SQL in Python, Python in SQL

Aaron Watters
Computer and Information Sciences
New Jersey Institute of Technology
University Heights
Newark, NJ, 07102
(201)596-2666

May 5, 1995

Abstract

This paper introduces the **Gadfly** prototype database manager which implements standard SQL in the Python prototyping language. It also briefly considers the possibility of embedding Python objects into a database management system in order to add object oriented features to relational technology, thus defining an object oriented database system.

Keywords: GADFLY, PYTHON, TK, GUI, DATABASE MANAGEMENT, DATABASE PUBLICATION, OBJECT ORIENTED DATABASES, SCRIPTING LANGUAGES.

1 The Gadfly Vision

Gadfly: (1) any of various flies that bite and annoy animals. (2) an irritating person, especially one who acts as a provocative stimulus. [Hybrid definition from the American Heritage and Webster's dictionaries.]

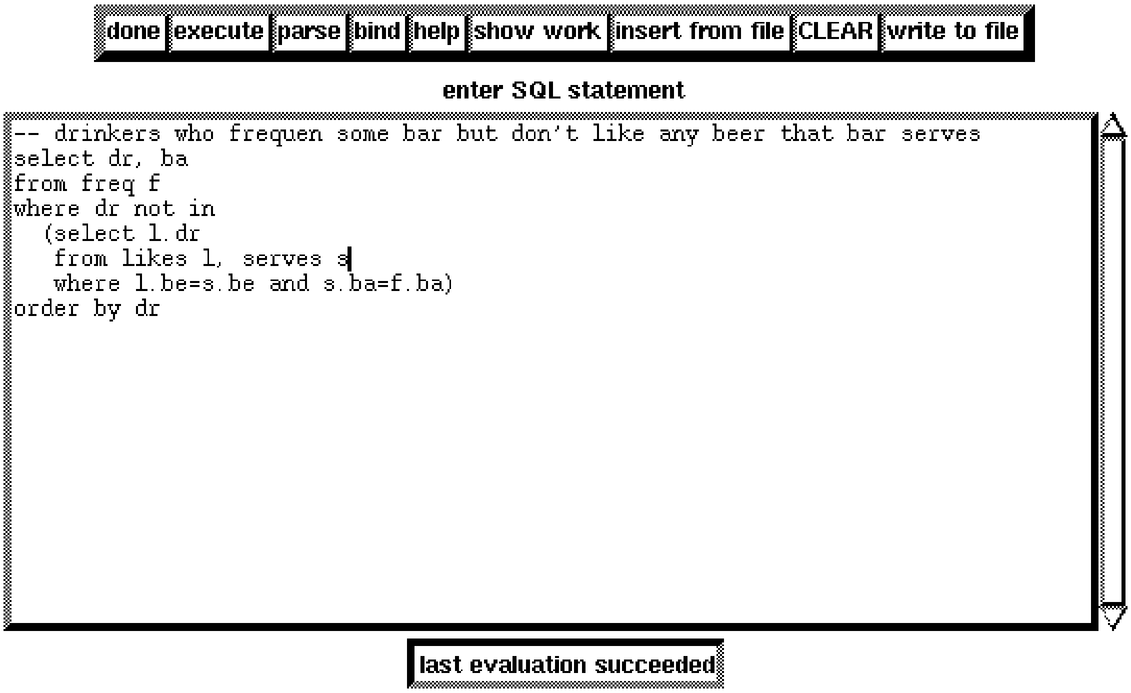
The Gadfly prototype is aimed at adding basic database functionality to the Python prototyping language using the SQL standard. Many possible uses for a light weight and portable database manager based in Python come to mind – from use in system administration tools, to use in generic data based applications, to use in multi-database interoperability.

One possible use for the prototype that might become feasible fairly soon could be as a vehicle for database publication. For example, it might be possible to fit Python and the prototype in a corner of a CD-rom (precompiled and configured for various platforms), with the remainder of the CD devoted to a fairly large database. Under such a scenario, the CD could encapsulate code for arbitrarily complex and convenient access to the data it contains, together with the ability to present and the manipulate the data using Python's scripting and graphical capabilities.

There is a much to be done before any of these visions will become a reality. This section discusses the current state of the prototype as well as planned extensions.

2 Status

Gadfly currently supports a significant subset of the SQL data manipulation language, as specified in MicroSoft's ODBC reference documentation[1], and in the near future it will support a much larger subset. The development is aimed at supporting SQL before adding more "modern" or experimental features, because (as Serge Abiteboul once put it) SQL is an intergalactic standard, and the prototype is much more likely



```
-- drinkers who frequen some bar but don't like any beer that bar serves
select dr, ba
from freq f
where dr not in
  (select l.dr
   from likes l, serves s
   where l.be=s.be and s.ba=f.ba)
order by dr
```

Figure 1: The SQL input screen.

to be used if it supports SQL than if it came with its own data manipulation language only. Gadfly has a simple, but nice, graphical user interface which runs under the Tk toolkit. Python programs may "embed" accesses to databases using Gadfly.[1]

At present the prototype is most appropriate for educational and experimental use. Nevertheless, hooks are in place for full database functionality including all of standard SQL with concurrency control and recovery, together with external interfaces to other data repositories of arbitrary kinds, as well as for the addition of "active" features such as triggers and external actions. I will devote most of the summer to improving the prototype and to adding the features listed above.

3 Implementation

If nothing else, Gadfly is an interesting example of a strategy for developing reasonably large applications in Python. The most interesting aspect of the prototype is that, although the size of the Python code is considerable, most of the "hard work" of query evaluation is done by the `kjbuckets` C extension module, which defines appropriate abstract data types for the problem at hand.

Figure 3 illustrates the dependencies between the major modules of the prototype.

Query Evaluation: The core to any SQL engine is the `SELECT` statement, and Gadfly supports most of the `SELECT` statement, and soon will support all of it. In particular, the query engine supports

- the standard `SELECT` (with `DISTINCT` or `ALL` including expressions and renaming), `FROM`, `GROUP BY`, and `ORDER BY` clauses;
- the `WHERE` clause with most standard predicates and expressions to arbitrary complexity, including the embedded subquery predicates `EXISTS` and `IN` and their negations.
- the `HAVING` clause (except that references to external aggregates in embedded subqueries are not yet supported).
- the standard aggregate functions such as `SUM(expr)` and `COUNT(DISTINCT expr)`.

Most other standard parts of the `SELECT` statement syntax can be (and will be) added fairly easily. Neither the `LIKE` predicate nor the `UNION` clause have been implemented, but they will not cause any difficulty. The general subquery constructs Θ`ANY` and Θ`ALL` need to be added, but do not present any difficulties either. Hooks are in place to support views and subselects in the `FROM` clause, but these features are not yet present.

The internal logic used to implement query evaluation is based on an extension of relational algebra using a cylindric formulation, a simplification of ideas described in [2] which in turn is inspired by Hull and Su [3] and Imielinski and Lipski [4]. The basis in cylindrical algebra maps easily to fast operations on graphs, sets, and mappings which allows the prototype to optimize equality based accesses reasonably well – especially since the basic set operations are implemented in a C extension to Python (`kjbuckets`) which does most of the "hard work".

The query planning and query evaluation use hash-based accesses, hashed semijoins, and hashed joins heavily. As yet there is no attempt to optimize "range queries" involving inequalities. Consequently of

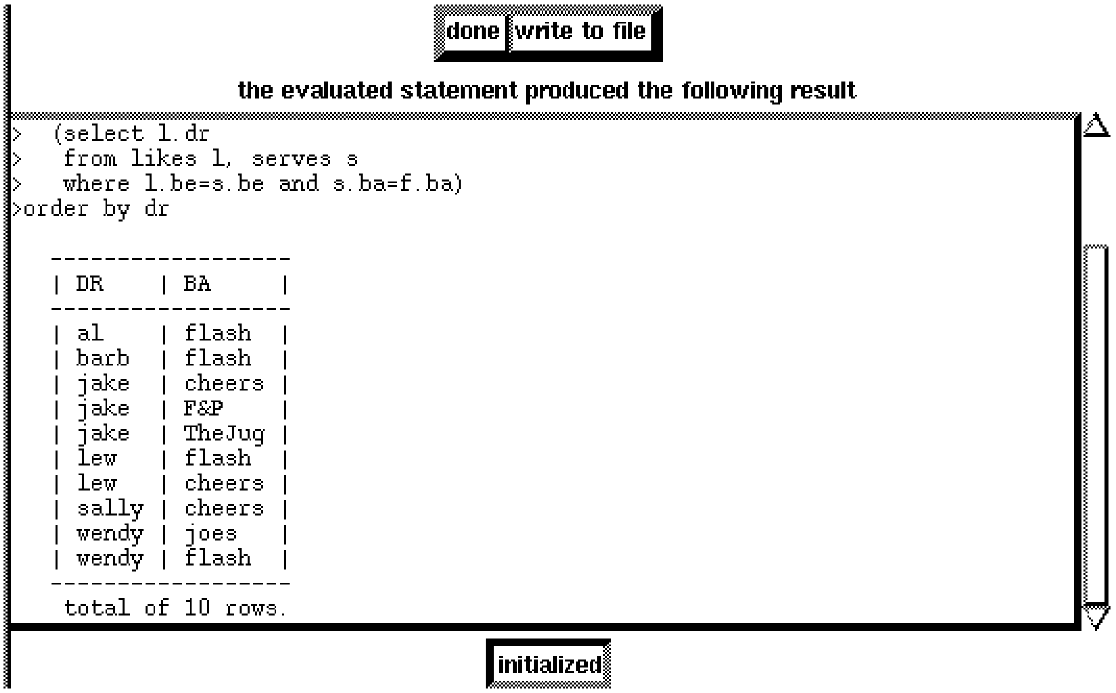
```
select * from r,s where r.a<=s.b and r.a>=s.b;
select * from r,s where r.a=s.b
```

the second query will execute much faster than the first, in general – since the equality testing of the second is "implemented in C" and translates to a "hash join" or r and s whereas inequality comparisons of the first are "interpreted in Python" over a cross product of r and s.[2] At binding time, static transformations are done to simplify expressions and conditions. Thus in the following "parse dump"

---

[1] But the present applications programmer interface needs to be cleaned up before it is documented and released.

[2] In the long term "range" based evaluation optimizations would be nice, but this is a low priority.



the evaluated statement produced the following result

```
>  (select l.dr
>   from likes l, serves s
>   where l.be=s.be and s.ba=f.ba)
>order by dr

   ------------------
   | DR    | BA     |
   ------------------
   | al    | flash  |
   | barb  | flash  |
   | jake  | cheers |
   | jake  | F&P    |
   | jake  | TheJug |
   | lew   | flash  |
   | lew   | cheers |
   | sally | cheers |
   | wendy | joes   |
   | wendy | flash  |
   ------------------
    total of 10 rows.
```

Figure 2: The SQL output screen.

```
> select * from dr where not (dr <> 'wendy')
parsed representation:
  select *
  from FREQ FREQ
  where (DR='wendy')
```

the (slow) non-inequality is translated to a (fast) equality.

Further discussion of the internals of the query processing engine are beyond the scope of this presentation.

Parsing: Gadfly parses SQL statements with the help of a Parser generator and a parsing module also written in Python.[3] Unfortunately, the parser (which uses no significant C extension aids) is a minor speed bottleneck which should be sped up at some point, perhaps by using `Lex` and `YACC` in place of the Python based parser generator.

User interface: The current graphical user interface to the prototype uses a small "forms" module built on top of Python's standard interface to the Tk graphical toolkit as described in Ousterhout [5]. Figure 1 shows the look of the SQL input screen and Figure 2 shows the screen that reports the result of evaluating the query listed in Figure 1. The interface at present is oriented towards demonstrating the capabilities of the system, and debugging, but it will be easy to extend the strategy to allow standard form-based accesses (such as inserting or modifying a row using a form). This interface is suitable for extension, both to provide more extensive native features for the prototype, and for use as an application programmer tool.

---

[3] These are modified versions of the `kwParsing` modules available at the the Python FTP sites – modified to do parser *generation* somewhat faster, and to report parse errors better.

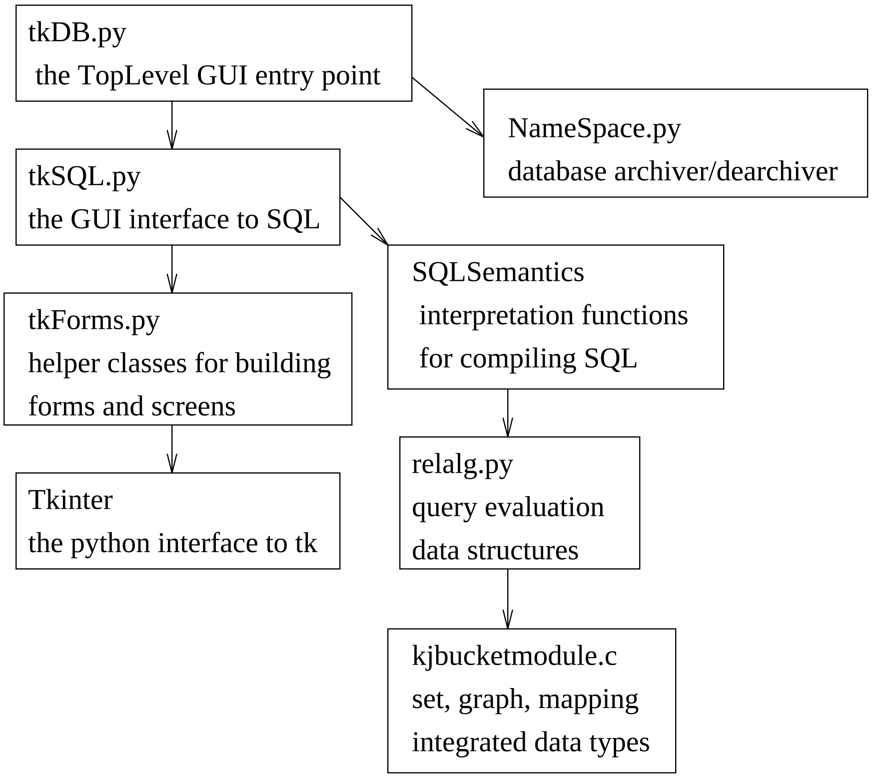


Figure 3: Gadfly primary modules.

4 Future

There are many planned extensions to the the prototype. They divide into basic extensions (which should be addressed before the system is used for non-toy applications), near future extensions (which may be included in near future releases of the prototype), and longer term extensions (ideas that may or may not ever be implemented).

4.1 Basic extensions

The following extensions to the prototype should be added before it is released for non-experimental use.

Near Complete SQL support: There are a number of SQL constructs that are not yet supported by the prototype. Often, the functionality to support the construct is present, but has not been "connected" with the syntax. Many more constructs will be supported, but some of the more idiosyncratic features may be deferred.

Concurrency control and recovery: Non-read-only applications require the ability to recover from a system failure. I plan to add a basic recovery strategy to the prototype using "incremental logging with deferred updates" as described in Korth and Silberchatz [6]. The prototype will use a time-stamp based protocol to prevent concurrent access conflicts – largely because time-stamping is more amenable to an "object oriented" view of data, as well as for loosely coupled distributed applications. Since the initial versions of the prototype will be restricted to single process access, the initial implementation may not provide full concurrency control, but will be designed to allow the addition of timestamping without requiring major restructuring of the code.

External data sources: The concept of a "table" will be implemented as a class that permits derived classes to define data accesses in arbitrary ways. Thus it will be possible for an applications programmer to define a class which, for example, parses and loads and dumps system administration configuration files (such as the unix `/etc/passwd` file or the various X configuration files, or the files of the linux `proc` file system) as SQL tables. An immediate internal application of this interface will be to add a persistent indexed relation class using the `dbm` Python indexed file interface.

Forms: With some minor extension the "forms" module of the prototype will allow applications programmers to develop their own input forms, as well as graphical reports. Eventually the Tk based interface should also be "cloned" to provide a version that uses the HTTP protocol to allow remote users to access data managed under the prototype using the World-Wide-Web – a conceptually straightforward exercise.

Improved applications programmer interface: Partially because I haven't decided what a good API should look like, the present API interface to the database engine is not yet suitable for external use. One or more applications programmer interface strategies will be added to the prototype to allow Python scripts to conveniently access and modify data managed under the prototype.

4.2 Near future extensions

A number of additional features will be added to the prototype in the near term. These include basic extensions to functionality, and experimental extensions.

Active Database Features: The prototype will implement transactions using a synchronous active database model (based on ideas described in [7] and Cacace et. al. [8]). This strategy will permit the database to check complex consistency constraints, to trigger other modifications automatically from user requested modifications, to define arbitrary "view updates," and to spawn external actions automatically from database interactions. As a simple example of an external action a new entry in a "customer" table may spawn electronic mail to a salesman telling the salesman to start harassing the new customer, or a new entry in the "reviews" table may spawn automatic mail to the author of a submitted journal paper, telling the author that the author's paper has been rejected (or possibly, accepted – but this is highly unlikely).

Relational Algebra: Primarily for educational use, and because it will be easy to add, the prototype will include a tk-based relational algebra interpreter. This component may be useful to database instructors with lots of students that can't seem to understand the relational algebra.

Multi-threaded multiple session support: The prototype will be extended to allow multiple concurrent access by making use of multi-threading and by completing the concurrency control and recover strategies outlined above. Eventually this extension will allow an instance of the prototype to function as a server which remote clients can access simultaneously.

4.3 Longer term extensions

Many possible extensions present themselves, below are some of the more compelling possibilities. Some of these stretch goals require significant thought and research, below are brief sketches of the concepts.

ODBC interface protocol: An ODBC driver for the prototype would allow other software to access data managed by the prototype without modification to any components. Thus, for example, Lotus Notes programs could connect to the prototype via the ODBC driver and extract data into a report. Hence a database published on a CDROM that included the prototype could be loaded onto a personal computer, and accessed using all sorts of standard software without the need for any special purpose code. If such an ODBC driver ever comes into being, the prototype will be a very compelling vehicle for database publication.

Distributed access: By folding the data access interface with the client/server access protocol it should be possible to make one instance of the prototype act as a client to several other remote server instances of the prototype – where the "servers" appear to be simple data sources to the client. Under this model the prototype could be used to build simple distributed databases.

Multi-database access: By interfacing API's for other database managers to Python, it should be possible to allow external databases managed (for example) by Oracle or Sybase to appear as data sources for the prototype. Using these interfaces the prototype could be used as a database "glue" tool which allows queries and accesses to span multiple autonomous databases.

5 Python as a Database

Let's forget about Gadfly for a moment and consider the more general possibilities of combining Python with a database manager. In particular, this section considers the possibility of defining a simple object oriented database system by embedding python structures within a relational database system.

A moments thought reveals that Python is "nothing more" than a relatively simple relational database with some useful C code tacked on. Figure 4 shows an incomplete entity relationship representation for the Python data model.

I have no really deep insights to offer here – this is more of a topic for discussion – but note that it would be a relatively simple exercise to embed all data from a Python process into any database manager that supported unbounded strings or binary large objects (BLOBs). Using such an archiving strategy together with the internal hooks already in place in Python it should be possible for a Python process to connect to a database and import modules, classes, instances and other objects from the database, modify them, and export them back to the database. This raises the possibility that Python could be combined with an arbitrary relational database manager to define an object oriented database system.

Analogous tools along these lines already exist for other languages. For instance Persistence is a package which combines $C++$ with database managers in such a manner. There is a great deal of excitement about Persistence at places such as AT&T Bell Laboratories – but my moles tell me that the actual use of Persistence is quite difficult. In my view, these difficulties derive from the peculiarities of $C++$ and from the fact that $C++$ is compiled,[4] and the greater elegance and simplicity of Python, as well as its interpreted nature, makes Python far more appropriate for embedding in a database.

---

[4] For example, a C++ object cannot be examined for its components except at compile time using complex precompilation tools. Thus Persistence requires many precompilation and code generation steps that would be unnecessary in an interpreted language such as Python.

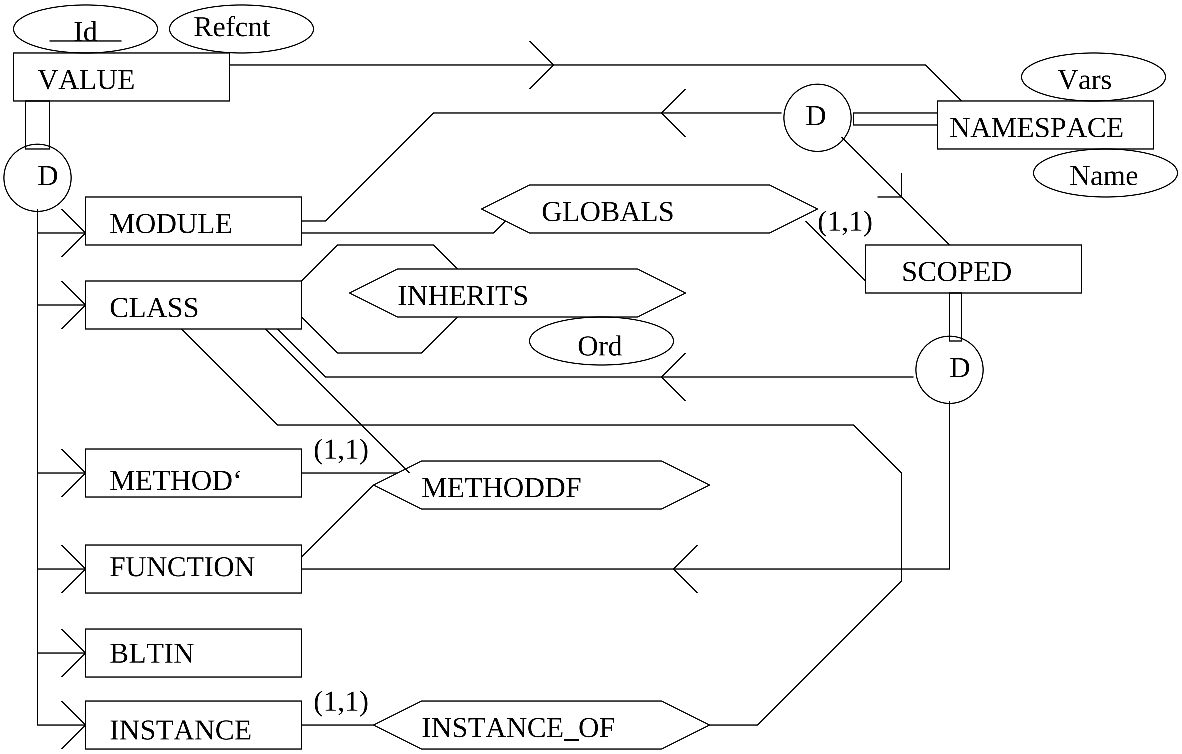


Figure 4: Python in an ER diagram.

There seems to be a great need for combining object oriented database systems while preserving the advantages of the standard relational approach. I believe that with some work Python could be used to address these needs.

6 Thoughts

I believe the Gadfly prototype (once it has filled out a bit) may be useful as a data management tool – for one, because it is fast enought to be useful. The embedded C extension `kjbuckets` provides the prototype with its speed, and were it not for `kjbuckets` the prototype would be far less compelling.

At a general level, I would draw from this observation that Python could benefit from a larger suite of abstract data types (ADTs) implemented in C. If more C implemented ADTs were available Python programmers could at once make their code faster, smaller, clearer, and more supportable by using them.

Among the ADTs I would like to see are

- Efficient order based mappings, perhaps using B-trees both implemented in memory and on the file system.
- A priority queue type.
- Matrix and other math support (perhaps via an interface to the standard CLAPACK math library).
- Types for efficient support of larger parsing problems and keyword matching – such as a trigram [9] based graph implementation where `G[".*thon.*"]` might retrieve a keyword list such as `["Python", "marathon", "thong"]`.

Other ADT ideas with suggested implementations can be found by cracking open any data structures book. Better C level aggregate conversion routines for existing Python types would also be useful.

With more ADTs Python would be an even better language for accomplishing Don Knuth's dream of "literate programming" and Python coding would be even more fun and rewarding.

References

[1] MicroSoft Corporation, *ODBC 2.0 Programmers Reference and SDK Guide.* MicroSoft Press, 1994.

[2] A. Watters, "Interpreting a reconstructed relational calculus," in *ACM SIGMOD International Conference*, pp. 367–377, 1993.

[3] R. Hull and J. Su, "Domain independence and the relational calculus," *Acta Informatica.* to appear.

[4] T. Imielinski and W. Lipski, "The relational model of data and cylindrical algebra," *Journal of Computer and Systems Sciences*, pp. 80–102, Feb. 1984.

[5] J. K. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley Professional Computing Series, 1994.

[6] H. Korth and A. Silberchatz, *Database Systems Concepts, second edition.* McGraw-Hill, 1991.

[7] A. Watters, "Synchronous logic for active databases," in *Information an Knowledge Management, CIKM-92*, pp. 568–576, 1992.

[8] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari, "Integrating object oriented data modelling with a rule based programming paradigm," in *SIGMOD, International Conference*, pp. 225–236, 1990.

[9] S. Adams and G. Popovici, "Information retrieval from a full text database using a trigram based index," in *Information an Knowledge Management, CIKM-92*, pp. 112–120, 1992.