Perl-SQL:

UNIX command line filters
for management of experimental data
in sparse and irregular relations

Andy Glew

May 12, 1998

1 Overview

1.1 What Perl-SQL is

Perl-SQL is an application program that allows users to perform SQL-like queries on databases stored in **ordinary text files**. This allows the user to manipulate the data using symbolic names, rather than using relative field numbers as in standard UNIX tools like sort and join.

E.g. `perl-sql 'SELECT Players,Points/AVG(Points) FROM "test.mnt" WHERE Points>3'`

E.g.
```
perl-sql 'select t1.count, t2.count from "profile1.db" as t1
                                  full outer join "profile2.db" as t2'
```

E.g. *`some-other-command`* | `perl-sql SELECT a, a/b` | *`yet-another-command`*

Perl-SQL supports the SELECT statement, with an optional WHERE clause. A wide variety of Perl and SQL expressions are supported. Aggregate operations to compute functions such as arithmetic, geometric, and harmonic means, standard deviations, sums, and max and minimums are supported. Explicit joins, including FULL OUTER JOINs, are supported. These operations are convenient for manipulating and comparing experimental data.

Apart from being a "plain text database", Perl-SQL's most notable feature is that it supports ***sparse and irregular databases***: unlike normal SQL databases, Perl-SQL does not require every tuple in a relation to have the same shape. Some of the database file formats that Perl-SQL supports are self schematizing, in that every record fully describes its format. It has been the author's experience that sparse and irregular databases occur naturally in his work, managing experimental data in computer architecture, and that managing such data with conventional databases is onerous.

Perl-SQL supports a **variety of file formats**, including the single-line nested tuples (SNTs) described above, multi-line nested tuples (MNTs), and fixed format single-line delimited records (SDRs). SDRs include common database formats supported by tools such as Excel, such as comma separated variables (CSVs) and tab separated variables (TSVs).

Many other programs, such as plotting packages, allow filter commands to be invoked on data. Perl-SQL brings many advantages to such data manipulations. Perl-SQL can also be used in UNIX pipelines. In the example below Perl-SQL is used to import particular variables by name from a database into the GNUPLOT graphing program, and immediately plot a graph – without having to

E.g.
```
gnuplot> plot '< ../perl-sql -format csv
       select Years, Points from "test.db" '
```

Compare to a similar command, from gnuplot help, using non-symbolic names:

Anti-Example:
```
gnuplot> plot '< awk "$0 !~ /^#/
       {print $1-1965, $2}" population.dat '
```

1.2 Who should consider Perl-SQL

Perl-SQL was designed to facilitate the management of databases containing experimental data from computer simulations. Many years of experience and frustration led to the author's refining of the concept of sparse and irregular databases and self-schematizing files, and to decisions such as using ordinary files. The author has encountered many people equally frustrated with data management tools. As such, you probably know who you are.

However, there is a much larger audience for Perl-SQL: people who should be frustrated with data management tools, but whose thresholds are set relatively high.

Probably, the best way of determining whether you are one of the people for Perl-SQL was written is the following test:

- Have you written more than a dozen Perl, AWK, shell or other scripts to extract data from a program that you have written, to convert it into a format used by some other tool?
- Have you used UNIX shell commands like sort, join, and awk enough to be frustrated and confused by which tools number fields starting at $0, and which at $1?
- Have you ever hacked up a tool to obtain a data value from a data value by copying an existing script, changing the name searched for, and creating a new script?
- Have you ever had to repair more than a dozen scripts produced as above when the output format of the tool that produced the original data changed?

I.e. are you somebody who regularly creates ad-hoc scripts to manipulate very similar pieces of data?

Perl-SQL will not eliminate the need to create scripts. Some complicated Perl-SQL scripts will still be necessary to store in files, i.e. be scripts. It will, I hope, reduce the need to write such scripts. Small queries can be directly typed in, and large queries will be reduced in size.

1.3 Usage

As is indicated in the examples above, Perl-SQL is invoked from the command line. Its help message describes the options:

```
-h / -help / --help     this help message

-format                 format to output results in
-format multiline_named_tuples
-format mnt             multiline named tuples
                        e.g {
                              field => var,
                            }
-format singleline_named_tuples
-format snt             e.g. { field => var }
-format singleline_delimited_records
-format sdr             generic single line records
                        TBD: specify options such as delimiter,
                        comment character, name specification, etc.
```

```
-format csv             comma-separated variables
-format tsv             tsv-separated variables
-headers                print comment headers fully describing data format
-syntax-check-only      check the SQL statement syntax, but do not
                        try to execute it.
```

The `-help` option prints the usual help message. And, as usual, the help message describes the options, but not the language or file formats or any of the most useful stuff.

The `-format` option specifies the format that the output database is produced in. The default format, MNT, is the most general, while the CSV and TSV formats are those most commonly supported by other tools.

The `-headers` option produces comments at the top of the output file, describing the database format. These comments can be used by Perl-SQL when the database is subsequently read in, reducing the need to guess at what the format is. For example:

```
# database file opened as: <test-csv-sprite
# format:                  singleline_delimited_records
# field-delimiter:         ,
# field-names:             Player, Years, Points, Rebounds, Assists, Championships
# ignore-whitespace:       0
```

The `-syntax-check-only` option indicates that the SQL statement should not be executed, but should only be subjected to a syntax check.

The `-headers` and `-syntax-check-only` options are binary options: i.e., variants `-no-headers, -headers=0`, and `-headers=1` are supported, with obvious meanings.

Any command line argument beginning with a '-' is interpreted as a possible option, and is rejected if it is not one of the above short list of supported options.

All other command line arguments are concatenated, with blanks separating them, to form a single SQL command string. (See special handling of filenames below.) Thus, both of the following are accepted:

E.g. `perl-sql 'SELECT  a - b AS ABdiff FROM "test.db" WHERE a > b'`

E.g. `perl-sql SELECT  'a - b' AS ABdiff FROM '"test.db"' WHERE 'a > b'`

Note the use of quotes to protect SQL statements that contain shell meta-characters such as '>', or words that would otherwise be interpreted as options beginning with '-'. Quoting the entire SQL statement is often best, but even here it is necessary to be careful with SQL single quotes for strings, and double quote for delimited identifiers.

This section's purpose was to describe the perl-sql command and its handling of command line options and arguments. A different section describes the SQL language implemented by Perl-SQL. However, let me briefly emphasize here the syntax for filenames within Perl-SQL. Filenames are considered to be identifiers. Ordinary identifiers follow the alphanumeric variable name pattern typical of many programming languages. Unfortunately, most common filenames are not alphanumeric variable names, but contain special characters such as slash "/", period ".", and dash "-". To support these, SQL "delimited identifiers" allow arbitrary character strings to be used as identifiers, when surrounded by double quotes '"'. Unfortunately, this frequent use of double quotes often makes the syntax, when the perl-sql application is invoked from the shell, ugly: e.g '"test.db"' (a double quoted delimited identifier filename inside a sing;e quoted shell word) rather than "test.db" (where the double quotes will be stripped away by the shell, if they are not externally quoted in some way).

Finally, two 'conveniences' have been added to the Perl-SQL application program:

First: If perl-sql is executed via a link, and the name of the link ends with an SQL command ('select', or 'update', or 'insert', or 'alter', or 'delete'), then this SQL command is assumed. I.e. the 'perl-sql' can be elided.

E.g. `select Player, Year < /unsup/perl-sql/test.mnt`
E.g. `select Player, Year from '"/unsup/perl-sql/test.mnt"'`

Note that normal UNIX input direction is used in the first example above. This is often convenient, reducing the need for mixing shell quotes and SQL quotes, as is done in the second example. Note that quoting of the command line arguments is still necessary.

Second: because mixing of shell quotes and SQL quotes is annoying, a special kluge has been added to the perl-sql application program command line parser – ***not*** the version of SQL supported internally. If a command line argument to the application program (argv[i] in C, $ARGV[$i] in Perl)

+ contains special characters requiring double quoting in Perl-SQL to be treated as a filename
+ isn't obviously an operator like '/' or '.'
+ follows a FROM
+ contains no whitespace

then it will be double-quoted by the perl-sql application program before being concatenated with the rest of the command line arguments and handled to the SQL parser. I.e. the double quotes in many of the examples above are unnecessary.

E.g. `select Player, Year from /unsup/perl-sql/test.mnt`
equals `perl-sql 'select Player, Year from "/unsup/perl-sql/test.mnt"'`

It should be emphasized that these two features are just conveniences. Using them may not always produce the clearest statements. If in doubt, specifying the perl-sql command explicitly, single quoting the entire SQL statement, double-quoting all filenames within the SQL statement, and escaping single quoted SQL strings, should work with most shells.

1.4 About this document

This document is a sort of combination of manual pages, design documents, wish list, and advertising pamphlet for Perl-SQL. It is all of these not because that's the right thing to do, but because the author is lazy.

Section 1.1, Overview, is sort of an advertising flyer for Perl-SQL. It tries to briefly describe the features of Perl-SQL in a format that might excite interest, but which cannot be used as documentation. It may appeal to a technical user who has already been looking for such a tool. Unfortunately, it will probably not appeal to the sort of technical user who needs a tool like this, but who has not verbalized this need. Section 1.2, Who should consider Perl-SQL, is supposed to appeal to this sort of potential user.

Section 1.3, Usage, describes the Perl-SQL application command line invocation, specifically command line options. Similarly, section 1.6, Local Issues, describes how Perl-SQL is installed locally, at the University of Wisconsin CS department, while section 1.7, Obtaining Perl-SQL, describes how outside users can access it. Again, it is targeted to people who don't want to read manuals, but want to get started quickly.

Section TBD is reference material about Perl-SQL. This reference material blends quickly into a design rationale, and a list of desired features and enhancements.

1.5 History

1. Started writing this document April 3, 1998.

2. First reasonably complete version of Perl-SQL documentation May 12, 1998.

1.6 Local Issues

At the University of Wisconsin Department of Computer Science, Perl-SQL has been installed as an unsupported software package, in `/unsup/perl-sql`. The present version is `/unsup/perl-sql-1.0`. Usage is invited, but Perl-SQL is not yet considered stable enough to place in the synthetic package `/unsup/std`. For example, the CVS trees of `/unsup/perl-sql` still point to the author's home directory.[1]

It will therefore be necessary to either put `/unsup/perl-sql/bin` in your path, or to invoke `/unsup/perl-sql/bin/perl-sql` directly.

The actual application program is `/unsup/perl-sql/bin/perl-sql`. A symlink `/unsup/perl-sql/bin/select` provides a shorthand, as described in section 1.3 above. Other potential shortcuts are not provided. In addition, `/unsup/perl-sql/bin/ssout-to-mnt` is a filter that converts the output of the SimpleScalar simulation tool to the MNT format supported by perl-sql.

1.7 Obtaining Perl-SQL

As mentioned above, Perl-SQL is locally installed at the University of Wisconsin as `/unsup/perl-sql`. This is accessible to AFS file systems around the world as `/afs/cs.wisc.edu/unsup/perl-sql`.

TBD: FTP, web, and CPAN PAUSE access.

The author can be reached as `glew@cs.wisc.edu`.

2 Introduction

Perl-SQL is a set of tools that I have developed, and, I hope, will continue to develop, for management of the experimental data that I collect in my research. I started writing it in frustration at (a) 15 years of using ad-hoc Perl and AWK scripts, and (b) never finding a database convenient or easy to use.

This document does not describe the finished project. I am writing it now because I want to use it as a bit of a design document, to see if I turn up anything obvious that I need by observing holes in the documentation.

Perl-SQL features:

1. All operations can be UNIX-like command line filters.
   They can be combined easily with other UNIX tools. Operations can be performed on explicitly

---

[1] If anyone knows a procedure to maintain two CVS trees that track each other, I would like to know of it. As it is, by having only one CVS tree I can make changes in my private source tree, and release them easily.

named files, or they can be performed on anonymous standard input and standard output, and placed in pipelines such as:

```
generate-data | select address, count  where count > 100 | plot
```

2. All relations are stored in standard UNIX text files.
   This avoids many of the issues of managing "real" databases: creating the database; allocating disk space for it; keeping the database server daemons running; performing backups; and, worst, the danger that you may not be able to read old database files if the database file format changes.

3. The database file formats are self-describing.
   Symbolic field names are placed in the file, and can be used in the commands, rather than the relative field numbers used by standard UNIX commands like sort.

4. The database file formats are self-schematizing.
   There is no need to create a database schema describing the fields of a relation, and then to import data. The database file formats implicitly describe their own schema.

5. The database file formats support irregular and sparse relations.
   A characteristic of evolving experimental data is that new measurements are added and old deleted over time. I.e. not every record of the database has the same fields, but nonetheless you want to be able to compare the records. Similarly, representing all fields in all records, including non-present fields, imposes a large space overhead in the "real" databases that I am familiar with. Perl-SQL handles such irregular and sparse relations naturally, and has convenient handling of nonexistent and undefined fields.

6. The file formats presently supported include Single-line Delimited Records (SDRs), Single-line Named Tuples (SNTs), and Multi-line Named Tuples (MNTs).
   Standard file formats such as comma separated variables (.CSV), tab separated variables (.TSV), and white-space separated variables are special cases of SDRs.

7. A powerful heuristic inspects the database files and nearly always succeeds in figuring out what the database format is, without having to rely on naming conventions.

8. All database file formats support internal comments: #, !, //, etc.

9. A subset of standard SQL's data manipulation language is supported.
   Supported: SELECT fields FROM table
   Supported: SELECT fields FROM table JOIN table ON ...
   equijoins only, on pre-sorted files
   Not supported: comma joins: SELECT fields FROM table1, table2
   why not: parsing, with limited query optimization ability
   Supported: UPDATE ... SET field ...
   Extensions are made to support command line filtering.
   Extensions are made to relax ordering and syntax rules for convenience.

10. Most Perl operators are supported.
    In particular, && and || operators make dealing with NULLs (nonexistent and undefined fields) considerably more convenient than in standard SQL.

11. OUTER JOINs are supported.
    OUTER JOINs are the basic tools for comparing irregular data. A surprisingly large number of SQL databases do not support them.

3 Motivation

Here are some of the reasons that I started writing Perl-SQL. I list them, because they may help others decide whether or not they want to use it.

- Tired of 15 years of writing AWK & Perl scripts to extract and manipulate data

  I have been working in computer systems performance analysis since the mid-80s, almost 15 years. In that time I have found that my coworkers and I constantly write and rewrite scripts to manipulate data. At first these were typically written in AWK; the arrival of Perl was a great improvement, but nonetheless I continue to find that many tasks are being constantly re-programmed.

  ☐ Too many different scripts

  It is not uncommon to have dozens of different but similar scripts. In the words of another student of computer architecture "I have ZILLIONS of scripts! Every time I run a new experiment I write new scripts!" Such scripts proliferate because they are ad-hoc, written as necessary. But, unfortunately, they often are not immediately thrown away, but must be maintained.

  A new script is typically copied from an earlier script, slightly changed to calculate slightly different performance metrics. For example, one of my friends allowed me to list the names of his core set of scripts for one simulator: `grab`*`iefflat`* `grab`*`imissinst`* `grab`*`imissrate`* `grab`*`ipc`* `grab`*`mispredinter`* `grab`*`mispredintra`* `grab`*`tefflat`* `grab`*`tmissrate`* `grab`*`tmisstask`*. Each script "grabs" a single performance metric (italicized) from the simulator output file. A metric is sometimes a single variable, and at other times must be calculated. Then another set of scripts massages the data for eventual manipulation by tools such as Excel. This friend estimates that it typically takes him an hour to go from a completed simulation run to a graphical depiction of the data that he can analyze – an hour full of running scripts, cutting and pasting, copying files, etc.

  At first I thought that my friends and I might just be lousy programmers, but experience at several companies and universities has indicated that, at least, this situation is widespread. Many other researchers are much less organized than my friend above: it is not unusual to open some researcher's working directories, to see 2 or 3 dozen scripts, with names like 'extract.conference-paper.expt1.graph3'.

  ☐ Scripts are hard to maintain

  I am sure that most people with computer experience will appreciate that such replicated scripts are hard to maintain, for example, when a change is made to the output format of one of the tools they are based on.

  ☐ Standard UNIX tools like AWK, SORT, CUT, and JOIN use field numbers instead of names

  Of course, such data manipulation scripts are often rolled using existing components, such as the standard UNIX tools SORT and JOIN. These tools, however, often use field numbers such as $0, $1, rather than symbolic variable names such as 'Experiment' and 'Cache_Miss_Rate'.

  ☐ Scripts can be buggy

  I have overheard conversations such as 'This data doesn't make sense. Have you fixed that bug in your [data extraction] script?' I have heard apologies at conferences, where the data plotted in the paper was incorrect because of a bug in the script, but the data presented on the overhead is

'alright now'. I myself have had bugs due to thinking that $3 was one variable, when in fact it was another.

Moreover, the duplication of scripts makes best-known methods hard to propagate. Even a simple task such as summing as list of numbers should not be programmed on an ad-hoc basis, if the numbers vary dramatically in size – numerically safe techniques such as sorting the list or using the Kahan summation formula should be used.

The reasons above indicate that *there's a whole lot of scripting going on*. This may not seem to be bad from the point of view of a professor, who has a lot of graduate students doing it for him, or even from the point of view of a senior researcher in industry, who may have underlings doing it for him. It is, however, bad if you are the graduate student or the underling; and it may be bad if it wastes the time of highly paid workers in industry.

I make these caveats because I have been accused of trying to automate things that should not be automated, eliminating script writing completely. In my defense I say that I do not think that all script writing can be eliminated. I think, however, that some frequently performed tasks can be improved upon. More Perl modules to make writing such scripts easier may help, but more widespread use of tools such as SQL will also help.

- Existing tools are unsatisfactory

  I have tried to use existing tools, such as relational databases and statistics package, on several occasions. Most recently I spent almost a month evaluating such tools, performing typical tasks. In the past I have had people working for me propose and implement relational database systems for managing the experimental data that we were working with. Furthermore, I have formally interviewed several other researchers who have gone the same route, who have used relational database management systems in the past, and who have given up. I believe that I have identified several reasons for this failure to keep using RDBMS:

  - RDBMS require too much administration for a single user or small group.

    Typical RDBMS require (1) a special "database" user – which unprivileged users of conventional OSes cannot set up without requiring special permission, and (2) require at least one running database server process. Furthermore, databases often (3) bypass the standard file system, which often (3') means that special backup procedures must be followed, and filespace must be specially allocated. None of this is too hard, just a hassle.

    Single user databases such as Microsoft Access require less administration, but are similarly less featured, and fall down on the other complaints.

  - RDBMS databases are not very portable or survivable across software installations.

    It is often a pain to convert a database developed with one RDBMS, say, Oracle, to a different commercial RDBMS, say, Informix – such as is required if, for example, you change universities or employers. Furthermore, RDBMS data files occasionally change format as new versions of the software are released. Problems may be encountered attempting to read very old, archived, RDBMS data. Dumping the database to a fairly universal format such as text files tends to answer both of these problems, but then raises the issue "Why not keep the data as text files to begin with?" so that you never have to worry about when to convert.

  - RDBMS are expensive.

It is hard to make the jump from essentially free tools like Perl, to 1000$ a seat, when the utility of RDBMS in this application area is not proven. Hard for industry, it is especially hard for academia.

- Classical relational databases are fixed format, whereas experimental data is often sparse and irregular.

  In my opinion, this is one of the biggest problems with the use of conventional RDBMS in my application area. Classical RDBMS require database schema to be carefully crafted. In my experimental work, however, I find that I am nearly always adding a new observation or metric every time I modify the simulator.

  Reformatting the database to add new fields on every run is possible, but most RDBMS make such operations (typically ALTER TABLE) slow and painful. Furthermore, this leads to a situation where many, if not most, fields contain no data – what I call "sparse" databases. In RDBMS this is typically implemented via NULLs. Many RDBMS, however, are not very efficient at handling such sparse databases, dull of NULLs, and impose a space penalty. Furthermore, while SQL does support use of NULLs in queries, the support is inflexible and inconsistent – so inconsistent that famous relational database advocates such as Date seriously propose abandoning NULLs completely. I, on the other hand, think NULLs and clean handling of sparse and irregular data are extremely important, and hope that there will be improvement in this area that will answer the valid concerns about their inconsistency.

  Normalization of the database to tables that look like

  ```
  Experiment_Number, Metric_Name, Metric_Value
  ```

  could be an efficient implementation of sparse and irregular data in a conventional fixed format RDBMS. Apart from the requirement of having multiple such tables for each data type of Metric_Value (some metrics are numeric (integer or floating point), while others are symbolic, while still other data types are natural, this normalized form makes queries quite unnatural.

  Finally, experimental data often has structure that the flat representations of conventional RDBMS do not support: arrays, histograms, etc. I.e. experimental data needs the sort of support provided by object relational databases.

- Many RDBMS implementations are incomplete

  While the relational model of data is definitely something that I would like to use to manage my experimental data, I found that many RDBMS implementations were incomplete. In particular, FULL OUTER JOINs are, to me, one of the most natural ways of comparing two datasets, and are especially important because the outer join indicates what data points are missing from one or the other experiment. A surprisingly large number of database products has no support for FULL OUTER JOINs.

My own experience, and that of the people I interviewed, is that conventional RDBMS are useful for "production" experiments: when a large number of similar experiments on various configurations is being run, when the schema of the database being collected is well known. But usage for "development" experiments is much more difficult. Furthermore, everyone says things like "Well, I tried used an RDBMS, and it worked fine, but then I changed sites (companies, universities, computer systems) and it was too much hassle. I always meant to get back to using a database, but I just never did.

Similarly, I considered using statistical data management packages such as SAS or SPSS. While these have many advantages, they have less support for ad-hoc queries on sparse and incomplete data than do RDBMS. The notes on my evaluations were posted to several newsgroups, and are summarized in section 6 on page 21.

Overall, therefore, I started writing Perl-SQL because I want to spend more time looking at results, and less time writing scripts to look at results. So therefore, of course, I spent a month working on it, and will probably spend more time in the future...

4 Database file and input/output formats

Perl-SQL supports three basic file formats:

1. Multi-line Named Tuples (MNT): the most flexible.
2. Single-line Named Tuples (SNT).
3. Single-line Delimited Records (SDR): the least flexible, but which subsume some formats widely supported by other databases.

All of these formats support variable names within the file: the Named Tuple formats on a record by record basis, the SDR format on an entire file basis.

All of these formats support comments, typically from # to end of line.

4.1 Named Tuples (MNT and SNT)

For example:

```
# Comment....
{
          experiment => 'Test OOO', IPC => 4.6,
          benchmark => 'gcc'  # more comments
},
{
          IPC => 9, benchmark => 'Test Superspeculative',
          benchmark => 'eqntott', cache_miss => 0.13,
}
```

The term "named tuples" is used rather than "records" here to emphasize that the data is irregular and relatively free format.

Many readers will recognize that MNTs are basically Perl syntax for literal hash arrays. In fact, in the original Perl-SQL implementation, each tuple is read in and is then subjected to Perl 'eval'. Using eval provides great flexibility, e.g. allowing Perl operators to be used to define keys and values, and allowing nested data structures using {} for hashes and [] for arrays, but does have some security issues. For example, an MNT could make reference to Perl variables in the Perl-SQL application program itself. (TBD: sandbox the data file parsing.)

Comments beginning with the character '#' and continuing to end of line are supported. ('#' is ignored within strings.)

Each multi-line named tuple begins with a line consisting solely of an open curly brace, '{'. All lines up to the next line consisting solely of a closing curly brace, '}' or a brace and a comma '},' are collected and eval'ed within Perl. To be compatible with Perl commas should separate tuples, although this may not be strictly enforced. The Perl syntax will be briefly described.

The tuple consists of (key, value) pairs. The key is placed to the operand of the '=>' operator. The key should be a string, and can be placed within single or double quotes, as described below; however, since =>

expects a string as its left operand, the quotes are optional for most keys. Numeric keys are accepted, but will be treated as strings.

The value is any Perl scalar value. Numbers are in conventional decimal integer (123), floating point (1.23E4), octal (017) ands hexadecimal (0xF00) forms, String values can be enclosed by single quotes or by double quotes. Inside double quotes special characters such as "\n" may be interpolated. (I recommend that single quotes be used wherever possible in the data files, if only for consistency with SQL, where single quotes indicate strings and double quotes delimited identifiers.)

Perl expressions such as '1 + log(2)' may be used. In particular, string concatenation using the '.' operator is supported.

Complex data structures may be represented as field values, using Perl's {} and [] constructors. Strictly speaking these return references, but for Perl-SQL's purposes {} may be considered to build hashes (key / value structures like those of the tuple), while [] builds arrays. Currently, Perl-SQL has minimal support for such complex data structures: basically they can be read in and printed out, but when printed out they occupy a single line. No operators currently support such complex structures.

TBD: actually use complex structures.

There is no requirement that the fields be specified in the same order in every tuple. Nor do all tuples need to contain the same fields.

Single-line named tuples (SNTs) are likewise also Perl hashes, but must be entirely specified on one line.

On output, the named tuple data file formats are printed out tuple by tuple, field by field. The keys are printed in alphabetic order. (TBD: should optionally provide a different order.) The values are whatever is printed by Perl's standard Data::Dump module.

4.2 Single-line Delimited Records (SDR)

SDR are a limited, fixed format, data type, but are provided because other programs commonly support them.

Each line of an SDR defines a record. Whitespace, tabs, commas, or nearly any other character may separate fields within a record.

Perl-SQL applies a heuristic to guess the format of a database file. This heuristic applies to all file types, but is most important in determining the type of SDR files.

If the first non-blank line begins with one of the strings '#', '!', or '//', this string is assumed to introduce comments.

The following special comment lines are supported (using whatever is the comment character):

```
# field-names: field1, field2, field3...
# field-delimiter: string
# comment: string
# format: {format-name}
# ignore-whitespace: [0/1]
```

The 'comment' special comment line is self-referential, but has some purpose as documentation. It does not affect the MNT or SNT formats.

The ‘format’ special comment can be set to ‘multiline_named_tuples’, ‘singleline_named_tuples’, or ‘singleline_delimited_records’. TBD: allow abbreviations like MNT, SNT, SDR, CSV, and TSV, to be set in format, as on command line.

The ‘field-delimiter’ special comment allows any character or string to be specified as separating fields within a record. Common choices include comma, colon, etc. The words ‘tab’ and ‘whitespace’ indicate the corresponding characters.

The ‘ignore-whitespace’ special comment can be set to 0 or 1. If set to 1, indicates that whitespace surrounding the delimiter character is ignored. If clear to 0, leading and trailing whitespace can be placed in a field.

The ‘field-names’ special comment (also ‘fieldnames’ or ‘fields’) indicates the names to be used to manipulate the fields of a record. Currently, commas must separate the fieldnames. If more fields exist in a record than are defined, names of the form ‘field1’, ‘field2’, etc. may be used.

If the special comments are not present, the heuristic will look at the first few non-comment lines, and will guess as to whether the delimiter is comma or some other special character. If this guess is inappropriate it is best to explicitly specify the delimiter and other format items via special comments.

If the first non-comment line of the file appears to consist of names of the form ‘letter followed by letters and numbers’, then this line is considered to define the fieldnames if not already defined.

```
Player,Score
Kareem Abdul-Jabbar,24
...
```

TBD: This ‘first line is fieldnames’ format is incompletely supported, and may have bugs if the data file is rescanned, as for an aggregate function.

Thus, it can be seen that the SDR data file format comprehends widely used data file formats such as CSV (comma separated variables) and TSV (tab separated variables). Currently, no quote syntax is defined within the SDR format, so this compatibility is only partial.

TBD: the format guessing heuristic should be extended to allow (a) complete specification of the input file format by the user, and (b) the ability to look at common filename suffixes, such as *.CSV and *.TSV.

Currently, the format of the output file can be specified on the command line as described in section 1.3 on page 2: basically, as MNT, SNT, SDR, CSV, or TSV.

TBD: it should be possible to fully specify any of the parameters for the SDR format (delimiter, comment character, first line comments, etc.), on the command line, implicitly via filename, or via some syntax inside SQL itself. Similarly, it has also been suggested that the name of the tool receiving the output be an option: e.g. ‘-format excel’ or ‘-format gnuplot’.

4.3 Format Conversion

Tools may be necessary to convert formats from the output files emitted by various programs such as computer architecture simulators, to the MNT, SNT, or SDR format supported by Perl-SQL. The MNT format is recommended, for obvious reasons of flexibility.

One such tool has already been written: `ssout-to-mnt`, which converts from the partially structured output format of SimpleScalar. Several comments may be useful for the design of other such converters:

- If the foreign file format is somewhat generic, for example containing lines of the format
  name: value
  or
  name <tab> value
  then this should be taken advantage of.

- If the foreign file format contains dates or comments, etc., try to extract and preserve them.

- Check for multiple definitions of the same field

- Embedding the original, entire, foreign file as a comment in the MNT file produced is an attractive way of ensuring that no information is lost.

TBD: allow such filters to be implicitly associated with file types (suffixes), so that they are always run, i.e. so that the intermediate MNT file format need not be produced.

4.4 More Formats Needed

TBD:

- Tuple per file ⇔ separate index files for sort orders

  Several users have commented that simulators typically produce one file per experimental run of a configuration for a given benchmark, and that assembling all of this data into a single big file for all of the same 'configuration' is onerous. They have requested that each simulator output file be considered as a single tuple.

  Similarly, for large data sets, external sorting is onerous[2]. This suggests that a sorted index file be prepared, perhaps consisting of `(filename,lseek,keyvalue)` and that this index be sorted separately. This index file format would naturally support tuple per file formats as well as views.

- Treat internal Perl arrays of hashes as relations

  For records like `{ name => 'Andy Glew', jobs => [ { ... } ]`
  would like to support queries such as

```
SELECT jh.manager,c.phone
       FROM applicants.jobs AS jh JOIN companies AS c
       ON jh.company_name = c.name
```

  (or whatever is the proper SQL 3 way).

- Treat internal Perl hashes as relations of (key, value)

  Want to support queries such as selecting the maximum of all fields in any tuple...

- Treat UNIX directories as tuples and/or relations, with file contents as blobs.

---

[2] Especially since standard UNIX sort does not support the named tuple formats, and since I have written no sort tool for them. (TBD: write a sort tool for named tuple formats.)

5 Perl-SQL syntax

Perl-SQL supports a very small subset of SQL: basically, some flavors of SELECT and DELETE.

TBD: Perl-SQL's grammar supports a much larger subset of the language, but most of it is not yet implemented (UPDATE, INSERT, ALTER TABLE), not for any major reason, but simply because I have not yet needed them.

```
SELECT ... FROM table WHERE ...

SELECT ... FROM table JOIN table ON ... WHERE ...

DELETE ... WHERE ...
```

TBD:

```
UPDATE,
ALTER TABLE ... DROP COLUMNS,
INSERT
```

should be straightforward to implement, and are already in the grammar.
I have had no pressing need for them, since they can easily be synthesized from other operations combining Perl-SQL SELECT and other UNIX command line tools such as /bin/cat, etc.

5.1 Use as filters

Many plotting tools allow filter commands to be invoked as data is read in:

```
gnuplot> plot '< ../perl-sql -format csv select Years, Points from \"test.db\" '
```

Compare to (from gnuplot help):

```
gnuplot> plot '< awk "$0 !~ /^#/ {print $1-1965, $2}" population.dat '
```

Q: do you like saying '$0' and '$1'?

In my dreams, I would like to be able to say something like

```
scatterplot select Years AS x, Points AS y from "test.db"
```

5.2 FROM and INTO clauses

Filenames are specified in the FROM and INTO clauses of such data manipulation statements as SELECT and DELETE.

The use of filenames as table names is fairly standard in the FROM clause.

Perl-SQL's INTO clause is not standard SQL. Rather than specifying a non-SQL variable into which the results of a query shall be put, in Perl-SQL the INTO clause specifies a file (table) into which the results should be put.

If the FROM clause is missing, standard input is used. Similarly, if the INTO clause is missing, standard output is used. Thus, 'SELECT' with no argument copies the table on standard input to standard output, while DELETE with no arguments produces an empty table on standard output.

TBD: support a special syntax such as – or STDIN or STDOUT.

5.3 Clause Ordering

Perl-SQL relaxes the ordering of clauses compared to standard SQL. The FROM, INTO, and WHERE clauses may occur in any order.

5.4 Filenames

Filenames are required in the FROM and INTO clauses of such data manipulation statements as SELECT and DELETE. (TBD: possibly in other places such as ALTER TABLE).

Host operating system (UNIX or NT) filenames are simply SQL identifiers within Perl-SQL. If the filename meets the usual rules for being an identifier (letter followed by letters or numbers), it can be used directly. E.g. if a file is called "database", then `SELECT * FROM database WHERE age > 29` is acceptable.

Most filenames contain special characters such as periods for suffixes or slashes for directory names. Such filenames can be represented as SQL delimited identifiers: any characters within double quotes can be used in the named of a delimited identifier. E.g. if the file has the much more common name of "database.db", then `SELECT * FROM "database.db" WHERE age > 29.`

Filenames, whether simple identifiers or delimited identifiers, can be used wherever appropriate.

E.g.
```
SELECT "database.db".age, "db2.csv".schoolyears
       FROM "database.db" JOIN "db2.csv" ON student_name
       WHERE "database.db".age > 29
```

However, it is usually convenient to define correlation names

E.g.
```
SELECT db1.age, db2.schoolyears
       FROM "database.db" AS db1 JOIN "db2.csv" AS db2
              ON db1.student_name = db2.student_name
       WHERE db1.age > 29
```

As noted in section 1.3 on page on page 2 this use of double quoted delimited identifiers for filenames is easily confused with shell quoting (in UNIX-like shells such as the Bourne shell, the C shell, or GNU bash). A special kluge is described in that section to reduce the need for such double quotes, but this kluge is in the perl-sql application program command line option interpreter, and is not part of the SQL subset supported by Perl-SQL. When in doubt, use single quotes on the command line.

5.5 Field names

As in standard SQL, field names can be simple identifiers (letter followed by letters and numbers) or delimited identifiers in double quotes.

In the data file formats, especially in MNTs and SNTs, care may need to be taken to correctly quote the field names (keys), if special characters are embedded. In SDRs, currently, no attempt is made to handle quoted strings.

The field names of the output file are derived directly from the expression used to evaluate them.

E.g. `/unsup/perl-sql/bin/perl-sql 'SELECT Player, Years/MAX(Years) FROM "/unsup/bin/test.mnt"'`

Produces
```
{
        Player => 'Larry Bird',
        'Years/MAX(Years)' => 0.6,
},
...
```

If such an expression is used for a subsequent operation, it will probably need to be made into a delimited identifier by double quoting:

E.g.
```
bin/perl-sql 'SELECT Player, Years/MAX(Years) FROM "test.mnt"'
| bin/select -format snt '"Years/MAX(Years)"'
```

Giving
```
{       'Years/MAX(Years)' => 0.6, },
...
```

If, however, an SQL derived-column-name is specified using the AS clause on an expression in the select list, then this name is used as the fieldname in the file output.

E.g.
```
select  -format snt Player, 'Years/MAX(Years)' AS YearsFraction
        < ../test.mnt
```

Giving
```
{       Player => 'Larry Bird',    YearsFraction => 0.6, },
...
```

Composite field names are specified as in SQL: `tablename . fieldname`

5.6 AS clause

This is explained in other sections, but is gathered here for convenience: Perl-SQL supports three forms of AS clause:

1. AS derived-column-name clause in the selection list of a SELECT statement

2. AS correlation name clause for filenames (table names) in the FROM clause

3. AS join-variable-name for equijoins.

   This last usage is not standard SQL, and performs a function of the USING clause in standard SQL.

5.7 Not supported

Great big swathes of SQL are not currently supported. This section lists some particular sections explicitly, mainly as I made notes of what things are not even supported by my grammar in Perl-SQL, let alone by database algorithms.

Perl-SQL does not support the data definition language: CREATE DATABASE, DROP DATABASE, CREATE TABLE, DROP TABLE, etc. No access control or security, beyond that afforded by the operating system for files, is provided.

TBD: it might be useful to support CREATE INDEX eventually, for query optimization. Initially,

this would probably be a static representation, not kept accurate.

TBD: eventually may want ALTER TABLE, to add columns (fields) to tuples in a file.

Row value constructors are NOT supported[3]. I.e.

```
SELECT * WHERE (c1,c2,c3) < (1,4,7)
```

TBD: fix

Similarly, table value constructors are not supported.

The pattern matching LIKE operator is not supported. Instead, Perl's =~ operator is supported, since it is more flexible.

TIME and INTERVAL data-types are not supported. The OVERLAPS operator for times is not supported.

ALL, SOME, and ANY quantified comparison predicates (essentially, reductions) are not supported.

TBD: could support, but would like to have generalized handling of reductions first.

GROUP BY and HAVING clauses are not supported.

INTERSECT and EXCEPT are not supported.

Only a limited number of joins: INNER, OUTER, LEFT, RIGHT are supported, and these only equijoins where the tables are already sorted on the join key.

No transactions, concurrency control...

Although eventually desirable, can probably be lived without for an awfully long time, just as, at the moment, workers in this field using Perl scripts live with whatever UNIX gives us in the file system (i.e. little or nothing).

No query optimization

TBD: this is somewhat desirable, e.g. to allow non-sorted joins to be performed.

Currently no nesting of query expressions is supported, e.g. SELECT ... FROM (SELECT)

Highly desirable. Depends on query optimization.

Relation valued expressions such as UNION are not yet supported

TBD: provide, although probably need to have sorting working in order to use in combination with JOINs.

5.8 JOIN syntax

Perl-SQL does not support the implicit comma syntax for Cartesian product JOINs:

Not supported:

```
SELECT * FROM t1, t2 WHERE t1.address = t2.address.
```

Instead, an explicit JOIN is required

```
SELECT * FROM t1 JOIN t2 ON t1.address = t2.address.
```

This limitation arises because we only support sorted equijoins, and do not have sufficient query optimization.

---

[3] I admit, I tried to add them, but broke function call syntax, so gave up.

Perl-SQL does not yet support the USING clause to indicate the JOIN variable, for all JOINs, including OUTER JOINs.

TBD: when supported, the USING clause implies an “eq” (string comparison) type join. The variable specified in the USING clause is implicitly coalesced – i.e. it represents the value being joined on, which is either the same in tuples drawn from each file, or exists in one file’s tuple and not in the other.

Multicolumn USING is not supported. Only a single column is supported.

Similarly, at the moment JOINs can only be done using ON t1.fa = t2.fb type clauses: simple equality of two fields. More complex join syntax is not supported. In particular, multiple field joins are not supported.

The implementation only supports sorted equijoins.

TBD: fix this (possibly by supporting index file formats)

It only supports joins where each join value is unique; i.e. it does not do the Cartesian product if there are multiple tuples in a file with the same join value.

Perl-SQL supports an extended syntax:

```
t1 JOIN t2 ON t1.fa == t2.fb AS fmerge
```

i.e. an AS clause is supported on the merge expression,
obviously meaningful only if the merge expression consists only of two variables with an equality. This AS clause specifies a name for the join variable, coalesced in the case of OUTER JOINs.

I.e. the JOIN/ON/AS clause is similar to a JOIN/USING clause, but permits different fields to be implicitly merged.

TBD: it has been suggested that Perl-SQL simply use the Perl CPAN modules for SET operations. I am somewhat reluctant to do this immediately, before some minimal query optimization is added, since the present sort-merge equijoin is not limited to the size of virtual memory.

5.9 Aggregates

I choose to define the aggregate functions to be null propagating. In this I followed the advice of Mr. Date, rather than the SQL standard. In the SQL standard, `SUM(A+B)` is not necessarily equal to `SUM(A)+SUM(B)`, because if A is null and B is non-null at a given record, then the value of B at this record is included in the latter sum but not the former. In Perl-SQL, by making SUM null propagating, this identity is maintained.

Similarly, MAX and MIN are null propagating,
giving `MAX(max(A,B)) = max(MAX(A), MAX(B)),`
where lowercase max is defined to be a scalar max over two arguments, null propagating, as compared to uppercase MAX over all fields.

If it is desired to obtain the SQL behavior, you can always add NOT NULL to the WHERE condition.

`COUNT(*)` simply counts the number of selected records. NULLs have no relevance here. Note that it is possible to have empty records, multiple times, in a file, and `COUNT(*)` will count them.

`COUNT(expression)` will count the number of non-null occurrences of the expression. Note that NULLs within the scalar expression propagate in most circumstances.

I am not at this time implementing the DISTINCT quantifier for aggregates.
E.g. `COUNT(DISTINCT Years)` is not implemented.

TBD: `COUNT DISTINCT` depends on support for sorting and indexes.

In addition to the standard SQL aggregate functions (COUNT, AVG, SUM, MAX, MIN), Perl-SQL implements some non-standard aggregates convenient for experimental data analysis:

- ARITHMETIC_MEAN

  Same as AVG, = $E(X)$

- HARMONIC_MEAN

  A mean convenient for rates

  $$1/E(1/X) = \frac{1}{\frac{1}{n}\sum_{i=1}^{n} 1/X_i}$$

- GEOMETRIC_MEAN

  A mean convenient for variables whose distribution is not linearly dense, the geometric mean is deprecated since it was commonly overused.

  $$\exp E\{\log(X)\} = \sqrt[n]{\prod_{i=1}^{n} X_i}$$

- STD_DEV

  The measure of variation for the ARITHMETIC_MEAN, I have chosen here to provide the population standard deviation, normalized by 1/n, $\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2}$,

  rather than the sample standard deviation, normalized by 1/(n-1) $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2}$

  TBD: eventually should provide both; I really want a better name than STDEV and STDEVP.

WARNING: All of these aggregate functions are evaluated in the straightforward way by scanning and summing. TBD: numerically accurate algorithms such as the Kahan summation formula should be used.

WARNING: the “first non-comment line of the file contains fieldnames” SDR format is broken with respect to aggregate functions: this first line is erroneously treated as a data record when summing.

Perl-SQL allows the rather trivial extension of mixing aggregates and non-aggregates in the same query. For example, the following is legal in Perl-SQL, although illegal in standard SQL 92:

```
SELECT Player, Points/MAX(Points) FROM "test.mnt"
```

Perl-SQL's rule is simple: if all expressions in an selection list are aggregates, then only one tuple is produced by a query, otherwise multiple tuples are produced.

Perl-SQL does not currently support nested aggregates such as `AVG(X+STD_DEV(Y))`.

Aggregates of any supported scalar expression are supported, such as `1/AVG(1/X)`, which is another way of specifying HARMONIC_MEAN. The names, however, may invoke more accurate algorithms in the future.

TBD: want more aggregate functions such as MODE, MEDIAN, Kth largest, etc.

5.10 Other Functions

All functions using syntax `function_name(argument_list,...)` that are not recognized as aggregate functions (see section 5.9) are passed through to Perl to evaluate. This means that standard Perl built-in functions such as `log()`, `exp()`, and `defined()` work as one would expect.

TBD: unfortunately, the Perl `exists()` predicate does not currently work. It will need to be specially handled. It is extremely desirable, as it allows a distinction to be made between a tuple that does not contain a field, and a field that contains an undefined value.

TBD: allow user defined functions, both scalar and aggregate.

5.11 Types and Expressions

Perl-SQL supports a mix of standard SQL and Perl operators

- infix operators

  addition +
  subtraction –
  "or", "and", "xor"
  logical short circuit &&, ||
  bitwise &, |, ^
  string "eq", "ne", "lt", "le", "gt", "ge"
  arithmetic == < <= > >=
  shift << and >> (TBD: Java >>>)
  multiplication *
  division /, modulus %, "div", "rem", "mod"
  pattern matching =~
  exponentiation **

- prefix unary operators

  unary plus (no action) +
  unary minus (negation) –
  logical inverse !
  bitwise inverse ~
  "not"

Priorities are as in Perl.

Perl-SQL uses Perl's weak typing: fields contain scalar values, and numbers are interchangeable with strings. (TBD: I would like to allow strings to be treated as numbers only if the entire string is a number, not just a prefix as in Perl.) This is straightforward, except with regards to comparisons, where Perl requires separate operators for string and arithmetic comparisons. Perl-SQL implements both 'eq' and '=='; SQL '=' is implemented as 'eq', and '<>' as 'ne'.

0 and non-zero are considered to be Boolean false and true.

TBD: currently || is defined as in Perl. SQL's string concatenation operator || is not supported. But then neither is Perl's string concatenation operator "." (period) since it conflicts with SQL's naming operator.

5.12 NULL handling

Perl-SQL's "null" is Perl's 'undef' value. The literal NULL is not supported (TBD).

Perl-SQL is slightly stronger than Perl with regards to NULL values. Perl treats 'undef' as 0 for many arithmetic operations. Perl-SQL, however, propagates NULLs for most operations.

`NULL + 2` produces `NULL`, not `2`

The only operators for which nulls do not propagate are the short circuit logical operations, || and &&. Expressions such as `fieldname||default` are often convenient, but are dangerous if fieldname might contain 0, since 0 is considered to be Boolean false.

5.13 Complex Data Structures

Although Perl-SQL allows fields to contain Perl hashes and arrays, arbitrarily nested, at the moment nothing can be done with these except to print them out.

TBD: define operators on complex data structures.

6 Issues and Enhancements

Issue: the frequent use of double quotes around filenames makes typing an SQL command in from the shell a pain.

TBD: **allow unquoted filenames with special characters**. This would require changing the lexer and parser so that strings of arbitrary non-whitespace characters are considered to be tokens when in a position where a filename would be expected.

E.g. `perl-sql select count from test.db` should be allowed,
instead of requiring `perl-sql select count from '"test.db"'`
or `perl-sql 'select count from "test.db"'`

Issue: Perl-SQL uses Perl internal 'eval' to evaluate records. This allows expressions to be used, and allows nested data structures, but allows a data file to modify Perl-SQL internal variables.

TBD: **sandbox the data file record evaluation**. (I would rather do this than fully parse the file, because I really do thing that access to all of Perl in the data file is a good thing.)

Issue: Perl-SQL's use of eval means that **lambda functions** could be specified as field values. This is potentially a very powerful feature.

TBD: define syntax to use such lambda function field values.

TBD: currently such lambda functions will not print out. (Wait for Perl Data::Dumper to support lambda functions?)

TBD: **partial wildcards on field names**

```
SELECT runlength* FROM "experiment.db"
```

TBD: **more general field name expressions** – basically, making it possible to treat any tuple as a table (key, value) and have queries on that.

```
SELECT (SELECT t.(r.name) / bpred_lookups FROM TABLE(t.runlength*net) AS r)
   FROM "experiment.db" AS t
```

In case the SQL syntax is not obvious, here is a non-SQL version of the above:

```
for each tuple t in "experiment.db"
   convert the table emitted by the following to a tuple {{
          for each tuple r in (convert tuple t to table(key,value))
                 if r.name =~ /runlength.*net/
                        emit tuple r.name / bpred_lookups
   }}
```

Personally, I find the FOR EACH syntax easier to read, and I reject descriptions of it as procedural, whereas the SELECT syntax is putatively non-procedural. Each is an attempt to describe what we want. Optimizations can be applied to either.

Issue: the MNT and SNT formats allow complex data structures to be read in and printed out, but the printout format is primitive (a single line), and there are no operators.

TBD: **allow indented printout of complex structures within MNTs**
TBD: **provide operators to use complex {} and [] structures**
TBD: **allow an array of hashes [{}] to be treated as an embedded relation**
TBD: **allow any hash to be treated as a relation of (key, value) pairs.**

Issue: MNTs are SNTs are flexible, but it is very easy to accidentally create a new field via a typo.

TBD: **optionally limit fieldnames to an approved list**, perhaps by using the field-names comment to provide a list of regexps that fieldnames must match.

Issue: MNT and SNT fields are printed in alphabetic order. This may be undesirable, e.g. for BiBTeX records.

TBD: **optionally provide fieldname printing order for MNT and SNT.**

Issue: named tuple formats do not allow **BiBTeX like tuple syntax**, using '=' or ':' or even nothing instead of '=>', and using other brackets than {}.

The 'format' special comment can be set to 'multiline_named_tuples', 'singleline_named_tuples', or 'singleline_delimited_records'. TBD**: allow abbreviations like MNT, SNT, SDR, CSV, and TSV, to be set in format comment**, as on command line.

Issue: in MNT and SNT formats, fields do not exist.

TBD: provide the exists(table,field) predicate

7 Perl-SQL internals

7.1 File and Module Structure

Currently, Perl-SQL is only a command line program, suitable for use as a UNIX filter.

It will be desirable, at some time, to fit Perl-SQL under the standard Perl DBI (Data Base Interface). It may similarly be desirable to allow direct access to Perl-SQL functionality via a Perl module interface, in the same way that Sprite, and other database interfaces, did.

Perl-SQL contains the following internal modules. These are currently all prefixed Perl_SQL:: .. I realize that I have violated just about every Perl naming convention:

perl-sql

The actual Perl SQL program.

ag_perl_sql_parser.pm

a parser generated by byacc. returns a parser object

ag_perl_sql_lexer.pm

a handwritten lexer. returns a lexer object.

dumb_line_re_lexer.pm

a dumb lexer module that fulfills my needs TBD: I'd like to eliminate it, and use an "official" Perl lexer module, but found them awkward.

plain_text_database.pm

returns a "ptd" handle object with various methods.

multiline_named_tuples.pm

singleline_named_tuples.pm

singleline_delimited_records.pm

methods for the various types of databases implemented by plain_text_database.pm

nice_IO.pm

used by plain_text_database.pm, essentially creates infinitely ungettable filehandles. Returns a handle object.

null_handling.pm

misc.pm

helper modules - not really object oriented.

db_algorithms.pm

implements database algorithms, such as scan select and sorted merge equijoin not OO

eval_functions.pm

functions used to evaluate the parse tree generated by ag_perl_sql_parser.pm - glue code connecting it to db_algorithms.pm

7.2 Design Flaws

I am sure that Perl-SQL's code will cause many programmers to gag. Some of the most egregious flaws are:

- Poorly defined API for db_algorithms.pm based on file handles

  The algorithms in this module (scan, join) accept databases as file handle objects, rather than filenames. This was done to permit STDIN and STDOUT to be passed as arguments, but has resulted in much duplication of code. This API should be redone passing file names.

- No use of inheritance in file handle package nice_io.pm

- Poor partitioning in eval_functions.pm

  This module does not cleanly separate operations that are strictly related to the YACC parse trees from operations related to the definition of null-propagating arithmetic and operations relating to aggregates.

There are undoubtedly many other flaws.

8 Other database tools

Perl-SQL is most closely related to Sprite, by shishir@ora.com. Sprite is also SQL implemented in Perl. Sprite is a much less complete implementation, doesn't implement anything nearly as close to full SQL. Sprite does, however, have a proper Perl module interface. Perl-SQL, by the way, contains no code from Sprite.

Perl-SQL is similar to tools such as RDB and SHQL, but seems to have greater ambitions of flexibility, generality, and compatibility with standard SQL.

Perl-SQL is *not* just an interface to some underlying database package, neither an existing SQL database, nor to some underlying database library

I did not, initially, want to write my own database. I spent more than a month doing product research seeking an existing tool that met my needs for "Experimental Data Management and Visualization". I used various web search engines, posted queries to a wide variety of Internet newsgroups, obtained a free subscription to the magazine "Scientific Automation", contacted vendors of many products, and spent days installing and running demonstration software, both on my PC, and in software showrooms. I posted a summary of my search to Internet newsgroups such as comp.arch, comp.data.management, and so on. I will only briefly summarize my search here.

Originally I hoped to find a single tool to do 'Experimental Data Management and Visualization'. I.e. I need not only to be able to manage my data, but I also need to prepare useful graphs, charts, scatter-plots, and statistical analyses. Moreover, I want interactive visualization, the ability to quickly zoom in and out of my graphs, e.g. by "mouse based zoom", drawing boxes around an area of the plot and having it rescaled.

I was quickly disappointed. Most of the tools that do good graphics have primitive data management, and vice versa. Many of the graphing tools can only handle dense, vector or matrix oriented data sets.

The graphing and analysis tools I evaluated included Sigmaplot, SPSS, SPSS/Deltagraph, BMDP, SAS, SAS/JMP, Datadesk, MATlab, XMGR, S-plus, Axum, Statistica, SPSS/Sysstat, Microsoft Excel, dataplot, Scilab, Minitab, HiQ,

I ended up purchasing Origin, from Microcal, as a plot package, and SPSS and SYSSTAT as analysis packages. The latter may have been a waste of money, since I haven't opened them yet.

I hope to be able to write macros in my plot packages (Origin and GNUPLOT) that will call Perl-SQL database query scripts.

The database tools that I evaluated included DEVise, MySQL, MiniSQL, GNU SQL, PostgreSQL, Beagle, and Microsoft Access. None of these supported OUTER JOINs, or sparse and irregular databases. In all cases maintaining the database was a pain.

I seriously considered using SYBASE, which is maintained by my local system administrators. However, the local SYBASE installation available to all CS educational users is sharply constrained in space, and would not be large enough to manage my data. Working with the system administrators to have a custom private database of sufficient size is an administrative hassle I do not need. SYBASE does not support OUTER JOIN or irregular and sparse databases. The data stored in the SYBASE files is not survivable: if I go to a company that does not have SYBASE installed, or if I want to take the data home, it is a real pain. My experience has been that such data eventually gets lost because migrating it falls to the bottom of the TO-DO list, and then falls off. Licensing restrictions are annoying: the SYBASE server does not use my PCs, but uses a slow machine across the network. Overall, too much hassle to use a database that doesn't do what I want in the first place!

There are a number of potentially useful freeware database tools:

- Sprite: a Perl script that does simple scan SELECTs and UPDATEs on plain text files. Doesn't do JOINs or handle sparse and irregular databases. The author of Sprite suggested that I start writing what has turned into Perl-SQL.

- SHQL: a quite old set of tools that implements SQL as UNIX Bourne shell scripts. The sort of thing that I want to do, except the SQL features are lacking (the usual restrictions, no OUTER JOIN or support for irregular databases), and I am scared away from shell scripts by previous bad experiences. (At least SQL is not CSH.)

- RDB (ftp://uiarchive.uiuc.edu/pub/lang/perl/CPAN/modules/dbperl/scripts/rdb/): limited file formats, no support for irregular databases, no OUTER JOIN, yet probably the closest thing to what I want. Has some features that I would like to eventually include in Perl-SQL.

- The list of freely available database software at http://bunny.cs.uiuc.edu/sigmod/databaseSoftware was nice, but I saw nothing that looked useful.

9 Conclusion

Perl-SQL is a tool that allows much useful relational database functionality to be made available to a group of people who have not, before now, used relational databases: researchers analyzing experimental data such as that obtained from computer architecture simulation studies. At the moment, this community seems to use ad-hoc scripts written in tools such as Perl and AWK and Excel to manipulate their data. Perl-SQL will, I hope, reduce the need to write such scripts, and will make it possible for such scripts to be more generic.

Most of Perl-SQL is straightforward, design decisions such as using ordinary ASCII text files, implementing FULL OUTER JOINS, etc.

Perl-SQL does incorporate one significant innovation with regards to conventional relational databases: Perl-SQL supports sparse and irregular data, by allowing every record of the database to define its own schema.

I hope that Perl-SQL will be of use to others who have had the same problems managing experimental data that I have. I also hope that conventional database systems will incorporate some of the ideas encapsulated in Perl-SQL: ease of administration, sparse and irregular databases, self-schematization, support of conventional text files.