cql – A Flat File Database Query Language

Glenn Fowler
gsf@research.att.com

AT&T Labs Research

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 }
```

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

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.