <trim-char(s)>]**)**

Examples

```
input section
           SERIAL  => &trim_trailing(RAW_SERIAL, 0)
```

&translate

New in v2. The ***translate*** macro returns the first argument *field* with all occurrences of each character in *from_list* replaced by its corresponding character in *to_list*. Characters in *field* that are not in *from_list* are not replaced. The argument *from_list* can contain more characters than *to_list*. In this case, the extra characters at the end of *from_list* have no corresponding characters in *to_list*. If these extra characters appear in *field*, then they are replaced by the last character in *to_list*, unless the modifier value of *d* is specified — in this case they are removed.

Format

&translate(<field>, <from-list>, <to-list> [**,** <modifier>]**)**

Examples

```
input section
    NO_NUM_NAME  =>  &translate(NAME, '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789')
       # remove number characters from NAME

    ENC_LICENCE  =>  &translate(LICENCE, '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', \
                                         '9999999999XXXXXXXXXXXXXXXXXXXXXXXXXX')
       # encode a field value
```

&soundex

New in v2.5. The ***soundex*** macro returns a character string containing the phonetic representation of *field*. This function lets you compare words that are spelled differently, but sound alike in English.

The phonetic representation is defined in The Art of Computer Programming, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

```
Retain the first letter of the string and remove all other occurrences of the following
letters: a, e, h, i, o, u, w, y.
Assign numbers to the remaining letters (after the first) as follows:
b, f, p, v = 1
c, g, j, k, q, s, x, z = 2
d, t = 3
l = 4
m, n = 5
r = 6

If two or more letters with the same number were adjacent in the original
name (before step 1), or adjacent except for any intervening h and w, then
omit all but the first.
Return the first four bytes padded with 0.
```

Format
&soundex(<field>**)**

Examples

```
filter
    LAST_NAME eq &soundex(SMYTHE)
```

&initcap

New in v2. The ***initcap*** macro will return the string expression *exp* with all the words capitalized in their first letter (with the rest of the word in lowercase).

Format
&initcap(<exp>**)**

Examples

```
input section
    ADDRESS => &initcap(join(' ', ADDRESS_LINE_1, ADDRESS_LINE_2, CITY, STATE, ZIP, COUNTRY))
```

&banding

The ***banding*** macro will return the band number (starting from 1) for *field*, depending on the value of *field* in relation to the *band-divisor*. The *band-divisor* must be a non zero numeric value. The returned band number is calculated as *int((field - 1) / band-divisor) + 1*.

Format
&banding(<field>, <band-divisor>**)**

Examples

```
input section
    LAST_SALE_PRICE_BAND => &banding(%propertyvalue(CONCATENATED_LINK)->SALE_PRICE, 50000)
```

&env

New in v2. The ***env*** macro will return the content of the environment variable *env_name*.

Format
&env(<env_name>**)**

Examples

```
input section
    USER_ID => &env(USER)
```

&option

New in v2. The ***option*** macro will return the value for the Pequel option *pql_option_name*.

Format
&option(<pql_option_name>**)**

Examples

```
input section
    SCRIPT_VERSION => &option(doc_version)
```

&sqrt &rand &log &sin &exp &cos &abs &atan2 &ord &chr &int

New in v2. Arithmetic functions.

The ***sqrt*** macro returns the square root of *expr*.

The ***rand*** function returns a random number between 0 and the value of the positive expression *expr* you pass; if you don't pass an expression, ***rand*** uses 1.

The ***log*** macro returns the natural logarithm of an expression.

The ***sin*** macro returns the sine of an expression *expr*.

The ***exp*** macro returns *e* to the power of *expr*.

The ***cos*** macro returns the cosine of a value in radians (two pi radians comprise a full circle).

The ***abs*** macro returns the absolute value of *expr*.

The ***atan2*** macro returns the arctangent of **Y/X** (the value returned is between -pi and pi).

The ***ord*** macro returns the ASCII value of the first character (only) of an expression *expr*.

The ***chr*** macro returns the character corresponding to the ASCII number you pass it in *expr*.

The ***int*** macro returns the integer (numeric) value of *expr*.

Format
&<macro>(<expr>**)**

&sign

The ***sign*** macro returns -1 if the argument field value is less than zero. If field value is zero , then the macro returns 0. If field value is greater than zero, then ***sign*** returns 1.

Format
&sign(<field>**)**

Examples

&trunc

The ***trunc*** macro returns the argument field value truncated to *dec* decimal places. If *dec* is omitted, then *field* is truncated to 0 places. *dec* can be negative to truncate (make zero) *dec* digits left of the decimal

point.

Format
&trunc(<field>, <dec>**)**

Examples

&arr_set_and
New in v2.5.

Format
&arr_set_and(<field> [**,** <field>, ...]**)**

Examples

&arr_set_xor
New in v2.5.

Format
&arr_set_and(<field> [**,** <field>, ...]**)**

Examples

&arr_set_or
New in v2.5.

Format
&arr_set_or(<field> [**,** <field>, ...]**)**

Examples

EXAMPLE PEQUEL SCRIPTS

Aggregates Example Script

Demonstrates aggregation and use of various aggregate function. For each PRODUCT_CODE group of records, determine: the minimum COST_PRICE, the maximum COST_PRICE, the average SALES_PRICE and SALES_QTY; accumulate the sum of SALES_TOTAL; calculate *range* for COST_PRICE. The input field SALES_TOTAL is a *derived input field*.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   nulls

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL => SALES_QTY * SALES_PRICE

sort by
   PRODUCT_CODE
   LOCATION

group by
   PRODUCT_CODE

output section
   string LOCATION           LOCATION
   string PRODUCT_CODE       PRODUCT_CODE
   decimal MIN_COST_PRICE    min COST_PRICE
   decimal MAX_COST_PRICE    max COST_PRICE
   decimal AVG_SALES_PRICE   mean SALES_PRICE
   numeric _AVG_SALES_QTY    mean SALES_QTY
   decimal SALES_TOTAL       sum SALES_TOTAL
   decimal SALES_TOTAL_2     sum SALES_TOTAL
   decimal RANGE_COST        range COST_PRICE
   numeric MODE_SALES_CODE   mode SALES_CODE
   numeric AVGS              = _AVG_SALES_QTY * 2
```

Apache CLF Log Input Example Script

Demonstrates reading Apache CLF Log file — split record on space delimiter, parse qouted fields and square bracketed fields. This is done by 1) specifying a space delimiter for the 'input_delimiter' and 2) specifying a double qoute (must be escaped) characted and a open square bracket character for the 'input_delimiter_extra' option. This option specifies other characters that may delimit fields. Pequel will match open bracket character specfication with their respective closing bracket.

Requires Inline::C and a C compiler to be installed because the 'input_delimiter_extra' option will instruct Pequel to generate C code.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   nulls
   transfer // Copy input to output
   input_delimiter( ) // Input delimiter is space.
   input_delimiter_extra(\"[) // For Apache Common Log Format (CLF).
   inline_CC(CC) // C compiler.
   inline_clean_after_build(0) // Pass-through Inline options:
   inline_clean_build_area(0)
   inline_print_info(1)
   inline_build_noisy(1)
   inline_build_timers(0)
   inline_force_build(1)
   inline_directory()
   inline_optimize("-xO5 -xinline=%auto") // Solaris 64 bit
   inline_ccflags("-xchip=ultra3 -DSS_64BIT_SERVER -DBIT64 -DMACHINE64")

input section
   IP_ADDRESS,
   TIMESTAMP,
   REQUEST,
   F4,
   F5,
   F6

output section
```

Array Fields Example Script

Demonstrates the use of array-fields. An array-field is denoted by the preceding '@' character. The 'salesman_list' field in this example is an 'array field' delimited by the default array field delimiter ','. Array type macros (&arr_...) will expect all arguments to be array-fields. Array macros can also be called as a method following the array-field.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   nulls

input section
   product_code,
   cost_price,
   description,
   sales_code,
   sales_price,
   sales_qty,
   sales_date,
   location,
   salesman_list,
   num_salesmen       => &arr_size(@salesman_list)
   salesmen_sorted    => &arr_sort(salesman_list) // implicit array -- all array macros expect array param vars
   salesmen_sorted_2  => @salesman_list->sort
   salesmen_uniq      => &arr_values_uniq(@salesman_list)
   salesmen_uniq_2    => @salesman_list->values_uniq
   salesmen_reverse   => &arr_reverse(&arr_sort(@salesman_list))

 sort by
   product_code

 output section
   string location             location
   string product_code         product_code
   string salesman_list        salesman_list
   numeric num_salesmen        num_salesmen
   string salesmen_sorted      salesmen_sorted
   string salesmen_sorted_2    salesmen_sorted_2
   string salesmen_uniq        salesmen_uniq
   string salesmen_uniq_2      salesmen_uniq_2
   string salesmen_reverse     salesmen_reverse
```

Pequel Script Chaining Example Scripts

This example demonstrates Pequel script 'chaining'. By specifying a pequel script name for the 'input_file' option, the input data stream will result by executing the specified script. Both scripts are executed simultaneously — with the input_file script as the child and this script as the parent. Beware of circular chaining! It is up to the user to ensure that this does not occur. Currently, 'sort by' is not supported in the parent script.

chain_pequel_pt1.pql

```
options
   input_file(sample.data)       // Need to specify this script is used as a pequel-table loader.
   optimize // (default) optimize generated code.

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL => SALES_QTY * SALES_PRICE

sort by
   LOCATION
   PRODUCT_CODE

group by
   LOCATION
   PRODUCT_CODE

output section
   string LOCATION         LOCATION
   string PRODUCT_CODE     PRODUCT_CODE
   decimal SALES_TOTAL     sum SALES_TOTAL
```

chain_pequel_pt2.pql

```
options
   input_file(chain_pequel_pt1.pql) // Need to specify this script is used as a pequel-table loader.
   header // (default) write header record to output.
   hash
   optimize // (default) optimize generated code.

input section
   LOCATION
   PRODUCT_CODE
   SALES_TOTAL

group by
   LOCATION

output section
   string LOCATION             LOCATION
   numeric COUNT_PRODUCT_CODE  distinct PRODUCT_CODE
   decimal SALES_TOTAL         sum SALES_TOTAL
```

Conditional Aggregation Example Script

Demonstrates the use of conditional aggregations. A conditional aggregate is done with the 'where' clause. This example analyses the COST_PRICE in various ways for the two states: NSW and VIC.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION

sort by
   PRODUCT_CODE

group by
   PRODUCT_CODE

output section
   string PRODUCT_CODE            PRODUCT_CODE
   numeric AVG_COST_PRICE         avg COST_PRICE
   numeric MIN_COST_PRICE         min COST_PRICE
   numeric MAX_COST_PRICE         max COST_PRICE
   numeric SUM_COST_PRICE         sum COST_PRICE

   numeric AVG_COST_PRICE_NSW     avg COST_PRICE where LOCATION eq 'NSW'
   numeric MIN_COST_PRICE_NSW     min COST_PRICE where LOCATION eq 'NSW'
   numeric MAX_COST_PRICE_NSW     max COST_PRICE where LOCATION eq 'NSW'
   numeric SUM_COST_PRICE_NSW     sum COST_PRICE where LOCATION eq 'NSW'

   numeric AVG_COST_PRICE_VIC     avg COST_PRICE where LOCATION eq 'VIC'
   numeric MIN_COST_PRICE_VIC     min COST_PRICE where LOCATION eq 'VIC'
   numeric MAX_COST_PRICE_VIC     max COST_PRICE where LOCATION eq 'VIC'
   numeric SUM_COST_PRICE_VIC     sum COST_PRICE where LOCATION eq 'VIC'

   numeric RANGE_COST_PRICE       = MAX_COST_PRICE - MIN_COST_PRICE
```

External Tables Example Script

Demonstrates the use of external tables. The default method for loading an external table is to embed the table contents in the generated code. SAMPLE1 is a example of an embedded table. External tables may also be loaded dynamically (at runtime) — the '_' table name prefix instructs Pequel to load the table dynamically. SAMPLE2 is an axample of a dynamic table. The optional environment variable 'PEQUEL_TABLE_PATH' may be set to the path for the location of the table data-source-files. This path will be used to locate the data-source-files unless the data source filename is an absolute path name.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.

load table
   // External embedded table -- key is field-1 (PRODUCT_CODE). 'STRING' is the key-field
   // type. 'sample.data' is the data-source-file to load the table from. Table has two
   // columns: DESCRIPTION (field #3 in source file), and LOCATION (#8 in source file).
   // The default for loading an external table is to embedd the table contents in the generated code.
   SAMPLE1 sample.data 1 STRING DESCRIPTION=3 LOCATION=8

load table
   // External dynamic table. The '_' prefix instructs Pequel
   // to load the table dynamically.
   _SAMPLE2 sample.data 1 STRING DESCRIPTION=3 LOCATION=8

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   S1_DESCRIPTION => %SAMPLE1(PRODUCT_CODE)->DESCRIPTION
   S1_LOCATION => %SAMPLE1(PRODUCT_CODE)->LOCATION
   S2_DESCRIPTION => %SAMPLE2(PRODUCT_CODE)->DESCRIPTION
   S2_LOCATION => %SAMPLE2(PRODUCT_CODE)->LOCATION

sort by
   PRODUCT_CODE

group by
   PRODUCT_CODE

output section
   string PRODUCT_CODE         PRODUCT_CODE,
   numeric RECORD_COUNT        count *
   numeric SALES_QTY_SAMPLE1   sum SALES_QTY where exists %SAMPLE1(PRODUCT_CODE)
   numeric SALES_QTY_SAMPLE2   sum SALES_QTY where exists %SAMPLE2(PRODUCT_CODE)
   string S1_DESCRIPTION       S1_DESCRIPTION
   string S1_LOCATION          S1_LOCATION
   string S2_DESCRIPTION       S2_DESCRIPTION
   string S2_LOCATION          S2_LOCATION
```

Filter Regex Example Script

Demonstrates use of filter and Perl regular expressions. The requlars expression can contain Pequel field names, macros and table names. This example also demonstrates the use of a simple 'local' table (LOC_DESCRIPT).

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.

init table
// Table-Name      Key-Value   Field->1               Field-2     Field-3
   LOC_DESCRIPT    NSW         'New South Wales'      '2061'      '02'
   LOC_DESCRIPT    WA          'Western Australia'    '5008'      '07'
   LOC_DESCRIPT    SA          'South Australia'      '8078'      '08'

filter
// Filter out all records except where LOCATION is 'NSW' or 'WA' or 'SA'
   LOCATION =~ /^NSW$|^WA$|^SA$/

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   LDESCRIPT => %LOC_DESCRIPT(LOCATION)->1 . " in postcode " . %LOC_DESCRIPT(LOCATION)->2

sort by
   SALES_CODE

group by
   SALES_CODE

output section
   string SALES_CODE              SALES_CODE
   string LOC_DESCRIPT            LDESCRIPT
   numeric NUM_PRODUCTS           distinct PRODUCT_CODE
   string _PRODUCT_CODE           PRODUCT_CODE
   string PROD_NUM                = _PRODUCT_CODE . "-" . NUM_PRODUCTS
   string LOC_NSW                 = %LOC_DESCRIPT(NSW)->1
   numeric AVG_COST_PRICE_NSW     avg COST_PRICE where LOCATION eq 'NSW'
   string LOC_WA                  = %LOC_DESCRIPT(WA)->1
   numeric AVG_COST_PRICE_WA      avg COST_PRICE where LOCATION eq 'WA'
   string LOC_SA                  = %LOC_DESCRIPT(SA)->1
   numeric AVG_COST_PRICE_SA      avg COST_PRICE where LOCATION eq 'SA'
```

Group By Derived Example Scripts

This example demonstrates the use of a derived (calculated) field as the grouping field. In this example it is assumed that the input data contains mixed case values for LOCATION. The 'hash' option is important here because grouping is based on exact values — that is, LOCATION's 'NSW' and 'Nsw' are not equal, but converting both to upper case make them equal. With the 'hash' option, the input data need not be sorted because the output is generated in memory using Perl's associative arrays. For this reason the 'hash' option should only be used when the total number of groups is small, depending on the amount of available memory.

Example Script 1

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   hash // Required because group-by field is derived.

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL => SALES_QTY * SALES_PRICE,
   FIXED_LOC_CODE => &uc(LOCATION)

group by
   FIXED_LOC_CODE

output section
   string FIXED_LOC_CODE FIXED_LOC_CODE
   decimal SALES_TOTAL sum SALES_TOTAL
```

Example Script 2

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   hash // Required because group-by field is derived.

init table // multi-column local table
// Table-Name  Key-Value    Field->1         Field->2
   TCITY       'SYD'        'Sydney'         'NSW'
   TCITY       'MEL'        'Melbourne'      'VIC'
   TCITY       'PER'        'Perth'          'WA'
   TCITY       'ALIC'       'Alice Springs'  'NT'

init table // single-column local table
// Table-Name  Key-Value    Field->1
   TSTATE      'WA'         "Western Australia"
   TSTATE      'NSW'        "New South Wales"
   TSTATE      'SA'         'South Australia'
   TSTATE      'QLD'        'Queensland'
   TSTATE      'NT'         'Northern Territory'
   TSTATE      'VIC'        'Victoria'

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL     => SALES_QTY * SALES_PRICE,
   FIXED_LOC_CODE  => %TCITY(LOCATION)->2 || LOCATION,  // lookup TCITY, return field-2
   STATE_NAME      => %TSTATE(FIXED_LOC_CODE) // lookup TSTATE, return field-1

group by
   FIXED_LOC_CODE

output section
   string  FIXED_LOC_CODE    FIXED_LOC_CODE
   string  STATE_NAME        STATE_NAME
   decimal SALES_TOTAL       sum SALES_TOTAL
```

Hash Option Example Script

This example demonstrates the use of the 'hash' option. With the 'hash' option input data sorting is not required — the data will be aggregated in memory. For this reason the 'hash' option should only be used when the total number of groups is small, depending on the amount of available memory.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   hash

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION

group by
   LOCATION

output section
   string LOCATION                 LOCATION
   numeric MIN_COST_PRICE          min COST_PRICE
   numeric MAX_COST_PRICE          max COST_PRICE
   numeric _DISTINCT_SALES_CODE    distinct SALES_CODE
   string SALES_CODE_1             first SALES_CODE where _DISTINCT_SALES_CODE == 1
   string SALES_CODE_2             first SALES_CODE where _DISTINCT_SALES_CODE == 2
   string SALES_CODE_3             first SALES_CODE where _DISTINCT_SALES_CODE == 3
   string SALES_CODE_4             first SALES_CODE where _DISTINCT_SALES_CODE == 4
   string SALES_CODE_5             first SALES_CODE where _DISTINCT_SALES_CODE == 5
```

Local Table Example Script

Demonstrates use of local tables. LOC_DESCRIPT is a local table. Each line in the ‘init table’ section contains an entry in this table. Each entry constist of table name, key value, field list values.
The ‘%’ character is used to denote a table name. The parameter contains the key value to look up.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.

init table // Local table:
// Table-Name      Key-Value   Field->1
   LOC_DESCRIPT    NSW         'New South Wales'
   LOC_DESCRIPT    WA          'Western Australia'
   LOC_DESCRIPT    SYD         'Sydney'
   LOC_DESCRIPT    MEL         'Melbourne'
   LOC_DESCRIPT    SA          'South Australia'
   LOC_DESCRIPT    NT          'Northern Territory'
   LOC_DESCRIPT    QLD         'Queensland'
   LOC_DESCRIPT    VIC         'Victoria'
   LOC_DESCRIPT    PER         'Perth'
   LOC_DESCRIPT    ALIC        'Alice Springs'

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   LDESCRIPT => %LOC_DESCRIPT(LOCATION) // Look up LOCATION in the table LOC_DESCRIPT

sort by
   LOCATION

group by
   LOCATION

output section
   string LOCATION                         LOCATION
   string DESCRIPTION                      LDESCRIPT
   numeric NUM_PRODUCTS                    distinct PRODUCT_CODE
   numeric AVG_COST_PRICE                  avg COST_PRICE
```

Pequel Tables Example Script

This script demonstrates the use of pequel tables. This scipt contains a ‘load table pequel’ section. The tables specified in this section will have their data loaded by executing the pequel script specified. The field names for the table columns are as per the load table script output format. The output format for a script can be displayed with the ‘-list output_format’ option on the command line. It is important that any Pequel script used in the ‘load table pequal’ to load a table must have an input_file option specification.

pequel_tables.pql

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.

load table pequel
   // Data for this table is loaded by executing the Pequel script 'sales_ttl_by_loc.pql'.
   // Pequel tables are loaded dynamically (at runtime).
   // LOCATION is the key field.
   TSALESBYLOC sales_ttl_by_loc.pql LOCATION
   TSALESBYPROD sales_ttl_by_prod.pql PRODUCT_CODE

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALESBYLOC => %TSALESBYLOC(LOCATION)->SALES_TOTAL,
   SALESBYPROD => %TSALESBYPROD(PRODUCT_CODE)->SALES_TOTAL,
   COMMENT => %TSALESBYLOC(LOCATION)->TOP_PRODUCT eq PRODUCT_CODE ? '**Best Seller' : ''

output section
   string PRODUCT_CODE             PRODUCT_CODE,
   decimal PRODUCT_SALES_TOTAL     SALESBYPROD,
   string LOCATION                 LOCATION,
   decimal LOCATION_SALES_TOTAL    SALESBYLOC,
   string COMMENT                  COMMENT,
```

sales_ttl_by_loc.pql

```
input
   input_file(sample.data) // Need to specify this script is used as a pequel-table loader.
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   hash // Required because group-by field is derived.

load table pequel
   TTOPPRODBYLOC top_prod_by_loc.pql LOCATION

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL => SALES_QTY * SALES_PRICE,
   TOP_PRODUCT => %TTOPPRODBYLOC(LOCATION)->PRODUCT_CODE

group by
   LOCATION

output section
   string LOCATION         LOCATION
   decimal SALES_TOTAL     sum SALES_TOTAL
   string TOP_PRODUCT      TOP_PRODUCT
```

top_prod_by_loc.pql

```
options
   input_file(sample.data)       // Need to specify this script is used as a pequel-table loader.
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   hash // Required because group-by field is derived.

input section
```

```
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL => SALES_QTY * SALES_PRICE

group by
   LOCATION

output section
   string LOCATION        LOCATION
   decimal _MAXSALES      max SALES_TOTAL
   string PRODUCT_CODE    first PRODUCT_CODE where sprintf("%.2f", SALES_TOTAL) \
                                              eq sprintf("%.2f", _MAXSALES)
```

sales_ttl_by_prod.pql

```
options
   input_file(sample.data) // Need to specify this script is used as a pequel-table loader.
   header // (default) write header record to output.
   optimize // (default) optimize generated code.

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   SALES_TOTAL => SALES_QTY * SALES_PRICE

group by
   PRODUCT_CODE,

output section
   string PRODUCT_CODE     PRODUCT_CODE
   decimal SALES_TOTAL     sum SALES_TOTAL
```

Oracle Tables Example Script

Demonstrates the use of external Oracle tables. WARNING: this feature is alpha and would (probably) require some hand coding adjustments to the generated code.

Requires Inline::C and DBI to be installed.

The 'load table oracle' section will load the ASCII data contained in the file specified by the second parameter ('sample.data' inexample SAMPLE1 below) into an oracle table. The generated inline C code will access this table via Oracle OCI. The Oracle table will be re-created with the same name as specified by the first parameter('SAMPLE1' in this example). The data will be loaded via Oracle sqlldr. The 4th parameter KeyLoc specifies the location of the key field in sample.data (field numbers starting from 1). The next parameter KeyType specifies the Oracle type and size to use when creating the table. The Columns list specifies field and field-number (in the SourceData file) pairs. The 'merge' option can be used when the table is sorted by the same key as specified in the 'sort by' section. This will result in a substantial performance gain when looking up values in the table.

```
options
   header // (default) write header record to output.
   optimize // (default) optimize generated code.
   inline_CC(CC) // C compiler.
   inline_clean_after_build(0) // Pass-through Inline options:
   inline_clean_build_area(0)
   inline_print_info(1)
   inline_build_noisy(1)
   inline_build_timers(0)
   inline_force_build(1)
   inline_directory()
   inline_optimize("-xO5 -xinline=%auto")    // Solaris 64 bit
   inline_ccflags("-xchip=ultra3 -DSS_64BIT_SERVER -DBIT64 -DMACHINE64")

load table oracle
// Declare SAMPLE1 table -- all parameters must appear on one line or use line continuation char '\'
// TableName   SourceData  ConnectString              KeyLoc KeyType      Columns
   SAMPLE1     sample.data 'user/passwd@DB1'          1      STRING(12)   DESCRIPTION=3 \
                                                                          LOCATION=8

load table oracle merge
// TableName   SourceData  ConnectString               KeyLoc  KeyType     Columns
   SAMPLE2     sample.data 'user/passwd@DB1'           1       STRING(12)  DESCRIPTION=3 LOCATION=8

input section
   PRODUCT_CODE,
   COST_PRICE,
   DESCRIPTION,
   SALES_CODE,
   SALES_PRICE,
   SALES_QTY,
   SALES_DATE,
   LOCATION,
   S1_DESCRIPTION => %SAMPLE1(PRODUCT_CODE)->DESCRIPTION
   S1_LOCATION => %SAMPLE1(PRODUCT_CODE)->LOCATION
   S2_DESCRIPTION => %SAMPLE2(PRODUCT_CODE)->DESCRIPTION
   S2_LOCATION => %SAMPLE2(PRODUCT_CODE)->LOCATION

sort by
   PRODUCT_CODE

group by
   PRODUCT_CODE

output section
   string PRODUCT_CODE         PRODUCT_CODE,
   numeric RECORD_COUNT        count *
   numeric SALES_QTY_SAMPLE1   sum SALES_QTY where exists %SAMPLE1(PRODUCT_CODE)
   string S1_DESCRIPTION       S1_DESCRIPTION
   string S1_LOCATION          S1_LOCATION
   numeric SALES_QTY_SAMPLE2   sum SALES_QTY where exists %SAMPLE2(PRODUCT_CODE)
   string S2_DESCRIPTION       S2_DESCRIPTION
   string S2_LOCATION          S2_LOCATION
```

PERL MODULE INTERFACE

The Perl module ***ETL::Pequel.pm*** provides developers with access to the ***Pequel*** functions from a perl program. The following synopsis should be self-explanatory. Basically an *ETL::Pequel* objcect is created via the *new* function, then items are added to each section as required. Use the *section* function to return a pointer to a section type, then use the *addItem* function for the section-type pointer to add items to the section. The *section* function requires a single parameter containing the section name. The *addItem* function requires one or more named parameters, some of which are optional. This is followed by a call to *prepare* and *generate*, and optionally *check*. Finally a call to *execute* will set off the transformation process.

Synopsis

```
use ETL::Pequel;
use strict;

my $p = ETL::Pequel->new();
my $s;

$s = $p->section(ETL::Pequel::OPTIONS);
$s->addItem(name => 'header',       value => 1);
$s->addItem(name => 'optimize',     value => 1);
$s->addItem(name => 'hash',         value => 1);
$s->addItem(name => 'nulls',        value => 1);
$s->addItem(name => 'doc_title',    value => "Aggregates Example Script");
$s->addItem(name => 'doc_email',    value => "sample\@youraddress.com");
$s->addItem(name => 'doc_version',  value => '2.4');

$s = $p->section(ETL::Pequel::DESCRIPTION);
$s->addItem(value => 'Demonstrates aggregation and use of various aggregate function.');

$s = $p->section(ETL::Pequel::INPUT_SECTION);
$s->addItem(name => 'PRODUCT_CODE',     type => 'string');
$s->addItem(name => 'COST_PRICE',       type => 'string');
$s->addItem(name => 'DESCRIPTION',      type => 'string');
$s->addItem(name => 'SALES_CODE',       type => 'string');
$s->addItem(name => 'SALES_PRICE',      type => 'string');
$s->addItem(name => 'SALES_QTY',        type => 'string');
$s->addItem(name => 'SALES_DATE',       type => 'string');
$s->addItem(name => 'LOCATION',         type => 'string');
$s->addItem(name => 'SALES_TOTAL',      type => 'string',
       operator => '=>', calc => 'SALES_QTY * SALES_PRICE');

$s = $p->section(ETL::Pequel::SORT_BY);
$s->addItem(fld => 'PRODUCT_CODE');
$s->addItem(fld => 'LOCATION');

$s = $p->section(ETL::Pequel::GROUP_BY);
$s->addItem(fld => 'PRODUCT_CODE');
$s->addItem(fld => 'LOCATION');

$s = $p->section(ETL::Pequel::OUTPUT_SECTION);
$s->addItem(type => 'string',  field => 'LOCATION',        clause => 'LOCATION');
$s->addItem(type => 'string',  field => 'PRODUCT_CODE',    clause => 'PRODUCT_CODE');
$s->addItem(type => 'decimal', field => 'MIN_COST_PRICE',  clause => 'min COST_PRICE');
$s->addItem(type => 'decimal', field => 'MAX_COST_PRICE',  clause => 'max COST_PRICE');
$s->addItem(type => 'decimal', field => 'AVG_SALES_PRICE', clause => 'mean SALES_PRICE');
$s->addItem(type => 'numeric', field => '_AVG_SALES_QTY',  clause => 'mean SALES_QTY');
$s->addItem(type => 'decimal', field => 'SALES_TOTAL',     clause => 'sum SALES_TOTAL');
$s->addItem(type => 'decimal', field => 'SALES_TOTAL_2',   clause => 'sum SALES_TOTAL');
$s->addItem(type => 'decimal', field => 'RANGE_COST',      clause => 'range COST_PRICE');
$s->addItem(type => 'numeric', field => 'MODE_SALES_CODE', clause => 'mode SALES_CODE');
$s->addItem(type => 'numeric', field => 'AVGS',            clause => '= _AVG_SALES_QTY * 2');

$p->prepare();
$p->generate();

if ($p->check() =~ /syntax\s+ok/i)
{
    $p->engine->printToFile("$0.2.code");
    $p->execute();
}
```

Function Reference

new

Create a new ***Pequel*** object. The *new* function requires no parameters. It will create an ***ETL::Pequel*** object and return a pointer to this.

```
my $p = ETL::Pequel->new();
```

The section name are pre-declared in the ***ETL::Pequel.pm*** module and include the following:

```
ETL::Pequel::OPTIONS
ETL::Pequel::DESCRIPTION
ETL::Pequel::INPUT_SECTION
ETL::Pequel::GROUP_BY
ETL::Pequel::SORT_BY
ETL::Pequel::SORT_OUTPUT
ETL::Pequel::LOAD_TABLE
ETL::Pequel::LOAD_TABLE_PEQUEL
ETL::Pequel::INIT_TABLE
ETL::Pequel::FILTER
ETL::Pequel::REJECT
ETL::Pequel::OUTPUT_SECTION
ETL::Pequel::HAVING
ETL::Pequel::DEDUP_ON
ETL::Pequel::USE_PACKAGE
ETL::Pequel::FIELD_PREPROCESS
ETL::Pequel::FIELD_POSTPROCESS
ETL::Pequel::DIVERT_INPUT_RECORD
ETL::Pequel::COPY_INPUT_RECORD
ETL::Pequel::DIVERT_OUTPUT_RECORD
ETL::Pequel::COPY_OUTPUT_RECORD
ETL::Pequel::DISPLAY_MESSAGE_ON_INPUT
ETL::Pequel::DISPLAY_MESSAGE_ON_INPUT_ABORT
ETL::Pequel::DISPLAY_MESSAGE_ON_OUTPUT
ETL::Pequel::DISPLAY_MESSAGE_ON_OUTPUT_ABORT
```

section

The *section* function of the *ETL::Pequel* object requires a single parameter containing the section name. It will return a pointer to the section object.

```
my $s = $p->section(ETL::Pequel::OPTIONS);
```

addItem

The *addItem* function of the section object is used to add items to a section. This function requires one or more named parameters. Some parameters are optional. The following table lists the parameter requirements for each section type:

```
options ( name [ value ] )
field_preprocess ( name type [ operator calc ] )
field_postprocess ( name type [ operator calc ] )
description ( value )
use_package ( value )
input_section ( name [ type operator calc ] )
output_section ( type field [ clause ] )
filter ( value )
reject ( value )
sort_by ( fld [ type sort ] )
group_by ( fld [ type sort ] )
sort_output ( fld [ type ] )
dedup_on ( fld [ type ] )
having ( value )
divert_input_record ( value )
copy_input_record ( value )
```

```
divert_output_record ( value )
copy_output_record ( value )
display_message_on_input ( value )
display_message_on_input_abort ( value )
display_message_on_output ( value )
display_message_on_output_abort ( value )
init_table ( name key values )
load_table ( name filename keycol keytype field_list )
load_table_pequel ( name scriptname keyfield [ keytype ] )
```

```
$s = $p->section(ETL::Pequel::INPUT_SECTION);
$s->addItem(name => 'PRODUCT_CODE',     type => 'string');
```

prepare

The *prepare* function of the *ETL::Pequel* object requires no parameters. It should be called after all the section items have been filled.

generate

The *generate* function of the *ETL::Pequel* object requires no parameters. It should be called after *prepare*. This function will generate the perl code for the *ETL::Pequel* object.

check

The *check* function of the *ETL::Pequel* object requires no parameters. It should be called after *generate*. This function will syntax check the generated perl program and return *'Syntax OK'* or an error message if the syntax check fails.

execute

The *execute* function will do just that — execute the generated program.

printToFile

The *printToFile* function of the *ETL::Pequel* object requires a single parameter containing the file name wich will contain the generated Perl program. It should be called after *generate*. This function will sace the generated code in the external file.

```
$p->engine->printToFile("$0.2.code");
```

INSTALLATION INSTRUCTIONS

Pequel is installed as a Perl module.

```
perl Makefile.PL
make
make test
make install
```

to specify different perl library path:

```
perl Makefile.PL PREFIX=/product/perldev/Perl/Modules
```

Installation Troubleshooting

When installing into non-default directory, i.e., if you used the ***PREFIX***, then you need to (probably) set the ***PERL_INSTALL_ROOT*** environment variable before 'make install'

```
export PERL_INSTALL_ROOT=/product/perldev/Perl/Modules
```

set this to whatever you specified for ***PREFIX*** above.
You will also need to set the ***PERL5LIB*** and ***PATH*** environment variables before executing *pequel*. To set ***PERL5LIB*** note the Installing messages displayed during the *make install*, and set this to the path up to and excluding *pequel*. For ***PATH*** add the directory containing the Pequel executable to the PATH variable — note the installation messages for *.../bin/pequel* — add this path to the ***PATH*** environment variable.

Example Installation

```
> perl Makefile.PL PREFIX=/usr/local/Perl
Checking if your kit is complete...
Looks good
Writing Makefile for ETL::Pequel

> make
Skip lib/ETL/Pequel.pm (unchanged)
Skip lib/ETL/Pequel/Main.pm (unchanged)
Skip lib/ETL/Pequel/Param.pm (unchanged)
Skip lib/ETL/Pequel/Type.pm (unchanged)
Skip lib/ETL/Pequel/Script.pm (unchanged)
Skip lib/ETL/Pequel/Field.pm (unchanged)
Skip lib/ETL/Pequel/Error.pm (unchanged)
Skip lib/ETL/Pequel/Engine.pm (unchanged)
Skip lib/ETL/Pequel/Engine/Inline.pm (unchanged)
Skip lib/ETL/Pequel/Collection.pm (unchanged)
Skip lib/ETL/Pequel/Code.pm (unchanged)
Skip lib/ETL/Pequel/Docgen.pm (unchanged)
Skip lib/ETL/Pequel/Parse.pm (unchanged)
Skip lib/ETL/Pequel/Pod2Pdf.pm (unchanged)
Skip lib/ETL/Pequel/Lister.pm (unchanged)
Skip lib/ETL/Pequel/Table.pm (unchanged)
Skip lib/ETL/Pequel/Type/Date.pm (unchanged)
Skip lib/ETL/Pequel/Type/Section.pm (unchanged)
Skip lib/ETL/Pequel/Type/Option.pm (unchanged)
Skip lib/ETL/Pequel/Type/Macro.pm (unchanged)
Skip lib/ETL/Pequel/Type/Aggregate.pm (unchanged)
Skip lib/ETL/Pequel/Type/Db.pm (unchanged)
Skip lib/ETL/Pequel/Type/Db/Oracle.pm (unchanged)
Skip lib/ETL/Pequel/Type/Db/Sqlite.pm (unchanged)
Skip lib/ETL/Pequel/Type/Table.pm (unchanged)
Skip lib/ETL/Pequel/Type/Table/Oracle.pm (unchanged)
Skip lib/ETL/Pequel/Type/Table/Sqlite.pm (unchanged)
Manifying blib/man3/ETL::Pequel::Pod2Pdf.3

> export PERL_INSTALL_ROOT=/usr/local/Perl
> make test
t/01_aggregates_1........ok
t/02_array_fields........ok
t/03_conditional_aggr....ok
t/04_filter_regex........ok
t/05_group_by_derived....ok
t/06_group_by_derived_2..ok
t/07_hash_option.........ok
```

```
t/08_local_table.........ok
t/09_macro_select........ok
t/10_output_calc_fields..ok
t/11_statistics_aggr.....ok
t/12_statistics_aggr_2...ok
t/13_transfer_option.....ok
t/14_simple_tables.......ok
t/15_external_tables.....ok
t/16_sales_ttl_by_loc....ok
t/17_pequel_tables.......ok
t/18_chain_pequel........ok
t/19_divert_record.......ok
t/20_copy_record.........ok
t/21_copy_output.........ok
t/22_output_combiner.....ok
All tests successful.
Files=22, Tests=22, 71 wallclock secs (64.37 cusr +  6.79 csys = 71.16 CPU)

> make install
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Param.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Code.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Collection.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Docgen.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Engine.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Error.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Field.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Lister.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Main.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Parse.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Pod2Pdf.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Script.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Table.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Aggregate.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Date.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Db.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Macro.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Option.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Section.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Table.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Table/Oracle.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Table/Sqlite.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Db/Oracle.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Type/Db/Sqlite.pm
Installing /usr/local/Perl/usr/perl5/site_perl/5.6.1/Pequel/Engine/Inline.pm
Installing /usr/local/Perl/usr/local/Perl/man/man3/Pequel::Pod2Pdf.3
Installing /usr/local/Perl/usr/local/Perl/bin/pequel
Installing /usr/local/Perl/usr/local/Perl/bin/pequelpod2pdf
Writing /usr/perl5/site_perl/5.6.1/sun4-solaris-64int/auto/Pequel/.packlist
Appending installation info to /usr/local/Perl/lib/sun4-solaris-64int/perllocal.pod

> export PERL5LIB=/usr/local/Perl/usr/perl5/site_perl
> export PATH=$PATH:/usr/local/Perl/usr/local/Perl/bin
> pequel -v
pequel Version 2.4-4, Build: Tuesday November  1 23:45:13 GMT 2005
```

Using Inline

Certain options (such as *use_inline*, *input_delimiter_extra*) will cause ***Pequel*** to generate embedded C code. The resulting program will then require the Inline::C module and a C compiler system to be available. Once you have Inline::C installed you can verify its availability to Pequel by running a compile-check on the *apachelog.pql* script

```
pequel -c examples/apachelog.pql
```

BUGS

- The Inline Oracle and Sqlite Tables functionality as of version 2.4-x requires further extensive testing.

- Array fields and macros not handling single element arrays.

- &period and &month not implemented.

- ***summary section*** is not implemented.

- If you specify ***group by*** you must also specify ***sort by*** (unless your input is already sorted in the required order or hash is specified).

AUTHOR

Mario Gaffiero <gaffie@users.sourceforge.net>

COPYRIGHT