Integrating Diverse Data Sources with Gadfly 2

Aaron Watters

http://www.chordate.com/

arw@ifu.net

ABSTRACT

This paper describes the primative methods underlying the implementation of SQL query evaluation in Gadfly 2, a database management system implemented in Python [Van Rossum]. The major design goals behind the architecture described here are to simplify the implementation and to permit flexible and efficient extensions to the gadfly engine. Using this architecture and its interfaces programmers can add functionality to the engine such as alternative disk based indexed table implementations, dynamic interfaces to remote data bases or or other data sources, and user defined computations.

Backgrounder

The term "database" usually refers to a persistent collection of data. Data is persistent if it continues to exist whether or not it is associated with a running process on the computer, or even if the computer is shut down and restarted at some future time. Database management systems provide support for constructing databases, maintaining databases, and extracting information from databases.

Relational databases manipulate and store persistent table structures called relations, such as the following three tables

```
 -- drinkers who frequent bars (this is a comment)
 select * from frequents

 DRINKER | PERWEEK | BAR
 ============================
 adam    | 1       | lolas
 woody   | 5       | cheers
 sam     | 5       | cheers
 norm    | 3       | cheers
 wilt    | 2       | joes
 norm    | 1       | joes
 lola    | 6       | lolas
 norm    | 2       | lolas
 woody   | 1       | lolas
 pierre  | 0       | frankies
)
```

```
 -- drinkers who like beers
 select * from likes

 DRINKER | PERDAY | BEER
 ===============================
 adam    | 2      | bud
 wilt    | 1      | rollingrock
 sam     | 2      | bud
 norm    | 3      | rollingrock
 norm    | 2      | bud
 nan     | 1      | sierranevada
 woody   | 2      | pabst
```

```
lola    | 5      | mickies

-- beers served from bars
select * from serves

BAR      | QUANTITY | BEER
=================================
cheers   | 500      | bud
cheers   | 255      | samadams
joes     | 217      | bud
joes     | 13       | samadams
joes     | 2222     | mickies
lolas    | 1515     | mickies
lolas    | 333      | pabst
winkos   | 432      | rollingrock
frankies | 5        | snafu
```

The relational model for database structures makes the simplifying assumption that all data in a database can be represented in simple table structures such as these. Although this assumption seems extreme it provides a good foundation for defining solid and well defined database management systems and some of the most successful software companies in the world, such as Oracle, Sybase, IBM, and Microsoft, have marketed database management systems based on the relational model quite successfully.

SQL stands for Structured Query Language. The SQL language defines industry standard mechanisms for creating, querying, and modified relational tables. Several years ago SQL was one of many Relational Database Management System (RDBMS) query languages in use, and many would argue not the best on. Now, largely due to standardization efforts and the backing of IBM, SQL is THE standard way to talk to database systems.

There are many advantages SQL offers over other database query languages and alternative paradigms at this time (please see [O'Neill] or [Korth and Silberschatz] for more extensive discussions and comparisons between the SQL/relational approach and others.)

The chief advantage over all contenders at this time is that SQL and the relational model are now widely used as interfaces and back end data stores to many different products with different performance characteristics, user interfaces, and other qualities: Oracle, Sybase, Ingres, SQL Server, Access, Outlook, Excel, IBM DB2, Paradox, MySQL, MSQL, POSTgres, and many others. For this reason a program designed to use an SQL database as its data storage mechanism can easily be ported from one SQL data manager to another, possibly on different platforms. In fact the same program can seamlessly use several backends and/or import/export data between different data base platforms with trivial ease. No other paradigm offers such flexibility at the moment.

Another advantage which is not as immediately obvious is that the relational model and the SQL query language are easily understood by semi-technical and non-technical professionals, such as business people and accountants. Human resources managers who would be terrified by an object model diagram or a snippet of code that resembles a conventional programming language will frequently feel quite at ease with a relational model which resembles the sort of tabular data they deal with on paper in reports and forms on a daily basis. With a little training the same HR managers may be able to translate the request "Who are the drinkers who like bud and frequent cheers?" into the SQL query

```
select drinker
from frequents
where bar='cheers'
  and drinker in (
      select drinker
      from likes
      where beer='bud')
```

(or at least they have some hope of understanding the query once it is written by a technical person or generated by a GUI interface tool). Thus the use of SQL and the relational model enables communication between different communities which must understand and interact with stored information. In contrast many other approaches cannot be understood easily by people without extensive programming experience.

Furthermore the declarative nature of SQL lends itself to automatic query optimization, and engines such as Gadfly can automatically translate a user query into an optimized query plan which takes advantage of available indices and other data characteristics. In contrast more navigational techniques require the application program itself to optimize the accesses to the database and explicitly make use of indices.

While it must be admitted that there are application domains such as computer aided engineering design where the relational model is unnatural, it is also important to recognize that for many application domains (such as scheduling, accounting, inventory, finance, personal information management, electronic mail) the relational model is a very natural fit and the SQL query language make most accesses to the underlying data (even sophisticated ones) straightforward.

For an example of a moderately sophisticated query using the tables given above, the following query lists the drinkers who frequent lolas bar and like at least two beers not served by lolas

```
select f.drinker
from frequents f, likes l
where f.drinker=l.drinker and f.bar='lolas'
  and l.beer not in
   (select beer from serves where bar='lolas')
group by f.drinker
having count(distinct beer)>=2
```

yielding the result

```
DRINKER
=======
norm
```

Experience shows that queries of this sort are actually quite common in many applications, and are often much more difficult to formulate using some navigational database organizations, such as some "object oriented" database paradigms.

Certainly, SQL does not provide all you need to interact with databases -- in order to do "real work" with SQL you need to use SQL and at least one other language (such as C, Pascal, C++, Perl, Python, TCL, Visual Basic or others) to do work (such as readable formatting a report from raw data) that SQL was not designed to do.

Why Gadfly 1?

Gadfly 1.0 is an SQL based relational database implementation implemented entirely in the Python programming language, with optional fast data structure accellerators implemented in the C programming language. Gadfly is relatively small, highly portable, very easy to use (especially for programmers with previous experience with SQL databases such as MS Access or Oracle), and reasonably fast (especially when the kjbuckets C accellerators are used). For moderate sized problems Gadfly offers a fairly complete set of features such as transaction semantics, failure recovery, and a TCP/IP based client/server mode (Please see [Gadfly] for detailed discussion).

Why Gadfly 2?

Gadfly 1.0 also has significant limitations. An active Gadfly 1.0 database keeps all data in (virtual) memory, and hence a Gadfly 1.0 database is limited in size to available virtual memory. Important features such as date/time/interval operations, regular expression matching and other standard SQL features are not implemented in Gadfly 1.0. The optimizer and the query evaluator perform optimizations using properties of the equality predicate but do not optimize using properties of inequalities such as BETWEEN or less-than. It is possible to add "extension views" to a Gadfly 1.0 database, but the mechanism is somewhat clumsy and indices over extension views are not well supported. The features of Gadfly 2.0 discussed here attempt to address these deficiencies by providing a uniform extension model that permits addition of alternate table, function, and predicate implementations.

Other deficiencies, such as missing constructs like "ALTER TABLE" and the lack of outer joins and NULL values are not addressed here, although they may be addressed in Gadfly 2.0 or a later release. This paper also does not intend to explain the complete operations of the internals; it is intended to provide at least enough information to understand the basic mechanisms for extending gadfly.

Some concepts and definitions provided next help with the description of the gadfly interfaces. [Note: due to the terseness of this format the ensuing is not a highly formal presentation, but attempts to approach precision where precision is important.]

The semilattice of substitutions

Underlying the gadfly implementation are the basic concepts associated with substitutions. A substitution is a mapping of attribute names to values (implemented in gadfly using kjbuckets.kjDict objects). Here an attribute refers to some sort of "descriptive variable", such as NAME and a value is an assignment for that variable, like "Dave Ascher". In Gadfly a table is implemented as a sequence of substitutions, and substitutions are used in many other ways as well.

For example consider the substitutions

```
A = [DRINKER=>'sam']
B = [DRINKER=>'sam', BAR=>'cheers']
C = [DRINKER=>'woody', BEER=>'bud']
D = [DRINKER=>'sam', BEER=>'mickies']
E = [DRINKER=>'sam', BAR=>'cheers', BEER=>'mickies']
F = [DRINKER=>'sam', BEER=>'mickies']
G = [BEER=>'bud', BAR=>'lolas']
H = [] # the empty substitution
I = [BAR=>'cheers', CAPACITY=>300]
```

A trivial but important observation is that since substitutions are mappings, no attribute can assume more than one value in a substitution. In the operations described below whenever an operator "tries" to assign more than one value to an attribute the operator yields an "overdefined" or "inconsistent" result.

Information Semi-order:

Substitution B is said to be more informative than A because B agrees with all assignments in A (in addition to providing more information as well). Similarly we say that E is more informative than A, B, D, F. and H but E is not more informative than the others since, for example G disagrees with E on the value assigned to the BEER attribute and I provides additional CAPACITY information not provided in E.

Joins and Inconsistency:

A join of two substitutions X and Y is the least informative substitution Z such that Z is more informative (or equally informative) than both X and Y. For example B is the join of B with A, E is the join of B with D and

```
E join I =
  [DRINKER=>'sam', BAR=>'cheers', BEER=>'mickies', CAPACITY=>300]
```

For any two substitutions either (1) they disagree on the value assigned to some attribute and have no join or (2) they agree on all common attributes (if there are any) and their join is the union of all (name, value) assignments in both substitutions. Written in terms of kjbucket.kjDict operations two kjDicts X and Y have a join Z = (X+Y) if and only if Z.Clean() is not None. Two substitutions that have no join are said to be inconsistent. For example I and G are inconsistent since they disagree on the value assigned to the BAR attribute and therefore have no join. The algebra of substitutions with joins technically defines an abstract algebraic structure called a semilattice.

Name space remapping

Another primitive operation over substitutions is the remap operation S2 = S.remap(R) where S is a substitution and R is a graph of attribute names and S2 is a substitution. This operation is defined to produce the substitution S2 such that

```
Name=>Value in S2 if and only if
    Name1=>Value in S and Name<=Name1 in R
```

or if there is no such substitution S2 the remap value is said to be overdefined.

For example the remap operation may be used to eliminate attributes from a substitution. For example

```
E.remap([DRINKER<=DRINKER, BAR<=BAR])
   = [DRINKER=>'sam', BAR=>'cheers']
```

Illustrating that remapping using the [DRINKER<=DRINKER, BAR<=BAR] graph eliminates all attributes except DRINKER and BAR, such as BEER. More generally remap can be used in this way to implement the classical relational projection operation. (See [Korth and Silberschatz] for a detailed discussion of the projection operator and other relational algebra operators such as selection, rename, difference and joins.)

The remap operation can also be used to implement "selection on attribute equality". For example if we are interested in the employee names of employees who are their own bosses we can use the remapping graph

```
R1 = [NAME<=NAME, NAME<=BOSS]
```

and reject substitutions where remapping using R1 is overdefined. For example

```
S1 = [NAME=>'joe', BOSS=>'joe']
S1.remap(R1) = [NAME=>'joe']
S2 = [NAME=>'fred', BOSS=>'joe']
S2.remap(R1) is overdefined.
```

The last remap is overdefined because the NAME attribute cannot assume both the values 'fred' and 'joe' in a substitution.

Furthermore, of course, the remap operation can be used to "rename attributes" or "copy attribute values" in substitutions. Note below that the missing attribute CAPACITY in B is effectively ignored in the remapping operation.

```
B.remap([D<=DRINKER, B<=BAR, B2<=BAR, C<=CAPACITY])
   = [D=>'sam', B=>'cheers', B2=>'cheers']
```

More interestingly, a single remap operation can be used to perform a combination of renaming, projection, value copying, and attribute equality selection as one operation. In kjbuckets the remapper graph is implemented using a kjbuckets.kjGraph and the remap operation is an intrinsic method of kjbuckets.kjDict objects.

Generalized Table Joins and the Evaluator Mainloop

Strictly speaking the Gadfly 2.0 query evaluator only uses the join and remap operations as its "basic assembly language" -- all other computations, including inequality comparisons and arithmetic, are implemented externally to the evaluator as "generalized table joins."

A table is a sequence of substitutions (which in keeping with SQL semantics may contain redundant entries). The join between two tables T1 and T2 is the sequence of all possible defined joins between pairs of elements from the two tables. Procedurally we might compute the join as

```
T1JoinT2 = empty
for t1 in T1:
    for t2 in T2:
        if t1 join t2 is defined:
            add t1 join t2 to T1joinT2
```

In general circumstances this intuitive implementation is a very inefficient way to compute the join, and Gadfly almost always uses other methods, particularly since, as described below, a "generalized table" can have an "infinite" number of entries.

For an example of a table join consider the EMPLOYEES table containing

```
[NAME=>'john', JOB=>'executive']
[NAME=>'sue', JOB=>'programmer']
[NAME=>'eric', JOB=>'peon']
[NAME=>'bill', JOB=>'peon']
```

and the ACTIVITIES table containing

```
 [JOB=>'peon', DOES=>'windows']
 [JOB=>'peon', DOES=>'floors']
 [JOB=>'programmer', DOES=>'coding']
 [JOB=>'secretary', DOES=>'phone']
```

then the join between EMPLOYEES and ACTIVITIES must containining

```
[NAME=>'sue', JOB=>'programmer', DOES=>'coding']
[NAME=>'eric', JOB=>'peon', DOES=>'windows']
[NAME=>'bill', JOB=>'peon', DOES=>'windows']
[NAME=>'eric', JOB=>'peon', DOES=>'floors']
[NAME=>'bill', JOB=>'peon', DOES=>'floors']
```

A compiled gadfly subquery ultimately appears to the evaluator as a sequence of generalized tables that must be joined (in combination with certain remapping operations that are beyond the scope of this discussion). The Gadfly mainloop proceeds following the very loose pseudocode:

```
Subs = [ [] ] # the unary sequence containing "true"
While some table hasn't been chosen yet:
    Choose an unchosen table with the least cost join estimate.
    Subs = Subs joined with the chosen table
return Subs
```

[Note that it is a property of the join operation that the order in which the joins are carried out will not affect the result, so the greedy strategy of evaluating the "cheapest join next" will not effect the result. Also note that the treatment of logical OR and NOT as well as EXIST, IN, UNION, and aggregation and so forth are not discussed here, even though they do fit into this approach.]

The actual implementation is a bit more complex than this, but the above outline may provide some useful intuition. The "cost estimation" step and the implementation of the join operation itself are left up to the generalized table object implementation. A table implementation has the ability to give an "infinite" cost estimate, which essentially means "don't join me in yet under any circumstances."

Implementing Functions

As mentioned above operations such as arithmetic are implemented using generalized tables. For example the arithmetic Add operation is implemented in Gadfly internally as an "infinite generalized table" containing all possible substitutions

```
ARG0=>a, ARG1=>b, RESULT=>a+b]
```

Where a and b are all possible values which can be summed. Clearly, it is not possible to enumerate this table, but given a sequence of substitutions with defined values for ARG0 and ARG1 such as

```
[ARG0=>1, ARG1=-4]
[ARG0=>2.6, ARG1=50]
```

```
[ARG0=>99, ARG1=1]
```

it is possible to implement a "join operation" against this sequence that performs the same augmentation as a join with the infinite table defined above:

```
[ARG0=>1, ARG1=-4, RESULT=-3]
[ARG0=>2.6, ARG1=50, RESULT=52.6]
[ARG0=>99, ARG1=1, RESULT=100]
```

Furthermore by giving an "infinite estimate" for all attempts to evaluate the join where ARG0 and ARG1 are not available the generalized table implementation for the addition operation can refuse to compute an "infinite join."

More generally all functions f(a,b,c,d) are represented in gadfly as generalized tables containing all possible relevant entries

```
[ARG0=>a, ARG1=>b, ARG2=>c, ARG3=>d, RESULT=>f(a,b,c,d)]
```

and the join estimation function refuses all attempts to perform a join unless all the arguments are provided by the input substitution sequence.

Implementing Predicates

Similarly to functions, predicates such as less-than and BETWEEN and LIKE are implemented using the generalized table mechanism. For example the "x BETWEEN y AND z" predicate is implemented as a generalized table "containing" all possible

```
[ARG0=>a, ARG1=>b, ARG2=>c]
```

where b<a<c. Furthermore joins with this table are not permitted unless all three arguments are available in the sequence of input substitutions.

Some Gadfly extension interfaces

A gadfly database engine may be extended with user defined functions, predicates, and alternative table and index implementations. This section snapshots several Gadfly 2.0 interfaces, currently under development and likely to change before the package is released.

The basic interface for adding functions and predicates (logical tests) to a gadfly engine are relatively straightforward. For example to add the ability to match a regular expression within a gadfly query use the following implementation.

```
from re import match

def addrematch(gadflyinstance):
    gadflyinstance.add_predicate("rematch", match)
```

Then upon connecting to the database execute

```
g = gadfly(...)
...
addrematch(g)
```

In this case the "semijoin operation" associated with the new predicate "rematch" is automatically generated, and after the add_predicate binding operation the gadfly instance supports queries such as

```
select drinker, beer
from likes
where rematch('b*', beer) and drinker not in
  (select drinker from frequents where rematch('c*', bar))
```

By embedding the "rematch" operation within the query the SQL engine can do "more work" for the programmer and reduce or eliminate the need to process the query result externally to the engine.

In a similar manner functions may be added to a gadfly instance,

```
def modulo(x,y):
    return x % y

def addmodulo(gadflyinstance):
    gadflyinstance.add_function("modulo", modulo)

...
g = gadfly(...)
...
addmodulo(g)
```

Then after the binding the modulo function can be used whereever an SQL expression can occur.

Adding alternative table implementations to a Gadfly instance is more interesting and more difficult. An "extension table" implementation must conform to the following interface:

```
# get the kjbuckets.kjSet set of attribute names for this table
names = table.attributes()

# estimate the difficulty of evaluating a join given known attributes
#  return None for "impossible" or n>=0 otherwise with larger values
#    indicating greater difficulty or expense
estimate = table.estimate(known_attributes)

# return the join of the rows of the table with
# the list of kjbuckets.kjDict mappings as a list of mappings.
resultmappings = table.join(listofmappings)
```

In this case add the table to a gadfly instance using

```
gadflyinstance.add_table("table_name", table)
```

For example to add a table which automatically queries filenames in the filesystems of the host computer a gadfly instance could be augmented with a GLOB table implemented using the standard library function glob.glob as follows:

```
import kjbuckets

class GlobTable:
    def __init__(self): pass

    def attributes(self):
        return kjbuckets.kjSet("PATTERN", "NAME")

    def estimate(self, known_attributes):
        if known_attributes.member("PATTERN"):
            return 66 # join not too difficult
        else:
            return None # join is impossible (must have PATTERN)

    def join(self, listofmappings):
        from glob import glob
        result = []
        for m in listofmappings:
            pattern = m["PATTERN"]
            for name in glob(pattern):
                newmapping = kjbuckets.kjDict(m)
                newmapping["NAME"] = name
                if newmapping.Clean():
                    result.append(newmapping)
        return result

...
gadfly_instance.add_table("GLOB", GlobTable())
```

Then one could formulate queries such as "list the files in directories associated with packages installed by guido"

```
select g.name as filename
from packages p, glob g
where p.installer = 'guido' and g.pattern=p.root_directory
```

Note that conceptually the GLOB table is an infinite table including all filenames on the current computer in the "NAME" column, paired with a potentially infinite number of patterns.

More interesting examples would allow queries to remotely access data served by an HTTP server, or from any other resource.

Furthermore an extension table can be augmented with update methods

```
table.insert_rows(listofmappings)
table.update_rows(oldlist, newlist)
table.delete_rows(oldlist)
```

Note: at present the implementation does not enforce recovery or transaction semantics for updates to extension tables, although this may change in the final release.

The table implementation is free to provide its own implementations of indices which take advantage of data provided by the join argument.

Efficiency Notes

The following thought experiment attempts to explain why the Gadfly implementation is surprisingly fast considering that it is almost entirely implemented in Python (an interpreted programming language which is not especially fast when compared to alternatives). Although Gadfly is quite complex, at an abstract level the process of query evaluation boils down to a series of embedded loops. Consider the following nested loops:

```
iterate 1000:
f(...) # fixed cost of outer loop
iterate 10:
   g(...) # fixed cost of middle loop
   iterate 10:
      # the real work (string parse, matrix mul, query eval...)
      h(...)
```

In my experience many computations follow this pattern where f, g, are complex, dynamic, special purpose and h is simple, general purpose, static. Some example computations that follow this pattern include: file massaging (perl), matrix manipulation (python, tcl), database/cgi page generation, and vector graphics/imaging.

Suppose implementing f, g, h in python is easy but result in execution times10 times slower than a much harder implementation in C, choosing arbitrary and debatable numbers assume each function call consumes 1 tick in C, 5 ticks in java, 10 ticks in python for a straightforward implementation of each function f, g, and h. Under these conditions we get the following cost analysis, eliminating some uninteresting combinations, of implementing the function f, g, and h in combinations of Python, C and java:

COST	FLANG	GLANG	HLANG
111000	C	C	C
115000	java	C	C
120000	python	C	C
155000	java	java	C
210000	python	python	C
555000	java	java	java
560000	python	java	java
610000	python	python	java
1110000	python	python	python

Note that moving only the innermost loop to C (python/python/C) speeds up the calculation by half an order of magnitude compared to the python-only implementation and brings the speed to within a factor of 2 of an implementation done entirely in C.

Although this artificial and contrived thought experiment is far from conclusive, we may be tempted to draw the conclusion that generally programmers should focus first on obtaining a working implementation (because as John Ousterhout is reported to have said "the biggest performance improvement is the transition from non-working to working") using the methodology that is most likely to obtain a working solution the quickest (Python). Only then if the performance is inadequate should the programmer focus on optimizing the inner most loops, perhaps moving them to a very efficient implementation (C). Optimizing the outer loops will buy little improvement, and should be done later, if ever.

This was precisely the strategy behind the gadfly implementations, where most of the inner loops are implemented in the kjbuckets C extension module and the higher level logic is all in Python. This also explains why gadfly appears to be "slower" for simple queries over small data sets, but seems to be relatively "faster" for more complex queries over larger data sets, since larger queries and data sets take better advantage of the optimized inner loops.

A Gadfly variant for OLAP?

In private correspondence Andy Robinson points out that the basic logical design underlying Gadfly could be adapted to provide Online Analytical Processing (OLAP) and other forms of data warehousing and data mining. Since SQL is not particularly well suited for the kinds of requests common in these domains the higher level interfaces would require modification, but the underlying logic of substitutions and name mappings seems to be appropriate.

Conclusion

The revamped query engine design in Gadfly 2 supports a flexible and general extension methodology that permits programmers to extend the gadfly engine to include additional computations and access to remote data sources. Among other possibilities this will permit the gadfly engine to make use of disk based indexed tables and to dynamically retrieve information from remote data sources (such as an Excel spreadsheet or an Oracle database). These features will make gadfly a very useful tool for data manipulation and integration.

References

[Van Rossum] Van Rossum, Python Reference Manual, Tutorial, and Library Manuals, please look to http://www.python.org for the latest versions, downloads and links to printed versions.

[O'Neill] O'Neill, P., Data Base Principles, Programming, Performance, Morgan Kaufmann Publishers, San Francisco, 1994.

[Korth and Silberschatz] Korth, H. and Silberschatz, A. and Sudarshan, S. Data Base System Concepts, McGraw-Hill Series in Computer Science, Boston, 1997

[Gadfly]Gadfly: SQL Relational Database in Python, http://www.chordate.com/kwParsing/gadfly.html