cql − A Flat File Database Query Language

Glenn Fowler
gsf@research.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

cql is a UNIX® system tool that applies C style query expressions to flat file databases. In some respects it is yet another addition to the toolbox of programmable file filters: *grep* [Hume88], *sh* [Bour78][BK89], *awk* [AKW88], and *perl* [Wall]. However, by restricting its problem domain, *cql* takes advantage of optimizations not available to these more general purpose tools.

This paper describes the *cql* data description and query language, query optimizations, and provides comparisons with other tools.

1 Introduction

Flat file databases are common in UNIX system environments. They consist of newline terminated records with a single character that delimits fields within each record. Well known examples are **/etc/passwd** and **/etc/group**, and more recently the *sablime* [CF88] MR databases and *cia* [Chen89] abstraction databases.

There are two basic flat file database operations:

update − delete, add or modify records

query − scan for records based on field selection function

For the most part UNIX system tools make a clear distinction between these operations. Update is usually done by special purpose tools to avoid problems that arise from concurrency. Some of these tools are admittedly low-tech: *vipw* write locks the **/etc/passwd** file and runs the *vi* editor on it; any other user running *vipw* concurrently will be locked out. On the other hand query tools usually assume that the input files are readonly or that they at least will not change during query access. *cql* falls into this category: it is strictly for queries and supports no update operations. Despite this restriction *cql* adequately fills the gap between *awk* and full featured database management systems.

In the simplest case a flat file database query is a pattern match that is applied to one or more fields in each record. The output is normally a list of all matched records. *grep*, *sh*, *awk*, and *perl* are well suited for such queries on small databases. These commands scan the database from the top, one record at a time, and apply the match expression to each record. Unfortunately, as the number of records and queries increases, the repeated linear scans required by these tools soon become an intolerable bottleneck. The bottleneck can be diminished by examining the queries to limit the number of records that must be scanned, but this requires some modifications, either to the database or to the scanning tools.

Some applications, such as *sablime*, ease the bottleneck by partitioning the database into several flat files based on one or more of the record fields. This speeds up queries that key on the partitioned fields, but hinders queries that must span the partition. Besides complicating the application query implementation, partitioning also imposes complexity on database updates and backup.

The *perl* solution (actually, *one* of the *perl* solutions − *perl* is the UNIX system swiss army knife) is to base the queries on *dbm* [BSD86] hashed files rather than flat files. Linear scans are then avoided by accessing the *dbm* files as associative arrays. A problem with this is that a *dbm* file contains the hashed field name and record data for each database record, so its file size is always larger than the original flat file. This method also generates a separate *dbm* file for each hashed field, making it unacceptable for use with large databases.

Other applications, such as *cia*, preprocess the database by generating B-tree or hash index files [Park91] for quick random access. Specialized scanning tools are then used to process the queries. The advantage here is that no database changes are required to speed up the queries. In addition, hash index files only store pointers into the database, so their size is usually smaller than the original database. The speedup, though, is not without cost. Some of the tools may be so specialized as to work for only a small class of possible queries; new query classes may require new tools.

Along with sufficient access speed, another challenge is to provide reasonable syntax and semantics for query expressions. For maximal transparency and portability the database fields should be accessed by name rather than number or position. Otherwise queries would become outdated as the database changes.

cql addresses these issues by providing a fast, interpreted symbolic interface at the user level, with automatic record hash indexing and query optimization at the implementation level. Query expressions are modeled on **C**, including a **struct** construct for defining database record schemas.

2 Background

As opposed to the UNIX system database tools like *unity* [Felt82], *cql* traces its roots to the C language and the *grep* and *awk* tools. As such *cql* is limited to readonly database access.

An example will clarify the differences between the various tools. The example database is **/etc/passwd** with the record schema:

```
name:passwd:uid:gid:info:home:shell
```

where `:` is the field delimiter, `uid` and `gid` are numeric fields, and the remaining fields are strings. The example query selects all records with `uid` less than 10 and no `passwd`.

Example solutions may not be optimal for each tool, but they are a fair representation of what can be derived from the manuals and documentation. The author has a few years experience with *grep* and *sh*, some exposure to *awk*, but had to resort to a netnews request for *perl*.

2.1 grep

```
grep '^[^:]*::[0-9]:' /etc/passwd
```

grep associates records with lines and has no implicit field support, so the select expression must explicitly list all fields. As it turns out the expression `uid<10` can be matched by a regular expression; more complicated expressions would require extra tool plumbing, possibly using the *cut* and *expr* commands. *grep* differs from the other tools in that a single regular expression pattern describes both the schema and query. This works fine at the implementation level but is cumbersome as a general purpose user interface.

2.2 awk

```
awk '
    BEGIN { FS = ":" }
    { if ($3 < 10 && $2 == "") print }
' /etc/passwd
```

Lines are the default *awk* record and `FS` specifies the field separator character. Numeric expressions are as in C and string comparison may also use the `==` and `!=` operators. Unfortunately the fields are named by number (starting at 1). If the database format changes then all references to `$`*number* must be changed accordingly. An advantage over *grep* is that fields are accessed as separate entities rather than being a part of the matching pattern.

2.3 shell

```
ifs=$IFS
IFS=:
while read name passwd uid gid info home shell junk
do    if   (( $uid < 10 )) && [[ $passwd == "" ]]
      then IFS=$ifs
           print "$name:$passwd:$uid:$gid:$info:$home:$shell"
           IFS=:
      fi
done < /etc/passwd
```

The shell (*ksh*[BK89]) version uses the field splitting effects of `IFS` and `read` to blast the input records. A nice side effect is that `read` also names the fields. If the database changes then only the field name arguments to `read` must change. Notice, however, that the shell has different syntax for numeric and string comparisons. Also, older shells [Bour78] have no built-in expressions and would require a separate program like *expr* to do the record selection.

2.4 perl

```
perl -e ’
    open (PASSWD, "< /etc/passwd") || die "cannot open /etc/passwd: $!";
    while (<PASSWD>) {
        ($name, $passwd, $uid, $gid, $info, $home, $shell) = split(":");
        if ($uid < 10 && $passwd eq "") {
            print "$name:$passwd:$uid:$info:$home:$shell";
        }
    }
’
```

The *perl* example [Chri92] is similar to *shell*, except that *shell* combines the record read and field split operations into a single `read` operation. As with *shell* string equality requires special syntax and `$` must prefix expression identifiers.

2.5 cql

```
cql -d "
    passwd {
        char*    name;
        char*    passwd;
        int      uid, gid;
        char*    info;
        char*    home, shell;
    }
    passwd.delimiter = ’:’;
" -e "uid < 10 && passwd == ’’" /etc/passwd
```

cql queries are split into two parts. The *declaration* section (`−d`) describes the record schema and the *expression* section (`−e`) provides the matching query. Using *cql* for this query is overkill, but it provides a basis for the more complex examples that follow.

2.6 Performance

Figure 1 shows the timing in user+sys seconds for the above examples, ordered from best to worst. The times were averaged over 5 runs on a lightly loaded 20 mip workstation with 2 cpus on an input file consisting of 19,847 records (1,525,549 bytes) in a local file system. *cat* is included as a lower bound.

`cat`	0.31
`grep`	1.77
`cql`	3.29
`awkcc`	3.37
`awk`	7.73
`perl`	9.09
`ksh`	19.98

Figure 1. Example timings

Although the compiled *awkcc* example runs more than twice as fast as the *awk* script it suffers by having a fixed select expression. Any change in the expression would force recompilation of the *awk* script to make a new executable. The timings also show that performance for the example query seems to be inversely proportional to tool functionality.

3 Optimization

Queries that check fields for equality are candidates for optimization. For example, most **/etc/passwd** queries are lookups for a particular `name`, `uid` or `gid`. As mentioned before, *perl* supports an associative array interface to *dbm* hash files, but converting to use this would require more than four times the file space of **/etc/passwd** itself and the query syntax would need to change to use the array notation. *cql* offers an alternative that only changes the schema declaration:

```
passwd {
    register char*  name;
    char*           passwd;
    register int    uid, gid;
    char*           info;
    char*           home, shell;
}
```

As with C the **register** keyword is a hint that marks variables that may be frequently accessed. For *cql* **register** marks fields that may be frequently checked for equality. *cql* generates a hash index file for each **register** field during the first database query. Subsequent queries use the index files to prune the scan to only those records with the same hash value as the **register** fields in the query expression. The index files are connected to a particular database; if the database file changes then the index files are regenerated by doing a full database scan. Because of index file generation the first query on schemas with **register** fields is always slower than subsequent queries.

The hash index file algorithm is due to David Korn and has been implemented as a library (*hix*) by the author. A *hix* file stores only hash codes and database file offsets, and its size ranges from 10% to 50% of the original database. The **/etc/passwd** example above has one record with the `name` **bozo**. The timings for the query `name=="bozo"` are listed in Figure 2.

no register fields	2.95
first register query	6.52
subsequent register queries	0.54
grep	1.64
awk	7.13
perl	7.56

Figure 2. Register query timings

The *hix* file generation slowed the first query by over 2 times but the subsequent queries were about 10 times faster. Even with *hix* file generation *cql* is still slightly faster than *awk* and *perl*. For the example **/etc/passwd** file size of 1,525,644 bytes the 3 *hix* files were a total of 788,952 bytes, or approximately 50% of the original database size.

4 Sub-schemas

Fields often contain data that can be viewed as another database record. *cql* supports this by allowing schema fields within schemas. The sub-schema fields are then accessed using the familiar C `'.'` notation. Our local **/etc/passwd** file formats the `info` field as:

```
info {
    char*  name, address, office, home;
}
info.delimiter = ",";
```

where the `info` sub-schema delimiter is `','`. An important difference with C declaration syntax is that the *cql* `char*` is a basic type. This means that all of the fields in this example have type `char*`, whereas in C only the first field would be `char*`.

Adding a second schema declaration introduces an ambiguity as to which schema applies to the main database file. By default the main schema is first schema from the top. `schema=`*schema-name*`;` can be used to override the default. The complete declaration now becomes:

```
passwd {
    register char*  name;
    char*           passwd;
    register int    uid, gid;
    info            info;
    char*           home, shell;
}
info {
    char*  name, address, office, home;
}
passwd.delimiter = ":";
info.delimiter = ",";
schema = passwd;
```

and the following queries are possible:

```
info.name=="Bozo T. Clown"
info.address=="* MH *"
```

where the second query illustrates *ksh* pattern matching on the address field.

Fields that refer to sub-schema data in different files are also possible. In this case the sub-schema field data is actually a key that corresponds to a field (usually the first) in the sub-schema data file. *cia* uses this format for its **reference** and **symbol** schemas. Sub-schema pointers are also used in *shadow password* implementations that split the **/etc/passwd** file into **/etc/passwd** that contains public information (no encrypted passwords) and **/etc/shadow** that contains privileged information (the encrypted passwords). An example **/etc/passwd** record might look like:

```
bozo:*shadow*:123:123:Bozo T. Clown, Big Top, 123-456::
```

with a corresponding **/etc/shadow** record:

```
bozo:abcdef.FEDCBA:Aug 11, 1993
```

In this case the `name` field doubles as the shadow key and the main schema `passwd` field is ignored. The *cql* declaration for these shadow passwords is:

```
passwd {
    shadow*         name;
    char*           passwd;
    register int    uid, gid;
    info            info;
    char*           home, shell;
}
info {
    char*  name, address, office, home;
}
shadow {
    char*           name;
    char*           passwd;
    date_t          expire;
}
delimiter = ":";
info.delimiter = ",";
schema = passwd;
passwd.input = "/etc/passwd";
shadow.input = "/etc/shadow";
```

C pointer notation is used to declare the `shadow` sub-schema reference and the predefined `date_t` type (described below) is used for the shadow password expiration field. The sub-schema reference also requires an input file that can be assigned by a *schema*`="`*pathname*`";` statement. A shadow equivalent of the original example query is:

```
uid<10 && name.passwd==""
```

Notice that `'.'` is also used to dereference sub-schema pointers (as opposed to `'->'` in C). Additionally, **register** is inferred for all sub-schema pointers. This allows *cql* to optimize queries by transforming equality expressions on sub-schema fields into hash index offset expressions. The **extern** keyword denotes the sub-schema key field in the sub-schema declaration; it may be omitted if the key field is the first sub-schema field.

Sub-schema data can also be specified in the declaration section:

```
info {
    char*    name;
    map*     type;
}
map {
    char*    name;
    char*    value;
}
info.type.input = { /* each "string" is a record */
    ";_ERROR_",   "g;global",   "t;typedef",
    "e;extern",   "s;static",   "l;libsym"
};
```

This is convenient for expanding database encodings in the user interface. For example, `type.value=="extern"` is more descriptive than `type=="e"`.

5 Language Description

The declaration language has already been introduced in the previous examples. Schemas are declared using the C **struct** style:

> *schema-name*
> `{`
>   *type-specifier*   *field-name* `[ ,` *field-name* `... ] ;`
>   `...`
> `}`

Schema, type and field names must match the regular expression `[a-zA-Z_][a-zA-Z_0-9]*`. The C reserved words `break, case, continue, default, else, for, if, return, switch, while,` are also reserved in *cql*, as are the following predefined types:

`char*`	A variable length string.
`date_t`	A date represented internally as seconds since the epoch. The data representation can be in any of the ‘‘standard’’ forms. `date_t` fields can be used in date comparisons such as `mtime<"yesterday"`.
`double`	A double floating point constant.
`elapsed_t`	Scaled elapsed time showing the two most significant time components. Examples are: `1.03s` one and three one hundreth seconds `2m20s` two minutes and 20 seconds `3w11d` three weeks and 11 days
`float`	Equivalent to `double`.
`int`	Equivalent to `long`.
`long`	A long integer constant.
`void`	The return type of user defined actions.

A *type-specifier* may be schema name. *schema-name*`*` declares a sub-schema field whose data is in a separate file whereas *schema-name* declares a sub-schema whose data is the field data itself. *type-specifier field-name*`[`*size*`]` declares an array field whose values are separated by a sub-field delimiter.

The schema name `cql` is also reserved. The fields in this schema are predefined by *cql* and provide access to run-time information. The fields are:

`elapsed_t clock`	The elapsed time since the start of this *cql* in hundredths of a second.
`date_t date`	The time this *cql* started.
`int errors`	The non-fatal error count.
`char* getenv(char*` *name*`)`	Returns the value of the environment variable *name*. For example, `cql.getenv("PATH")` is the value of the `PATH` environment variable.
`int line`	The current declaration or expression input line.
`int offset`	The current record offset in the main schema input file starting at 0.
`char* path(char*` *name*`, int` *length*`)`	Returns value of the file pathname *name* truncated to *length*. The truncation attempts to preserve the significant parts of the pathname.
`int record`	The current record number in the main schema input file starting at 1.
`int select`	The number of records selected by the select expression.
`int size`	The size in bytes of the current input record.
`date_t time`	The current time.

Schema attributes are also assigned in the declaration section.

```
schema = schema-name;
```

sets the main schema name. If omitted the main schema defaults to the first declared schema (from the top). Schema delimiters are defined by:

```
qualified-schema-name.delimiter = "delimiter";
```

The default main schema delimiter is `:` and the default sub-schema delimiter is `;`. Schema input is defined by:

```
qualified-schema-name.delimiter = "path";
```

where *path* is the pathname of the schema data file, or by:

```
qualified-schema-name.input = {
    "record-1",
    "record-n",
    };
```

where *record-i* are the schema records. Input data may be shared by:

```
a.input = b.input = ...
```

The default main schema input is the standard input; indirect sub-schema inputs must always be defined.

The expression language is basically C with the exception that `char*` operands are allowed for the `==,!=,<,<=,>=,>` operators. The expression may be labeled by:

```
label: expression;
...
```

or equivalently by:

```
void label() { expression };
...
```

the former being more convenient for command line expressions. The value of a labelled expression is either the value of a **return** statement evaluated in the expression or the value of the last statement evaluated in the expression. The default expression label is **select**. The **select** expression is applied to each record to determine the matching records. The default **select** is `1`, i.e., all records match. The **action** expression is then applied to each record matched by **select**. The default **action** lists the matching records on the standard output. The **begin** expression, if specified, is evaluated before the database is scanned and the **end** expression, if specified, is evaluated after all records have been scanned.

It is important to note that *cql* is not a complete C interpreter. Just enough is borrowed from C to get the query job done. The implementation is based on a C expression library that was originally written for the *tw* [FKV89] replacement for the *find* command. This library constitutes a large portion of the work; in fact the *cql* prototype was written in one day (although the addition of hash indexing and query optimization took considerably longer).

6 Reports

A query language is incomplete without some form of reporting. *cql* provides the `printf` function to output field values. `printf` is most often used in **action** expressions to output portions of selected fields. For example,

```
select: uid < 10 && name.expire < "in 2 days";
action: printf("%s will expire on %s\n", name, name.expire);
```

Format specifications are type checked and string to integer conversions are supported. For example, `name.expire` printed as `%d` lists the seconds since the epoch, but printed as `%s` lists the date in the standard date format.

Headers and footers are done by using `printf` in the **begin** and **end** expressions. Fancier operations like pagination, text filling and font highlights are left to tools designed specifically for that job.

Selected records may be sorted by specifying the sort order:

```
sort = { field, ... };
```

7 Alternate Database Formats

Three alternate database formats are supported. The formats are orthogonal and may appear in any combination. The first, *virtual database files*, **Figure 3**, allows multiple schemas to be defined in a single file. This format is similar to UNIX system archive file format but is designed for quick access to the individual schemas. In the past *cia* generated two database files with the fixed names **reference.db** and **symbol.db**. This made it difficult to copy and backup abstraction databases. By using a virtual database *cia* now combines the two files into a single file with an application specific name, e.g., **ksh.db** for the *ksh* abstraction database. A virtual database file acts like a directory in the *cql* interface. For example, the `symbol` schema data in the `ksh.db` virtual file is named `ksh.db/symbol`. A virtual database file is also used to store the index files for each input database. The index file is named by inverting the case of the corresponding database file name suffix. For example, the indices for `ksh.db` would be `ksh.DB`.

The second format, *partitioned database files*, **Figure 4**, allows a single database schema to span more than one file. The implementation is complicated by the fact that separate index files may be associated with each partition. Partitions are named as a single file pathname by separating the partition component pathnames with `':'`, as in "`ksh.db:libast.db:libdl.db`".

Finally, a *union database file*, **Figure 4**, allows schema records to be intermixed in a single file. The first union record field is used to identify the schema for each record. This format is convenient for applications that generate streams of different record types.

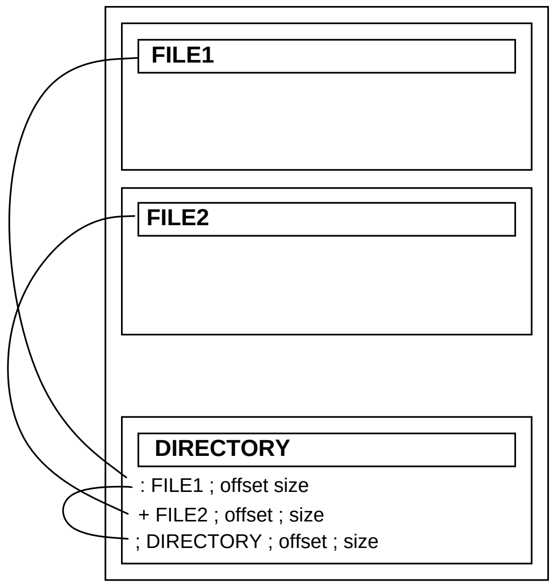

Figure 3. Virtual database format

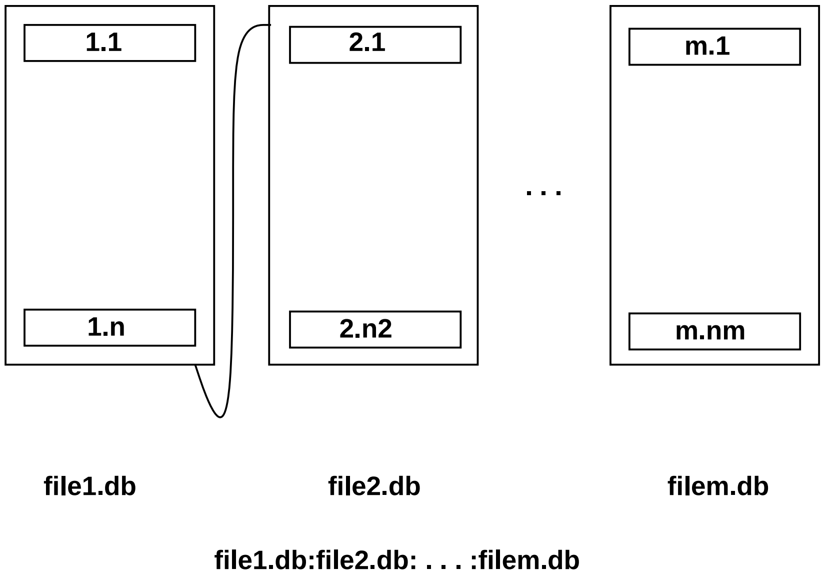

Figure 4. Partition database format

8 Future Work

It should be possible to access binary and fixed length fields with no delimiters. These are not supported yet.

A transitive closure operation would also be useful.

9 Conclusion

cql is an attractive scripting tool for database queries. Because it is an interpreted query language it supports rapid prototyping; because it is fast it allows prototypes to become production code. Its interpreted nature also

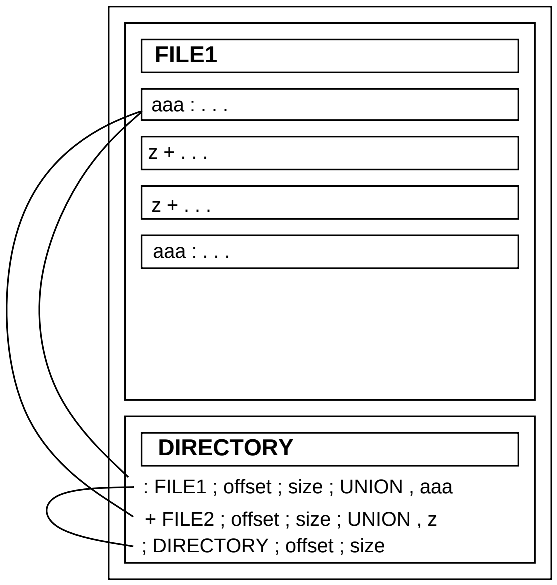
FILE1
aaa : . . .
z + . . .
z + . . .
aaa : . . .
DIRECTORY
: FILE1 ; offset ; size ; UNION , aaa
+ FILE2 ; offset ; size ; UNION , z
; DIRECTORY ; offset ; size

Figure 5. Union database format

makes it easy for database users (as opposed to database providers) to write and test new queries.

10 Acknowledgements

Thanks go to my colleagues in the Software Engineering Research Department: Phong Vo for the excellent *sfio* implementation that pepped up *cql* I/O; David Korn for the hash index file algorithm; Robin Chen for translating database lingo and concepts into terms a shell and C hack could understand.

References

[AKW88] A. V. Aho, B. W. Kernighan, P. J. Weinberger, *The Awk Programming Language*, Addison-Wesley, 1988.

[BK89] Morris Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.

[Bour78] S. R. Bourne, *The UNIX Shell*, AT&T Bell Laboratories Technical Journal, Vol. 57 No. 6 Part 2, pp. 1971-1990, July-August 1978.

[BSD86] *The UNIX Programmer's Reference Manual: 4.3 Berkeley Software Distribution*, UC Berkeley, California, 1986.

[CF88] Steve Cichinski and Glenn S. Fowler, *Product Administration through Sable and Nmake*, AT&T Bell Laboratories Technical Journal, Vol. 67 No. 4, pp. 59-70, July-August 1988.

[Chen89] Yih-Farn Chen, *The C Program Database and Its Applications*, Proc. of Summer 1989 USENIX Conf.

[Chri92] Tom Christiansen, private correspondence.

[Felt82] S. Felts, *The Unity DBMS*, AT&T Bell Laboratories Technical Memorandum, TM82-59312-1, 1992.

[FKV89] Glenn S. Fowler, David G. Korn and Kiem-Phong Vo, *An Efficient File Hierarchy Walker*, Proc. of Summer 1989 USENIX Conf.

[Hume88] Andrew Hume, *Grep Wars*, EUUG Conference Proceedings, London, England, Spring 1988.

[Park91] *Off The shelf: B-tree data-file managers*, Tim Parker, UNIX Review, Vol. 9 No. 3, pp. 55-58, March 1991.

[Wall] Larry Wall, *The Nutshell perl Book*.