Data integration using scanners with SQL output - The Bioscanners project at sourceforge

Detlef Groth (1), Stefanie Hartmann (1), Martin Friemel (2), Natascha Hill (1), Stefan Müller (1), Albert J. Poustka (3), Georgia Panopoulou (3)

[1]University of Potsdam, Bioinformatics Group, c/o Max Planck Insitute of Molecular Plant Physiology, Am Mühlenberg 1, D-14476 Potsdam-Golm, Germany

[2]Max Planck Insitute of Molecular Plant Physiology, Bioinformatics Group, Am Mühlenberg 1, D-14476 Potsdam-Golm, Germany

[3]Max Planck Institute for Molecular Genetics, Ihnestr. 63/73, D-14195 Berlin, Germany

Summary

There is currently no standardized approach for parsing output that the numerous bioinformatical tools generate. Because the framework approach of the Bio-toolkits has some shortcomings, we searched for alternative approaches. To this end, we evaluated scanner generators for various programming languages with respect to their potential of creating standalone, small, and fast applications that can easily be delivered on any modern and many ancient operating systems. We developed sample applications that generate standard SQL database code and thereby greatly simplify the parsing work of data integration and data analysis. At the sourceforge project page the source code and some binaries for a selection of our applications are freely available at `http://bioscanners.sourceforge.net`.

1 Introduction

Biologists and bioinformaticians are frequently challenged with reading, analyzing, reformatting and storing of the output of various biological tools that utilize different formats. After data parsing, results from other analysis steps and the corresponding software tools often need to be integrated. Unfortunately, no standard or solution for this problem is currently available.

In principle, the following general approaches exist for parsing and integrating bioinformatics data: a) Standalone applications connected via standard Unix-pipes [14, 5], b) framework libraries including previously developed functions, classes, and tools connected with the aid of a programming language [9], c) web services where existing web applications are connected by standardized protocols. Because web services are not solving the data integrations problem for user data and are not suited for high throughput processing, we will not discuss them here.

We will briefly review the approaches using standalone applications and framework libraries, and we will describe our approach of using scanner generators, and how it solves the problems that arise when standalone applications and framework libraries are used. In computer terminology, a parser is an application which evaluates the grammar rules of a text or a computer programming language. We will therefore use the term “scanner” for our programs because the grammar of the data files will not be checked.

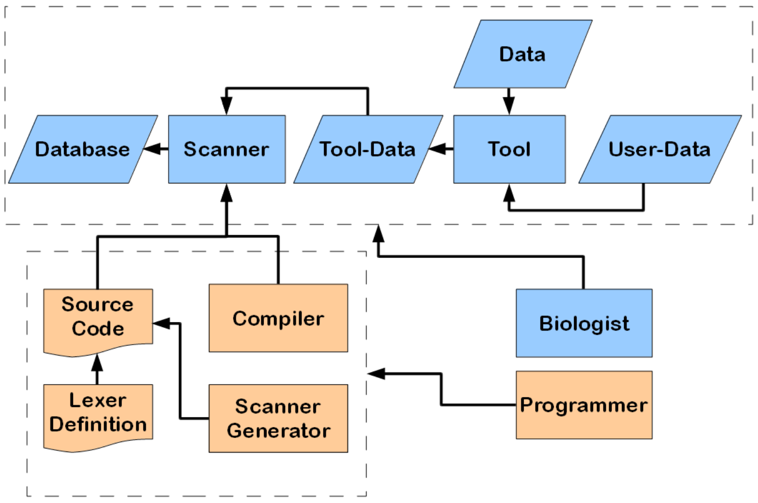


Figure 1: Data analysis and programming pipeline using a scanner. Working procedures performed by the programmer and the working steps by the biologist are outlined in different boxes.

1.1 Standalone applications

In accordance with the Unix philosophy, standalone applications are kept as small as possible and perform a single, well-defined task. Communication between applications is mainly realized by commandline pipes. But this solution is in many cases not flexible enough. It cannot be applied or easily modified when tasks become more specific, when the input data format changes, or when different types of experimental data needs to be integrated or connected. Without having a reliable storage mechanism for the data or a programming framework, a tedious and error-prone manual processing and inspection procedure is required.

1.2 Framework libraries

Programming frameworks like BioPerl [13], BioPython [1], or BioJava [11], commonly called Bio-Toolkits[9], solve the tedious task of integrating data from different data sources and tools by utilizing a language-specific programming interface. A programmer writes the code and connects different parts of the framework by calling functions and methods of different objects or packages. The output of one method can be passed to another method or object. However, this approach has several drawbacks: users are often not willing or able to install large software libraries to solve small problems. Based on specific requests by the biologist, repeated small adjustments by the programmer in the source code are required. This results in a constant dependency of the biologist on a programmer. Furthermore, the huge and complex programming interface takes a lot of time to get comfortable with, and it often makes the data analysis extremely slow.

1.3 Our approach: Programs developed using scanner generators

We have used different scanner generators to write our Bioscanner-applications (Table 1). Scanner generators are used for the lexical analysis of text, for instance by a compiler, and can produce also small standalone applications on their own.

Scanners and Compilers				
scanner	language	url	compiler	wc-sizes
Flex	C	http://flex.sourceforge.net	GNU gcc 4.1.2	16 / 489
Flexpp	C++	http://flex.sourceforge.net	GNU g++ 4.1.2	21 / 1002
RE2C	C	http://re2c.org	GNU gcc 4.1.2	7 / 7
TPLex	Pascal	http://www.musikwissenschaft.uni-mainz.de/ag/tply	FPC 2.0.4	109 / 109
JFlex	Java	http://jflex.de	SUN javac 1.6.0	7 / NA
GPLEX	C#	http://plas.fit.qut.edu.au/gplex	Mono gmcs 2.0.1	13 / 6525

Table 1: Overview of the used scanner generators and their targeted programming languages. Sizes of the word counter "wc" sample applications as dynamic / static applications (kb)

The advantage of using a scanner generator to create a scanner application is that the resulting code for the applications have fewer lines of code, they are therefore faster to write, easier to maintain, and also faster at execution speed than handwritten applications [2]. By using the scanner generator with its restricting syntax, the influence of the personal programming style on code quality and maintenance is also minimized. The steps for a programmer to create a working scanner are outlined in the lower part of Figure 1. The tasks for the biologist and the programmers are here strictly separated.

1.4 Our approach: Database output

The easy and fast scanning of data is as important for the user as the storage of the scanning result in a format that can be used to selectively extract information at any time. These results again are data with relations to other data from the same or other experiments, making it ideal for import and further analysis in a database. Unfortunately, the output of such scanners or parsers is only seldom directly imported into a relational database system.

Our scanners create SQL (Structured Query Language) code to generate data tables, insert data, and index the data table to allow fast data retrieval from the database. Using a SQL relational database to store the output of a data scanner application offers several advantages. SQL databases as the data integration platform ensure the best possible performance in data exploration, especially in a relational context, as well as in maintaining data integrity. As databases are extremely flexible in data abstraction and data structure changes, the user can easily modify the existing database schema, which is given by default, either by simple renaming columns or tables or by creating views which join different tables. This simplifies the data integration into an already existing analytical pipeline. Furthermore, complex query statements can be stored as views and reused later on. Sophisticated visual tools allowing easy and interactive data explorations are also available. Problems like: "Considering a minimal bit score of 200 for the BLAST results, which sequences have a hit against database A but not against database B" can be easily solved within a relational database but are difficult and tedious to answer with customized programming scripts.

Taking into account all the considerations discussed above, we developed our data scanners for various file formats. Our applications are easy to install, they are small, fast, and they emit standard SQL-code suitable to feed SQL compliant databases.

2 Materials

An overview of the used scanner generators and compilers is given in Table 1. We tested six scanner generators which can be used to produce executables at least for Windows, Linux, and Mac-OSX operating systems, and which support five different programming languages (C, C++, C#, Java, Pascal). For simple output formats, scanning scripts with tabular output were written using Bash-shell functions that are portable to Windows, Mac-OSX, and Unix operating systems.

For time- and memory tests, we used 32-bit computers with Intel processors running the operating systems Windows-XP Professional with SP 2 and Linux Fedora core 8, 64-bit computers with Intel processors with the operating systems Mac-OSX Darwin kernel 8.11, and Linux with CentOS 5.1. Additionally a Sparc computer with Solaris 9 and an Alpha computer with OSF-1 V4.0 were used. All time tests were done on the 32-bit Linux machine using Tcl-scripts for measuring time and memory parameters. The Linux machine had 2 GB memory, 2 Intel Xeon CPUs with 2.8 GHz with the Fedora core 8 operating systems on Linux kernel 2.6.26. Some timing tests were also performed on the 32-bit Windows machine with a single Intel Pentium 2 Ghz processor. Memory tests were made on the Linux 32-bit machine using the Unix tool ps. The timing and memory scripts are available from the sourceforge project page.

Testing of the database import was done for SQLite (`http://www.sqlite.org`) on Windows-XP, Linux-32bit and Linux-64bit machines and for PostgreSQL (`http://www.postgresql.org`) and MySQL (`http://www.mysql.com`) on the Windows-XP machine.

3 Results

3.1 Scanner generators and programming languages

We evaluated different scanner generators for their suitability to implement efficient and highly platform portable data scanners. Because it was recently shown that scripting languages are much slower than semi- or fully-compiled languages [8], we used only non-scripting languages in our final analysis. This decision was further supported by the observation that for two of the most used scripting languages in bioinformatics, Perl and Python, no scanner package is available which accepts the format similar to the simple format of the most widely used scanner generator Flex.

3.2 File sizes

As can be seen from the results in Table 1, the file sizes of the compiled applications varies considerably. We included in this table the file sizes for a sample word counting application ("wc") both as dynamic and static builds. For dynamic builds either the required libraries or the runtime must be available for the platform, whereas for static builds the appropriate libraries or runtime environments do not have to be installed on the target platform. The RE2C/C and Pascal executables do not depend on external libraries or runtimes, therefore there are no dynamic builds possible. RE2C executables were the smallest. In contrast to static applications programmed in C and Pascal the static binaries for C++ and C# were very large. If the Mono-runtime

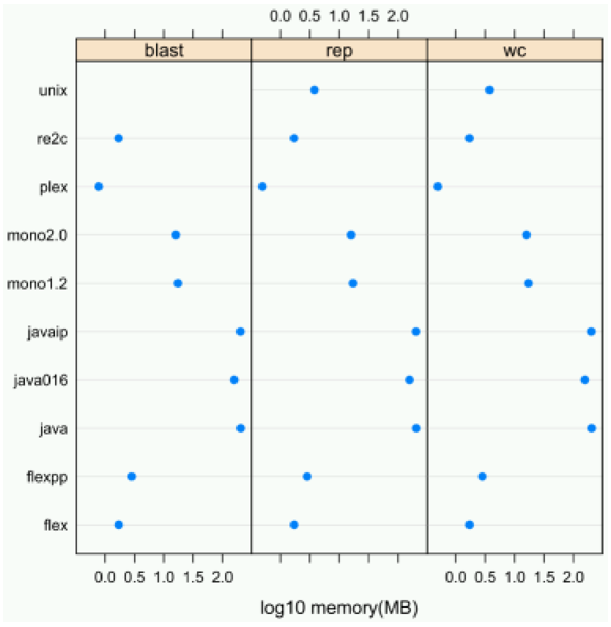


Figure 2: Memory consumption. The names of the scanners that were evaluated are given on the y-axis. Unix tools evaluated were "wc" and "sed", re2c: RE2C C programs, plex: Pascal programs, mono1.2 and mono2.0: programs executed with the 1.2 or 2.0 Mono interpreter, java: java programs, javaip: java programs running in interpreted mode using the flag "java -Xint", java016: java programs running with reduced startup memory using the flag "java -Xmx16m", flex and flexpp: the C and C++ applications generated with the Flex scanner generator. Each scanner was assessed for its memory requirement for three different tasks listed in the first row: a simple blast scanner ("blast"), a replacing application ("rep"), and a word counter implementation ("wc"). The results were largely the same for all the three tasks evaluated.

(`http://www.mono-project.com`) was linked in, the simple word counter application for C# was around 6 MB in size, making it not a valuable option for delivering applications to platforms where the Mono-framework is not installed.

3.3 Performance

We compared the performance and memory consumption of our applications for three different and frequently-used data analysis tasks. First, we implemented a word counter program, similar to the known "wc" terminal program. Second we made a program that replaced all lower case characters to an "x" to mimic a typical format conversion application. Finally, we wrote a simple BLAST output file scanner that extracts all hits with scores and E-values as well as all query identifiers with their exact position in the BLAST report-file. For the first two tests the performance of standard tools, "wc" and "sed", was used for reference.

We found that the memory requirements for the semi-compiled languages were generally much higher in comparison to the compiled languages (Fig. 2). This was especially true for Java-applications based on the JFlex scanner generator, where even with optimized runtime configurations more than 150 MB of memory were consumed. The Mono-runtime required around 20 MB of memory in both tested versions (1.2 and 2.0) whereas the memory footprint of the compiled languages was always below 3 MB, for the Unix tools always below 4 MB. The lowest footprint had the Pascal application (plex) with less than 1 MB of memory footprint. We did not find significant memory or performance differences between dynamic and static builds,

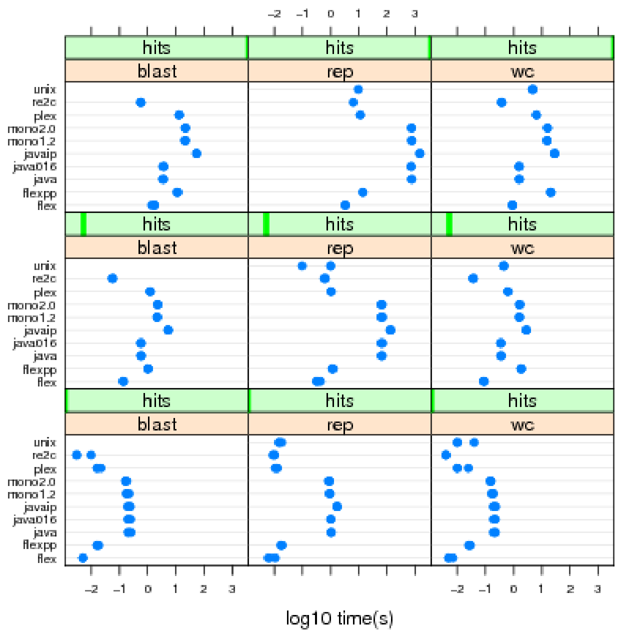


Figure 3: Time requirements for different scanners. Scanner generators were tested for three tasks. The same abbreviations as in Figure 2 are used for programs (y-axis) and tasks (boxes). Bottom row: the results for the hits of single qouery BLAST file (ca 40 kB). Middle row: results for files with hits for 100 queries (ca 4 MB). Top row: times for files with 1000 queries (ca 40 MB). All times were determined three times independently, and all individual results are plotted. Because the results did not vary much, generally only a single point for the three replicates is visible.

and the memory requirements for our three sample tasks mentioned above were quite similar.

Because the processing time of an application is often more important than the memory requirements and the application size, we evaluated the execution speed of the sample applications in more detail (Fig. 3). In all examples tested, RE2C-scanners were the fastest, followed by Flex scanners. For small files, the runtime of the program is mostly determined by the applications startup time whereas with larger files the input and output performance of the programming languages are more important. For the small, single hit file, the startup costs are much higher for the semi-compiled languages in comparison to the compiled languages. For larger files this changed considerably. Especially the performance of the JFlex based scanners improved. When the number of output operations was kept small in the “wc” and “blast” examples, the Java scanners with “just in time compilation” (JIT) enabled were only around 2 times slower in comparison to Flex-scanners and around 5 times slower in comparison to RE2C scanners. Java scanners running in interpreted mode and C++ and Mono-scanners were the slowest.

We also performed preliminary tests on a Windows-XP machine using the .NET framwork. The results did not differ much from the results on the Linux machine.

3.4 Code amount

In addition to the final application performance, the programmer’s efficiency in code writing is of great importance. Recent research suggests that the amount of code to write does not differ significantly between semi-compiled and compiled languages [12, 8], and these differences are even smaller when scanner generators are used. We therefore did not perform a detailed comparison on the amount of code in our current study. As an example we just note that the RE2C- and the JFlex-based “wc” implementation each required around 10 application-specific lines of code.

Based on the results of our scanner generator evaluation, we chose the RE2C scanner generator for implementing our Bioscanner applications. In brief, we found that the applications generated using the RE2C scanner generator are easy to compile and install, they have fast processing speed, small binary sizes, and low memory consumption.

3.5 Databases

Instead of inventing a new output format, we chose one of the most widely accepted standards: SQL. Our Bioscanner applications generate database code that can be imported into SQL databases. We successfully tested SQLite, PostgreSQL, and MySQL as the most widely used, freely available database engines. The necessary indices to perform effective database join operations are also created after the scanning procedure. As SQLite and MySQL allow queries across different databases, the data from different projects can be merged or stay separated, based on the user requirements. The user can, for instance, join a BLAST output database with an existing UniProt database and a pathway database, without being forced to put all the data in the BLAST result database beforehand.

3.6 Installation and Usage

The C-code and binaries for various computer platforms for our applications can be downloaded at the sourceforge project page. The applications can be compiled with any recent C compiler on any modern computer platform. The applications were written to accept command-line parameters for the input files and a prefix-argument for the table name. This optional prefix argument allows the user to keep data from different program runs in the same database. If invoked from the command-line without arguments the applications display a help message about their usage. In case of creating or updating a SQLite database the database code can be sent directly into the “sqlite3” command-line application (Fig. 4). For MySQL and PostgreSQL, the database can be loaded into a running command-line client - “mysql” or “psql” respectively. Details of installation and usage can be taken from the documentation available from the website or together with the downloaded applications.

3.7 Comparison with other parsers

We compared the performance of our RE2C based BLASTScanner with BLAST parsers provided by the Bio*-toolkits and with the Zerg-parser [10]. As existing SQL-bindings of the Bio*-toolkits are not yet accessible in a consistent manner to the end user, the usage of those toolkits as well as Zerg-parser require coding in Perl, Python, Java, or C. In contrast, the BLASTScanner simply runs from the command-line. Furthermore, while the processing time of the BLASTScanner was comparable to the Zerg-parser, the BLASTScanner was orders of magnitudes faster in comparison to the Bio*-toolkit parser. The BLASTSCanner application

```
01  $ BLASTScanner --infile osat_aa.fasta-scer_aa.fasta.blastp
             --prefix scer | sqlite3 mysample.sqlite3
02  $ BLASTScanner --infile osat_aa.fasta-cele_aa.fasta.blastp
             --prefix cele | sqlite3 mysample.sqlite3
03  $ sqlite3 mysample.sqlite3
04  sqlite> select distinct query_id from cele_hits
       where score > 100
  except select distinct query_id   from scer_hits
       where score > 100 limit 5;
05  cele10017
06  cele10035
07  cele10036
08  cele10037
09  cele1005
10  sqlite> select count(*) from (select distinct query_id from
    cele_hits where score > 100  except select distinct query_id
    from scer_hits where score > 100);
11  4237
```

Figure 4: Sequences of rice (”osat”) were compared to sequences of yeast (”scer”) and of a nematode (”cele”). The BLAST result files from these searches were first imported into a sqlite3 database using the first two lines shown. The database (named ”mysample.sqlite3”) was then opened (line 3), and two queries were executed: In line 4, all hits to rice for which a hit in the nematode but not in the yeast were found, are identified. The results are printed to the screen but were here limited to the first five lines, shown in lines 5 to 9. In line 10, the total number of such hits are identified, which is shown to be 4237 unique nematode sequences, as shown in line 11.

scans, for instance, within 2 seconds a 150 MB BLAST file on a modern 2.8 GHz machine and translates it into database code.

3.8 TabCsvScanner and BioBash

During our daily work with different file formats we observed often cases where tools already produced quite simple output formats that didn’t have to be converted. The GFF-file format (`http://www.sanger.ac.uk/resources/software/gff`), for example is a tabulated format. In other cases formats could be translated with a little bit of shell code into a tabulated format quite easily. We did this, for instance, for the output formats of the cellular localization predictors TargetP, SignalP [6] and chloroP [7]. These tools provide an output format that contains predictions of signal peptides, scores for different localizations, and predictions for possible cleavage sites. After translating these ones into tabulated format, we created the generic TabCsvScanner which transforms tabular input with the aid of column definitions “on the fly” into SQL code. This scanner can be used in cases where the input format is either already tabulated or can be translated with a few simple shell commands into such an format. All our simple scanning scripts, including the a more sophisticated Gff2IntronScanner-function are stored in a file called BioBash.sh. Having these simple scripts in one separate file makes it easy to add additional functions and to stay organized with all the different tools.

3.9 Practical evaluation: case study

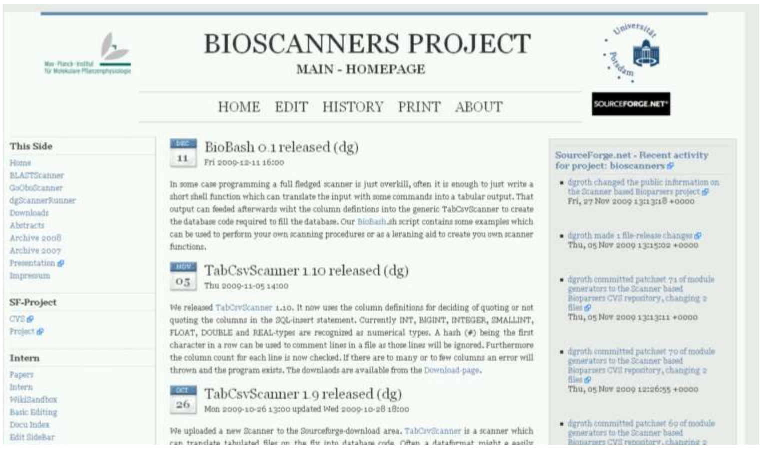
We recently used our BLASTM8Scanner application for a large-scale identification of candidate regions for an exon-primed intron- spanning PCR analysis in the wet-lab. Pre-selected introns and their flanking exons from the genome of “Canis lupus familiaris” (dog) were screened against the “Felis silvestris cattus” (cat) scaffolds using a local BLAST installation. From this search, introns that diverged between the cat and dog genome, but which were flanked by conserved exons, were to be identified. Intron-spanning PCR-primers could then be designed within these conserved exons. The aim was therefore to identify all dog introns for which exactly two BLAST hits from the same cat scaffold, at least 50 bp apart, were found. Multiple hits to the same cat scaffold were allowed if a shorter local alignment was completely nested within a longer local alignment. Because of possible mis-priming, all other multiple hits were disallowed.

This time-consuming task was initially manually performed for a small number of intron sequences. For a much more efficient solution of this problem, the BLASTM8canner application was subsequently used. First, all 81 BLAST output files, in M8-table form, were imported into a SQLite database with a single terminal command, taking less than 10 seconds to complete. In order to identify hits with the required characteristics, SQL-statements were then used to create four consecutive views in the database, implementing in a stepwise manner the process of selecting the desired BLAST hits. Specifically, from 20,275 pre-selected dog introns, 2,283 had exactly two unique hits to a single cat scaffold as described above. Additional 84 dog introns had nested hits to a single cat introns. The corresponding sequences for the 2,367 candidate dog introns can now be aligned for automated primer-design.

Although most biologists are not expected to generate the SQL statements for these analyses, they can easily comprehend the general approach and the purpose of the stored SQL statements and database views. Moreover, they can graphically access the database and the stored views using a graphical database tool on their own computer, for instance using the SQLite plug-in for the Firefox browser. For the programmer, the advantage lies in not having to generate customized programming scripts, where the intermediate results are not transparent for or accessible to the biologist.

4 Discussion

This work addresses a well-known and long-standing problem: the extraction of information from the output of bioinformatics tools. The number of output files of such tools is enormous, and the necessity for developing universal and easy-to-use data scanners is obvious. However, an agreement about a standard procedure to create, update, and maintain such parsers is still missing. While it has been known for a long time that scanner generators can be used to analyze textual data in an efficient and reproducible way, they are surprisingly rarely used in bioinformatics. Scanner generator based applications are easier to program while they are flexible and extremely fast processing tools. But ”efficiency” refers not only to the application speed, but also programmer’s working speed. As the amount of code and the number of required functions is greatly reduced, the programmer’s productivity and the ease of long-term maintenance is improved. The low complexity of the scanner code base allows even less experienced programmers to fix and improve existing code of other developers. The restrictive framework of


BIOSCANNERS PROJECT

MAIN - HOMEPAGE

HOME EDIT HISTORY PRINT ABOUT

This Side

Home
BLASTScanner
GoOboScanner
dgScannerRunner
Downloads
Abstracts
Archive 2008
Archive 2007
Presentation
Impressum

SF-Project

CVS
Project

Intern

Papers
Intern
WikiSandbox
Basic Editing
Docu Index
Edit SideBar

DEC 11 BioBash 0.1 released (dg)
Fri 2009-12-11 16:00

In some case programming a full fledged scanner is just overkill, often it is enough to just write a short shell function which can translate the input with some commands into a tabular output. That output can feeded afterwards wiht the column defintions into the generic TabCsvScanner to create the database code required to fill the database. Our BioBash.sh script contains some examples which can be used to perform your own scanning procedures or as a leraning aid to create you own scanner functions.

NOV 05 TabCsvScanner 1.10 released (dg)
Thu 2009-11-05 14:00

We released TabCsvScanner 1.10. It now uses the column definitions for deciding of quoting or not quoting the columns in the SQL-insert statement. Currently INT, BIGINT, INTEGER, SMALLINT, FLOAT, DOUBLE and REAL-types are recognized as numerical types. A hash (#) being the first character in a row can be used to comment lines in a file as those lines will be ignored. Furthermore the column count for each line is now checked. If there are to many or to few columns an error will thrown and the program exists. The downlaods are available from the Download-page.

OCT 26 TabCsvScanner 1.9 released (dg)
Mon 2009-10-26 13:00 updated Wed 2009-10-28 18:00

We uploaded a new Scanner to the Sourceforge-download area. TabCsvScanner is a scanner which

SourceForge.net - Recent activity for project: bioscanners

- dgroth changed the public information on the Scanner based Bioparsers project
  Fri, 27 Nov 2009 13:13:18 +0000
- dgroth made 1 file-release changes
  Thu, 05 Nov 2009 13:15:02 +0000
- dgroth committed patchset 71 of module generators to the Scanner based Bioparsers CVS repository, changing 2 files
  Thu, 05 Nov 2009 13:13:11 +0000
- dgroth committed patchset 70 of module generators to the Scanner based Bioparsers CVS repository, changing 2 files
  Thu, 05 Nov 2009 12:26:55 +0000
- dgroth committed patchset 69 of module generators to the Scanner based

Figure 5: The bioscanners project page at sourcefoge

the scanner generator limits possible influences of personal programming styles. As an example the first author of this paper did not have any previous experience with the C# programming language, but it was possible for him to install a current versions of the Mono-runtime, and to program all three sample applications within just one working day.

4.1 Bioscanner advantages

As textual output files are one of the most frequently generated data files in bioinformatics, approaches which are optimized for scanning and interpreting textual data offer a valuable choice for an effective implementation for such scanning tools. Because the programming with the scanner generators seems to be of similar complexity in all programming languages tested, we chose the scanner according only to its performance and the simplicity of installing the application on the users side. Based on these criteria we selected the REC2 scanner generator. The C code generated with RE2C should compile with any modern C compiler on any current computer platform. Such a compiler should be pre-installed on many computer platforms. To simplify the installation even more, we provide for some applications also binaries. To give an example, the BLASTScanner for 6 platforms together with the C source code and some documentation is packed as a Zip-archive (all less than 100 kB large in size).

The small size and the single file approach allows an easy handling of ongoing changes in the format of bioinformatics tools. A possible approach would be, for instance, a rename of the binary to reflect the version of the tool it actually supports. If renamed, for example, to “BLASTScanner-2.2.17”, another, newer binary can be downloaded to support future format changes of the BLAST output format without the need of supporting all possible formats in one single application. This approach is far more difficult to achieve with large programming frameworks like BioPerl.

As our data scanner applications were faster than the import procedure of the database application, there remains no additional speed optimzation work to be done. For users not comfortable with command-line applications we have created a small cross platform GUI tool, which simplifies the database import procedure. This application can be downloaded from the project page as well.

For simple scanning tasks the programming and setup of a standalone application would be to much effort, we implemented such tasks using simple Bash-functions containing some shell commands which can be used in a cross platform manner on any modern operating system. The shell functions just create a tabulated output format which is then passed into the TabCsvScanner to easily migrate the data into the database. Using Bash-functions allows also having all function definitions in one file together which aids greatly in updating and maintainance.

4.2 Database advantages

The scanning step is just a task to extract information from the data and to make relationships to other experiments later on. With regard to that target it was very important for us to achieve this by utilizing technologies which had proven to be of high performance and comfortable even for the less experienced user. The adapted standard in this area are relational databases supporting a standardized query language, i.e. SQL. As far as we know, scanners and standard conform SQL output have not been brought together in bioinformatics before. Either efficient scanners were programmed with non-standard output [10, 4], or less effective parsers have been connected to just one relational database system [3].

From our experience it is easier to teach biologists SQL rather than a programming language, especially when a graphical interface with direct access to the data can be used for the SQL statements. Furthermore, the relational questions that are relevant for biological research are best answered in the database environments that have been optimized for many years by a large number of professional developers. As it is easy to export data or the result of such complex queries, databases are a perfect choice to support other tools, such as genome-viewers, by exporting the relevant data in a format required by those tools. Not focusing on a certain SQL dialect, we ensured that the SQL output can be imported into three of the main free database systems, MySQL, PostgreSQL, and SQLite. Other systems are likely to work as well.

Although new alternative approaches to store and query large scale data exist, like XML/X-Query, they currently are not used widely and did not offer the same convenience, speed and data abstraction possibilities as the SQL approach. However, support for such efforts could be easily added to our tools using XML as the output format. For some tools like BLAST, this might be not required as they have a XML output option. For others this might be fast choice to transform their data into XML.

5 Conclusions

Our scanners are currently the only available applications to translate the output from bioinformatic tools with a single simple invocation into standard SQL code suitable for direct import into a SQL database. The main advantages of using our Bioscanners are their ease in installation and usage, their speed, and their standardized output format. To support widely used scripting languages we also developed scanner generators for Ruby, Perl, Python and Tcl. Scanners written in those languages are highly suitable for model implementations which can be later translated into RE2C code if required. The source code for our application is freely available at the sourceforge project page (http://sf.net/projects/bioscanners) under an open source BSD-license.

Acknowledgements

The authors thank Marco Ende and Peter Krüger for technical support.

References

[1] S. Bassi. A primer on python for life science researchers. *PLoS Comput. Biol.*, 3:e199, Nov 2007.

[2] P. Bumbulis and D. D. Cowan. Re2c: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, March–December 1993.

[3] M. Catanho, D. Mascarenhas, W. Degrave, and A. de Miranda. BioParser: a tool for processing of sequence similarity analysis reports. *Appl. Bioinformatics*, 5:49–53, 2006.

[4] M. Dubnick. Btab–a Blast output parser. *Comput. Appl. Biosci.*, 8:601–602, Dec 1992.

[5] A. M. Durham, A. Y. Kashiwabara, F. T. G. Matsunaga, P. H. Ahagon, F. Rainone, L. Varuzza, and A. Gruber. Egene: a configurable pipeline generation system for automated sequence analysis. *Bioinformatics*, 21(12):2812–2813, 2005.

[6] O. Emanuelsson, S. Brunak, G. von Heijne, and H. Nielsen. Locating proteins in the cell using TargetP, SignalP and related tools. *Nature Protocols*, 2:953–971, 2007.

[7] O. Emanuelsson, H. Nielsen, and G. von Heijne. ChloroP, a neural network-based method for predicting chloroplast transit peptides and their cleavage sites. *Protein Science*, 8:978–984, May 1999.

[8] M. Fourment and M. Gillings. A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9:82, 2008.

[9] H. Mangalam. The Bio* toolkits–a brief overview. *Brief. Bioinformatics*, 3:296–302, Sep 2002.

[10] A. Paquola, A. Machado, E. Reis, A. Da Silva, and S. Verjovski-Almeida. Zerg: a very fast BLAST parser library. *Bioinformatics*, 19:1035–1036, May 2003.

[11] M. Pocock, T. Down, and T. Hubbard. Biojava: open source components for bioinformatics. *SIGBIO Newsl.*, 20(2):10–12, 2000.

[12] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany, Mar. 2000. ftp.ira.uka.de.

[13] J. Stajich. An Introduction to BioPerl. *Methods Mol. Biol.*, 406:535–548, 2007.

[14] L. D. Stein. How perl saved the human genome project. *The Perl Journal*, 1:1–1, 1997.