The Giles Production System Compiler

Rob King
KoreLogic Security, Inc.

Abstract

Production rule systems have found widespread use in event correlation, automated planning, expert systems, and other areas of artificial intelligence. This paper describes Giles, a compiler that is used to create production rule systems. Its defining feature is that its compilation target is not machine code, but instead a modern relational database management system (RDBMS). While there has been much research into using production rule systems internally to improve RDBMSs, and in using RDBMSs as storage backends for production rule systems, Giles has a different goal. The compiler utilizes these systems' ability to embed procedural logic ("triggers") to implement a production rule system entirely within a normal database for these systems. In other words, Giles can implement an entire production rule system in an unmodified RDBMS, working with existing database constructs.

This approach conveys numerous benefits, such as allowing the programmer to use existing ubiquitous database access interfaces, to reduce runtime software dependencies, and to give transactional and data reliability guarantees to the production rule system. It also creates production rule systems that are well suited for working with large amounts of data over long periods of time, and that are easily embedded inside other projects.

This paper discusses the usage of the Giles compiler and the systems it creates, as well as the implementation of the compiler. A strong focus is made on the techniques used to ensure the resulting production rule systems are efficient and work well with modern query optimizers.

1 Introduction

Giles[1] is a compiler that creates production systems (or "engines" in Giles parlance), with a special focus on using these engines for event correlation, forensic behavior detection and log analysis, and expert systems. Giles's defining feature is that the output of the compiler is not a runnable production system, but rather a schema for modern, unmodified, SQL-based[2] relational database management system (RDBMS). This schema utilizes modern databases' ability to embed reactive procedural logic ("triggers") to implement the described production system as a normal database. Databases created using this schema *are* production systems, with no additional software required at runtime.

This approach has immediate advantages. Perhaps the most important is that programmers can use normal database access interfaces, which are ubiquitous; this immediately makes Giles engines accessible to far more programmers than those of traditional systems. The engines are also able to take advantage of modern databases' data safety and reliability guarantees, as well as their transactional semantics. This means that Giles engines are capable of dealing with huge amounts of data safely, even in the face of system crashes, and over long periods of time. These benefits are important given the expected use cases of Giles engines: pre-defined production systems that are built into and provide intelligence for larger, potentially long-running and unattended, systems.

Much research has been performed in modifying databases to use production systems internally (i.e. invisibly to the user) for query optimization, view materialization, and other such topics; see for example [5]. Likewise, there has been considerable research in using relational databases to store state for production systems (e.g. [7]). The problem addressed by Giles is somewhat different: to implement an *entire* production system solely in an *unmodified* RDBMS, meaning it must work with the constructs available to a normal user of such a system. Giles does not require the underlying RDBMS to be modified, and does not require any external "driver" or "logic" program. The schema output by the compiler implements the entire production system, in any compatible RDMBS.

This paper is divided into two parts. The first part describes the usage of the Giles compiler and of the engines it creates. The second part describes the compiler's output in detail, with a special focus on the techniques used to make the compiled engines work well with database query optimizers.

1.1 Overview of Production Systems

Production systems (or *production rule systems*) are software systems that consist of *facts* and *rules* (or *productions*, hence the name).

A fact is a single datum describing a discrete piece of information in the problem domain of the system. Some example facts might be:

- Suzanne is an astronaut.
- Train #447 arrived at platform $9\frac{3}{4}$ at 08:00.
- User 'jdoe' logged in via terminal #16 at 12:33.

The set of all known facts is referred to as the *working memory* of the system.

Facts are generally represented as structured pieces of information consisting of some number of *fields*. In Giles, every fact belongs to exactly one *class*, and classes determine what *fields* that fact has. Fields are identified by name.

Productions are simple "if-then" rules. The "if" part, known as the *predicate*, describes a pattern of facts. The predicate can specify an arbitrary number of pattern clauses matching an arbitrary number of facts, and the facts to be matched can be related to one another by various relational operators. The "then" part, referred to as the *action*, describes an action to take when a set of facts in working memory matches the associated predicate. Different systems define different possible actions, but essentially all (Giles included) define at a minimum *assert* (add a new fact to working memory) and *retract* (remove an existing fact from working memory) as actions. When a production's action is taken, the production is said to have *fired*.

The general mechanism of operation then in a production system is thus:

Step 1 Determine which productions have predicates satisfied by some set of facts in working memory. Halt if no predicates are satisfied.

Step 2 Determine which of those productions should fire.

Step 3 Perform the actions specified by those productions.

Step 4 Go to step 1.

This loop is referred to as the *recognize-act* cycle.

Step 2 in this cycle is referred to as *conflict resolution*. Different systems have different conflict resolution algorithms, but Giles takes the simple approach of firing all productions whose predicates are true. Adding different, selectable conflict resolution algorithms to Giles is an area of active research (see [6] for a survey of different algorithms).

Given this, the goal of a production system is to efficiently find all matching predicates as facts are added and removed from working memory. This is more difficult than it may sound, as productions may have complicated predicates correlating multiple facts and the wrong pattern matching algorithm can result in super-linear behavior.

Matching sets of facts to productions is an example of the many-to-many matching problem. Various algorithms have been created to solve this problem; the canonical one is Rete ([4], though [3] may provide a more accessible introduction). Giles uses a variant of the Rete algorithm designed to be efficient when used with modern database query optmizers; much of this paper is dedicated to describing the implementation of this algorithm.

2 The Compiler and Its Output

2.1 An example engine

Engine descriptions can be presented to Giles in a variety of formats. Currently, the best-supported format uses YAML[3] files to describe engines. Valid engines must have at least one class of facts and one rule.

The engine presented in Listing 1[4] below is a simple engine that provides two fact classes, `IsHuman` and `IsMortal`, and a single rule, `AllHumansAreMortal`. This engine, when given a fact asserting someone is human, will also assert a new fact stating that that person is mortal.

The two fact classes each have a single field of type[5] `STRING` named `Person`. The only rule's predicate matches a single fact of class `IsHuman` and extracts the contents of the matched fact's `Person` field into a local variable. The rule's action asserts a new fact of type `IsMortal`, populating its `Person` field with the contents of the local variable `Person`.

The predicate has no complex conditions, and therefore will match each `IsHuman` fact asserted in the system. For each set of matching facts, a new "instance" of the rule is created with its own set of local variables. Thus, each `IsHuman` will create a new instance of the rule, and each instance's `Person` local variable will be populated with that fact's `Person` field.

This engine could be stored in a file called `mortal.yml` and compiled with Giles using a command line similar to the following:

```
giles -o mortal.sql mortal.yml
```

The output of this compilation step is a SQL database schema, by default for the ubiquitous and excellent SQLite[6] system. The output schema can be read into SQLite[7] to create a database:

```
sqlite> .read mortal.sql
```

The database is now a functioning production system. To test this, we can assert that Socrates is human:

```
sqlite> INSERT INTO
    Giles_IsHuman_Facts(Person)
    VALUES('Socrates');
```

and then check to see if the engine has concluded that he is also mortal:

```
sqlite> SELECT * FROM
        Giles_IsMortal_Facts;

Person = Socrates
id     = 1
```

This indeed shows that there is an `IsMortal` fact asserting that a person named Socrates is mortal. The `id` field is automatically present in all fact classes, and provides a unique identifier for every fact.

As is hopefully obvious from the example above, inserting a row into a table is equivalent to asserting a fact; deleting a row is likewise equivalent to retracting a fact. Facts asserted directly by the user are referred to as *axioms* or *axiomatic facts*.

We can ask the system to justify how its knowledge that Socrates is mortal:

```
sqlite> SELECT Justification
    FROM Giles_IsMortal_Justification
    WHERE id = 1;

    Fact 'IsMortal #1' was produced
    by rule 'AllHumansAreMortal':
    All humans are mortal.
    Justification:
        * The named person is human.
          (IsHuman #1)
```

This justification can be applied recursively. We could ask for the justification of `IsHuman #1`, and so on until the ultimate source of a given fact is found (that is, the axioms and rules used to derive that fact).

If we wish, we can retract the initial assertion that Socrates is mortal:

```
sqlite> DELETE FROM
        Giles_IsHuman_Facts
        WHERE Person = 'Socrates';
```

The retraction of this initial fact will automatically cause the retraction of any dependent facts:

```
sqlite> SELECT COUNT(*)
        FROM Giles_IsMortal_Facts;
COUNT(*) = 0
```

This retraction applies no matter how many degrees of separation (i.e. causal chains of fired rules) intervene between the retracted fact and the dependent fact.

It is a general guarantee of Giles engines that all facts that can be derived from the known axioms are derived (again, regardless of degree of separation), that no unprovable facts are derived, and that this set of derived facts is updated automatically as axioms are asserted and retracted. Moreover, thanks to Giles's use of the underlying database engine's transactional semantics, asserting or retracting facts is guaranteed to either have all consequences accounted for or to fail, even in light of system crashes and power outages. This functionality makes Giles particularly well-suited to simulation problems and to experimentation, as users can add and remove axioms without fear of lost data or unjustifiable facts.

Giles's purpose in deriving all derivable facts is a reflection of its original purpose in building forensic and behavior-detection event correlation engines. In forensic situations, investigators generally want to know all possible causes of a given set of events — something that could be both benign and malicious should certainly not be ignored. This guarantee also serves diagnostic expert systems well. For example, a patient may have a single illness that causes all of his or her symptoms, or may have a cluster of illnesses. Deriving all possible facts would reveal all possible combinations of diagnoses that could explain these symptoms.

The guarantee that all derivable facts are derived extends even to automatic retraction of facts. For example, say rule R has as its action the retraction of fact F, and rule R fires as a consequence of fact F'. If fact F' is later retracted (either manually or as the consequence of some rule), rule R will "un-fire" and fact F will be restored. This implies that Giles engines must store enough information to re-assert facts that are retracted in response to a rule's firing. This is the case, although the compiler generates code that stores the bare minimum amount of information necessary.

Note that this reassertion guarantee does not apply to facts that are deleted manually, that is, via a user-provided `DELETE` statement. Giles guarantees that all facts derivable from a set of axioms are in fact derived; a manual deletion removes an axiom, and thus changes the set of derivable facts.

Listing 1: An example engine

```
Facts:
    IsHuman:
        Person: STRING

    IsMortal:
        Person: STRING

Rules:
    AllHumansAreMortal:
        Description: All humans are mortal.

        MatchAll:
            - Fact:    IsHuman
              Meaning: The named person is human.
              Assign:
                Person: !expr This.Person

        Assert:
            IsMortal:
                Person: !expr Locals.Person
```

2.2 A More Complex Example

The example engine in Listing 1 shows many of Giles's features, but fails to show some of Giles's more advanced features, like multi-fact predicates, multi-rule engines, negative matches, YAML compact syntax, etc. This section describes a more complex engine (in Listing 2) that illustrates some of these features. For the sake of brevity, we will not delve too deeply into this engine's operation; it is instead presented and briefly explained, leaving further investigation to the reader.

This engine infers that a user's password has been compromised if the following are true:

- A time window W has been specified.
- User U logs in to terminal T.
- User U logs in to terminal T', where $T \neq T'$.
- The login to T' is after the login to T, but within W seconds.

Note that the facts matched by this rule (or any rule) can be asserted in any order; Giles engines are completely order-insensitive. This explains the explicit `LogTime` field used in this example; this field is presumably populated by the user from the log entry and is used to provide explicit temporal ordering of the asserted facts. Thus the rule still works and orders the facts correctly, regardless of the order with which they were actually asserted. This is useful for situations where network delays or long-running jobs can result in out-of-order delivery of information from various sensors.

This engine will produce one alert per suspect login-pair per time window. That is, if the user logs into Terminal T, then T' and then T'', there will be three alerts: one for (T, T'), one for (T, T''), and one for (T', T''). If two time window facts were added in this example, there would be six alerts, and so on.

As the result set is updated atomically and accurately, users could experiment with different time windows — adding and removing time windows will be reflected in the result set correctly. This is a useful feature, as it allows engines to be "tunable" and allows real-time experimentation with these tunables to, for example, test different configurations or generate different reports.

2.3 Efficiency

Giles engines are designed to trade space for time. It is difficult to provide a formal analysis of the time complexity of an engine, as it depends entirely on the rule set and the facts asserted. However, in the general case for a well-written engine, experimentation has shown that the time needed to assert or retract n facts is on the order of $O(n)$ in the common case. This is opposed to a more naive matching algorithm, where the time needed would be a function of the *total* number of facts in the system.

This temporal efficiency comes at the cost of increased storage space. The exact storage requirements are difficult to estimate, again because the space needed depends

Listing 2: A more complex example

```
Facts:
  Alert:
    Message: STRING

  TimeWindow:
    WindowLength: INTEGER

  UserLogin:
    Username: STRING
    Terminal: STRING
    LogTime:  INTEGER

Rules:
  UserLoggedInToTwoTerminalsWithinWindow:
    MatchAll:
      - Fact:    TimeWindow
        Meaning: A time window has been specified.
        Assign:
          WindowLength: !expr This.WindowLength

      - Fact:    UserLogin
        Meaning: A user logged in to one terminal.
        Assign:
          Username: !expr This.Username
          Terminal: !expr This.Terminal
          LogTime:  !expr This.LogTime

      - Fact:    UserLogin
        Meaning: That user logged into another terminal within a time window.
        When:    !expr This.Username == Locals.Username AND
                       This.Terminal != Locals.Terminal AND
                       This.LogTime  >= Locals.LogTime  AND
                       This.LogTime  <= Locals.LogTime + Locals.WindowLength

    Assert:
      Alert:
          Message: !expr ( "User " . Locals.Username .  " logged in twice!")
```

on the rule set and the set of asserted facts. Giles produces some code to reduce its space usage, and real-world testing has shown that well-written rule sets can reduce space usage by an order of magnitude or more for some problem sets.

The matching algorithm and the space- and time-saving optimizations are discussed in detail in the second part of this paper.

2.4 Recursion

Rules can, as actions, assert new facts or retract existing ones to and from working memory. As working memory is changed, new predicates may be matched and new new rules fired, which will again change working memory, and so on. If a rule set contains *assert-match* cycles, these changes could continue indefinitely.

The Giles input language therefore supports an additional primitive action: *assert-distinct*. This action asserts a new fact if and only if an identical fact does not already exist. This mechanism can be used to break out of a recursive loop.

Note that engines using recursion still respect the Giles guarantee that all derivable facts are always derived. If a fact produced by a distinct assertion is retracted by the action of some rule (either directly or indirectly), the engine will check to see if any rule would have produced an identical fact but for the now-retracted fact. If so, that rule is retroactively fired and the identical fact is asserted. Any fact that can be asserted by any number of distinct assertions is only completely retracted if no rule could possibly produce it given the current set of axioms.

2.5 Limitations

One notable limitation faced by Giles is a byproduct of its use of an existing RDBMS: new rules cannot be added at runtime because, in general, RDMBS's do not allow trigger programs to modify the database schema. This limits the kinds of problems for which Giles can be used; machine learning applications are particularly impacted. However, this is not seen as a major limitation because it is beyond the principle use case envisioned for Giles.

Another notable limitation is a consequence of Giles's mandate on maintaining consistency at all costs: engine databases grow without bound. There is no mechanism by which a fact can be really and truly deleted without external action; even rules with a retraction action still must store some bookkeeping data in case the rule is ever "un-fired". Likewise, facts are always kept, in case they are needed by some future instance of a rule. This limitation is likewise not seen as much of a problem, as one of the primary use cases of Giles is to perform long-term log analysis and behavior detection where the behavior may take a long time (perhaps several months or years) and be made up of many smaller facts.

In practice, long-running engines often have external scripts that delete facts older than some age, which keeps database size limited at the cost of potentially missing some inferences. It should also be noted that Giles engines do discard facts that can be cheaply proven to never match any rule in the engine (see Section 3.2).

2.6 Other Features

Giles has numerous other interesting features that could not be included in this paper for lack of space; they are mentioned only briefly here:

Foreign Function Interface Giles engines can declare external linkage to single-valued functions present in the host database (e.g., the standard SQL `length` function). These functions can then be used in expressions in the language with full type checking.

Disjoint Negative Predicates In addition to the conjunctive `MatchAll` clause (which can contain any number of patterns), predicates can also contain disjunctive `MatchNone` clauses, which will cause a predicate to fail if any of the contained patterns match.

Regular Expression Support If the target database system supports it, expressions can use full regular expressions to perform string matching.

Tunable Parameters Parameters are user-definable special classes of facts that can have constraints on their values, and can be guaranteed to be single-valued or multi-valued and indexable via a string key. They are useful to present a "tuning" interface to a given engine (for example, an adjustable time window or severity threshold). Parameters are normal facts under the hood, and changing a parameter's value results in the appropriate changes to the set of derived facts.

Offline Upgrades While it is not possible to add new rules to an existing system, Giles supports offline upgrades in which a given system's axioms can be loaded into a new system with a potentially different rule set.

3 The Implementation

3.1 Rete in SQL

Giles is a compiler, but there is nothing particularly interesting about its implementation. The novel features

of Giles are all in the code it generates, or, more accurately, in the database schemas it generates. Therefore, this part of the paper discusses the schemas constructed by the compiler.

The basic design of a Giles engine is an implementation of the Rete algorithm in SQL. The Rete algorithm provides its optimizations by trading space for time using extensive memoization of partial match results in a discrimination network. Facts are inserted and retracted from the "top" of the network, and actions sit at the "bottom". The network consists of *action nodes*, α- and β-*memory nodes*, and *join nodes*.

An action node is a node activated when some rule's predicate matches; the node is responsible for performing whatever the rule's action is.

Each α-memory node consists of all of the facts of a given class. The α-memories consist of sets of facts matching some (possibly composite) constant predicate. In Giles there is only one predicate: "class". Each α-memory stores all of the facts of a given class. All of the α-memories taken together form the *working memory* of the system.

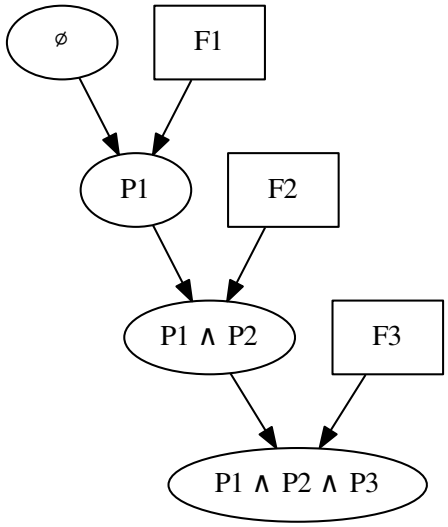
Each β-memory node consists of sets of facts that match some conjunction of tests in a rule. Each β-memory adds one more test, such that the final β-memory stores those sets of facts that match a rule's entire predicate. In Giles, β-memories also store the contents of local variables for a given instance of a rule; each β-memory holds the local variables that have been assigned by the current and previous match statements. These nodes memoize the results of each sub-predicate, allowing checks against smaller and smaller sets of facts to find matches to subsequent sub-predicates, reducing the number of comparisons that must be performed and thus time needed. The inclusion of local variables in these memories is an additional source of state that can be indexed to further improve performance (see 3.6).

The join nodes are connected to one α-memory and one β-memory as inputs, and one β-memory or one action node as an output. Whenever one of the input memories is updated, the join node performs a join operation on its two inputs, extracting new partial matches, and produces new outputs to be added to its output node (either as new contents for a β-memory, or as inputs for an action to be taken).

As a simple example, a rule that specifies the conjoined predicate "$P1(F1) \wedge P2(F2) \wedge P3(F3)$" (where Px are sub-predicates and Fx are fact classes) would result in a Rete network like the one shown in Figure 1 (note that α-memories are square, β-memories are ovals, and join and action nodes are not shown).

The basic idea of a Giles engine is straightforward: each α- and β- memory is a separate table, and each join and action node is a trigger. When a user asserts a new fact, it is by inserting a new row into an α-memory table. Upon insertion, one trigger per associated β-memory fires and checks to see if this new fact can be joined with any partial matches in the associated β-memory. Successful join results are inserted into the output β-memory or action node. Insertions into a β-memory fire another trigger for the associated α-memory, checking to see if any existing facts can now be joined with this new β-memory record. Successful join results are again inserted into the output node.

Figure 1: Rete graph for $P1(F1) \wedge P2(F2) \wedge P3(F3)$



Action nodes are implemented as triggers watching the lowest β-memory for a rule. When results are inserted into this memory, triggers fire that effect the designated action. For *assert* actions, new rows are inserted into the appropriate α-memories and for *retract* actions, rows are deleted from those memories. This action can cause further triggers watching those α-memories to fire, causing more activity.

Provided there are no cycles in actions and signature matches, the compiler will generate a set of triggers forming a directed acyclic graph. That is, for any assertion of a fact, no trigger will be activated twice. For rule sets that contain cycles, a given trigger may be activated many times — potentially infinitely many. For this reason, the compiler will not accept rule sets containing cycles without an explicit acknowledgement from the user.

Normal SQL foreign key references are used to handle deletions of dependent facts when their support is removed. Each non-axiomatic fact has references to a row in a β-memory that represents the action that produced it; if that row is deleted, the fact is retracted. Like-

wise, each row in a β-memory contains references to each matched fact, and will be automatically deleted if any of its matched facts are retracted.

Additional node types are defined to handle re-assertion of programatically-retracted facts. These nodes are implemented by triggers on β-memories that copy retracted axioms[8] to *shadow tables* when a retraction action is taken, and copy them back if the reason for their retraction is removed.

3.2 Fact Pruning

The first optimization made by the Giles engine is called α*-pruning*.

At compile time, the compiler examines all predicates across all rules, and extracts all of the constant tests from them. For example, in the Giles match predicate

```
This.FirstName LIKE "James%" AND
This.Age > 3                 AND
This.LastName == Locals.Last
```

the tests on `This.Name` and `This.Age` are constant, while the test on `This.LastName` is not — it references a local variable.

The compiler gathers all of the constant tests across all matches in all rules and builds an "α-predicate" per fact class is true for a fact if that fact matches at least one of the conjoined predicates (in the example above, the α-predicate would be `This.Firstname LIKE "James%" AND This.Age > 3`).

This class α-predicate is then tested before every insertion of a fact; if it fails, it is proven that the given fact can never match any rule's predicate and the insertion is simply ignored. This cheap, constant-time test immediately forgoes any further processing on facts that cannot possibly affect future computation.

3.3 Query Optimization and Index Usage

Database query optimization is a complex topic (though [2] offers an excellent survey of the subject). However, the general goal is to make a query completely covered by an index or, barring that, covering as many of the query's clauses as possible. Portions of a query that are not covered by an index result in a linear scan over the partial results. In the worst case, this can result in a *table scan* in which the entire contents of a table are scanned for a match.

Various SQL operations result in scans and different database systems' optimizers choose indexes differently. For the purposes of this document, we focus on the optimizer for SQLite[9]. While this optimizer may have limitations, it serves as an excellent example and the techniques used here apply to many other targets as well.

Indexes can be *multi-column*. For example, an index might be built over multiple columns like this:

```
CREATE INDEX idx ON Table (
    ColA, ColB, ColC
);
```

This would create an index that allowed rapid querying of `Table` if the columns used in the query were some subset of `ColA`, `ColB`, and `ColC`. However, in the SQLite query optimizer, there are additional constraints. An index will be used if and only if all of the following conditions are true:

- The list of tested columns is a prefix of the list of indexed columns.
- The order of the tested columns matches the order of the (prefix of the) indexed columns.
- At most one of the tests is an inequality.
- If any of the tests is an inequality, it must be the last test.

This means that the following query cannot fully use the index created above:

```
SELECT * FROM Table WHERE
    ColA != 'foo' AND
    ColB  = 1     AND
    ColC  = 2;
```

because the first test is not an equality. Only the first test (`ColA`) will use the index.

This query likewise cannot use the index:

```
SELECT * FROM Table WHERE
    ColB = 1 AND
    ColC = 2 AND
    Col1 = 1;
```

because the columns are tested in a different order[10] from how they were indexed.

The following query cannot make full use of the index:

```
SELECT * FROM Table WHERE
    ColA = 'bar' AND
    ColC = 2;
```

In this case, only the first column of the index (`ColA`) can be used, because `ColB` is skipped.

Another common pattern in queries that prevents index usage is querying a table for a computed value, where the row of the table is used as an input to the computation. For example:

```
SELECT * FROM Table WHERE
    LENGTH(ColA) = 1;
```

This cannot use the index created above because it does not use *any* of the indexed values. Recall that the values of `ColA` were indexed, but the value of the computation

`LENGTH(ColA)` was not indexed anywhere. In this case, the RDBMS must instead perform a table scan, extracting the each value of `ColA`, applying `LENGTH`, and comparing the result.

The compiler has numerous strategies to ensure that all of the generated queries are as covered by indexes as possible.

3.4 Query Rewriting

Recall that all predicates in rule matches are conjunctions of tests on facts. Because conjunction is commutative, it means that the tests can be reordered in any way. Giles takes advantage of this to rewrite tests to better make use of indexes.

For example, this Giles predicate:

```
This.Age    > 3         AND
This.Name   = 'James'   AND
This.Height < 96        AND
This.City   = 'Austin'
```

would be rewritten

```
This.City   = 'Austin' AND
This.Name   = 'James'  AND
This.Age    > 3        AND
This.Height < 96
```

All of the constant tests would be moved to the front of the query and then lexically sorted. This ensures that the longest possible prefix of tests are equality tests, which is a prerequisite of optimal index usage. The lexical sorting is a simple tactic that helps with index minimization, discussed below.

3.5 Index Minimization

Whenever a table (i.e. an α- or β-memory) is used in a query, Giles makes note of the columns queried. At the end of processing, the compiler will output a minimal list of indexes such that all of the queries are covered by at least one index, and such that no index is a perfect prefix of a longer index. "Covered" in this case means that the (rewritten) expression has all of its equality tests (plus one inequality if present) covered by an index.

For example, the queries:

```
SELECT * FROM Table WHERE
    ColA = 'foo' AND
    ColB = 1     AND
    ColC < 3;
```

and

```
SELECT * FROM Table WHERE
    ColA = 'foo' AND
    ColB = 1     AND
    ColC = 3     AND
    ColF = 7     AND
    ColZ > 10;
```

would result in an index being created on `Table`:

```
CREATE INDEX idx ON Table (
    ColA, ColB, ColC, ColF, ColZ
);
```

This index covers both queries, because the first query uses a set of columns that form a prefix of those used by the second. However, given the following queries:

```
SELECT * FROM Table WHERE
    ColA = 'foo' AND
    ColC = 3     AND
    ColB < 1;
```

and

```
SELECT * FROM Table WHERE
    ColA = 'foo' AND
    ColB = 1     AND
    ColC = 3     AND
    ColF = 7     AND
    ColZ > 10;
```

the compiler would build two indexes:

```
CREATE INDEX idx1 ON Table (
    ColA, ColC, ColB
);
```

and

```
CREATE INDEX idx2 ON Table (
    ColA, ColB, ColC, ColF, ColZ
);
```

because the first query tests the table's columns in a different order (enforced by the mix of equalities and inequalities in the query).

3.6 Synthetic Assignments

Take the example engine in Listing 3.

This (rather contrived) engine detects potential members of families by matching last names. Pay particular attention to the predicate on lines 26–29. This predicate has on its right hand side a computed value that uses local variables in the computation.

This might be compiled into a query similar to:

```
SELECT * FROM Alpha, Beta
WHERE
  Alpha.Family =
    UPPER(
        SUBSTR(Beta.Last, 1, 1))
    ||
    LOWER(
        SUBSTR(Beta.Last, 2, 100))
```

As was shown in Section 3.3, this query cannot be covered by an index and results in a table scan, because the

Listing 3: An example of a difficult optimization problem

```
Facts:
  KnownFamilies:
      Family: STRING

  PersonIsNamed:
      First:  STRING
      Last:   STRING

  MightBelongToFamily:
      First:  STRING
      Family: STRING

Rules:
  PersonMightBelongToFamily:
      Description: A named person might belong to a known family.

      MatchAll:
          - Fact:    PersonIsNamed
            Meaning: A person has been named.
            Assign:
              First: !expr This.First
              Last:  !expr This.Last

          - Fact:    KnownFamilies
            Meaning: A family with a matching name exists.
            When:    !expr This.Family == UPPER(SUBSTR(Locals.Last, 1, 1))
                                          .
                                          LOWER(SUBSTR(Locals.Last, 2, 100))
            Assign:
              Family: !expr This.Family

      Assert:
          MightBelongToFamily:
              First:  !expr Locals.First
              Family: !expr Locals.Family
```

Listing 4: The previous example's assignments after the synthetic assignment optimization

```
Assign:
    First:        !expr This.First
    Last:         !expr This.Last
    _Synthetic_1: !expr UPPER(SUBSTR(This.Last, 1, 1))
                        .
                        LOWER(SUBSTR(This.Last, 2, 100))
```

query tests against a computed value in which a column takes part. To avoid situations like this, Giles automatically detects these sorts of predicates and builds synthetic local variables.

The compiler detects when the right hand side of a predicate has computed values based on local variables, and rewrites the assignments in the previous match clause to include "synthetic assignments" that precompute the values. In other words, the assignments on lines 20–22 in the example are augmented as if they had been written as shown in Listing 4.

The predicate is then compiled to something like this:

```
SELECT * FROM Alpha, Beta
WHERE
    Alpha.Family = Beta._Synthetic_1
```

which can easily be covered by an index (that the compiler will automatically generate).

4 Conclusions and the Future

The primary goal of Giles has been to create a tool that makes the creation of production systems that can be used from many languages simple. We believe that goal has been met. Giles has been used in our organization for over two years to create engines that are embedded in production software, and these engines have proven reliable and efficient. The engines themselves have been distributed (as part of a larger product) to third parties, and have functioned without issue even when presented with millions of facts for correlation. Several rule sets consisting of hundreds of rules have been written and tested in real-world environments and worked perfectly.

This is not to say there is not room for improvement. Giles has several issues that we are in the process of addressing.

The rule specification language is needlessly verbose, and especially wasteful of vertical space. Alternative syntaxes and input formats are being investigated, including XML and LISP-like S-expressions.

The lexical sorting on query rewriting helps minimize the number of indexes that must be created and maintained, but a more sophisticated algorithm would probably do considerably better in common cases.

We are also investigating other algorithms (e.g. [1]) that could be used by Giles to perform pattern matching.

However, the most important area of further work is the targeting of new RDBMSs. To date, Giles has only been used in a production capacity with SQLite. Research is underway to determine how easily and how well other systems could be targeted; the most active area of research is focusing on PostgreSQL[11].

All in all, Giles has proved a remarkably useful tool in the niche for which it was created. It grew from a proof-of-concept to a full-fledged tool very quickly and underwent at least two major rewrites. We are excited to see how it may grow in the future.

Availability

Giles has been released under a permissive, open source license and is freely redistributable. For more information, see `http://www.korelogic.com`.

References

[1] ACHARYA, A., AND TAMBE, M. Collection oriented match. In *Proceedings of the Second International Conference on Information and Knowledge Management* (New York, NY, USA, 1993), CIKM '93, ACM, pp. 516–526.

[2] CHAUDHURI, S. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 1998), PODS '98, ACM, pp. 34–43.

[3] DOORENBOS, R. B. Production matching for large learning systems. Tech. rep., Pittsburgh, PA, USA, 2001.

[4] FORGY, C. L. Expert systems. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, ch. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pp. 324–341.

[5] HANSON, E., AND WIDOM, J. An overview of production rules in database systems. *The Knowledge Engineering Review 8* (1992), 121–143.

[6] MCDERMOTT, J., AND FORGY, C. Production system conflict resolution strategies. *SIGART Bull.*, 63 (June 1977), 37–37.

[7] SELLIS, T., LIN, C. C., AND RASCHID, L. Implementing large production systems in a dbms environment: Concepts and algorithms. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM, pp. 404–423.

Notes

[1] Giles is named after the character Rupert Giles from the *Buffy the Vampire Slayer* television and comic book series. The names are pronounced identically, with a soft *G*.

[2] This paper assumes the reader has some familiarity with SQL. If needed, `https://www.sqlite.org/lang.html` provides a good overview of the dialect of SQL used for the examples in this paper.

[3] `http://www.yaml.org/`

[4] The example listings in this paper are all YAML documents. Of particular note is the `!expr` annotation. This is an example of a YAML *tag*. The compiler uses the `!expr` tag to signal that a dynamic expression follows. Without such a tag, values are interpreted from syntax rules as strings or numeric literals.

[5] Giles has a simple but sound type system of four types: `BOOLEAN`, `INTEGER`, `REAL`, and `STRING`. Unlike SQL itself, the Giles system does not allow `NULL` values, and the type system enforces this.

[6] `http://www.sqlite.org/`

[7] These example SQLite sessions have been slightly edited to better fit in the space provided.

[8] Note that only axioms are copied to shadow tables, because all derived facts can be recalculated when the axioms are re-asserted.

[9] `https://www.sqlite.org/optoverview.html`

[10] Note that many query optimizers will automatically detect this case and reorder the queried columns.

[11] `http://www.postgresql.org`