Database Rules and Time: Some Proposed Extensions to the SQL Standard

Liam O'Neill B.Sc.

Submitted to Dublin City University
for the degree of
Master of Science (Computer Applications)

This research was carried out under the supervision of
Dr. A.F. Smeaton in the School of Computer Applications
at Dublin City University

February 1994

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science (Computer Applications) is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible] Date: 11/2/94

ACKNOWLEDGEMENTS

I would like to thank my academic supervisor, Dr. Alan Smeaton, for his expert advice and many kind words of encouragement. I am grateful to the Irish Aviation Authority who sponsored my work on this thesis and to colleagues and friends for their valuable assistance.

A very special thanks to my wife, Geraldine, for her understanding and support.

CONTENTS

ABSTRACT

Title Database Rules and Time :
Some Proposed Extensions to the SQL Standard

Author Liam O'Neill

The subject of this thesis is the incorporation of temporal semantics into database rules and how the resultant syntax might be reconciled with the evolving SQL standard. In particular, it explores time-driven rules and the time-relationship between triggering events and associated actions.

A review of the key research results in the area of database rules and the syntax developed for the major prototype implementations is conducted, and a working syntax , free of any limitations within the SQL standard, developed. Next, an operational definition is evolved through the application of this working syntax to two sample domains rich in 'temporal rules'. In each case a graphical representation of the domain is presented using an adapted object-oriented modelling technique followed by a mapping into the working temporal syntax.

Attention is then turned to the SQL-92 standard and its future successor SQL3. An assessment is made of their implications for the working syntax developed in the earlier chapters - with particular reference to the specification of time and the use of database triggers.

When an attempt was made to re-cast the working syntax into SQL, a satisfactory mapping, which succeeded in preserving the semantics of the original, could not be achieved. Support for time-based triggers; cyclic operations; delayed actions and rule lifetimes necessitated the development of appropriate modifications to the basic SQL3 draft syntax. The proposed extensions capture all of the semantics required for the specification of time-based rules.

The example applications indicated that an extended SQL-compliant language approach allied to a sound object-oriented modelling formalism had a broad applicability. Furthermore, it was apparent that the addition of a temporal dimension to rule actions was a key enabling factor in increasing their semantic power.

Chapter 1
Introduction

1.1 General

An interesting development in Database Management Systems in recent years has been the trend towards the migration of integrity maintenance logic out of traditional application code and into the database server itself. This has been achieved by allowing developers to program the server as well as specifying the user-interface and front-end logic.

The parallels with the introduction of Data Normalisation, where duplication and inconsistencies are resolved by way of the simple strategy of actually having only one copy of the data, are striking. The real benefits made possible by the process of what could be called 'Logic Normalisation' are already to be seen with commercial Database Management Systems, chiefly in reduced maintenance, greater flexibility and responsiveness and enhanced security. Indeed, the ability to provide central integrity control has made such servers a key enabling technology for the spread of client-server applications.

The factor which has made the database come alive and cease to be a passive data repository has been the ability to capture rules, the event/action pairs long the subject of AI research. Database management systems have thus become sensitised and, in broad terms, we will be concerned in this study, with the range of stimuli to which they might usefully react.

1.2 Subject of Thesis

The subject of this thesis is the incorporation of temporal semantics into database rules and how the resultant syntax might be reconciled with the evolving SQL standard. In particular, it explores time-driven rules and the time-relationship between triggering events and associated actions.

A number of sub-topics flow naturally from the above and the intention is to:

(i) Provide a suitable working syntax for exploring temporal rules.
(ii) Define their semantics by an appropriate means.
(iii) Attempt to provide, as a research aid, a formalism for their graphical representation.
(iv) Evaluate SQL as a medium for temporal rule specification.
(v) Document any required SQL extensions.

The strategy adopted is to develop a working syntax so as to avoid being bound by any limitations in the current SQL standard and the SQL3 draft, clarify the associated semantics by way of sample applications, and then to attempt a mapping into SQL, highlighting any extensions identified in the process. The implementation of this strategy is reflected in the chapter contents set out in the next section.

1.3 Structure of Thesis

In Chapter Two we establish our context, describing how the thesis relates to the broader field of knowledge (Database Management Systems) to which it belongs.

Chapter Three is a review of the key research results in the area of database rules and a study of the syntax developed for the major prototype implementations. We follow this up in Chapter Four by describing a working syntax based on the material covered in the previous chapter.

The focus in Chapter Five is on semantics. We develop our operational definition through the application of the working syntax to two sample domains which are rich in temporal 'rules'. In each case a graphical representation of the domain is presented using an adapted object-oriented modelling technique followed by a mapping into the working temporal syntax.

In Chapter Six we look at some of the highlights of the SQL-92 standard [MELT93] and its future successor SQL3 [EISA93]. We then go on to assess the implications for the working syntax developed in previous chapters - in particular, how this relates to the way in which the standards handle the specification of time and the use of database triggers. Up to SQL-92 the standard has continued to be based on the relational model. The next version, with the working title SQL3, will remain relational-centred but will add object oriented features. More significant, from our viewpoint, will be its introduction of a standard for database rules.

Taking the evolving SQL standard as a starting point we go on to attempt to map the semantics of the operational definition into SQL. The outcome of the attempted mapping is discussed and any required syntactic extensions detailed.

In the final chapter, we summarize our conclusions and suggest areas of further work.

Chapter 2
Database Management Systems

In this context-setting chapter, the emphasis will be upon database modelling concepts as this has most relevance to the focus of later chapters, but we will also touch on areas such as Distributed Databases. We will begin our discussion of Database Management Systems (DBMS) with a review of the relational model.

2.1 Relational-Centred DBMS

The relational model is now some 20 years old and in that time the early prototypes, System R and Ingres ([ASTR76], [STON76]) have given rise to the mature DBMS products widely used today. In this section we will look at this model in its original form, highlight its perceived shortcomings and discuss various proposals for overcoming them.

2.1.1 Relational Model Characteristics

A concise description of the relational model can be found in [CODD79]. It is emphasised that the algebraic operators are as intrinsic an element of the model as are the structures and that there is a close relationship with first-order predicate logic.

A database is said to be fully relational if it supports:

(1) the structural aspects of the relational model;

(2) the insert-update delete rules;

(3) a data sub-language at least as powerful as the

relational algebra, even if all facilities the language may have for iterative loops and recursion were deleted from the language.

Structure

Domain -	a pool of values of similar type e.g. integers, on which all field values are defined. The domains of the relational model are simple.
Relation -	the cartesian product produced from n domains produces a set of tuples comprising of all possible combinations from the n domains. A relation defined on these n domains is thus a subset of the cartesian product. The number of domains, not necessarily distinct, defines the degree of the relation.
Attribute -	one of the characteristics of the entity which the relation represents. The values are drawn from the domains on which the attributes are defined. Codd looks on attributes as different uses of the underlying domains.
Tuple -	a set of attribute/value pairs with the values drawn from the domains on which the attributes are defined.

Using these elements Codd summarises the term relation as a set of tuples each containing the same group of attributes. Because the attribute values are atomic any relation can be represented as a table.

Four rules govern the properties of a relation:

(1) There are no duplicate tuples.

(2) There is no ordering significance amongst the tuples of a relation.

(3) There is no ordering significance amongst the attributes of a tuple.

(4) All attribute values are drawn from simple domains.

A relational database, then, is a time-varying collection of data which presents itself as a group of tables where values are atomic in nature. The property of closure holds under the operators of the relational algebra (discussed later in this section) - operations on relations produce other relations.

Data models generally have three dimensions - structure, integrity constraints and manipulation primitives.

Two rules constrain operations on base tables:

(1) **The Entity Integrity Rule**
No primary key value of a base relation can be wholly or partially null. This rule is needed to preserve the semantic link between an individual tuple and its corresponding real-world entity set instance.

(2) **The Referential Integrity Rule**
If an attribute of a base relation is a foreign key matching the primary key of another base relation, then its value must be equal to some value taken by that primary key or be null. In practical terms this

^Crule states for instance that an employee can only be assigned to a valid department or else remain unassigned e.g. an induction training period may be needed to decide on where to send the employee during which time the department is null.

Based on the above definitions the model can be viewed as consisting of the following:

(1) a collection of time-varying relations;

(2) the two integrity rules. These are often referred to as the insert-update-delete rules;

(3) the relational algebra.

Codd defined a set of operators for manipulating tables which he called the Relational Algebra [CODD70], [CODD79]. As relations are sets the operators UNION, INTERSECTION, DIFFERENCE and CARTESIAN PRODUCT can be used. In the case of relations as distinct from normal sets the UNION operator is restricted to union-compatible relations i.e. their attributes must correspond in number and type (domain).

The other operators are THETA-SELECT (OR RESTRICT) which produces a subset of tuples. It picks out specific rows based on a restriction predicate in the query e.g.

R	(A	B	C)	
	p	1	2	
*	p	2	1	*
	q	1	2	
	r	2	5	
	r	2	3	

R [B.C]

(A	B	C)
p	2	1

[CODD79]

PROJECTION - Formally defined as R [A1, A2..An] is the relation made up of the specified attributes of R after the elimination of any duplicate rows e.g.

R	(A	B	C)
	p	1	2
	p	2	1
	q	1	2
	r	2	5
	r	2	3

R	[B]	(B)
		1
		2

[CODD79]

THETA-JOIN - this is the concatenation of tables based on the relationship specified for the linking attributes (of compatible domain). If the relationship specified is that of equality then this operator is referred to as EQUI-JOIN.

```
                *--------------*
R      (A    B     C)        S      (D    E)
        p    1     2                 2    u
        p    2     1                 3    v
        q    1     2                 4    u
        r    2     5
        r    3     3
```

(EQUI-JOIN Case)

R [C = D]S (A B C D E)

p	1	2	2	u
q	1	2	2	u
r	3	3	3	v

(GENERAL Case)

R [C . D] S

(A	B	C	D	E)
r	3	3	2	u
r	2	5	2	u
r	2	5	3	v
r	2	5	4	u

[CODD79]

NATURAL JOIN - this is similar to EQUI-JOIN except that one of the linking columns is removed e.g. column D in the EQUI-JOIN example above.

DIVIDE - Codd describes this as the algebraic counterpart of the universal quantifier and defines it as follows:

Taking 2 relations R (A, B) and S (C) with B and C domain compatible then R [B DIVIDE C]S is the greatest subset of R[A] such that its Cartesian product with S[C] is in R.

Example

R

(A	B)
p	1
p	2
p	3
q	1
r	1
r	3

S

(C)
1
3

R [B DIVIDE C]S

(A)
p
r

[CODD79]

Note:
The Cartesian product q1, q3 not being a subset of R does not qualify q for inclusion in the result of the DIVIDE operation.

2.1.2 Evaluation of the Basic Relational Model

The main achievement of the relational model was to provide a modelling methodology that offered data independence to both the designer and the user. Users did not need to know and designers could initially ignore the implementation details of storage structures and access paths required for acceptable performance. To simplify is to remove unnecessay detail and depending on the context of the application significant loss of semantic richness could result. This is most noted in cases where mapping onto the tables of the relational model is not entirely natural.

> "One problem inherent in modelling any subset of the real world is the difference between the human's perception of the enterprise and the computer's need to organise the structures in a particular way for efficient storage and performance. This gives rise to three database modelling levels that reflect the user's conceptual model, the machine's physical model and the mapping from one to the other".
>
> [PECK88]

An emphasis on retaining more of the human perception tied up in the conceptual level was the driving force behind research on other data models such as that of Chen [CHEN76] and the functional [SHIP81], semantic [HAMM81], object-oriented ([DKIM90], [COPE84], [BANE87] and others) and the extended relational [STON86] approaches.

One of the main limitations of the relational model which such later work addressed is the fact that a user needs to be aware of the foreign keys which implicitly define the relationships between entities and to use this knowledge to make the connections apparent. By contrast in, for example, the E-R model [CHEN76] the connections are example, the E-R model [CHEN76] the connections are explicitly defined within the model itself and no such semantic pre-processing is called for.

> "Current database systems are primarily an effort to implement an abstract data type over the memory of a machine rather than to support easy and natural modeling of real-world enterprises. They hide the complexity of file systems and indexing techniques and provide a degree of physical data independence. The next generation of database systems will be knowledge management systems with more support for data semantics, inferencing and general purpose programming."
>
> [PECK88]

Copeland and Maier [COPE84] highlighted specific shortcomings with the relational model:

(1) **Lack of type definition facilities**
Types such as integers, date and money are standard in current systems. However, it is not possible to enhance the set of operators e.g. 'day of week' already defined for these types or indeed to declare new types. By definition, values must be drawn from atomic domains which excludes the use of structured data types e.g. arrays of simple data types.

(2) **Structural limitations**
The primitives of the relational model cannot adequately capture real-world objects. The framework provided by the tables of the model offer simplicity but impose rigidity - other models can cope much better with such practical requirements as the facility to store an extra middle name in some records but not in others.

(3) **Modelling power**
Simplification of the target domain leads to compromise. Looking at the 'middle name' example

again, two people who are in reality distinguishable by full name can become 'identical' in the database.

Logical pointers are required to show relationships between entities - e.g. dept-name could be used to link employee records with their corresponding department record. The designer has to find or artificially create this logical pointer and it is consequently vulnerable to update anomalies e.g. if it is decided to rename the 'Personnel' department to 'Human Resources'.

(4) **Lack of a temporal dimension**
Although manual systems are based on historical data the relational model ignores this aspect of applications. Deletion was originated to allow the re-use of scarce computer resources but this is no longer a major concern. The extra cost can be justified by the importance that users attach to having access to historical views of the enterprise and the benefit of built-in error recovery offered.

On the question of semantic weakness, Codd [CODD79] has highlighted the fact that his original model was not without semantic features and instances domains, keys and the notion of functional dependence as examples. He recognises, however, that a greater concern for problems of the external level is required and that those cannot be solved by structural approaches alone.

> "Structure without corresponding operators or inferencing techniques is rather like anatomy without physiology. Some investigators have retained clear links with the relational model and have therefore benefitted from inheriting the operators of this model - just as the relational model retained clear links with predicate logic and can therefore inherit its inferencing techniques".
>
> [CODD76]

2.1.3 Extended Relational Models

In response to the perceived shortcoming of the relation model Codd presented an enhanced model which added the concepts of relationships and integrity rules [CODD79]. He referred to this new model as RM/T (sometimes referred to as the Tasmanian model). In later versions this model has become known as RM2.

His original model handled relationships in a value based way, through joins expressed in the data manipulation language. Through the process of normalisation entities are factored to avoid redundancy and ensure consistency. A side-effect is that what the user sees as a single entity e.g. the data about say a book is in fact stored in several tables, the relationships or linkages between which are not obvious to the user.

In RM/T relationship types are defined in addition to regular entities. Codd introduced the ideas of E-relations and P-relations. There is an E-relation for each entity type which holds the entity identifiers for each instance of that type. This use of an internal identifier has been followed in the object oriented model ([ZDON90],[KIMW90]). P-relations then define the properties (attributes) for each entity type and hold the values for each instance.

Example

E-RELATION

Book	Book-ID

P-RELATIONS

AUTH	Book-ID	Author
TITL	Book-ID	Title
PUBL	Book-ID	Publisher

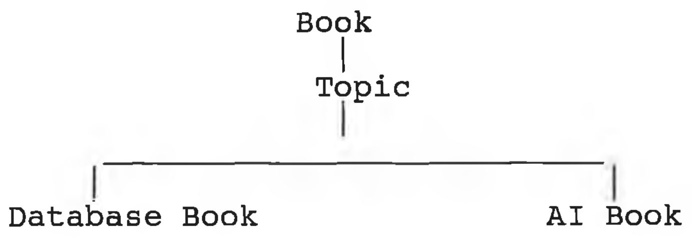
Moving on to the specification of relationships RM/T introduces associative entity types for many-to-many relationships and designative entity types for one-to-many relationships.

As an example of the latter Peckman and Maryanski [PECK88] show that by adding a 'DESIGNATING' clause to the definition of the E-relation Book the relationship between a writer and his book is captured.

```
CREATE E-RELATION Book
DESIGNATING (Author via Writer-ID)
```

Here the link to AUTHOR is set down at data definition time leading to a more semantically explicit schema. In addition to this, built in integrity rules are proposed.

RM/T also supports IS-A relationships through E-relation definition



For example, the following syntax defines a new subtype 'AI-Book' value-based on TOPIC:

```
CREATE E-RELATION AI-Book
SUBTYPE OF BOOK PER
CATEGORY TOPIC
```

The trend towards extending the relational model is driven by the sensible notion of achieving greater semantic modelling power without throwing away what has been achieved so far with the original model. In short, Codd was saying that a new model was not necessary.

The argument for the extended relational approach can be summarised as follows:

- existing benefits are retained,
- it is a natural evolution of what users already know and like about a robust existing model,
- there is no need to learn a totally new DML.

2.2 Alternative DBMS Models

In this section we will examine a range of non-relational models on which a DBMS can be based. As will be seen, these offer greater expressive power and functionality but introduce a greater level of complexity.

2.2.1 Semantic Data Models

Peckman and Maryanski [PECK88] provide a useful review of research on the semantic data model highlighting some common characteristics and providing an evaluation of future prospects for the various proposals put forward.

Their basis for comparison between semantic models focuses on the following observation.

"Every semantic model has <u>objects</u> (or entities), <u>relationships</u> (functional or relational), <u>dynamic properties</u> and a means for handling <u>integrity constraints</u>. Relationships can be characterised by the abstractions they are capable of representing and the means by which they do so. Dynamic properties can range from the simple specification of insertion and deletion constraints to the modelling of operations and transactions. Constraints can be collected from the user and represented and/or automatically implied by the semantics of the model's relationships. Both the level and mechanisms of information representation are used to characterise and compare models."

[PECK88]

Elaborating on these key concepts they provide an eight-point framework of fundamental semantic data modelling characteristics.

(1) Representation of Unstructured Objects
(2) Relationship Representation
(3) Standard Abstractions Present
(4) Networks or Hierarchies of Relationships
(5) Derivation/Inheritance
(6) Insertion, Deletion and Modification Constraints

(7) Degree of Expression of Relationship Semantics
(8) Dynamic Modelling

An interesting example of a semantic database proposal is SDM [HAMM81] which illustrates in concrete terms the application of the above concepts.

2.2.2 Deductive Databases

Although primarily a field of academic research the topic has yielded important practical benefits to commercial RDBMS (- null and missing values, integrity constraints and optimization). Their strengths are likely to lie in the implementation of very large expert systems where the database management requires DBMS level functionality.

Also known as logic or expert databases these hold data in two forms:

(1) Explicitly stored data.

(2) Data deriveable from the above and defined as logic procedures.

Although a promising concept, performance, as in the case of semantic databases, continues to be a problem. Such difficulties are to be expected. For instance it is not possible to have the power of recursion without the overhead of potentially redundant processing. Many algorithms have been proposed to alleviate this problem without major success. However, a hardware solution is emerging in the form of massively parallel systems.

A comprehensive introduction to deductive databases is given by Gallaire, Minker and Nicolas [GALL84] who describe the important role of mathematical logic in query languages, handling null values, integrity constraints, query optimization and database design.

2.2.3 Semantic Query Optimisation

This is a spinoff of deductive database research. The idea being to use rules to help in optimization rather than working on the query itself. It is analogous to the way 'old hands' in any organisation are able to say things like 'you won't find it there!' and, as might be expected, the basic approach to date has been heuristic in nature.

Although the idea looks simple at first sight it has proved complex to implement efficiently. However, the advantages to be gained are striking as the following examples based on Shenoy and Ozsoyoglu [SHEN89] illustrate.

```
     Schema
          Employee (Ssn, Name, Job, Sal)
          -    cardinality    =    36000
          -    secondary index on Job

     Constraint 1
          "Only managers make over £30,000

     Query 1
          select employees with salary >= £40,000

          Note:     No index on employee.sal.
                    Index on employee.job.
```

Strategy

Regular RDBMS:-	Table scan of Employee
Semantic:-	Constraint condition indicates the best access path is the secondary index on employee.job to pick out

'manager' tuples and then scan this subset for salary condition. This is an example of semantic index introduction.

As the number of managers is a small proportion of total employees the response time for such a query would be significantly improved. It could be argued, in the case of such a trivial query, that a simple secondary index on employee.sal would have much the same effect but what if a second constraint was added along the following lines:

Constraint 2

Only Mr. Brauer can make over £60,000

The semantic query optimizer could now use this constraint to zero in on this tuple in response to a query requesting employees earning, say, £70,000.

Shenoy and Ozoyoglu summarise this mechanism in the following terms:

> "...dynamic and heuristic interaction of three entities - schema, semantics and query..."
>
> [SHEN89]

As expected the savings achieved increase with the size of the data set as the optimization cost is fairly constant. A recent study [MCMA92] evaluated the potential of semantic query optimization on a public health database using ORACLE and concluded that significant reductions in query times were achievable. It has been argued [SHEK88] that this optimization strategy will find its niche in the field of Very Large Databases.

2.3 Distributed DBMS

"To someone with a new hammer everything looks like a nail".

Anon

One of the lesser known innovations of the Saatchi and Saatchi advertising empire was the term 'globalisation'. They envisaged all products from hamburgers to consultancy services being marketed McDonald's style - companies should think globally not just multinationally.

This sort of evolution has profound implications for database systems. A global enterprise would naturally hold Customer databases in each of their local centres e.g. Hong Kong, Milan and Sydney with most day-to-day access being confined to staff employed in these centres. However, for control and planning purposes, it will also be necessary to interrogate several of the databases as a unit to get, say, an age profile of the Customer base. One feature which a Distributed Database Management System (DDBMS) must support therefore, is location transparency. Of course, a distributed database could equally be implemented as part of an Office Information System (OIS) on one floor of an office block. Stonebraker [STON88] has proposed that meetings could be scheduled by intersecting the diaries on individual's workstations if such personal databases formed the nodes of a 'local area' distributed database.

Normal 'economies of scale' do not apply to computer hardware. Upgrading a mainframe is now seen as prohibitively expensive when there is an option of offloading onto cheap desktop machines using the client-server approach. However, client-server only shares the processing load, leaving the DBMS still resident on a backend mini or mainframe. Using distributed database management as an enabling technology, work on spreading the database itself onto workstations is well advanced e.g. ORACLE on nCUBE.

Robert Epstein, who helped build the relational database prototype 'Ingres' at Berkeley in the mid 1970's recently made the striking observation that 99% of the world's data is held outside of relational databases and that in ten year's time this figure could be down to 90%. [EPST90]. Stonebraker sees an opportunity here for heterogeneous DDBMS. Users should be able to work with their organisation's legacy and new databases through some generic interface without having to be aware of whether they are connecting to an Rdb or DB2 database or indeed both at the same time.

Moving on to specifics, even an outline functional specification for a DDBMS begins to reveal the practical issues that need to be resolved and reconciled:

(1) Location Transparency:
Users should not have to supply location specific information e.g. a node name, in queries against remote data.

(2) Performance Transparency:
Performance should be independent of where the user chooses to submit the query. This implies the existence of some form of global optimization which takes line speeds, processor speed, I/O speed into consideration.

(3) Copy Transparency:
It should be possible to distribute copies of data to all sites if required - this is to allow continued service during site failure.

(4) Transaction Transparency:

The 'all or nothing' nature of transactions should hold irrespective of whether a multi-site or single site update is involved.

(5) Fragment Transparency:
For performance reasons, a single table may be distributed across several machines e.g. the ORACLE architecture referred to previously. This distribution scheme turns what would normally be a heavy query such as a table scan on a serial machine into an ideal parallel processing task. A user should be unaware of any fragmentation of the target query object.

(6) Schema Change Transparency:
In a traditional DBMS a user only has to change one catalog to modify the schema. This level of simplicity must be retained in the distributed case.

(7) Local DBMS Transparency:
The distributed DBMS should not be affected in any way by the nature of the local data managers at the individual nodes.

Much progress has been made in the areas summarized above and the emergence of mature DDBMS is on the horizon. However, a question mark still hangs over the degree of uptake of the offerings from the major vendors. My own instinct is that the spin-off role of this technology in taking advantage of loosely coupled multiprocessor architectures at local sites for high availability and performance will be of equal importance to the primary objectives in relation to geographically dispersed databases.

2.4 Object-Oriented DBMS

A rule of thumb in coping with complexity is to isolate it into compartments which at least look simpler from the outside. Object-oriented programming systems have successfully used this strategy to achieve significant real-world modelling power - the challenge now is to make these objects persistent in an efficient and commercially viable manner.

2.4.1 Background

An object-oriented database is a data repository where complex objects are stored directly i.e. the physical data model mirrors the logical schema. This implies that the constructs and concepts of Object Orientated Programing Systems (OOPS) are carried forward into object-oriented database technology so a review of these characteristics is appropriate.

An OBJECT is basically a complex variable but it can have a CLASS and a STATE as well as the usual TYPE and VALUE. Skarra and Zdonik [SKAR87] discuss the problem of maintaining consistency between a set of persistent objects and a set of type definitions that can change. They argue that this can be resolved by the use of version control on the types and the definition of related error handlers which are version specific i.e. designers are given the means to define the correspondence between different type versions.

Relational database designs are full of surrogate keys (Social Security Number, Payroll Number, Personnel File Number can all be used to identifier a specific employee) which are used to bind and track an entity across the database schema. The weakness of this approach is that it tends to confuse data values with identity. In an Object-Oriented Database (OODB) this mechanism is no longer necessary as objects are assigned unique identifiers of which the user is never aware ([KHOS86,MAIE87]). Merging of OODBs naturally raises the issue of identifier conflict - the simple procedure of assigning a new set of identifiers to the imported data is just one possible work-around.

Dayal et al [DAYA87] argue that the requirements for modelling complex objects can be met by minor enhancements to the DAPLEX data language [SHIP81] which supports generic operations on entities, relationships between entities and entity and relationship level constraints.

CLASSES can also be simple (primitive) or complex. The notion of a CLASS extends the idea of a datatype to include the behaviour of any object defined on that CLASS. The behaviour is captured as a set of operators called METHODS which can be changed at will.

The purpose of a class is "so that each object need not carry around its own methods ...."[MAIE87]. Once defined in this way a method is applied to an object by way of a MESSAGE which is like a procedure call which elicits a SIGNAL from the object. In the introduction to a paper describing the object-oriented data model O2, [LECL88], Lecluse distinguishes between the terms 'type' and 'class'. The intensional notion 'type' provides a blueprint (the 'class defining object' of Maier and Stein [MAIE87]). The extensional notion 'class' describes the set of all objects which can conform to the 'type' at a given time.

Classes can have subclasses which inherit their properties. Cardelli [CARD88] describes this as a biology and taxonomy approach but argues that multiple inheritance is necessary to describe real world class hierarchies effectively. Implementing this property is, however, much more difficult than straightforward single inheritance. Snyder [SNYD86] examined the relationship between inheritance and encapsulation developing a set of requirements for full support of encapsulation with inheritance. The problem he studied was the inherent conflict between the concepts of strong encapsulation and the information sharing between objects in a class hierarchy. He looks upon inheritance as a 'contract between a class and its children' - like any contract it limits the scope of actions, in particular, changes to a class.

2.4.2 The Zdonik/Maier Threshold and Reference Models

It can come as something of a surprise to find the comedian Chevy Chase quoted in a paper, by Andrews and Harris [ANDR87], on the question of impedance mismatch in object-oriented systems, but the phrase "You're both right, it's a dessert topping **and** a floor wax!" somehow says it all. There is still a lot of confusion as to what objects are, due in part to the way that the ideas associated with objects have been developed by workers in the diverse fields of programming languages, artificial intelligence and, more recently, databases [KIM90].

In [ZDON90] Zdonik and Maier provide a useful framework against which putative OODBMS can be evaluated - the 'Codd's Laws' of the object-oriented database world. They first propose a 'Threshold Model' which is a set of minimal requirements that every ODDBMS must have. Building on this model they set down the capabilities of a 'Reference Model' which is a yardstick for commercially acceptable systems.

The Threshold Model

(i) An ODDBMS must provide database functionality.

(ii) Objects must have a unique and permanent identity independent of value.

(iii) It must provide encapsulation and all abstract objects should be defined on this basis.

(iv) It must support objects with complex state i.e. the full state of an object is made up of it's local state plus that due to inter-object references.

The Reference Model
The Reference Model builds on the Threshold Model's foundations of object identity, encapsulation, complex objects and standard database management utilities adding the following features:

(i) **Structural representation for objects**
Rather than being limited to holding the state of an object in a simple data structure it should be possible to build up a compound representation for those defined in terms of component objects (i.e. defined by nested application of constructors) which is somewhat analogous to the 'nested-dot' concept used in GEM and Postgres ([ZANI83], [ROWE87])

(ii) **Persistence by reachability**
All objects, irrespective of their type, should be permanently reachable through a distinguished root object.

(iii) **Typing of objects and variables**
Every object knows its type and every variable and argument in a method definition has a type. A variable has effectively two types - it's declared type and it's immediate type (the type of the object that is it's current value). Static type checking at

run-time must ensure that a variable's immediate type is a subtype of it's declared type. Messages must also be checked to ensure that they are meaningful to the target object.

(iv) **The existence of three hierarchies**
The model must support the following hierarchies:
- a specification hierarchy of types
- an implementation hierarchy of representations and methods
- a classification hierarchy of actual groups of objects

(v) **Polymorphism**
The reaction of an object to a given message depends on how the corresponding method is implemented. This allows the same message to trigger off a user-definable variety of object behaviours. This feature must be provided.

(vi) **Collections**
Built-in types must be supported for aggregate objects such as sets, lists and arrays. These greatly enhance semantic modelling power.

(vii) **Name Spaces**
The model proposes database variables which can be assigned a value in a database bind and then persist for use in subsequent sessions. In relational systems the only names that persist are base tables and views. The presence of database variables supports a richer set of query targets.

(viii) **Queries and Indexes**
The query language must be high level and amenable to optimization. In relational systems queries

consist of well defined operators working on relations which are simple tabular structures. In OODBMS queries may involve both newly defined operators and abstract structures resulting in a whole new algebra. Apart from a well designed object-orientated algebra two other factors are important. For efficient optimization the query optimizer must be 'trusted' to peel back the layers built up by the process of encapsulation to become aware of any storage structure details which it could find useful. It must also be possible to create indexes on collections.

(ix) **Relationships**
Single valued, multi-valued and symmetric named relationships must be fully supported. This emphasises the fundamental importance of relationships in data modelling.

(x) **Versions**
The idea of a version set must be supported to hold the history of an object. Previous versions can be retrieved by moving a pointer over the members of the version set.

2.4.3 Problems Facing OODBMS

Object-oriented database systems still lag far behind RDBMS when it comes to implementation issues. Work aimed at making up the shortfall is summarized below.

Query optimization
In the absence of a successful 'object algebra' and a simple DML, direct use of defined methods presents another level of indirection to the problem of query optimization. People such as Manola and Dayal [MANO86] have been addressing this issue with proposals for an 'object algebra'. A hybrid approach has been suggested by Fishman

[FISH87] based on the IRIS project. He proposes an object-oriented extension to SQL 'OSQL' together with a relational storage manager, an achitecture which permits the use of the standard relational algebra and query optimization procedures against an object-oriented schema.

The query optimization problems stem from the notion of encapsulation which is fundamental to the object-oriented model - the optimizer needs to know about the detail masked by the messages. In light of this the necessity for a 'trusted system component' is now accepted which is allowed to see the internal workings of objects.

Storage Management
There is still a long way to go towards a mature storage technology for OODBs. This is further complicated in the distributed case where the object's methods must also be replicated/updated across all sites to ensure consistent local access [LYNG84].

Exploitation of Emerging Technologies

The use of parallel processing of queries in set-oriented systems is now well established. However, it remains to be seen if OODBs can successfully exploit this technology - splitting method evaluation between processors may not justify the scheduling overhead involved [ZDON90]. The use of optical storage technology on the other hand will lend itself to the idea of retaining all versions of objects (no deletion semantics), an important facility in CAD environment.

Finally, OODBs face a problem of cultural acceptance in certain quarters. Stonebraker recalls the almost religious fervour of the opposing camps in the COBOL/CODASYL Vs RELATIONAL war of minds which culminated in the 'Great Debate' at the ACM/SIGMOD Conference in Ann Arbor, Michigan in 1975 -

"The debate was significant in that it highlighted once again that neither camp could talk in terms the other could understand".

[STON88]

Stonebraker himself is playing a leading role on the relational side this time around and is largely determining a strategy of moving away from a purist stance by making some object-oriented beliefs part of a born-again relational dogma.

2.5 Temporal Databases

"The Machiguenga verb system was complicated and misleading, among other reasons because it readily mixed up past and present. Just as the word for 'many' - tobaiti - was used to express any quantity above four, 'now' also included at least today and yesterday, and the present tense of verbs was frequently used to recount events in the recent past. It was as though to them only the future was something clearly defined."

Mario Vargas Llosa, *The Storyteller*.

Temporal database systems are becoming increasingly important as a means of handling versioning of data in application areas such as Computer Aided Design. However, the most striking feature proposed for temporal database systems is the idea of update as succession rather than replacement. Nothing is deleted but rather migrates to less volatile storage - physical memory to magnetic disk to optical disk. This is yet another example of hardware-driven innovation in database technology. These database systems are sometimes called historical databases because recorded data is never deleted but is timestamped with creation/deletion times [CLIF83]. The timestamping enables these databases to handle queries like:

Has Jims salary ever risen?
Did Richard work in sales last year?
When was Paul hired?
Have Jim and Paul ever earned the same salary?
Will the average salary in Finance surpass X this year?

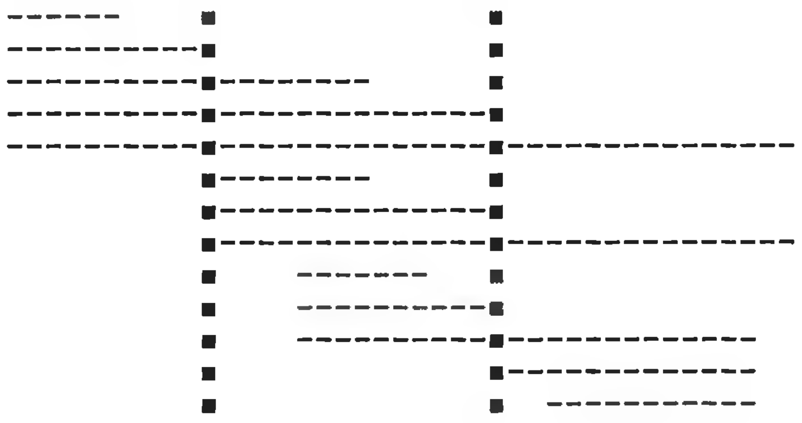
Implementation

Implementing temporal database features using timestamps in a relational database can be cumbersome as the folowing example illustrates:

EMP	Sal	Start	End
John	10K	D2	D8
John	11K	D8	D12
John	12K	D12	NULL
Mary	15K	D3	D6
Mary	15K	D8	D11
Mary	15K	D11	D15

Now, assume that we want to record the fact that John was not employed during [D4,D6] we find that this is not recorded explicitly, so we need to insert two more tuples.

On top of this we have the fact that two time intervals can be placed together in thirteen distinct ways which must all be handled in some consistent way:



The result is that it is not always obvious how to handle a query such as "retrieve all salaries during the interval [D1,D2]".

Copeland and Maier [COPE84] describe extensions to Smalltalk-80 to support temporal concepts. Data elements are set rather than single valued with the binding between an element name and its associated value indexed by time.

For example, using their notation, E!Salary@T would represent an employee's salary when the database was in the state current at time T. More specifically, the link between an attribute and a given value begins at the transaction time for the value, and ends when a new value/transaction time pair becomes current.

Stonebraker [STON87a] outlines the design of the Postgres storage system including time management features. The Postgres DML 'postquel' provides a simple syntax for queries on historical data:

```
retrieve  (Emp.Salary)  using    EMP[T]
where           Emp.Name  =    "Mike"
```

The state of the EMP relation at time T is the scope of the query target. Although times are held as 32 bit unsigned integers built-in conversion functions allow T to be specified in a range of standard formats.

The Postgres storage system is designed to facilitate versioning by storing an additional 8 fields in each record. They store details of the lifetime of a record from the commit time of the transaction which created it to the commit time of the transaction which superceded it. These values can be used for efficient validity checking during query execution. The algorithm can be extended without difficulty to deal with queries that request records valid in the interval $[T_1, T_2]$ rather than valid at time T.

Although Stonebraker does not explicitly include the catalog tables in this discussion, it would be useful to be able, for instance, to plot cardinalities against time over various phases in the lifetime of a database. Used in combination with alerters (rules whose actions are messages rather than database updates) runaway table growth could then be automatically highlighted for the attention of the DBA.

Typical of more recent research is the paper-based work of Jensen ([JENS91], [JENS92]) which, although without an underlying theoretical formalism, suggests a way forward. The approach proposed is to take the relational model and extend it to handle temporal information calling it DM/T (data model with time) in order to support the notion of a transaction taking time to execute instead of being an atomic event. In order to do this new relational operators (UNIT, FOLD, UNFOLD, WHEN, etc) are proposed and described.

Chapter 3
Rules in Database Systems

In this chapter we review the key research results relating to the incorporation of rules into database systems, examine the syntax and semantics of the major prototype implementations, and finish up with a look at some of the implementation issues arising.

3.1 Background

It is important right from the start to clear up any confusion which might exist as to the difference between an expert system and a database management system that can support rules.

As stated in [STON83], rule systems are nothing new in Artificial Intelligence where they typically take the form of a set of first order logic formulae. Stonebraker emphases the 'inference engine' role of an expert system's data manager i.e. its function is to see what rules can fire at any given time and then proceed to process them.

In a DBMS the emphasis has traditionally been exclusively on representation by pure data. The data manager's function is to apply a search logic in response to data requests expressed in a declarative language. Rules get fired, not by an 'inference engine' but by database queries (in the broad sense which includes inserts, updates and deletions) against the database. What we are discussing is a mechanism which provides guaranteed consistency between rules and data [STON92].

Using rules in a DBMS holds the promise of turning a database from a passive data repository into something active.

Silberschatz [SILB91] lists triggering, data mining and deduction as potential features. To be useful these should be capable of supporting complex triggering of actions on events in a controlled fashion. In addition to such **imperative** rules he highlights the equally significant potential of what he refers to as **declarative** rules ("if A is true then B is true") which open up the possibility of storing information without specific data. Indeed, he highlights the handling of large numbers of such rules as a major challenge for Next Generation Database Systems.

From a broader perspective these two classes of rules should be viewed as part of a comprehensive integrity maintenance system which is emerging as a standard requirement for DBMS - data alone is not enough.

As to implementation, he makes three interesting observations. Firstly, the rule processing cannot be delegated to a separate sub-system such as an expert system shell. These use a memory based approach which will not be an option with large systems. In general terms, such implementation considerations are tending to force the rule manager and data manager roles together which is not a bad thing as it is consistent with the goal of integration of knowledge and data. Further, this also ties in with the current trend towards 'normalising' business logic into the database server and out of the application code where a given 'rule' can have multiple and potentially inconsistent occurrences. Stonebraker [STON83] points out that attaching an inference engine to a data manager leads to the DBMS becoming not only much larger and more complex, but one which is attempting to reconcile two essentially different sub-components. He proposed the more elegant approach of extending the functionality of the data manager to cater for rules.

Secondly, he mentions the problem of maintaining the consistency of a rule set as new rules are added. The possible de-stabilising effects need to be addressed. Returning to the metaphor of normalising application logic this concern is understandable in terms of the potential 'chaos' which could result if programmers were allowed to fling new logic into an application without re-testing for overall consistency off-line. Silberschatz foresees an equivalent requirement for change control in light of the extensive rule-sets likely to become commonplace and their critical role in future systems. Thirdly, rather than throwing away the obvious benefits of declarative query languages, such as SQL, he favours extending the concept as a language strategy for next-generation systems.

By way of illustrating the challenge facing the prototype builders Widom and Finkelstein [WIDO89] list some of the major questions that needed to be resolved:

> "What causes a rule to be triggered? Is it a database state, a transition from one state to another, either, both?
>
> If rules can be triggered by state transitions, what exactly constitutes a transition? An operation on a single tuple? A set-oriented database update? A transaction?
>
> When are rules executed? At any time? Only after certain operations? Only at transaction boundaries?
>
> What happens if several rules are triggered at the same time? Are all rules executed? If so, is there an order? Is only one rule executed? If so, how is it chosen?
>
> What happens if execution of a rule causes another rule to trigger? How does the new rule interact with the other triggered rules? Can a rule trigger itself?
>
> If rules are not always executed as soon as they are triggered, what environment is used when a rule is finally executed?
>
> If several rules are triggered simultaneously, what happens if execution of one rule's action negates another rule's condition? "
>
> [WIDO89]

3.2 Rule Applications

The arrival of the Client-Server architecture has led to an explosion in end-user computing. The problem is that the responsibility for the maintenance of integrity which was previously enforced by programmers in their application code cannot simply be passed on to the end-users. Consequently, integrity checking has had to be provided more centrally, specifically by greatly enhancing the degree to which it can be programmed into the database server itself.

Aside from the support of simple referential constraints a whole range of general integrity constraints is becoming commonplace such as triggers, assertions and alerters. The obvious advantage of such facilities over simply embedding constraints in application code is in flexibility - the rules only need to be specified in one place and can be adjusted overnight, as it were, in response to changing needs. This will free up staff resources currently tied up in maintenance programming.

A frequent requirement in commercial applications is the provision of running aggregates e.g. the year-to-date expenditure from a budget subhead. Generally two options are available. The first is to put in end of day routines which tot up the necessary balances and store them in some sort of summary table against which queries can be run. With smaller databases it may be acceptable to allow users to run such queries interactively against the raw data and display the results. Both approaches have their draw-backs.

Rules can be used to set up triggers that make it easy to clock up such aggregates incrementally during the normal processing of database queries. The following example, using the syntax of the Ingres Knowledge Management Extension [KMAN91] keeps a running total of the number of employees that report to each manager in the context of the following schema:

```
Manager(name,dept,no_of_employees)
Employee(ename,mgr,age,salary)

AFTER INSERT,UPDATE (mgr) of Employee
     EXECUTE PROCEDURE p_check_mgr
          (mname = NEW.mgr)
PROCEDURE p_check_mgr (mname char(20)) AS
     BEGIN
          UPDATE Manager SET no_of_employees =
                    no_of_employees + 1
          WHERE  name = :mname;
          IF (iirowcount = 0) THEN
             RAISE ERROR 1 'RE-Enter Manager'
          ENDIF
     END
```

The rule facilities which are becoming available in the commercial product Ingres flow from on-going work with a Next-Generation Database prototype called POSTGRES ([WENS88],[STON90a],[STON90b],[STON91],[STON92]) which will be discussed in more detail in the following sections.

However, an example like the above does not reveal the true power and potential of rules which is to bring the event-monitoring of real time systems into the realms of the DBMS. To take a simple example, a stock control reorder mechanism can be built into the database server by specifying values for the three basic elements of a rule:

```
AFTER UPDATE (quantity) OF Stock
     WHERE NEW.quantity < 100
     EXECUTE PROCEDURE p_reorder_stock.
```

Moving on to the question of security, rules allow the DBA to add two more mechanisms to the existing ability to grant verb level privileges to users. The degree of auditing that becomes possible using rules is limited only by performance considerations. An example would be the logging of all variation orders approved on capital projects.

```
ON UPDATE (budget) OF Project
     EXECUTE PROCEDURE p_log_changes
```

With the accompanying procedure definition:

```
PROCEDURE p_log_changes
     BEGIN
        INSERT  old_budget,  new_budget,  whoby,
                project, date
        INTO    Variation_log_table VALUES(...)
     END
```

A DBA can easily restrict privileges such as updates of a Salary table to specified users but this is revealed as rather crude when compared to the ability to specify value-based security controls like the following. Here we only allow managers to update the salaries of employees who report directly to them:

```
AFTER UPDATE (salary) OF Emp
     EXECUTE    PROCEDURE    p_check_ok
          (man_no=NEW.man_no)
PROCEDURE p_check_ok (Man_no integer) AS
     BEGIN
          SELECT man_no
          FROM   Manager
          WHERE
                 Manager.man_no = :man_no AND
                  Manager.name              =
                        dbmsinfo('username');
          ENDIF
     END
```

A more complex application is the solution of Tree Walking type problems [KMAN91] such as 'Parts Explosion'. Essentially, this involves implementing a tree search with the root set at the part level that is to be exploded - the rule is triggered by a query against this root which sets off a (potentially recursive) call to a pre-defined procedure that will retrieve all of the sub-parts.

The same ideas can easily be extended to Graph Traversal problems as occur in transportation systems, project management and general network type applications. The approach taken is practical and involves selecting a suitable method, say Dijkstra's Algorithm, and implementing it directly with rules.

State Transition Diagrams offer another illustration. In database applications, extensive programming is used to enforce constraints such as that the status of an order has a natural progression of states from 'approved' to 'picked' to 'dispatched' etc. and that a particular firm may have its own particular policies regarding exception to these. Surprisingly, all of this code can be replaced by a rule which puts an integrity check across these old and new states combined with a simple trigger to link it to the 'Order' table.

So far we have been looking a rules in isolation. In [STON90b] the assertion is made that all data management applications are essentially three dimensional in nature and merit a three dimensional implementation. The point is made that the 'real world' must be modelled in terms of data plus objects plus rules. The example of a newspaper layout application is presented. Whereas a traditional data manager could handle the costing and billing of advertisers, an object manager is required to manipulate pictures and other graphical components. The third dimension is then supplied by the rule set which guides the layout process itself e.g.

> "... the ad copy for two major department stores can never be on facing pages"
>
> [STON90b]

This approach is evident in the design philosophy of POSTGRES.

Reference was made previously to the use of rules in a deductive manner. This makes possible the derivation of data which is not explicitly stored in the database. A glimpse of how this could be useful is given in [STON91] which presents an elegant solution to the problem of keeping the salaries of two employees automatically synchronised - the key concept is that there is actually only ever one value in storage. Stonebraker & Kemnitz propose the following Postquel [WENS88] syntax:

```
ON RETRIEVE TO Emp.salary WHERE
EMP.name = "JOE"
THEN DO INSTEAD RETRIEVE
(Emp.salary)
WHERE Emp.name = "FRED"
```

As can be seen the salary for employee 'JOE' does not need to be stored as a separate data item. This process can, of course, involve a whole series of linked steps depending on the needs of the particular application.

To complete the picture reference has to made to the potential contribution of rules to the field of Semantic Query Optimization ([SHEN89],[CHAK90],[GRANT92]). A recent study [MCMA92] has shown that where response time improvements occur over conventional methods that these are very significant - up to 100% in cases where the existence of a rule obviates the need for any table access in response to a query. However, as mentioned in [SILB91], where large numbers of rules are involved something comparable to the RETE Match strategy [FORG82] will be necessary in order to exploit rules in this way.

Finally, applying rules in distributed systems raises some interesting issues. For performance reasons, asymmetrical multiprocessor systems, where a database system is implemented across multiple processors are receiving attention as a cost-effective solution for very large database applications. As rules are fired by triggers attached to tables vendors will have to offer an efficient mechanism for implementing these triggers when the tables are striped over a multiplicity of separate nodes. Another point in relation to peer to peer distributed systems such as those implemented using Sybase data servers is that, to take a simple example, a rule on database 'A' can have its condition part dependent on database 'B' and its action part happen in database 'C' all on different nodes.

3.3 Next-Generation Prototype Systems

The following sections draw heavily on published results from the two main research vehicles in this area - Starburst and POSTGRES.

The Starburst [HAAS90] project is being undertaken at IBM's Almaden Research Centre and has as its ultimate goal the building of a highly extensible DBMS which can cater for the non traditional requirements outlined in [SILB91]. The approach being taken is to "... explore extensibility in every aspect of database management" [HAAS90] with such features as complex objects, user-defined datatypes, main-memory databases, parallelism in addition to support for rules. The intention is to support everything from CAD/CAM to office systems without compromising on the existing strengths of traditional systems in the areas of concurrency control, optimization, recovery and authorization. There is to be no compromise on performance either and a secondary goal of Starburst is to review and enhance the best ideas put forward for building performance systems and use these to tackle the Very Large Database (VLDB) problem. It has been on-going since 1985 when resources become available towards the end of the R* distributed database project.

In providing support for production rules a new syntax and semantics has been developed. The set-oriented nature of the relational database DML has been carried forward into the rules system. The approach is to implement forward chaining triggers at set, rather than at record level. The corresponding actions can of course involve sets of updates. The syntax is an extension of standard SQL.

Starburst has several rule systems, a shortcoming recognized by the research team, and the goal is to move towards a unified rule processor. The intention is to provide a supporting set of design utilities which would make it easier for the DBA to avoid undesirable interactions or redundancy between rules.

In contrast to Starburst, the POSTGRES project ([STON90b], [STON91],[STON92]), also begun in 1985, is being built by a group of 4 part-time students with a full-time team leader who have nonetheless constructed a prototype comprising of some 180,000 lines of C code. Work is based in the University of California at Berkeley. In addition to the DBMS, a front-end development environment called PICASSO is being developed to exploit the full range of features being built into POSTGRES. Some of these features are already finding their way into the related commercial

DBMS ASK Ingres.

The project goal is to extend the relational data manager to include object (bitmaps, icons, text and polygons) and knowledge management. Knowledge management supports both the enforcing of integrity constraints and the derivation of data not explicitly stored in the database. The eventual aim is to make available a DBMS which will enable three dimensional applications i.e. the real world equals data plus knowledge plus objects. The most remarkable feature so far implemented is the storage manager which is based on a 'replace as delete' mechanism making possible temporal queries.

The evolution of POSTGRES is the result of two fundamental design decisions. Firstly, all database access is from a query language - POSTQUEL [WENS88]. Although a single query language is used at database level the fact that a DBMS usually sits in a multi-lingual environment was recognized and the ability to call POSTGRES from many different languages is envisaged. Secondly, the POSTGRES data model is built using a small number of concepts just as the relational model was. The concepts consist of types, functions and inheritance which suggests that POSTGRES can be considered either as object oriented or extended relational depending on the chosen definition.

Turning to the rules system, not all of the implementation decisions taken have proved to be successful. Building the rules system with a single syntax, although POSTGRES uses both query re-write and trigger mechanisms, has worked well. However, after some years of development work, Stonebraker [STON90b] admits that the rule system as originally implemented was unnecessarily complicated. It also failed to provide support for some expected functionality such as updates on views and, as might be expected from a prototype, the rule system still lacks acceptable performance. Consequently, a 'version two' rule system called PRSII is under development.

The general impression is one of a model which is excessively complex and sophisticated - Stonebraker makes the point that it has taken much longer to build than the original relational prototypes. Conversely, it has taken less than half the number of years for the benefits to be reflected in commercial systems.

3.4 Syntax for Rule Specification

Production rules are of the form 'when X then do Y'. The X part is generally referred to as the trigger, the Y part the action to be performed when X holds ([WIDO89],[STON92]). The exact syntax found will depend on the language used to implement the rules system, be it extended SQL, as in Starburst, or Postquel [WENS88] as used in POSTGRES. Further differences are introduced on account of the range of special features or extensions that occur in the two prototypes.

The Starburst syntax for rule actions is defined in terms of the operation block. An operation block is any sequence of SQL update, delete and insert primitives which together go to make up a database transition (as distinct from a transaction). The following syntax for an operation block is given in [WIDO89] :

```
op_ block ::= sql_op; sql_op;...;sql_op

sql_op    ::= update_op|delete_op|insert_op

update_op ::=  update table
               set columns = expressions
               where predicate
```

```
delete_op ::=  delete from table
               where predicate

insert_op ::= insert into table
              values <V1,V2,...,Vn>
             |insert into table
                 (select_op)
```

The triggering of rules is specified in terms of pre-defined operations on tables. This requires a syntax that sets down both the operation and the target table. Starburst uses the concept of a transition predicate eg. an append to a table - more formally:

```
trans_pred  ::=   updated table
                | deleted from table
                | inserted into table
```

The terms predicate, table and column have the same meaning as in the standard SQL syntax for relational databases.

So far we have not considered conditional triggering of rules. The syntax must support the addition of qualifications to the trigger section. To accomplish this a '.. where predicate...' clause can optionally be included.

Putting it all together then gives a complete primary syntax for the specification of production rules.

```
prod_rule ::=  when trans_pred     {Trigger}
               where predicate     {Condition}
               then  op_block      {Action}
```

```
trans_pred ::= updated table
              |deleted from table
              |inserted into table


op_block   ::= sql_op; sql_op;...;sql_op

sql_op     ::= update_op|delete_op|insert_op

update_op  ::= update table
               set columns = expressions
               where predicate

delete_op  ::= delete from table
               where predicate

insert_op  ::= insert into table
               values <V1,V2,...,Vn>
              |insert into table
                   (select_op)
```

Standard SQL statements are always interpreted in terms of the current state of the database against which they are being run. Rule systems deal with state-transitions and it becomes necessary to be able to refer to four different tuple sets.

In an update statement the old values of the updated tuples as well as the new values assigned by the update must be accessible. The same applies to the removed values referenced in a delete statement and the values appended by an insert statement.

To accommodate these requirements reference to these tuple sets needs to be added to the basic syntax. In Starburst this achieved by the use of the keywords **inserted, deleted old updated** and **new updated** which are placed before the tablename specified in the rule's trigger clause.

This construct enables some very complex rules to be defined on account of the ability to compare old and new values in update statements but a simple example is adequate to illustrate the syntax.

The following statement implements a cascade delete policy. Here we want to enforce the rule that whenever a Department is deleted that its assigned employees are also removed.

```
WHEN DELETED FROM Dept
THEN DELETE FROM Emp
     WHERE dept_no IN
     (SELECT dept_no FROM DELETED Dept)
```

This also illustrates the set-orientation of the syntax - the same statement can cater for both single tuple and multiple tuple deletions.

The syntax used in POSTGRES differs not only for the trivial reason that it is QUEL rather than SQL based but because it reflects the presence of many novel ideas. For a start, there are three categories of rule in POSTGRES - **always**, **once**, and **never** rules [WENS88].

'Once' rules are designed to fire when a qualification predicate attached to the trigger becomes true. After the rule fires it is automatically deleted.

The POSTGRES 'never' rule category can be view as access control statements implemented as rules. The term **is never** is added to the rule definition syntax as shown by an example from [WENS88]. This rule refuses access by the user 'Spyros' to salary details of first floor department employees.

```
define rule Y is never
     retrieve (emp.salary)
     where emp.dept = dept.dname
     and   dept.floor = 1
     and user0 = "Spyros"
```

Most rules will fall into the category of standard rules which are classified as **always** rules in POSTGRES.

The full syntax for assigning a rule to a class has the following structure.

```
define rule rule_name is always|once|never
query
[priority = number]
```

The syntax provides for the optional specification of a priority for a rule in the range 0 (default value) to 15. These values can be used in conflict resolution.

Within each category the individual rules are specified using the following compact syntax.

```
on append|retrieve|replace|delete to database_object
     [where expression]
     then do [instead] expression
```

Put more simply, all rules are of the form 'on event do action'. In [STON92] all of the possible combinations of such events and actions are explored yielding four variations in all as both events and actions are database operations which can only be retrieve or update (in the broad sense) statements. The 'database_object' can be a view as well as a base table or attribute.

Update/Update rules produce **forward chaining**, a mechanism supported in both POSTGRES and Starburst as well as the commercial products ASK Ingres and Sybase. Because such rules have an update in both the trigger and the action it becomes possible to set off a chain reaction of updates involving any number of rules. Stonebraker's familiar salary propagation example is a rule of this type.

```
on replace to Emp.salary where
     Emp.name = "Joe"
then do replace Emp (salary = new.salary) where
     Emp.name = "Sam"
```

Update/Retrieve rules drive **alerters**. By defining such a rule a user is saying that if the specified event occurs 'then I want to know about it'. Stonebraker [STON92] notes that, so far, only POSTGRES and HiPAC support this feature.

Although a rule of the form Retrieve/Retrieve might initially appear fairly innocuous, this construct turns out to be very powerful indeed. By inserting the keyword **instead** into the do clause backward chaining becomes possible.

This presents another strategy for solving the Joe/Sam problem. Rather than explicitly storing the two salaries and keeping them in step POSTGRES stores a single value and a policy.

```
on retrieve to EMP.salary where
     EMP.name = "Sam"
then do instead  retrieve (EMP.salary) where
     EMP.name = "Joe"
```

The presence of a backward chaining mechanism raises the issue of recursion and the marketers of the POSTGRES commercial offspring Ingres Knowledge Manager have been quick to seize upon such a differentiating factor.

Recursion allows a single rule to perform tasks such as extracting an employee's chain of command from a personnel database. In the context of POSTGRES syntax the practical implication is that specifying the same attribute name in the action as well as the event clause of a rule may lead to recursion.

So far only the POSTGRES rules system supports backward as well as forward chaining.

The final category provides a useful means of implementing an audit trail feature - on retrieve .. do update. Stonebraker [STON92] offers the following example to illustrate how this might be used.

```
on retrieve to EMP.salary
then do append to AUDIT
     (name = current.name,
      salary = current.salary,
      user = user())
```

The usefulness of rules systems in authorization checking has also been studied in the Starburst prototype and is discussed in [WIDO89].

As can be seen with both prototypes, the syntax is continuously evolving to incorporate new features. It is interesting to note that the POSTGRES derived 'Knowledge Manager' facility now available with ASK Ingres allows rules to be specified in SQL.

3.5 Rule Execution Semantics

We now focus on the way in which rules behave and interact in a database rules system. Although this has yet to become a mature research area it is possible to summarize the key issues involved.

While rules are activated by database operations the designer still has the discretion to decide when the rules system should be allowed in, leading in turn to a variety of different end-states. For example POSTGRES allows free interleaving between the data and rules systems so that rule activation is exactly as implied by the syntax - immediate activation. However, this means that rules work at a record level rather than at set level as in Starburst. This difference is also reflected in ASK Ingres (POSTGRES approach) and Sybase (Starburst approach). An intermediate strategy which enjoys some of the benefits of the set-oriented approach is to allow in the rule manager between commands. For example, if a command updated all tuples in a table the rules system would await this update and then fire once rather than as each tuple was touched by the update.

Turning to the approach used in Starburst we find that the separation is at transaction level. The semantics of the Starburst rules system can be visualised in terms of a state-transition diagram with the transitions corresponding to committed database transactions. These state transitions provide the triggers.

In Starburst "..rules are considered and executed just before considering and executing each externally-generated transaction" [WIDO89]. Widom and Finkelstein use the word 'externally-generated' to distinguish normal database transactions from the rule system database transactions which may result from rule firing. Irrespective of their origin, however, all transactions are treated in the same way - a state transition which produces an effect and a new state. This implies that rules can in turn trigger other rules producing forward chaining in Starburst.

A final option is to de-couple data and rule transactions in which case the rules system will not automatically kick in at the end of a transaction as just described but will catch up later. This mechanism has been studied in HiPAC [MCCA89].

In [STON92], Stonebraker points to a specific security implication of this decoupling. He takes the example of a rule-implemented audit trail to log user accesses to the salary attribute of an employee table. Using the same transaction for both data and rule operations effectively allows users to cover their tracks by aborting the transaction after accessing the salary thus leaving no trace in the audit trail. It has to be said that this is really a special case as it would be unacceptable to allow a rule triggered off by an aborted transaction to career on across the database. If a separate transaction strategy is to be adopted then it is vital to have effective co-ordination between the data and rule systems to abort rules where necessary.

So far, it has been implied that semantic differences exist only between different implementations but this is not entirely true. It was mentioned previously that POSTGRES supports both forward and backward chaining rules. In both cases there is a determining value. In backward chaining this has to be the value which is actually stored in the database as distinct from the value(s) that may be derived from it. In forward chaining it is the value cited in the rule trigger. If its value is deleted and the dependent attribute then examined a dichotomy is observed. In the forward chaining case the value found will be the last one stored as a result of the rule. However, with backward chaining, the rule will return the current value of the determining value which will be null.

Finally, what happens when several rules are enabled at the same time. Even with only two rules the possible semantic variations begin to mount up.

One solution would be to fire only one of the rules as is the case in POSTGRES if the **exception** syntax is used instead of the original priority mechanism.

A strict ordering could be imposed as in the case of Starburst where a **before** and **after** syntax is available. Alternatively a laisser faire approach could be adopted letting all of the enabled rules fire in a random manner. Once again, the effects will differ between the three cases.

The designers of Starburst also considered other options such as 'least recently triggered' and 'most recently triggered' along with total, partial and weighted ordering. They concluded that although the only way to guarantee deterministic behaviour was to use total ordering that this meant the loss of the flexibility to add rules independently.

Widom and Finkelstein [WIDO89] consider flexibility to be a vital design objective for rule systems.

> "...we might want additional flexibility in the time at which rules are triggered and in the correspondence between rules and transactions... For example, we might want the ability to specify that a rule's action should be executed in a separate transaction. Also, in some cases, it might be advantageous to execute several externally-generated transactions before considering triggered rules, or , conversely, we might prefer to consider rules earlier than the commit point of an externally-generated transaction."
>
> [WIDO89]

Overall, the literature reflects an uneasy sense of critical mass which could precipitate an uncontrollable semantic explosion at any moment. Even Stonebraker is using language such as '.. semantic morass..' and '..too complex for any database administrator to understand..' which re-enforce the impression of moving away from the mature and well understood environment of the traditional relational DBMS. Characteristically, in [STON92], he counters this by making two incisive proposals to researchers. The simplest way forward, he contends, would be to use a scoping approach. Simplify the problem by rejecting, as semantically unworkable, certain complicating cases. His second suggestion, another type of abstraction, would be to come up with a higher level language that would provide a form of insulation analogous to the abstraction of the intricacies of the base machine provided by an operating system.

3.6 Implementation of Rules Systems

The primary implementation problem is that the DBMS is now being asked to take on the workload associated with a rule base on top of existing commitments. There is also the very practical constraint that in next generation database systems the rule base will, most probably, need to held on disk rather than main memory due to its size.

In light of this the prototype builders have examined a succession of solutions. Stonebraker gives an up to date evaluation of three such schemes in [STON92], **brute force**, **discrimination network** and **marking**.

Brute force involves keeping a list of every defined rule. When a database operation occurs this event is matched against the condition part of each of the rules. Although simple to implement, this strategy proves to be too slow once the number of rules begins to grow. The list needs to be structured so as to enable faster access.

Discrimination networks have been studied for some time by Artificial Intelligence researchers notably by Forgy [FORG82]. Nonetheless, his Rete Match Algorithm assumes that all of the rules are in memory. As mentioned at the outset this would be unsuitable for very large databases.

A third technique called **marking** has been adopted in the POSTGRES prototype. Here no lists are kept but instead each rule is 'processed' against the database and every record which is touched by the qualification clause is identified. I suspect that the adoption of this mechanism to some extent accounts for the record level activation of rules favoured in POSTGRES as opposed to the set-level emphasis of Starburst.

Records are marked with identifiers for the rules to be triggered. This requires more storage space but obviates the need to perform any rule search at run time. Stonebraker recognizes the problems inherent in keeping the marking up to date as updates are made to the database. For example, if a rule's scope applied to employees with salaries less than £20,000 and, say, Jones got a pay rise then his record would need to undergo marking adjustment. Nevertheless, this is nothing totally new as the marking can be considered as just another kind of index on the table with an inevitable but acceptable maintenance overhead.

For the implementation of backward chaining rules POSTGRES employs a technique called **query rewrite**. This entails fleshing out the user command with the logic of the relevant rules. This is analogous to how query modification is currently used to implement user privileges in relational database systems. In POSTGRES, this involves running the **do instead** clause of the rule. As hinted at in regard to user privileges, query rewrite does not have to be confined to the implementation of backward chaining rules and an evaluation of the pros and cons of this technique can be found in [STON90b]. An indication of the alternative proposals evaluated, particularly from a performance standpoint can be judged from [STON86].

Query rewrite is also used in Starburst. Perhaps a certain convergence is to be expected between the two prototypes when they come under the harsh light of the performance benchmark arena. As Haas et al., in a review of the Starburst prototype, put it "... these problems are not peculiar to our application but are in fact generic to all rule systems." [HAAS90].

As previously indicated Starburst actually has several rule systems with separate rule processors. The various features implemented, such as the use of prioritized queues are outlined in [HAAS90]. This 'multiple rule processor' approach tends to give an erroneous impression of a lack of focus in Starburst but, it is, after all, a research vehicle and building several rule processors in different ways increases the potential for generating efficient solutions. In ways, both POSTGRES and Starburst are at a stage of development similar to the early relational prototypes when people accepted that they liked how they looked but now wanted to see them run. The solution may come in the form of a hardware driven solution such as parallelism. Indeed, some research into the exploitation of such technology for parallel rule processing has already been conducted eg. [ISHI91].

We now have our starting point, a framework for exploring what happens when we bring time into the picture, and in the next chapter we will move on to consider the potential of temporal rules and the question of how they might be specified to a DBMS.

Chapter 4
A Working Syntax for Temporal Rules

We begin this chapter with a brief look at some of the opportunities which the introduction of a time dimension to database rules presents. These will be elaborated upon in later chapters when specific application domains are discussed in detail. We go on to present a working syntax for the examination of time-enabled database rules.

4.1 Motivation

As will become clear, the simple act of bringing time to bear on what we have said so far regarding database rules opens the door to a interesting range of possibilities.

We will see how cyclical rule firing enables batch jobs to be specified at database rather than operating system level; how time-based alerters can be used to implement scheduled reviews, deadline notification and general timing constraints; how deferred rule actions support the roll-out of a series of procedures over time e.g. reminders of increasing severity, enforcement or compliance with regulations by specific dates; how database rules are enabled to take part in workflow monitoring - checking output/progress at regular intervals; time-based rule enabling - rule lifetimes, rule dormant/active periods; time-specific processing - rules fired by specific dates such as retirement dates or implementation dates for new tax rates.

On a more specialised level, temporal rules also present the possibility of providing sophisticated tools such as 'DBA Advisor' applications. Examples would be the automated testing and logging of response-time figures - rules as continuous benchmarks; monitoring table growth by way of rules defined on the catalog (a temporal rule would fire, say, every hour and check the rate of table growth) and support for real time (automated) performance tuning.

Finally, although some commercial DBMS can already detect run-away queries and stop them, a more ambitious idea might be to use rules to attach specific time-outs to updates that not only aborted the update but re-submitted it later at a less-busy time.

The syntactic enhancements called for to effectively deliver the kind of functionality that we have touched on above is the subject of the remainder of this chapter.

4.2 Modified Syntax for Temporal Extensions

In this section we set out the changes in syntax required to support the temporal extensions, define each of the new terms introduced and explain their usage by way of examples.

The complete extended syntax is as follows:

```
     DEFINE RULE IS ALWAYS|ONCE|NEVER|TEMPORAL
               [,CYCLE=time interval,LIMIT = max number of
                      cycles(default = ∞)]
               [,TIMES=time range|specific times]
               [,DAYSOFWEEK= days]
               [,DATES = date range|specific dates]
               [,LIFETIME = start, finish]

               ON   query|OCCURRENCE|OCCURRENCE AND query|
                          OCCURRENCE OR query

               [WHERE condition]
               [incidence = n out of m]

               DO action
               atime (default = now)
```

CYCLE

CYCLE is the time interval between activations of a recurrent rule where the firing is on a regular time basis.

```
[, CYCLE = time interval ]
```

time interval

The parameter 'time interval' is expressed in terms of

```
months:days:hours:minutes:seconds
```

Example:

```
DEFINE RULE IS TEMPORAL, CYCLE = 01:00:00

ON OCCURRENCE

DO
     DELETE FROM Pending
     WHERE  Pending.closed = 'Y'
```

This rule does garbage collection on a table of pending requests by clearing out 'closed' items every hour.

LIMIT

LIMIT is the number of cyclic recurrences defined for a rule. It is one method of specifying a lifetime for a rule.

```
[, LIMIT = max number of cycles ]
```

max number of cycles

The parameter 'max number of cycles' is an integer value representing the number of cyclic recurrences which the rule is allocated. If no LIMIT is defined then the default value is infinity ie. the rule will continue to recur until it is deleted.

Example:

```
DEFINE RULE IS TEMPORAL, CYCLE = 01:00:00, LIMIT = 5
ON OCCURRENCE
DO
     SELECT count(*) FROM Pending
     WHERE  Pending.closed != 'Y'
```

Such a rule might be defined by a supervisor to monitor how staff are progressing with a build-up of requests. The supervisor wants to be notified with an hourly count of the outstanding items for the duration of the following 5 hour period.

TIMES

TIMES specifies either an effective time range for cyclic rules or explicit times at which to fire for other rules.

```
[, TIMES = time range|specific times]
```

time range

The parameter 'time range' is an interval or series of intervals during which cyclic firing of a rule is enabled.

Example 1:

```
DEFINE RULE IS TEMPORAL, CYCLE = 01:00:00,
          TIMES = 09:00 - 17:00

ON OCCURRENCE

DO
     SELECT count(*) FROM Pending
     WHERE  Pending.closed != 'Y'
```

This example is a modification of the previous rule which now runs an unlimited number of times but is only enabled during normal business hours.

specific times

The parameter 'specific times' allows the specification of one or more explicit times which will serve as a trigger for non-cyclic rules.

Example 2:

```
DEFINE RULE IS TEMPORAL
          TIMES = 09:00;17:00

ON OCCURRENCE

DO
     SELECT etime, count(*) FROM Pending
     WHERE  Pending.closed != 'Y'
```

This rule takes an opening and closing balance for the day of requests outstanding. The special variable **etime** captures the date and time of the rule triggering event.

DAYSOFWEEK

DAYSOFWEEK specifies the days of the weeks on which the rule is enabled.

```
[, DAYSOFWEEK = days]
```

days

The parameter 'days' is a list of day identifiers.

Example :

```
DEFINE RULE IS TEMPORAL, CYCLE = 01:00:00,
          TIMES = 09:00 - 17:00
          DAYSOFWEEK  = Mo;Tu;We;Th;Fr
ON OCCURRENCE

DO
     SELECT count(*) FROM Pending
     WHERE  Pending.closed != 'Y'
```

This example shows the rule enabled during normal business hours from Monday to Friday only.

DATES

DATES specifies either an effective date range for cyclic rules or explicit times at which to fire for other rules.

```
[, DATES = date range|specific dates]
```

date range

The parameter 'date range' is a date interval or series of intervals during which cyclic firing of a rule is enabled.

Example 1:

```
DEFINE RULE IS TEMPORAL, CYCLE = 01:00:00,
          DATES != ('01-Aug' - '15-Aug')

ON OCCURRENCE
DO
     SELECT count(*) FROM Pending
     WHERE  Pending.closed != 'Y'
```

This rule suspends the running of the hourly workload check for the duration of the summer plant closure. This example also indicates the potential for building real-time process control applications from a collection of such rules.

specific dates

The parameter 'specific dates' allows the specification of one or more explicit dates which will serve as a trigger for non-cyclic rules. However, if no year is specified they become implicitly cyclic with a time interval of ' 1 year'.

Example 2:

```
DEFINE RULE IS TEMPORAL
          DATES = '01 Jan'

ON OCCURRENCE

DO
     {New Year Routine}
```

This example shows a rule driven implementation of a 'New Year Routine'.

Example 3:

```
DEFINE RULE IS TEMPORAL
          DATES = '01 Jan 1997'

ON OCCURRENCE

DO
     SELECT count(*) FROM PC_Inventory
     WHERE Anti_Glare_Compliance != 'Y'
```

This example shows a rule which will re-awaken when the EC transition period for compliance with the directive on the provision of anti-glare screens to employees expires. The value returned by count(*) should be zero when the rule fires.

LIFETIME

LIFETIME specifies the timespan during which the rule is in force.

```
[, LIFETIME = (start, finish)]
```

start, finish

The parameters 'start' and 'finish' specify the date on which the rule is to come into force and the date when it expires respectively. The default value for 'start' is **now** and for finish is **infinity**.

Example :

```
DEFINE RULE IS TEMPORAL, LIFETIME = ('01-Jan-1993',-)

ON OCCURRENCE AND INSERT INTO Contract
```

```
    DO
          {Single European Market Query}
```

This example shows a rule which remains dormant until a scheduled statutory date. Perhaps a rule might exist to enforce a competition directive which comes into force from that date onwards. The boolean combination in the event clause is discussed later.

WHERE CLAUSE

The WHERE clause allows the specification of an optional condition on the rule trigger as in the traditional syntax. However, for recurrent rules additional conditional statements can be used. We may wish to define an action threshold - only fire the rule if the condition is satisfied on a specified percentage of evaluations [RMON92].

```
    [,incidence = n out of m]
```

n

'n' is the threshold number of evaluations to true

m

'm' is the sample size

For instance, we could test for the size of a transaction table every 10 mins and take an appropriate action if it were empty more than 90% of the time.

DEFERRED ACTION

The existing syntax for rules does not support deferred or scheduled performance of the rule action(s).

```
DO action
```

atime (default = now)

atime

No value need be specified if the action is to be performed immediately . The extended syntax can also support a single action at specific 'atime' or multiple actions at multiple 'atimes'. In the latter case these can be specified in terms of the time associated with the triggering event eg. etime + '3 months', etime + '6 months'. Thus, a series of actions can be scheduled to occur over time.

Boolean Combination of Triggering Events

Rules fire when their defined triggering event evaluates to true. This, as the syntax suggests, has implications for the specification of rules.

ON query|**OCCURRENCE**|**OCCURRENCE AND query**|
OCCURRENCE OR query

ON query

This is the standard syntax used in traditional rule definition. Rules are triggered by query events against specific database objects eg. 'ON update to Employee.salary...' .

ON OCCURRENCE

In this case the rule will fire immediately on the pre-defined time based event. An example would be a rule defined to fire every night at 11.30 pm to kick off an end of day update.

ON OCCURRENCE AND query

Here the rule will be enabled by the time based event but will not fire unless the 'query' clause becomes true. An example use would be putting a time frame on the simple 'ON update of Employee.salary' rule mentioned earlier. The rule is enabled in the time frame but will not actually fire unless the query event occurs as well.

And finally,

ON OCCURRENCE OR query

Such rules will fire on whichever event happens first.

In the next chapter we will explore the semantics of the syntax defined above by attempting to apply it in two application areas which are rich in temporal rules. As part of this process we will evolve an appropriate formalism for the documentation of such rules.

Chapter 5
An Operational Semantics for Temporal Rules

In this chapter we will explore the semantics of temporal database rules by way of an empirical approach. Two example domains are used - Personnel Management and Programme Scheduling. In each case a graphical schema is presented followed by a mapping into the temporal syntax.

The chapter begins with a discussion of the formalism chosen for specifying the schema - OSA (Object-Oriented Systems Analysis) which supports time-based triggers and constraints. The suitability of this methodology owes much to its object-oriented basis. As will be seen, for instance, in the Personnel Management example the database rules can be thought of as being defined for the Class 'Employee' - logically part of the methods for this Class. When a new employee is hired the employee object identifier ('employee #') is used to instantiate a rule-set for this new employee from the 'Employee' Class rule templates.

5.1 Graphical Representation of Temporal Rules

In order to explore and illustrate the use of temporal database rules a formalism supporting the semantics of event-driven actions was required. A methodology called OSA (Object-Oriented Systems Analysis) [EMBL92] was selected.

In particular, the adapted state-net diagrams which are employed in OSA to capture the details of object behaviour (including time-based triggers and real-time constraints) proved an efficient mechanism for the identification and subsequent specification of temporal rules. OSA is semantically more powerful than the classical approaches to systems analysis of DeMarco and Gane ([DEMA79];[GANE79]) and is more implementation independent than Coad and Yourdon's OOA (Object Oriented Analysis) methodology [COAD90].

If a data-flow diagram (DFD) is examined it is apparent that any of the processes documented will contain elements of the behaviour of a variety of the objects in the application domain. The emphasis in OSA is on taking the objects themselves as the starting point and building a comprehensive standalone description of the properties and behaviour of each.

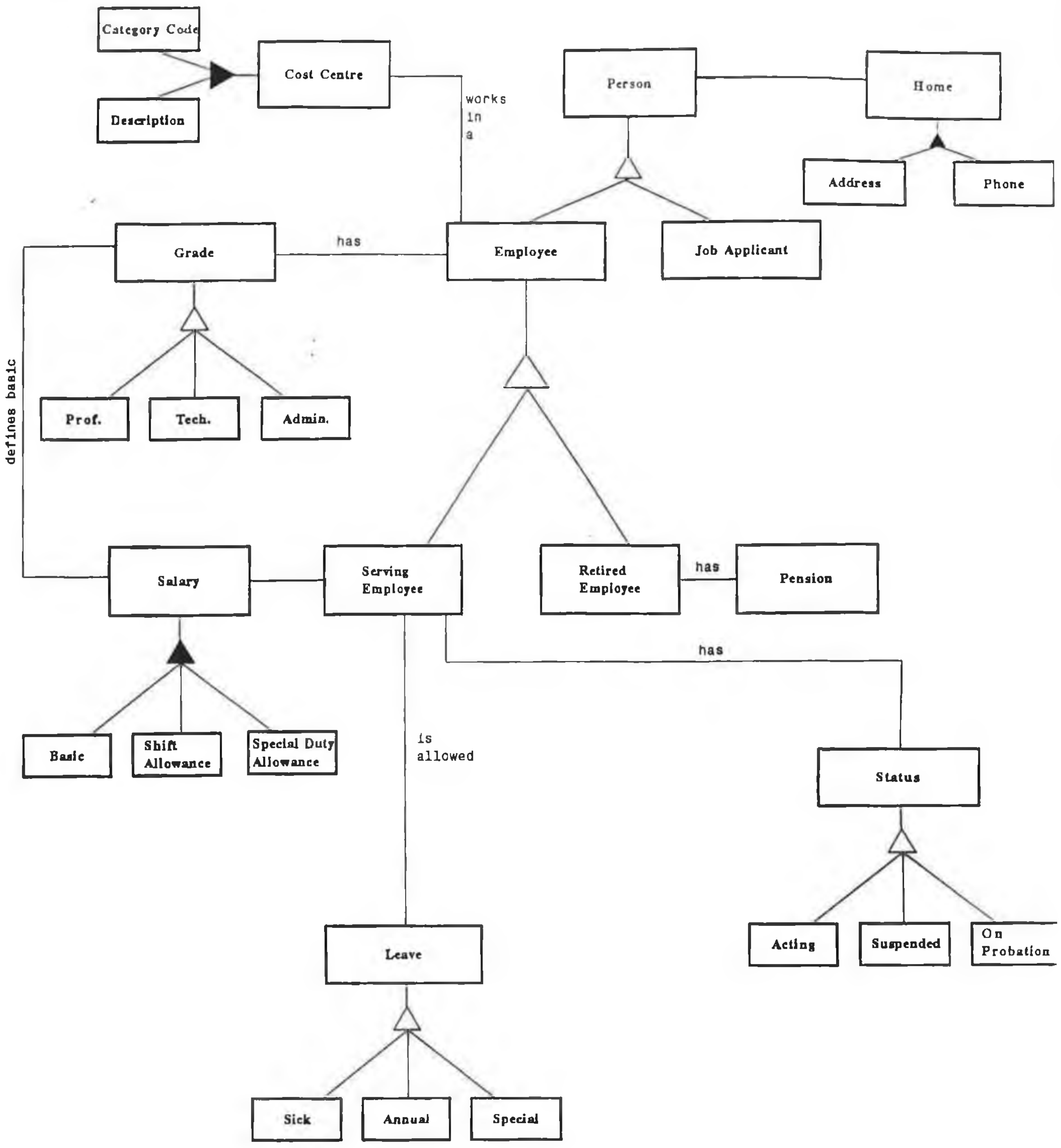
The benefits of this approach include the resultant direct correspondence between real world objects and the analysis documentation produced to describe them; a concentration on the 'what' rather than on the 'how' during the analysis stage and the increased semantic modelling power that comes with support for aggregation, generalisation and classification.

The basis of OSA then, is the compact representation of object classes; relationships between object classes;

object behaviour and object interactions. The starting point is the object class - represented by a rectangle.

A labelled line connecting two object class rectangles denotes the relationship-set between them. The relationship may be simple such as 'ownership' between a person and a consumer product but OSA is equally comfortable with 'Is A' (represented by a triangle) and 'Is Part Of' (shown as a dark triangle) relationships. These four symbols (rectangle, triangle, dark triangle and connecting line) are combined in the first category of diagram used in the OSA formalism, the Object Relationship Model (ORM). A partial ORM for the Personnel example is given in Figure 1 which illustrates the ease with which Generalisation/ Specialisation, Inheritance and Aggregation can be specified in OSA.

The second type of diagram available to the analyst using OSA is the Object Interaction Model which captures the interplay between different objects in the system. As before, the objects themselves are represented by rectangles. A single new symbol is introduced - the zigzag line - to represent the interaction. For instance, a manager putting work in a secretary's in-basket is an object interaction. The manager puts the work in the in-basket which the secretary removes from time to time. What the object interaction model does not tell us is what triggers the secretary to remove items from the in-basket in the first place or how long the manager can wait for this to happen. To answer such question requires some form of behaviour modelling and this brings us to the third element of OSA called the Object Behaviour Model which uses three basic concepts - states, triggers and actions, the very language of database rules.



PARTIAL ORM FOR PERSONNEL DOMAIN

Figure 1.

5.2 The OSA Object Behaviour Model

The behaviour of an object has three dimensions

(a) the states which it can assume

(b) the conditions which cause it to change state

(c) the actions associated with the object in these states or in changing between states.

In OSA a state-net is drawn for each object class. The rectangle representing the object class is exploded in the state-net diagram to reveal the detail within. States are represented by rectangles with rounded corners,transitions are shown as partitioned rectangles with the trigger above the line and the action(s) below. The transition paths are represented by directed arcs. Importantly, real-time constraints on the object's behaviour can be added to the basic state-net.

Building a state-net begins with consideration of the valid states which an object can exhibit in the target system. For instance, a sales order would have states such as 'open' , 'filled' and 'invoiced'. The next step is to look at how an object moves in and out of these states, more specifically, what events trigger these transitions. For instance, the event of raising an invoice triggers the transition of an order from 'filled' to 'invoiced'.

OSA recognizes two types of actions: noninterruptible and interruptible. The former are atomic in nature - they either complete or rollback whereas the latter category can be suspended and resumed as required. Noninterruptible actions are associated with transitions while interruptible actions relate to states - indeed the state itself may represent the continuous performance of some action which is interrupted and resumed as the object leaves and returns to that state. Object concurrency, both interobject and intraobject, is supported - not alone can the different states be occupied by any number of class instances at the same time but a given object can be in more than one state at any instant, for example, speaking on the phone and opening in-coming mail.

The firing of transitions is far from automatic. The trigger must first be enabled by its designated prior state. Additional conditions may also need to be satisfied and indeed a trigger may be viewed as a boolean expression which evaluates to true or false. This echoes the trigger and condition syntax used for specifying database rules.

As will be seen in the Employee state-net some additional conventions are required. The initial state of the object is shown as a solid line rather than a round-cornered rectangle. An event monitor **@hire** is used to detect the arrival of a new employee in the system. This has the subsequent state of 'On Probation' but no prior state -

initial transitions are always enabled. Terminal states can be recognized by the absence of any arrow leaving that state.

One final state-net symbol remains to be discussed. Analogous to the idea of levelling in data flow diagrams OSA supports 'states-within-states'. It is possible for an object to enter a new state without actually leaving its current one. This layering of states is shown as an extra arc outside the symbol representing the enabling state.

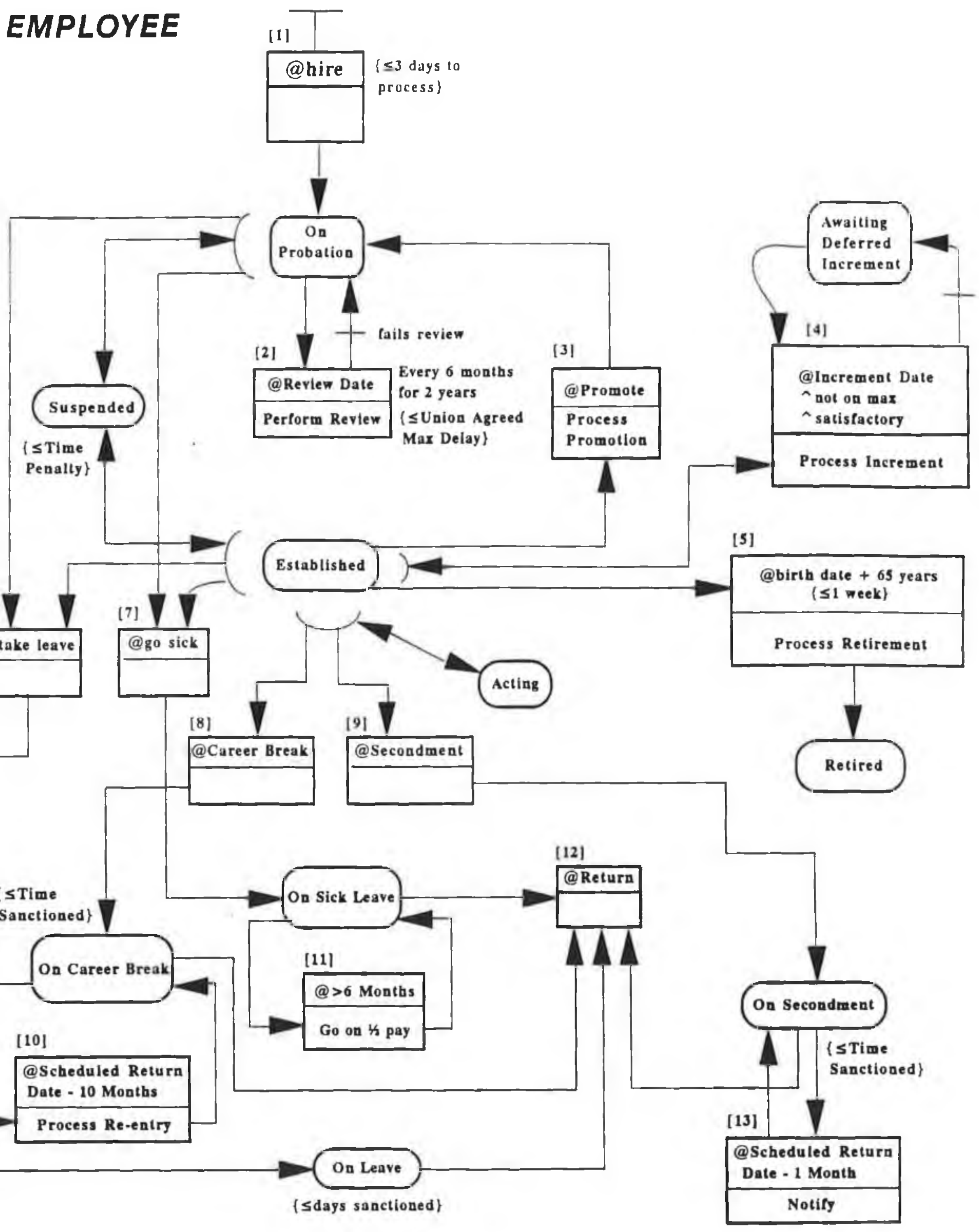
5.3 Modelling Real-Time Constraints

Once the state-net has been drawn timing constraints can be added to capture any important temporal aspects of the object's behaviour. Timing constraints can be specified for triggers, actions, states and the duration of state-transition paths.

In each case the constraint is specified using an expression enclosed in braces ({}) which is associated with the appropriate symbol on the state-net. For instance, the real-time constraint {<= 1 hour} might appear beside the 'on lunch-break' state of an employee. Similarly, a constraint of {<= 15 minutes} might be specified for the action of filling a sales order.

Constraints on triggers specify the acceptable 'response time' between the firing of the trigger and the commencement of the accompanying transition. Finally, a constraint can be defined over the duration of the transition as a whole covering the time to respond plus the time to leave the old state, perform all of the transition actions and enter the new state.

The role of this constraint mechanism in temporal rule specification is further explored in the examples which follow.



PARTIAL OSA STATE-NET FOR THE PERSONNEL DOMAIN

Figure 2.

5.4 Worked Examples

Two case studies are used viz. Personnel Management and Programme Scheduling. In each case an OSA state-net is presented (Figures 2 and 3), and the accompanying temporal rules defined.

Personnel Management

The rules are categorized into transition-centred rules (which are basically time-based triggers) and state-centred rules which consist of timing constraints. We will begin by looking at the transitions in Figure 2 .

Transition [1]

This is a timing constraint on a trigger which states that the specified action must commence within a specified time. In this specific instance, the constraint states that the induction procedure should commence within three days of hiring a new employee. The following rule checks three days after the hire date that the induction procedure has indeed commenced for a new employee. The first step might be to place him/her on the payroll.

```
DEFINE RULE check_induction_init IS TEMPORAL
ON Append to Employee
DO exec proc check_induction_begun(emp#)
   atime = etime + '3 days'
```

Transition [2]

An employee on probation has a review every six months for a period of two years [rule (a)]. There is also an agreed maximum time of two weeks in completing the review on foot of a Union agreement [rule (b)].

Rule (a)

```
DEFINE RULE probation_review IS TEMPORAL
       CYCLE = '6 months'
       LIMIT = '4 cycles'
ON OCCURRENCE
DO INSERT INTO Pending_review
VALUES (emp#, 'Due since ',date(etime))
```

Rule (b)

```
DEFINE RULE Union_Agreement IS TEMPORAL
ON Append to 'Pending_Review'
DO exec proc check_review_complete(employee#)
   atime = etime + '2 weeks'
```

Transition [4]

Employees on a salary scale are due an increment on their designated increment date if they are not already on the maximum point of their pay scale and if their work is satisfactory.

```
DEFINE RULE increment_rule IS TEMPORAL
     DATES = employee.increment_date
ON OCCURRENCE
WHERE {Not on max pay and satisfactory}
DO exec proc pay_rise(emp#)
```

This shows the way in which an object Class rule template can instantiate an object specific rule by filling in the blanks such as the increment date above when a new employee is created. In this way the rules can be viewed as an extension of the methods for the Class with the triggering dates specific to each employee being bound to the relevant Class rule.

Transition [5]

There are two temporal rules involved here:

(a) An employee must retire at 65 years of age.

(b) The procedures involved, such as putting the employee on pension must begin within a week of the retirement date.

Rule (a)

```
DEFINE RULE retirement_rule IS TEMPORAL
     DATES = employee.retire_date
ON OCCURRENCE
DO exec proc retirement_procedure(emp#)
```

Alternatively, if the number of employees was small this rule could be implemented as follows:

```
DEFINE RULE retirement_rule2 IS TEMPORAL
     DATES = employee.birthday
ON OCCURRENCE
WHERE ('today' - employee.birthdate) >= '65 yrs'
DO exec proc retirement_procedure(emp#)
```

In the second example the check is made every year for each employee.

Rule (b)

```
DEFINE RULE max-delay IS TEMPORAL
ON Append to Pending_Retirement_Procedures
DO exec proc check_procedures_begun(emp#)
   atime = etime + '1 week'
```

This implements the timing constraint of one week placed on the delay in getting the various tasks associated with a retirement underway.

Transition [10]

Ten months before an employee is due to finish a career break preparations for their return must commence. For example, their name must be appended to the short-list table for vacancies at their grade

```
DEFINE RULE career_break_rule IS TEMPORAL
     DATES = return_date - '10 months'
ON OCCURRENCE
DO INSERT INTO Short_List
     VALUES employee#, grade, return_date
```

Transition [11]

If an employee is on sick-leave for more than six months they are put on half pay.

```
DEFINE RULE extended_sick_leave IS TEMPORAL
     DATES = sick_leave_begin + '6 months'
ON OCCURRENCE
WHERE sick_leave_return IS NULL
DO exec proc half_pay (emp#)
```

An efficient implementation of such a rule would probably de-activate the rule when the employee returned from sick leave.

Transition [13]

An employee on secondment (loan) to another organisation must be notified to come back one month before their scheduled return date.

```
DEFINE RULE secondment_rule IS TEMPORAL
     DATES = secondment_end_date - '1 month'
ON OCCURRENCE
DO exec proc notify_employee (emp#)
```

When an employee goes on secondment this rule for the Class Employee is instantiated for the given employee using the parameters 'emp#' and 'secondment_end_date'.

The following examples look at how temporal rules may be associated with object states.

Suspended State

For various reasons an employee may be taken off the payroll temporarily for a specified period. There is therefore a real-time constraint on the time the employee should be kept in that state. The following temporal rule expresses this constraint.

```
DEFINE RULE suspended IS TEMPORAL
ON Update to Employee.status
WHERE new.status = 'Suspended'
DO exec proc revoke_suspension(emp#)
     atime = etime + (suspension_period)
```

The last line states that the action part of the rule will not fire until the suspension period has elapsed.

Career-Break State

An employee may voluntarily take a year or longer off without pay to pursue other interests and be re-instated on return. The break cannot be longer than the period sanctioned.

```
DEFINE RULE career_break IS TEMPORAL
ON Update to Employee.status
WHERE new.status = 'On Career Break'
DO exec proc re-instate(emp#)
     atime = etime + interval('sanctioned_break')
```

Secondment State

Secondment to another organisation is for a sanctioned loan period. The employee must then return to his/her substantive position. Once again there is a real-time constraint placed on the time spent in the secondment state.

```
DEFINE RULE secondment_rule IS TEMPORAL
ON Update to Employee.status
WHERE new.status = 'on secondment'
DO exec proc resume_substantive(emp#)
     atime = etime + secondment_period
```

On-Leave State

Career Breaks and Secondment arrangements are relatively rare. However, paid leave in its various forms - especially annual leave and sick leave, are not only very common but require close monitoring. These are characterised by high volume/short duration time spans making the requirement one of exception handling - a regular check for cases of leave taken beyond the amount sanctioned. The temporal rule system must provide the functionality of the traditional end_of_day/ week batch report. A rule-level implementation is presented below.

```
DEFINE RULE leave_check_rule IS TEMPORAL
     CYCLE = '1 week'
ON   OCCURRENCE
DO  exec proc leave_check
```

The above example illustrates the importance of supporting the full semantics of time-based triggering. In this example the system designer could elect to use either the actual return date or a time cycle as the appropriate triggering mechanism.

Programme Scheduling

"Life has been a bit quiet here lately. The only thing that changes from day to day are the television programmes"

Anon

Although a television programme schedule changes every day there is an underlying framework of fixed points on which it is built. These consist of transmission start and approximate closedown, newstimes and regular commercial breaks. This backdrop is further classified into Weekday, Saturday and Sunday patterns. The second case study takes the example of creating an active database of these schedule frameworks. The rules for this database will be exclusively of the alerter category and require a time-based triggering mechanism.

The templates for Weekday, Saturday and Sunday schedules are set out below:

Transmission Times

Category	Start	Closedown	Synonyms
Weekday	12.05	c. 11.45 Closedown depends on how thing actually transpired on the day. An adjustment for 'injury time' needs to be added to the nominal <weekdayclose> value.	**<weekdaystart>** **<weekdayclose>**
Saturday	13.05	c. 00.30 (Sun)	**<satstart>** **<satclose>**
Sunday	11.00	c. 00.30 (Mon)	**<sunstart>** **<sunclose>**

Newstimes

Category	Times	Synonyms
Weekday	13.00,15.00,18.01,21.00	**<weeknews1>** **<weeknews2>**

		<weeknews3>
		<weeknews4>
Saturday	14.25,18.01,21.00,00.20(Sun)	**<satnews1>**
		<satnews2>
		<satnews3>
		<satnews4>
Sunday	13.40,18.01,21.00,00.20(Mon)	**<sunnews1>**
		<sunnews2>
		<sunnews3>
		<sunnews4>

Commercial Breaks

Category	**Interval**	**Synonyms**
Weekday	25 mins	**<weekcommcycle>**
Saturday	20 mins	**<satcommcycle>**
Sunday	30 mins	**<suncommcycle>**

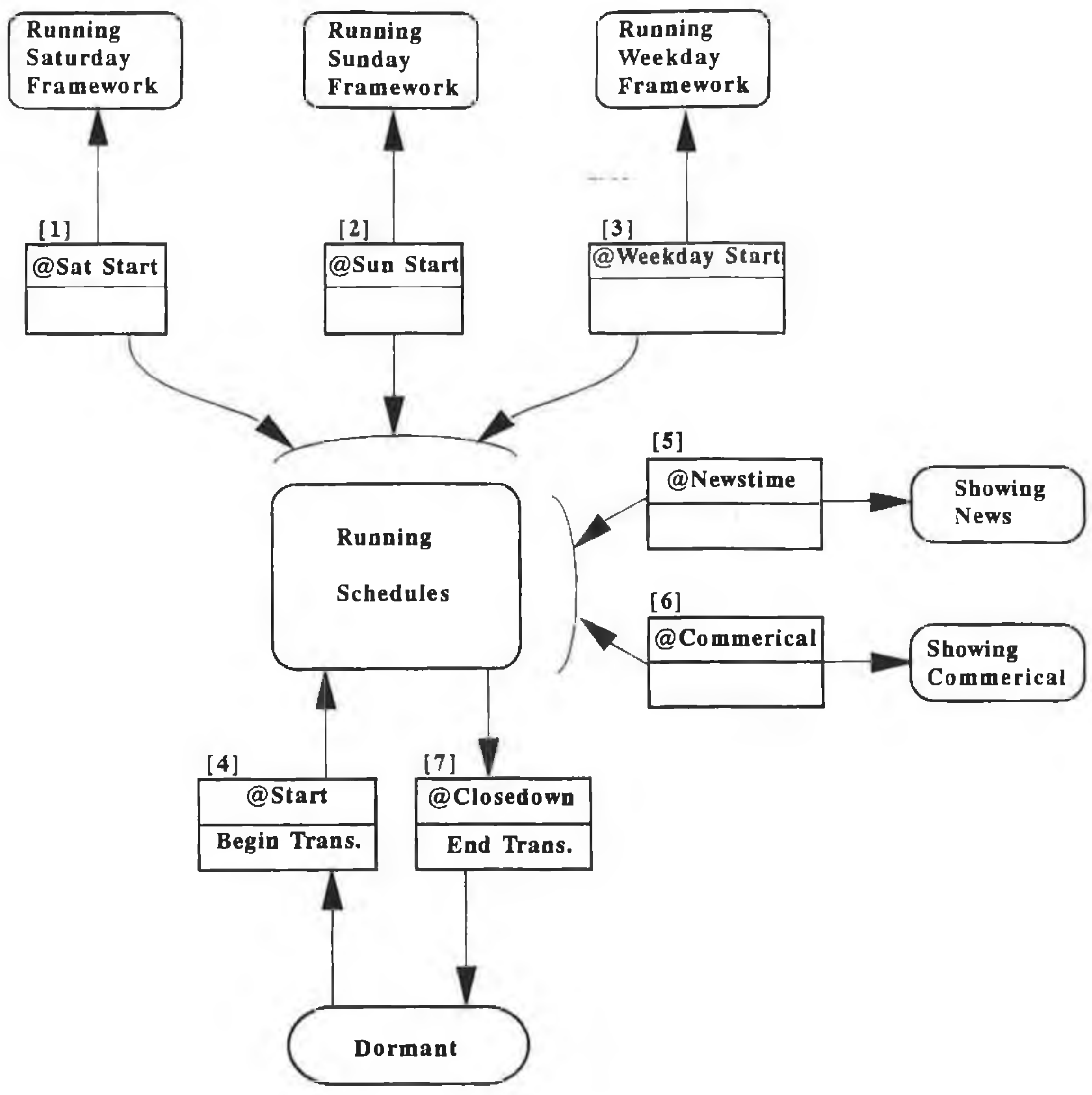
Moving on to the state-net (Figure 3) the semantics of transitions [1],[2],[3],[4] and [7] can be expressed in the following rule set.

(a) Alerter for Transmission Start

(i) Weekdays

```
DEFINE RULE weekday_start_rule IS TEMPORAL
     TIMES = <weekdaystart>,
DAYSOFWEEK = [MON..FRI]
ON OCCURRENCE
DO  Message 'Weekday Transmission Start Due'
```

SCHEDULE FRAMEWORK



PARTIAL OSA STATE-NET FOR SCHEDULING DOMAIN

Figure 3

(ii) Saturdays

```
DEFINE RULE saturday_start_rule IS TEMPORAL
     TIMES = <satstart>,
DAYSOFWEEK = [SAT]
ON OCCURRENCE
DO  Message 'Saturday Transmission Start Due'
```

(iii) Sundays

```
DEFINE RULE sunday_start_rule IS TEMPORAL
     TIMES = <sunstart>,
DAYSOFWEEK = [SUN]
ON OCCURRENCE
DO  Message 'Sunday Transmission Start Due'
```

(b) Alerter for Transmission Closedown

Allowance for two factors is necessary. Firstly, closedown is an approximate time and an adjustment for delays etc. needs to be made. Secondly, closedown on Saturdays and Sundays occurs on the following morning. There is no overlap in the schedule frameworks as can be seen from rules (i) and (iii) where in each case the temporal trigger will not fire until the TIMES clause evaluates to TRUE.

(iv) Weekday Closedown

```
DEFINE RULE weekday_close_rule IS TEMPORAL
```

```
    TIMES = <weekdayclose> + time('variance'),
DAYSOFWEEK = [MON..FRI]
ON OCCURRENCE
DO Message 'Weekday Transmission Closedown Due'
```

(v) Saturday Closedown

```
DEFINE RULE sat_close_rule IS TEMPORAL
DAYSOFWEEK = [SAT]
ON OCCURRENCE
DO Message 'Saturday Transmission Closedown Due'
atime =  TOMORROW + <satclose> +time('variance')
```

(vi) Sunday Closedown

```
DEFINE RULE sun_close_rule IS TEMPORAL
DAYSOFWEEK = [SUN]
ON OCCURRENCE
DO Message 'Sunday Transmission Closedown Due'
atime = TOMORROW + <sunclose> + time('variance')
```

(c) Alerters for Newstimes

(vii) Weekday Newstimes

```
DEFINE RULE weeknews_rule IS TEMPORAL
    TIMES = [weekdaynews1, weekdaynews2,
             weekdaynews3, weekdaynews4],
    DAYS = [MON..FRI]
    ON OCCURRENCE
```

```
DO Message' Newstime Due'
```

The same rule template can be used for Saturday and Sunday Newstimes.

(d) Alerter for Commercial Breaks

(viii) Weekday Commercial Breaks

```
DEFINE RULE weekcommcycle_rule IS TEMPORAL
     CYCLE = <weekcommcycle>,
DAYSOFWEEK = [MON..FRI]
ON OCCURRENCE
DO Message ' Commercial Break Due'
```

Similarly for Saturday and Sunday rules.

The foregoing assumes that the Weekday, Saturday and Sunday schedule frameworks are consistent throughout the year whereas in fact they may change between Summer and Winter Schedules. This is an example of where it becomes necessary to specify a **LIFETIME** clause.

What is required is a rule-set for each schedule category with the ability to set the start and end dates for each.

Summer Schedule

```
LIFETIME = [<summerstartdate>,<summerenddate>]
```

Winter Schedule

```
LIFETIME = [<winterstartdate>,<winterenddate>]
```

The syntax provided requires that this clause be included in every rule. This raises the issue of opening up the syntax to allow some clauses to apply to multiple rules.

So far, we have allowed ourselves a fairly free hand in how we expressed our rules. We have seen how, combined with a suitable analysis formalism, temporal rules can readily capture a broad range of application requirements in an almost intuitive manner. The time has now come to see if this can be reconciled with the rigorous requirements of the evolving SQL standard.

Chapter 6

Temporal Rules and the SQL Standard

In this chapter we will look at some of the highlights of the SQL-92 standard and its future successor SQL3. We will then go on to assess the implications for the temporal syntax developed in previous chapters - in particular, how this sits with what the standards have to say in relation to the specification of time and the use of database triggers.

6.1 Overview of SQL-92 and SQL3

In taking an overview of SQL-92 we will see how the new standard has resolved the previous lack of application language features; look at what is happening with Joins and the relational operators; the enhanced integrity features; treatment of privileges; the important area of transaction management; the topical issue of connections to remote databases; how SQL-92 has rationalised error handling; we will look beyond the single language database with a review of the standard's significant internationalisation features and finally focus on SQL-92's support for temporal data types.

Up to SQL-92 the standard has been based on the relational model. The next version, working title SQL3, will remain relational-centred but will add object-oriented features. More significant, from our viewpoint, will be is its introduction of a standard syntax for the definition of database rules.

6.1.1 Advanced Langauge Features of SQL-92

As we will see, the main enhancements to the existing standard come in the form of the CASE and CAST expressions, row value constructors, parameters, special values and the SQL functions.

This is part of a strategy of inclusion of programming constructs to achieve reduced dependence on host languages. These advanced value expressions are among the major enhancements in SQL-92.

1) **CASE** - a conditional expression.

2) **CAST** - a data conversion expression.

3) **ROW VALUE CONSTRUCTOR** - allows a user to deal with an entire row of data as a unit.

CASE

(a)<u>CASE and Search Conditions</u>

This allows a user to store a code and expand to a description without the need for host language intervention. For instance:

Marital Status 1=single, 2=married, 3=widowed, 4=divorced.

Allows conversion of 'null' to say '0' during retrieval as in the example which follows:

```
UPDATE employees
SET    salary = CASE
                  WHEN dept = 'video'
                     THEN salary * 1.1
                  WHEN dept = 'music'
                     THEN salary * 1.2
                  ELSE 0
                END
```

(b)<u>CASE and Values</u>

We can use shorthand version for simple value comparisons e.g.

```
SELECT title
     CASE movie_type
          WHEN 1 THEN 'Horror'
          WHEN 2 THEN 'Comedy'
          WHEN 3 THEN 'Romance'
```

```
                WHEN 4 THEN 'Western'
                ELSE NULL
           END,
              our_cost
      FROM movie_titles
```

(c)NULLIF

A special form of the CASE construct used, for instance, to allow nulls to be physically stored as, say, -1 (for some historical reasons) yet be retrieved as null.

```
NULLIF(our_cost, -1) is equivalent to

CASE WHEN our_cost = -1 THEN NULL ELSE our_cost END.
```

(d)COALESCE

This is a shorthand for an often used variation of the CASE statement.

```
COALESCE (value1,value2,value3) is equivalent to:

       CASE WHEN value1 IS NOT NULL
                THEN value1
            WHEN value2 IS NOT NULL
                THEN value2
            ELSE value3
       END
```

In other words, if value**1** is not null then the value of COALESCE is value**1**. If value**1** is null then value**2** is checked. This continues until either a non-null value value**j** is found- in which case the value returned by COALESCE is value**j** - or every value, including value**n** is found to be null - in which case, the value returned by COALESCE is null itself. The use of COALESCE in OUTER JOINS is discussed in a later section.

CAST

This is a sort of counter-balance to the inherent strong typing of the SQL language. The user can now mix exact numerics and characters in a single expression by CASTing to appropriate datatypes. Once again this reduces dependence on host language capabilities.

It is particularly useful when we want to UNION two tables whose columns may differ in datatypes or for passing eg. data of type DATE to a host language which treats dates as character strings.

ROW VALUE CONSTRUCTORS

These facilitate multiple column value comparisons. Take, for example, the situation where one wished to compare all columns in one table with all columns in another table - we can compare full rows rather than column values.

SQL-89

```
WHERE c1 = 'CA' AND c2 = 'CB' AND c3 = 'CC'
```

SQL-92 allows

```
WHERE (c1,c2,c3) = ('CA','CB','CC')
```

A row value constructor is essentially a parenthesised list of values. It can be used in many places where a value is permitted. Indeed, the individual values don't have to be literals but can be parameters/host variables or even subqueries.

Parameters in SQL

These (also called host variables) allow values to be passed between host variables and SQL statements. A ':' prefix is used.

Example:
```
UPDATE EMPLOYEES
SET salary = salary * :raise
WHERE dept = :department;
```

SQL-92 uses the following three categories of SQL parameter:

1) status parameter (returns status information)
   - SQLCODE - Currently deprecated i.e. it might

not be supported in future versions of the standard.

- SQLSTATE - New to SQL-92

2) data parameter

```
e.g. INSERT INTO MOVIE_STARS
     VALUES (:title,  :year, :last_name,
     :first_name);
```

3) indicator parameter
- returns -1 if a null value is retrieved. This is necessary because 3GLs don't understand 'null'.

- informs host program of truncation of a returned value.

Special Values

Examples of these special values are **session_user** and **current_timestamp**.

Functions

Set Functions

Count, Max, Min, Sum, Avg

Value Functions

There are three types in SQL-92:

1) Numeric Value Functions

2) String Value Functions

3) Datetime Value Functions which will be discussed later.

Numeric Value Functions

These always return a numeric value. Examples of these would be **POSITION, CHARACTER_LENGTH, OCTET_LENGTH, BIT_LENGTH** and **EXTRACT**.

Example:

EXTRACT (YEAR FROM DATE'1992-06-01')

This returns a numeric value of 1,992.

String Value Functions

For instance, **SUBSTRING** which extracts a substring, **UPPER** and **LOWER** which do case conversion and **TRIM** which strips off characters (TRIM (BOTH 'T' FROM 'TEST') yields 'ES')

SQL-92 introduces internationalization to the SQL standard and this is reflected here by the functions **TRANSLATE** and **CONVERT**.

6.1.2 Working with Multiple Tables: The Relational Operators

SQL-92 makes it easier to pull information together from across a relational database. This section introduces the new Join Operations as well as covering the use of the UNION,INTERSECT and EXCEPT operators.

<u>JOIN OPERATIONS</u>

SQL-86 and 89 handled table joins by way of SELECT...WHERE. SQL-92 expands the ways in which this can be done. There are now eight join types supported.

1) Old Style Joins

```
SELECT ... WHERE ... = ...
```

2) Cross Joins

   Produces Cartesian product of tables specified

   e.g.

```
  SELECT *
    FROM table1 CROSS JOIN table2;
```

   This is, of course, equivalent to the old style join with the absence of a WHERE clause.

3) Natural Joins

   This selects rows from two tables that have equal values in the relevant columns. For this to work column names must be the same.

```
SELECT * FROM table1 NATURAL JOIN table2
```

NATURAL can also be used to qualify other join types such as <u>inner</u>, <u>outer</u>, and <u>union.</u>

4) Condition Join

Any columns may be used to match rows from one table against those from another.

```
SELECT *
FROM table1 JOIN table2
ON table1.c1 = table2.c3
```

T1

C1	**C2**
10	15
20	25

T2

C3	**C4**
10	BB
15	DD

Joined Table

C1	**C2**	**C3**	**C4**
10	15	10	BB

5) Column Name Join

Natural Joins use all columns with the same names for matching. With column name we can limit the columns used in matching to a specified sublist.

```
SELECT *
FROM t1 JOIN t2
USING (c1,c2)
```

The old-style JOIN and the CROSS JOIN are essentially the same thing. The NATURAL JOIN uses any columns with the same name in the two source tables for an implicit equijoin, while the COLUMN NAME JOIN allows you to specify a USING clause so that you can further restrict the columns used to a subset of those with the same name.

By contrast the CONDITION JOIN lets you specify an arbitrary search condition to determine just how the rows of the two tables will be joined. In many ways, the ON clause is redundant with the where clause; however, the ON clause is useful to specify conditions specifically related to the join and use the WHERE clause for additional filtering of the rows returned.

6) The Inner Join

All of the above are known in SQL-92 terminology as INNER JOINS. For clarity this can be made explicit as in the following example.

```
SELECT *
FROM t1 INNER JOIN t2
USING (c1, c2);
```

7) Outer Joins

These differ in that they preserve unmatched rows from one or both tables depending on whether the keyword LEFT, RIGHT or FULL is used. Inner joins disregard all unmatched rows.

Left Outer Join

This preserves unmatched rows from the LEFT table.

```
SELECT *
FROM   t1 LEFT OUTER JOIN t2
ON t1.c1 = t2.c3;
```

T1

C1	**C2**
10	15
20	25

T2

C3	**C4**
10	BB
15	DD

Joined Table

C1	**C2**	**C3**	**C4**
10	15	10	BB
20	**25**	**null**	**null**

By definition each row in the first table in a LEFT OUTER JOIN must be included in the result table.

Right Outer Join

IN this case the second named table has its rows preserved.

Full Outer Join

This is a combination of left and right outer joins. The resultant table contains all of the values from each table filled out with nulls where necessary.

8) Union Join

SQL-89 limited UNION to cursor operations - SQL-92 permits UNION operations to be performed within query expressions. This is an example of how the standard lags behind commercial applications.

In order to use this operator tables must be union compatible.

```
SELECT *
FROM music_titles

UNION

SELECT *
```

```
FROM discounted albums;
```

By default UNION eliminates duplicate rows but SQL-92 allows the use of **UNION ALL** which preserves duplicate rows. This is one of the most useful features introduced in SQL-92. A query may involve a union of tables of, say, monthly sales figures where it is possible for duplicate figures to occur thus leading to an error when the figures are brought together by the UNION operator prior to producing totals.

Another variation UNION CORRESPONDING allows us to specify a subset of compatible columns for the UNION operation i.e. the non compatible columns can be excluded automatically.

Union Join works just like a full outer join, with a difference -

1. It creates a virtual table with the union of all columns from the source tables.

2. It creates a row in the new table with the values from the respective columns from each source table, with null values assigned to columns within each row from the other table.

T1	
C1	C2
10	15
20	25

T2	
C3	C4
10	BB
15	DD

Joined Table

C1	**C2**	**C3**	**C4**
10	15	null	null
20	25	null	null
null	null	10	BB
null	null	15	DD

As can be seen, unlike the FULL OUTER join, no effort is made to match columns.

INTERSECT & EXCEPT Operators

Starting with two tables - these make it easy to say which rows are common to both tables and which rows are in one table but not in the other.

INTERSECT returns all rows that exist in the virtual table formed by the intersection of the two source tables.

```
SELECT *
FROM   music_titles
```

```
INTERSECT

SELECT *
FROM discontinued_albums
```

ALL and CORRESPONDING act in the same way as with the UNION operator eg. INTERSECT CORRESPONDING (column_list) and allow the user to focus the intersect on the attributes of interest.

The syntax for EXCEPT is similar. EXCEPT returns all rows that are in the first table <u>except</u> those that also appear in the second table.

6.1.3 Constraints, Assertions and Referential Integrity

The application of constraints, or rules to the structure of a database and its contents is now the established mechanism for achieving database integrity. This move is reflected in SQL-92 where integrity is enforced via the DBMS rather than by applications running against the database.

This permits a kind of 'logic normalisation'. The integrity logic is centralised and terse making rapid rule changes possible.

Column and Table Constraints in SQL-92

For Primary Key constraint enforcement NOT NULL and UNIQUE are already present in SQL-89 but they can now be used separately.

```
CREATE TABLE movie-titles
     (title CHARACTER(30) NOT NULL,
      |,
      |
     )
```

UNIQUE

```
CREATE TABLE distributors
     (dist_name CHARACTER VARYING(30) UNIQUE)
```

CHECK

This is effectively the SQL-92 Business Rule specification mechanism and is very flexible in its range of uses.

FORMAL SYNTAX:

```
     CHECK(search_condition)
```

where search_condition ca be any valid SQL expression.

Examples

```
     CREATE TABLE movie_titles
          (...
           our_cost DECIMAL (9.2)
               CHECK(our_cost < 100.00),
            |
            |
            )
```

This places a domain constraint on the specified column. What happens when the table is empty? Well, the SQL standard holds that if there are no rows present then no row violates the constraint.

Multiple CHECK clauses can be specified for each column and they are then effectively ANDED together.

So far we have been looking at a single table. CHECK's search condition parameter can just as easily reference data from other tables for a domain constraint. Importantly, the constraint is dynamic - the CHECK is run again after any changes to these other tables. Indeed this also holds if the constraint is based on other column values in the same table.

<u>RANGE OF VALUES</u>

It is often desirable to set down bounds of reasonable values for a given column e.g. employee ages or salaries.

SQL-92 provides the BETWEEN qualifier to the CHECK option for this purpose.

```
     CHECK (age BETWEEN 17 and 65)
```

Similarly, a positive or negative value constraint can readily be specified as in 'BETWEEN -10000 AND 0'.

<u>LIST OF VALUES</u>

A set of acceptable values can be specified for any column as follows.

```
CREATE TABLE movie_titles
               (title          CHAR(30) NOT NULL,
                |
                    |
                    movie_type    CHAR(10)
                        CHECK (movie_type IN
                              ('Children', 'Comedy','Musical',
                               'Romance', 'Western',
                               'Adventure','Other')),
                    available     CHAR(1)
                        CHECK (available IN ('Y','N'))
```

The construct used for the 'available' attribute highlights the fact that SQL-92 does not support true Boolean values.

SQL-92 CHECKs are not limited to this sort of hard-coded format. The following is also valid for the 'movie-type' constraint:

```
     CHECK (movie_type = SOME
           (SELECT * FROM category))
```

<u>CONSTRAINT NAMES</u>

SQL-92 permits the optional specification of names for user defined constraints. The previous example could be re-cast in the following format.

```
CREATE TABLE ...
        |
        |
        CONSTRAINT check_movie_type
        CHECK (movie_type IN
              ('Children'.......
```

Why bother with them? One obvious advantage in allowing users to specify their own constraint names is that constraint violation messages become much more meaningful than is possible with system generated defaults.

The name is also used as a handle to SET the constraint to DEFERRED or IMMEDIATE or to DROP the constraint.

Constraints are, by default, NOT DEFERRABLE - they are checked at the end of each SQL statement. Alternatively, DEFERRABLE, can be specified which effectively delays constraint checking until COMMIT time.

ASSERTIONS

An assertion is a constraint which is not tied to a particular table (i.e. part of a CREATE TABLE statement) but rather enforces a rule across some portion of the schema.

For instance, if we wished to limit the total value of video and CD stock to £50,000 a suitable assertion would be:

```
     CREATE ASSERTION maximum_inventory
          CHECK ((SELECT SUM(our_cost)
                  FROM   movies)
                +(SELECT SUM(our_cost)
                  FROM music_titles)
                < 50,000 )
```

PRIMARY KEY

SQL-92 provides a very natural and direct approach to the designation of a unique identifier allowing the use of the keywords PRIMARY KEY within the CREATE TABLE statement.

FOREIGN KEY

The basic requirement is for the foreign key to include enough columns in its definition to uniquely identify a row in the referenced table and that the referencing table never contains values in these columns which are not represented in the primary key values of the referenced table.

```
     CREATE TABLE movie_stars
          (
           |,
             |
           CONSTRAINT titles_fk FOREIGN KEY (movie_title)
               REFERENCES movie_titles (title)
```

In other words the table movie_stars has a foreign key (movie_title) which must correspond to the primary key (title) of the movie_titles table.

REFERENTIAL CONSTRAINT ACTIONS

These are new to SQL-92 and mean that the DBMS has to ensure that tables are kept in 'sync' over time. SQL-92 not only ensures that referenced rows cannot be deleted but provides the option of allowing the deletion to go ahead on condition that a pre-defined replacement is specified.

```
CREATE TABLE movie_titles
     ( title        CHAR(30) NOT NULL,
       |
       |
       distributor  CHAR VARYING(25)
          REFERENCES distributors
       |
       |)
```

What this means is that as long as we have titles for a distributor that firm cannot be deleted from the distributors table. However, the following may be desirable. We may want the freedom to drop any distributor and transfer their titles to a default supplier.This can be achieved with only minor modifications.

```
CREATE TABLE movie_titles
     (title    CHARACTER(30) NOT NULL
      |
      |
      distributor CHARACTER VARYING(25)
                  DEFAULT 'Big East, Inc.'
          REFERENCES distributors
            ON DELETE SET DEFAULT,
      |
      |
      )
```

SET NULL

SQL-92 allows the direct specification of a 'set foreign key to null' strategy for the maintenance of referential integrity. The syntax is as follows.

```
CREATE TABLE movie_titles
     (title         CHARACTER(30) NOT NULL
      |
      |
      distributor   CHARACTER VARYING(25)
          REFERENCES distributors
               ON DELETE SET NULL,
      |
      |
      )
```

CASCADE

So far we have been dealing with the case of simple deletion from the referenced table but SQL-92 handles updates as well. For instance, if a name forming part of the primary key of a referenced table had to be updated it can now be left to the DBMS to make the necessary amendments in the referencing tables to maintain the linkages.

```
CREATE TABLE movie_titles
     (title         CHARACTER(30) NOT NULL,
      |
      |
      |
      distributor   CHARACTER VARYING(25)
          REFERENCES distributors
               ON UPDATE CASCADE,
      |
      |
      )
```

If a distributor changed its business name this would be automatically reflected in any rows of movie_titles which referenced that distributor in its original guise. Cascades can, of course, involve multiple tables with the updates of one table triggering off a chain reaction in line with the foreign key constraints defined at the CREATE TABLE stage.

6.1.4 Privileges, Users and Security

This is concerned with the control of access to various categories of SQL-92 database objects:

> Tables, Columns, Views, Domains, Character Sets
> Collations and Translations.

The last three are new in SQL-92 and are discussed in section 1.9 (Internationalisation).

Privileges could not be revoked in the SQL-89 standard but this has been rectified in SQL-92. The concept of granting a privilege to PUBLIC is also now supported.

<u>GRANT</u>
Syntax in SQL-92

```
GRANT privilege_list
ON    object
TO    user_list [WITH GRANT OPTION]
```

The privilege_list can include

> SELECT/DELETE/INSERT/UPDATE/REFERENCES and USAGE

In all but the last case the 'object' would be a table. In the case of the USAGE privilege the 'object' would be a DOMAIN, CHARACTER SET, COLLATION or a TRANSLATION.

'REFERENCES' limits the use of references via foreign keys to the specified table to specified users. This is necessary because a user could deduce the contents of this table by trial and error using the referential integrity check as an indicator.

<u>USAGE</u>

Only those users granted USAGE privilege on the domains, character sets, collations and translations can 'see' them and use them in their data definitions or in their SQL programs. This avoids problems such as an unauthorised user issuing a DROP domain statement.

The privileges on a view are equivalent to the privileges held on the base table(s) on which it is defined. In SQL-92 the view privileges are automatically updated to reflect changes in adjusted rights to the base table(s) i.e. as if the view were re-created after each change in privilege.

<u>REVOKE</u>

Two qualifiers, RESTRICT and CASCADE are provided with REVOKE. RESTRICT will disable REVOKE if the privilege in question was passed on. With CASCADE the privilege is removed from all users to whom it was passed on through the WITH GRANT option. Usefully, SQL-92 allows a user to revoke the WITH GRANT OPTION privilege.

As can be seen, although security in SQL-92 is syntactically simple - just GRANT and REVOKE, there is a good deal of complexity introduced by the WITH GRANT OPTION clause.

6.1.5 Transaction Management

SQL has never used an explicit statement to start a transaction but each ends with a COMMIT or ROLLBACK statement.

In SQL-92 a transaction has three characteristics

(a) Mode
(b) Isolation Level
(c) Diagnostics Area

The default which the standard provides is

(a) Read and update permitted
(b) Maximum isolation from concurrent transactions
(c) Diagnostics area with default size is set up

Any deviation from these norms is specified through the SET TRANSACTION statement. Transactions can be set to READ ONLY, READ WRITE with ISOLATION levels ranging from the lowest level of READ UNCOMMITTED, through READ COMMITTED, REPEATABLE READ up to the highest level possible - SERIALIZABLE.

Example:

```
SET TRANSACTION
```

```
READ ONLY,
ISOLATION LEVEL READ UNCOMMITTED
```

i.e. no updates will be allowed within the transaction and a dirty read is acceptable. This could be useful for some reports such as rough statistical queries where we want to scan a table without causing any locking delays for other users. An isolation level of READ COMMITTED will eliminate the dirty read phenomenon i.e. only committed transactions are read.

The REPEATABLE READ level guarantees that if the same row is read more than once within a transaction then its value will be the same. SERIALIZEABLE is the highest level and the one provided by current commercial database systems.

6.1.6 Connections and Remote Database Access

There is a move towards the adoption of distributed database environments. SQL-92 addresses the question of multiple connections although there is more work needed to achieve seamless use of heterogenous databases.

SQL-89 did not have the concept of a session in which your application ran or of a connection from your application to a session. This requirement has arisen due to the general adoption of the client-server paradigm.

Establishing connections

In SQL-89 a DBMS was expected to accept SQL statements without any prior definition of a context. The context was provided by DBMS specific software. SQL-92 recognizes the need for a degree of DBMS independent set-up and it is therefore worthwhile looking at this process in a little more detail.

First the user needs to establish a connection between the program (client) and the DBMS (server). SQL-92 specifies that the connection is to a default server (left to the implementation to define). So, if the user does not execute an explicit CONNECT statement SQL-92 executes one for them against the defined server.

The presence of a CONNECT statement allows you to establish a connection to several servers at a time. This is where the notion of a session comes in. Each connection has a session associated with it which is analogous to a user having multiple logins in, say, VMS.

The session associated with the current connection is called the current session whereas other connections are known as dormant sessions. The SET CONNECTION statement is used to switch between sessions.

For efficient use of system resources a DISCONNECT statement is provided for use in client programs. One problem with early client-server implementations was the possibility of connections to clients being left hanging if someone decided to re-boot their PC. For this reason a tidy up is carried out when a client program terminates which will disconnect any sessions not explicitly released from the client side.

How the client exploits the ability to have multiple sessions is left as an implementation-dependent feature. A transaction may be limited to one connection in which case disconnecting or session switching will raise an error. Alternatively, the implementation may treat statements executed across all of the connections as part of a global transaction. In summary, CONNECTING starts a session, DISCONNECTING ends a session.

6.1.7 Diagnostics and Error Management

This is concerned with error and condition reporting and uses the two functions SQLCODE and SQLSTATE which allow the developer to include robust error handling in SQL applications. SQL-92 classifies and interprets errors and supplies warning descriptions. The earlier standard reported errors via the status parameter SQLCODE where 0 represented successful completion, 100 meant a no-data condition (i.e. no rows found on which to operate) and all negative values represented an error condition. As implementors were free to use their own set of negative numbers for specific error conditions portability was compromised. The approach taken in the SQL-92 standard was to create a second parameter SQLSTATE rather than attempting to impose a retrospective standard for SQLCODE. Indeed, SQLCODE has been deprecated i.e. it will eventually be deleted from the standard.

Pre-defined values are supplied for SQLSTATE and rather than using an integer like SQLCODE it uses a 5 character string (the uppercase letters A-Z and the digits 0-9). For greater semantic power the code is divided into a two-character <u>class code</u> and a three-character <u>subclass code</u>. The standard reserves for itself all class codes beginning with A-H or 0-4. For these classes any subclass code starting with the same character is standard-defined. Implementors are free to define class codes beginning with the remaining letters and digits.

On its own, SQLSTATE has limited powers in reporting error situations and SQL-92 has addressed this issue with the GET DIAGNOSTICS statement. These diagnostics are retrieved from a diagnostics area which is structured so as to provide header information on the last SQL statement executed as a whole and detail entries for each error, warning or success code associated with that statement. The diagnostics area is emptied at the start of each new SQL statement.

6.1.8 Internationalisation

Before SQL-92, database products were designed for English language use based around 8-bit ASCII characters.

Unfortunately, the Japanese language, for instance, requires not only support for thousands of characters but requires more bits to encode each character. The challenge facing SQL-92 was to support not just one but multiple languages at the same time - a standard for internationalised DBMS.

Character Sets, Collations and Translations

In order to be meaningful every character string has a character set associated with it. Generally, the user has no control over which character set is used as vendors currently decide this in advance.

A character set has three basic attributes:

- The repertoire of characters or what characters it is capable of representing.

- The form of use or method of representation e.g. one, two bytes per character.

- The default collation or sort order. This assumes we are comparing two strings from the same repertoire a default which can be over-ridden.

For example:

```
CREATE TABLE t1 (
     col1      CHARACTER(10)
     col2      CHARACTER  VARYING(50)  CHARACTER  SET
               KANJI
                 )
```

Further, SQL-92 supports the translation of character strings from one character set to another e.g. from Hebrew to Latin characters.

6.1.9 The Specification of Time in SQL-92

The introduction of standards for datetimes and intervals posed some difficulties, not least the existence of myriad implementation conventions, and consequently a canonical form of expression was finally adopted. We will start by looking at how datetimes are handled before moving on to the topic of intervals.

Datetimes

SQl-92 Data Type	**Literal Example**
DATE	DATE '1929-10-29'
TIME	TIME
TIMESTAMP	TIMESTAMP '1987-10-19 16:00:00.00'
TIME WITH TIME ZONE	TIME '10:45 - 07:00'
TIMESTAMP WITH TIME ZONE	TIMESTAMP '1993-04-05 03:00:00 + 01:00'
INTERVAL	INTERVAL '10:30' MINUTE TO SECOND

Note: The specification of time zones uses a plus/minus UCT offset. Universal Coordinated Time (UCT) replaces the earlier GMT standard.

A precision can also be specified for temporal data types eg.

TIME(2)	TIME '14:35:10.55'
TIME(3)	TIME '12:20:00.000'

DateTime Value Functions

These are functions which return a value of type DATE.

<u>Examples</u>

Function	**Returns**
CURRENT_DATE	The current date
CURRENT_TIME(2)	The current time to 2 decimal places
CURRENT_TIMESTAMP	The current timestamp for the timezone of your session in the following format:

Year:Month:Day:Hrs:Mins:Secs:Fraction of a second to specified precision

Datetime Value Expressions

These operate on date-oriented data types DATE, TIME, TIMESTAMP and INTERVAL

The result is always a datetime.

```
CURRENT_DATE + INTERVAL '1' DAY gives tomorrow.

TIME '10:45:00' AT LOCAL
- Store 10:45 in my local time zone.

TIME '10:45:00' AT TIME ZONE INTERVAL '+09:00' HOUR TO
MINUTE
- Store 10:45 in Tokyo time.
```

Interval Value Expressions

If you subtract one datetime from another you will get an interval as a result. SQL-92 divides intervals into two categories: year-month intervals and day-time intervals. It does not allow these to be mixed in a single expression.

A year-month expresses an interval as a number of years and an integral number of months. This is an exact representation as a year always has only 12 months.

Similarly, a day-time interval expresses an interval as a specific number of days,hours, minutes and seconds - there are always the same number of hours in a day, minutes in an hour an so on. However, we cannot know how many days are in a month unless we know which month it is.

The following rules govern intervals and datetimes in expressions:

1) datetime - datetime -> interval

2) datetime - interval -> datetime

3) datetime + interval -> datetime

4) interval *¦/ scalar -> interval

Example:

CURRENT_DATE - DATE_RELEASED
- Gives the time for which a movie or CD has been available.

More correctly:

(CURRENT_DATE - DATE_RELEASED) YEAR TO MONTH

The qualification YEAR TO MONTH is required because subtraction of two dates can result in an invalid interval (an interval that is neither a year-month interval nor a day-time interval).

6.1.10 SQL3 - A Look to the Future

New versions of SQL generally appear every 3 years. The next version of the standard has the working title SQL3 and its expected publication date is 1995/96.

The thrust of SQL3 [EISE93] is twofold,

1) enhanced relational capabilities.
2) support for objects.

Enhanced Relational Support

Triggers

SQL-92 lacks a definition of database rules (triggers). The standards committee did not foresee their rapid uptake and vendors have had to use the SQL3 draft which does include support for triggers.

The SQL3 definition is as follows:

```
CREATE TRIGGER trigger_name time event
     ON table_name action
```

The 'time' is not a clock time but simply specifies that the action happens either BEFORE or AFTER the event. An 'event' can be an INSERT,DELETE or UPDATE. The 'table-name' refers to the table against which the event occurs.

A 'granularity' can be specified for the action with a default of FOR EACH STATEMENT. The alternative of FOR EACH ROW causes the action to be performed for each row inserted, deleted or updated by the event whereas the default causes a single firing of the rule. This is very significant as the default maintains the set-level nature of SQL statements i.e. the default behaviour of a trigger is to react to the INSERTION, DELETION etc. of <u>sets</u> of rows rather than single rows. Sybase is an example of a commercial product which currently supports this level of trigger abstraction.

<u>Recursive Operations</u>

SQL3 introduces the RECURSIVE UNION operation which implements a long-awaited <u>'bill of materials'</u> functionality. It effectively allows a user to traverse a tree of rows in a database.

The syntax proposed is:

```
(initial RECURSIVE UNION correlation_names
     [columns] iteration
     [search] [limit])
```

The term 'initial' is a query expression specifying the starting point of the search. The correlation names are used in the accumulation of rows into the result. The term 'iteration' is a query expression that set out how child rows of any parent row are to be located.

The search strategy (DEPTH FIRST, BREATH FIRST etc.) is specified by the 'search' parameter. Finally 'limit' will allow the user to control how long the search should continue thus avoiding runaway queries.

New Data Types

Two new data types are so far proposed in SQL3: Enumerated and Boolean.

For instance, an attribute of MOVIE_TITLES might be IN_STOCK which would be defined as Boolean (True or False).

The enumerated data type allows the definition of a domain which has a fixed set of values. For example:

```
CREATE DOMAIN movie_types
     (children, comedy, horror, musical)

CREATE TABLE movie titles
     (title    CHARACTER VARYING(30),
               type MOVIE_TYPES,
      ...)



.

.

.

INSERT INTO movie_titles VALUES
```

```
(...,movie_types::musical,...)
```

Note the use of the '::' to specify the enumerated value within the domain.

Other features such as stored procedures, so vital in distributed environments, are also expected.

Support for the Object Paradigm

Intensive work is on-going aimed at lifting SQL beyond its relational database roots and providing the basis for combined relational and OOA features in one DBMS. To date the following concepts have received attention and can be expected to be fully supported in the new standard.

- User-defined abstract data types
- Encapsulation
- Object Identity
- Unification of SQL tables and abstract data types
- Subtypes and Supertypes
- Inheritance of type attributes and methods
- Parameterised types.
- Type generators
- A control language for implementation of methods
- Computational completeness
- Functions and procedures written in SQL
- Static and dynamic binding of methods
- "Built-in" data type generators (eg for sets, multisets

and lists)
- SQL variables, temporary variables.

Add to this new standards for multimedia and GIS Geographical Information Systems) and it can be expected that SQL3 will involve a quantum leap in complexity within the standard. To deal with this, a layered approach is emerging which will most likely see these extensions built on top of existing standards but it is not possible to say how things will turn out for another couple of years.

So far, we have outlined SQL-92 and established the direction in which it is evolving. It is now time to bring our temporal database rules back into the picture, viewed, this time, against the more formal backdrop of the SQL standard.

6.2 Implications for the Temporal Syntax

We next look at how triggers are defined in SQL and how this sits with our Working Syntax. We then evaluate how successfully SQL can capture the semantics of the sample application domains and finally, define any syntactic extensions highlighted by this process.

6.2.1 Trigger Definition

In this section we will review the SQL3 trigger definition syntax in detail, teasing out the underlying elements. The result of this exercise will be used as a framework for suggested extensions to the draft standard.

We will take as our starting point an example given in [EISE93]:

Assume the two tables:

```
     tab1 (a CHAR(2),b CHAR(2))
     tab2 (c CHAR(2),d CHAR(2))
```

and the trigger definition

```
CREATE TRIGGER tr1
AFTER UPDATE OF a
ON tab1
```

```
REFERENCING OLD AS pre_a, NEW AS post_a
UPDATE tab2 SET d = post_a WHERE
     d = pre_a
FOR EACH ROW;
```

This can be analysed and abstracted as follows:

CREATE TRIGGER tr1 **(trigger_name)**
AFTER **(time)** UPDATE OF a **(event)**
ON tab1 **(database object)**
update tab2 ... **(action)**

giving -

CREATE TRIGGER **(trigger_name)**
(time) (event)
ON **(database object)**
(action)

where

(time) represents the time relationship between the occurrence of the triggering event and the performance of the associated action. In the draft SQL3 syntax this time relationship is limited to the form 'immediately before' or 'immediately after' the event. The syntax proposed here seeks to extend this relationship.

(event) The intention of the proposed syntax is also to extend the set of triggering events to include temporal events. So far, it seems as if SQL3 will limit events to "INSERT, UPDATE and DELETE actions on a specified base table" [EISE93].

In essence, SQL3 defines a rule in terms of a trigger name, a triggering event, an action initiated by the trigger and an expression of how the trigger and action relate in time.

More formally,

```
  SQL3_TRIGGER_STATEMENT ::= CREATE TRIGGER

                             <Trigger_Name>
                             <Event/Action_Time_Relationship>
                             <Triggering_Event>
                             <Triggered_Action>
```

In the following section we will re-visit the worked examples given in the previous chapter and attempt to re-cast these in a form compliant with SQL-92 time specification and SQL3's emerging framework.

6.2.2 Application to Worked Examples

A representative sample of statements from the Personnel and Programme Scheduling worked examples have been selected. In each case the working syntax used is given before being dismantled into the elements of an SQL3 Trigger and finally re-assembled taking care to meet the existing requirements of SQL-92. For brevity, some of the following examples assume the promised inclusion of remote procedure calls (RPCs) in the eventual SQL3 standard.

Example 1

This rule expresses a time constraint of 3 days within which the induction procedure for a new employee must begin.

Working Syntax

```
DEFINE RULE check_induction_init IS TEMPORAL
ON APPEND TO Employee
DO exec_proc check_induction_begun (emp#)
     atime = etime + '3days'
```

Trigger Elements

Trigger Name : check_induction_init

Triggering Event	: Append to Employee
Triggered Action	: exec proc check_induction_begun (emp#)
Event/Action Time Rel.	: Perform the action 3 days after the triggering event.

Requires an Extended SQL3 Syntax

First cut :

```
     CREATE TRIGGER check_induction_init
     AFTER (+ 3 days) INSERT INTO Employee
     exec proc check_induction_begun (emp#)
```

After adjusting for SQL-92 time interval specification:

```
     CREATE TRIGGER check_induction_init
     AFTER INTERVAL + '3' DAY INSERT INTO EMPLOYEE
     exec_proc check_induction_begun (emp#)
```

Example 2

An employee on probation has a review every six months for a period of 2 years. This rule is set in motion when an employee is hired or promoted.

Working Syntax

```
DEFINE RULE probation_review IS TEMPORAL
          CYCLE = '6 month'
          LIMIT = '4 cycles'
ON OCCURRENCE
DO INSERT INTO Pending_Review
VALUES (emp#, 'Due since', date(etime))
```

Trigger Elements

Trigger Name	: probation_review
Triggering Event	: An absolute date which occurs once every six months for a period of two years.
Triggered Action	: INSERT INTO Pending_Review VALUES (Emp#, 'Due since', date)
Event/Action Time Rel.	: Immediately after.

Required an Extended SOL3 Syntax

```
CREATE TRIGGER probation_review
AFTER (cycle ('6 months'),(limit('4 cycles'))
INSERT INTO Pending_Review
VALUES (emp#, 'Due since',CURRENT_DATE)
```

Notes: 1. Cyclic operations are not supported in the SQL-92 or SQL3 standards.

2. CURRENT_DATE is an SQL-92 datetime value function.

Example 3

Employees move to the next point on their current pay scales on their designated increment date if they are not already on the maximum point and if their work is satisfactory.

<u>Working Syntax</u>

```
DEFINE RULE increment_rule IS TEMPORAL
     DATES = employee.increment_date
ON OCCURRENCE
WHERE {not on MAX and satisfactory}
DO exec proc pay_rise (emp#)
```

<u>Trigger Elements</u>

Trigger Name	: increment_rule
Triggering Event	: An absolute date - the employee's official increment date.
Triggered Action	: Move up to next point on pay scale

on condition that not on Maximum point of scale already and work is satisfactory.

Event/Action Time Rel. : Immediately after.

<u>Requires an Extended SQL3 Syntax</u>

```
CREATE TRIGGER increment_rule
AFTER employee.increment_date
WHERE {not on max and satisfactory}
exec proc pay_rise (emp#)
```

Adjusted for SQL-92 compliant specification of parameters this becomes:

```
CREATE TRIGGER increment_rule
AFTER DATE :employee.increment_date
WHERE {not on max and satisfactory}
exec proc pay_rise (emp#)
```

Example 4

An employee must retire at 65 years of age.

<u>Working Syntax</u>

```
DEFINE RULE retirement_rule IS TEMPORAL
          DATES = employee.retire_date
ON OCCURRENCE
DO exec proc retirement_procedure (emp#)
```

<u>Trigger Elements</u>

Trigger Name	: retirement_rule
Triggering Event	: An absolute date - the employee's retirement date.
Triggered Action	: Retirement procedure initiated
Event/Action Time Rel.	: Immediately after.

<u>Requires an Extended SQL3 Syntax</u>

```
CREATE TRIGGER retirement_rule
AFTER employee.retirement_date
exec proc retirement_procedure (emp#)
```

Adjusted for SQL-92 :

```
CREATE TRIGGER retirement_rule
AFTER DATE :employee.retirement_date
```

```
exec proc retirement_procedure (emp#)
```

Example 5

Ten months before an employee is due to finish a career break preparations for their return must commence.

Working Syntax

```
DEFINE RULE career_break IS TEMPORAL
     DATES = employee.return_date - '10 months'
ON OCCURRENCE
DO INSERT INTO Short_List
VALUES employee#, grade, return_date
```

Trigger Elements

Trigger Name	:	career_break
Triggering Event	:	An absolute date - given by a date expression representing a date ten months prior to the employee's expected return date.
Triggered Action	:	INSERT INTO Short_list VALUES employee#, grade, return_date

Event/Action Time Rel. : Immediately after.

Requires an Extended SQL3 Syntax

```
CREATE TRIGGER career_break
AFTER  employee.return_date - '10 months'
INSERT INTO Short_list
VALUES employee#, grade, return_date
```

After application of SQL-92 this becomes:

```
CREATE TRIGGER career_break
AFTER DATE (:employee.return_date - INTERVAL '10'
            MONTH)
INSERT INTO Short_List
VALUES employee#, grade, return_date
```

Example 6

The following rule expresses the real-time constraint on the period for which an employee can be kept off the payroll for any reason. When an employee is suspended there should be no delay in re-instatement once the suspension period has elapsed.

Working Syntax

```
DEFINE RULE suspended IS TEMPORAL
ON UPDATE TO Employee.status
WHERE new.status = 'suspended'
DO exec proc revoke_suspension (emp#)
     atime = etime + (suspension_period)
```

Trigger Elements

Trigger Name	: suspended
Triggering Event	: Employee.status being set to 'suspended'
Triggered Action	: Revoke the suspension
Event/Action Time Rel.	: The action is to be performed at a time 'suspension_period' later than the event.

Requires an Extended SQL3 Syntax

```
CREATE TRIGGER suspended
AFTER (+ suspension_period) UPDATE OF status ON
Employee
```

```
REFERENCING OLD as pre_status, NEW AS post_status
WHERE post_status = 'suspended'
exec proc revoke_suspension (emp#)
```

In SQL-92 compliant terms

```
CREATE TRIGGER suspended
AFTER INTERVAL :suspension_period DAY UPDATE OF status
ON Employee
REFERENCING OLD as pre_status, NEW AS post_status
WHERE post_status = 'suspended'
exec proc revoke_suspension (emp#)
```

Example 7

Moving on to our hypothetical Programme Scheduling worked example, we begin with a simple alerter for transmission start times.

<u>Working Syntax</u>

```
DEFINE RULE weekday_start_rule IS TEMPORAL
     TIMES = <weekdaystart>,
DAYSOFWEEK = [MON..FRI]
ON OCCURRENCE
DO Message 'Weekday Transmission Start Due'
```

Trigger Elements

Trigger Name	: weekday_start_rule
Triggering Event	: An absolute time 'weekdaystart' for days in the range MON..FRI
Triggered Action	: Alerter message
Event/Action Time Rel.	: Immediately after

Requires an Extended SQL3 Syntax

```
CREATE TRIGGER weekday_start_rule
AFTER (weekdaystart, DAYSOFWEEK([MON..FRI])
Message 'Weekday Transmission Start Due'
```

SQL-92 does not have the sort of datetime functions found in commercial DBMS for extracting the day of the week from an absolute date and consequently the above syntax has been left unaltered.

Example 8

The following alerter for newstimes is triggered by a set of times on a range of days.

Working Syntax

```
DEFINE RULE weeknews_rule IS TEMPORAL
```

```
          TIMES = (weekdaynews1, weekdaynews2,
                   weekdaynews3, weekdaynews4),
     DAYSOFWEEK = [MON..FRI]
     ON OCCURRENCE
     DO Message 'Newstime Due'
```

<u>Trigger Elements</u>

Trigger Name	: weeknews_rule
Triggering Event	: An absolute time in the set (weekdaynews1,weekdaynews2, weekdaynews3,weekdaynews4) for days in the range MON..FRI
Triggered Action	: Alerter message
Event/Action Time Rel.	: Immediately after

<u>Requires an Extended SQL3 Syntax</u>

```
     CREATE TRIGGER weeknews_rule
     AFTER TIMES(weekdaynews1,weekdaynews2,weekdaynews3,
                 weekdaynews4),DAYSOFWEEK([MON..FRI])
     Message 'Weekday Newstime Due'
```

The SQL-92 standard allows SETS to be defined as a comma delimited list enclosed in brackets eg.

```
.. WHERE type IN ('horror','comedy','western')
```

so here we could write

```
     AFTER (TIME :weekdaynews1, TIME :weekdaynews2,
            TIME :weekdaynews3, TIME :weekdaynews4)
```

although, again, an equivalent of the necessary DAYSOFWEEK function is lacking.

Example 9

In the final example from our worked examples we look at rule lifetimes. We wish to make the previous rule apply to the summer time schedule only.

Working Syntax

```
     DEFINE RULE weeknews_rule IS TEMPORAL
          TIMES = (weekdaynews1, weekdaynews2,
                   weekdaynews3, weekdaynews4),
     DAYSOFWEEK = [MON..FRI],
     LIFETIME   = (summerstart_date, summerend_date)
     ON OCCURRENCE
     DO Message 'Newstime Due'
```

Trigger Elements

Trigger Name	: weeknews_rule
Triggering Event	: An absolute time in the set (weekdaynews1,weekdaynews2, weekdaynews3,weekdaynews4) for days in the range MON..FRI during the lifetime of the rule.
Triggered Action	: Alerter message
Event/Action Time Rel.	: Immediately after

<u>Requires an Extended SQL3 Syntax</u>

```
CREATE TRIGGER weeknews_rule
AFTER TIMES(weekdaynews1,weekdaynews2,weekdaynews3,
           weekdaynews4),DAYSOFWEEK([MON..FRI]),
           LIFETIME(:summerstart_date,:summerend_date)

Message 'Weekday Newstime Due (Summer Schedule)'
```

SQL3 does not support the concept of a lifetime for a trigger. The above syntax is notional.

6.2.3 The Extended Syntax

Having reviewed the semantics of our worked examples we are now in a position to summarize the required syntactic extensions which they highlight. We will take the draft SQL3 trigger definition statement as our starting point, merge in the temporal clauses developed in previous chapters, and finally, give the complete extended syntax proposed.

The draft SQL3 standard gives the following syntax for a database trigger:

```
TRIGGER_STATEMENT ::= CREATE TRIGGER <trigger_name>

                      <time>

                      <event>

                      ON table_name [referencing]

                      <action>
```

where:

<trigger_name>	= The name of the trigger
<time>	= BEFORE|AFTER indicating whether the trigger is fired before or after the specified event occurs. **We will need to extend this to allow an optional delay between the event and the action.**
<event>	= INSERT|DELETE|UPDATE, indicating that execution of an INSERT statement, a DELETE statement or an UPDATE statement will fire the trigger. **We will need to extend this to cater for temporal events.**
<table_name>	= Identifies the table which the DBMS must watch for the triggering event.
[referencing]	= For UPDATE statements it may be necessary to refer to the pre and post values to allow comparisons within the trigger statement. The full syntax is:

```
                  REFERENCING OLD [AS]
                  old_correlation_name
                  [NEW [AS] new_correlation_name]

                  Or

                  REFERENCING NEW [AS]
                  new-correlation_name
                  [OLD [AS] old_correlation_name]


<action>      =   [WHEN (search_condition)]

                  (statement [,statement]...)

                  [granularity]
```

where

```
(search_condition) = an expression
                     controlling the
                     conditional firing
                     of the trigger.
```

and

```
[granularity]     = FOR EACH ROW|FOR EACH
                    STATEMENT
```

The trigger fires for each row inserted/deleted/updated or once for the statement as a whole.

So far, the syntax allows only for INSERT, UPDATE and DELETE type events on a specified table, we will refer to them as 'data_events'. Actions occur immediately BEFORE the event or immediately AFTER the event. We can take zero therefore as the default value of the first clause we will introduce, an optional **delay** clause where 'delay' is a time period specified as an SQL-92 compliant datetime.

In order to bring 'time_events' into the picture we need to re-state 'event' as follows:

```
<event>       ::= <time_event> | <data_event>


<time_event> ::= [CYCLE = <time_interval>,
                  LIMIT = <max_cycles>]

                 [,TIMES = <time_range>|<specific_times>]
```

```
[,DAYSOFWEEK = <specified_days>]


[,DATE|DATES =
   <date_range>,<specific_dates>]

[,LIFETIME  = <start_date>, <finish_date]
```

The parameters used in <time_event> have the following meanings:

CYCLE

CYCLE is the time interval between activations of a recurrent rule where the firing is on a regular time basis.

```
[, CYCLE = <time_interval> ]
```

<time_interval>

The parameter 'time_interval' is expressed in terms of

```
months:days:hours:minutes:seconds
```

LIMIT

LIMIT is the number of cyclic recurrences defined for a rule. It is one method of specifying a lifetime for a rule.

```
[, LIMIT = <max_cycles> ]
```

<max_cycles>

The parameter 'max_cycles' is an integer value representing the number of cyclic recurrences which the rule is allocated. If no LIMIT is defined then the default value is infinity ie. the rule will continue to recur until it is deleted.

TIMES

TIMES specifies either an effective time range for cyclic rules or explicit times at which to fire for other rules.

```
[, TIMES = <time_range>|<specific_times>]
```

<time_range>

The parameter 'time_range' is an interval or series of intervals during which cyclic firing of a rule is enabled.

<specific_times>

The parameter 'specific_times' allows the specification of one or more explicit times which will serve as a trigger for non-cyclic rules.

DAYSOFWEEK

DAYSOFWEEK specifies the days of the weeks on which the rule is enabled.

```
[, DAYSOFWEEK = <specified_days>]
```

<specified_days>

The parameter 'specified_days' is a list of day identifiers.

DATES

DATES specifies either an effective date range for cyclic rules or explicit times at which to fire for other rules.

```
[, DATES = <date_range>|<specific_dates>]
```

<date_range>

The parameter 'date_range' is a date interval or series of intervals during which cyclic firing of a rule is enabled.

<specific_dates>

The parameter 'specific_dates' allows the specification of one or more explicit dates which will serve as a trigger for non-cyclic rules. However, if no year is specified they become implicitly cyclic with a time interval of ' 1 year'.

LIFETIME

LIFETIME specifies the timespan during which the rule is in force.

```
    [, LIFETIME = (<start_date>, <finish_date>)]
```

<start_date>, <finish_date>

The parameters 'start_date' and 'finish_date' specify the date on which the rule is to come into force and the date when it expires respectively. The default value for 'start_date' is **today** and for finish_date is **infinity**.

In the case of 'data_events' SQL3 provides a complete syntax as shown below:

```
<data_event> := INSERT|DELETE|UPDATE

                ON <table_name>

                [REFERENCING OLD [AS] old_correlation_name
                  [NEW [AS] new_correlation_name] |
                 REFERENCING NEW [AS] new_correlation_name
                  [OLD [AS] old_correlation_name]
```

We can now bring the various elements together into an **Extended SQL3 Trigger Statement** which has the form shown overleaf:

```
TRIGGER_STATEMENT ::= CREATE TRIGGER <trigger_name>
                      BEFORE|AFTER [delay]
          {
            [CYCLE = <time_interval>,
             LIMIT = <max_cycles>]
            [,TIMES = <time_range>|<specific_times>]
            [,DAYSOFWEEK = <specified_days>]
            [,DATE|DATES =

               <date_range>,<specific_dates>]
            [,LIFETIME  = <start_date>, <finish_date]

            | (INSERT|DELETE|UPDATE)

            ON <table_name>

            [REFERENCING OLD [AS] old_correlation_name
             [NEW [AS] new_correlation_name] |
            REFERENCING NEW [AS] new_correlation_name
             [OLD [AS] old_correlation_name]

          }
             [WHEN (search_condition)]

               (statement [,statement]...)

               [granularity]
```

The syntax on the previous page shows how the temporal elements and the SQL-92 and SQL3 components can be combined without the introduction of unnecessary complexity. The worked examples used to develop this extended syntax serve to demonstrate its intuitive nature. These issues are discussed further in the concluding chapter.

Chapter 7
Conclusions

In this chapter we outline the conclusions which we have reached and draw attention to some areas where further research could be of benefit.

7.1 Temporal Database Rules and SQL

In our review of the SQL standard we looked at how it is constantly evolving, highlighting the many innovative developments, in areas ranging from internationalisation to integrity checking, which are being built into the language. The SQL-92 standard, in particular, had a lot to say regarding how time should be specified.

For the specification of triggers, however, we had to base our discussion on SQL3, an advanced draft version of the next revision of the SQL standard, which is due for publication around 1995. In spite of this, it proved possible to abstract SQL3's trigger format (which has largely crystallised at this stage) into the following components:

1. A Trigger Name
2. A Triggering Event
3. A Triggered Action

4. An Event/Action Time Relationship

As we have seen, SQL3 limits the time relationship to 'immediately before' and 'immediately after' the triggering event. Moreover, it limits triggering events to INSERT, UPDATE and DELETE actions.

When the above abstraction mechanism was applied to the worked examples developed in the earlier chapters, it proved possible to re-cast them into the 'shape' required by SQL3 and to achieve compliance with the SQL-92 datetime specification syntax, but not to capture the complete semantics of the original.

The underlying limitations in SQL for the purposes of specifying temporal rules (lack of support for delayed actions; time-based triggers; cyclic operations and rule lifetimes) necessitated the use of an extension of the basic SQL3 draft syntax.

In the final section of Chapter Six the results of all of the previous work were brought together to set out a formal specification of the proposed extended syntax for database trigger definition highlighting where this departed from the evolving SQL framework. The proposed extensions capture all of the semantics required for the specification of time-based rules.

7.2 The Graphical Modelling Formalism

In Chapter Five we discussed the formalism chosen for specifying temporal rules - OSA (Object-Oriented Systems Analysis) which supports the concepts of temporal events and constraints. We emphasised that the suitability of this methodology owed much to its object-oriented basis.

We saw how it was possible to adapt the state-net diagrams used in OSA to capture the details of object behaviour (including time-based triggers and real-time constraints) and demonstrated how this could be used as an efficient mechanism for the identification and subsequent specification of temporal rules.

We examined how OSA behaviour modelling combined three views of an object:

(a) the states which it can assume
(b) the conditions which cause it to change state
(c) the actions associated with the object in these states or in changing between states.

This results in what is called an Object Behaviour Model which uses these three basic concepts - states, triggers and actions, the very language of database rules. Importantly, real-time constraints governing the object's behaviour can be added to the basic state-net which represents this model.

As with database rules the firing of object state transitions is far from automatic. The trigger must first be enabled by its designated prior state. Additional conditions may also need to be satisfied and indeed a trigger may be viewed as a boolean expression which evaluates to true or false. This strongly echoes the trigger and condition syntax used for specifying database rules.

Once the state-net was drawn timing constraints were added to capture any important temporal aspects of the object's behaviour. In our example domains timing constraints were specified for triggers, actions, states and the duration of state-transition paths. Constraints on triggers specified the acceptable 'response time' between the firing of the trigger and the commencement of the accompanying transition.

In adapting OSA as a formalism for temporal database rules the benefits were twofold. Firstly, a graphical representation mechanism emerged and secondly, it provided the necessary framework for the systematic identification and specification of temporal rules.

7.3 Research Directions in Database Rules

In this section we take a brief look at some of the possible ways in which the field of database rules might develop. These range from further developments in the areas of syntax and semantics; CASE Tools and Knowledge Acquisition; High Level Languages; Explanation Facilities and the special requirements of Distributed and Object-Oriented databases.

The issue of extending the syntax to support the combination of transition predicates with boolean operators is discussed in [WIDO89]. For example, a free-format specification of such a rule, 'R1', might appear something like the following:

```
R1  when inserted in t1
    **or** deleted from t2
    where predicate  then op-block
```

As can be seen this also opens up the possibility of referring to multiple tables in a trigger. The points made by Stonebraker [STON92] regarding the need for a simplifying assumption regarding semantics have already been touched upon in Chapter Three. Another issue is the provision of what he refers to as an **explain** facility to allow hypothetical rule firing - 'what would happen if I fired this rule?'. It is to be expected that this will be adopted as a standard service in future systems given the body of work already built up by AI researchers.

Widom and Finkelstein also suggest that, as rules generally enforce constraints, it should be possible to just type in the constraint and have the rules automatically generated. To successfully bring the benefits of CASE to this arena, however, requires a good metaphor for graphically specifying rules and for getting user confirmation of their correctness during design - something equivalent to the way dataflow diagrams etc. are used in SSADM and other methodologies. This should also support schema generation. The OSA formalism, as adapted in Chapter Five, might be worthy of further investigation for this purpose.

Before specifying constraints we have to establish what they are and it may be that rules systems will carry the bottleneck associated with knowledge acquisition into the database domain. However, many proposals for streamlining this process have been put forward e.g. the use of Kelly's repertory grids [FORD91] and the outlook looks positive.

Researchers seem to agree ([STON92],[WIDO89]) on the usefulness of providing a High Level Language for rules at a level of abstraction above the syntax used in prototype systems. Like any language, this should provide a debugging facility for use by the 'database production rules' programmer which would issue warnings of potential loops and rule conflicts.

Rule systems will also need to prove themselves in the new territories opened up by advances in object-oriented and distributed database technologies. The ability to define global rules in a location transparent manner needs to be examined. For object oriented systems rules may need to cope with the added complexity introduced by object versions.

Finally, it might be worthwhile at this point to stand back and have a look at where all of this work is likely to yield the greatest benefit. A recent paper [STON92] sees these systems being used mainly for 'simple' rule bases replacing most of what currently goes to make up application programs, with front-end rule systems being used for 'hard expert system' shells such as automated geologist or physician type problems. Perhaps we will find the client-server paradigm surfacing here, giving us hybrid systems that offer co-ordination between such back-end and front-end rule bases.

7.4 Conclusions and Suggested Future Work

The following conclusions are drawn:

(a) OSA can be adapted as an effective formalism
for modelling temporal database rules.

The formalism adapted for the graphical representation of database rules - OSA (Object-Oriented Analysis) lent itself to the compact definition of time-based triggers and constraints. The suitability of this methodology owed much to its object-oriented basis, indeed, database rules might be viewed of as 'methods' for the Classes which a database seeks to model, and triggering events thought of as 'messages'. Indeed, this echoes the trend to increased object-orientation in SQL itself.

(b) The extended SQL syntax was capable of expressing all of the time-based rules thrown up during OSA analysis.

Recent work by Chandra and Segev [CHAN93], combining the Postgres extensible database with an elegant calendric approach to the specification of time points, confirms the semantic power of time-triggered database rules. However, Segev (pers. comm.) has found the use of Postgres somewhat restricting as an implementation medium for their ideas.

The example applications given here indicated that an Extended-SQL compliant language approach allied to a sound object-oriented modelling formalism has a broad applicability. Furthermore, it was apparent that adding a temporal dimension to rule actions was a key enabling factor in achieving increased semantic power.

As to suggested future work, it would be interesting to see what could be achieved in developing the graphical formalism into a full CASE concept by evolving a formal specification of temporal rule semantics using, say, the language 'Z', and combining this with the graphical front-end. Useful objectives for such research might be, firstly to develop a specification for a CASE tool which could take a graphically represented rule set as input and generate a set of production rules and, secondly, to evaluate its potential, for instance, as a methodology for partitioning out and building the server-side logic (business rules) for Client-Server applications.

References

[ANDR87] Andrews, T., and Harris, C. "Combining Language and Database Advances in an Object-Oriented Development Environment." *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, Fl, October, 1987*

[ASTR76] Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Potzolu, G.R., Traiger, I.L., Wade, B.W. and Watson, V. "System R: A Relational Approach to Database Management" *ACM Transactions on Database Systems, 1(2), 1976, pp. 97-137*

[BANE87] Banerjee, J. "Data Model Issues for Object-Oriented Applications" *ACM Transactions on Office Information Systems, 5(1), 1987, pp. 3-26*

[CARD88] Cardelli, L. "A Semantics of Multiple Inheritance" *Information and Computation, 20(8), 1977, pp. 564-576*

[CHAK90] Chakravarthy, U.S., Grant, J. and Minker, J. "A Logic-based Approach to Semantic Query Optimization" *ACM Transactions on Database Systems, 15(2), 1990, pp. 162-207*

[CHAN93] Chandra, R. and Segev, A. "Managing Temporal Financial Data in an Extensible Database" *Proceedings of the 19th International Conference on Very Large Databases, Dublin, Ireland, 1993, pp. 302-313*

[CHEN76] Chen, P.P.S. "The Entity-Relationship Model - Toward a Unified View of Data" *ACM Transactions on Database Systems, 1(1), 1976, pp. 9-36*

[CLIF83] Clifford, J. and Warren, D.S. "Formal Semantics for Time in Databases" *ACM Transactions on*

Database Systems, 8(2), 1983

[COAD90] Code, P. and Yourdon, E. "OOA: Object Oriented Analysis" *Prentice Hall, Englewood Cliffs, N.J., 1979*

[CODD70] Codd, E.F. "A Relational Model of Data for Large Shared Data Banks" *Communications of the ACM, 13(6), 1970, pp. 377-387*

[CODD79] Codd, E.F., "Extending the Database Relational Model to Capture More Meaning" *ACM Transactions on Database Systems, 4(4), 1979, pp. 397-434*

[COPE84] Copeland, G. & Maier, D. "Making Smalltalk a Database System" *Proceedings of the ACM SIGMOD International Conference on the Management of Data. ACM, New York, 1984, pp. 316-325*

[DAYA87] Dayal, U, and Maniola F. "Simplifying Complex Objects: The Probe Approach to Modelling and Querying Them". *Proceedings of the German Database Conference, Burg Technik and Wissenschafts, Darmstadt, April, 1987*

[DEMA79] De Marco, T. "Structured Analysis and System Specification" *Prentice Hall, Englewood Cliffs, N.J., 1979*

[EISE93] Eisenberg A., & Kulkarni, K. "SQL-92 and SQL3 (its eventual successor)" in *Tutorial Notes, 19th International Conference on Very Large Databases, August 1993, Dublin, Ireland.*

[EMBL92] Embley, D. W., Kutz, B.D. and Woodfield, S.N. "Object-Oriented Systems Analysis : A Model-Driven Approach" *Yourdon Press, Prentice Hall Building, Englewood Cliffs, N.J., 1992*

[EPST90] Epstein B. "Trends and Implications in Database & Repository Technology" *Butler Cox Foundation Opening Address, 1990. Audio Cassette, Sybase Inc.*

[FORD91] Ford, K.M., Petry, F.E., Adams-Webber, J.R. and Chang, P.J. "An approach to Knowledge Acquisition Based on the Structure of Personal Construct Systems" *IEEE Transactions on Knowledge and Data Engineering, 3(1), 1991, pp. 78-88*

[GALL84] Gallaire, H., Minker, J. and Nicolas, J.-M. "Logic and Databases: A Deductive Approach" *ACM Computing Surveys, 16(2), 1984, pp. 153-185*

[GANE79] Gane, C. and Sarson, T. "Structured Systems Analysis: Tools and Techniques" *Prentice Hall, Englewood Cliffs, N.J., 1979*

[GRANT92] Grant, J. and Minker, J. "The Impact of Logic Programming on Databases" *Communication of the ACM, 3, 1992, pp. 66-81*

[HAAS90] Haas et al. "Starburst Mid- Flight : As the Dust Clears" *IEEE Transactions on Knowledge and Data Engineering, 2(1), 1990, pp. 143-160*

[HAMM81] Hammer, H. & McLeod, D. "Database Description with SDM: A Semantic Database Model" *ACM Transactions on Database Systems, 6(3), 1981, pp. 351-386*

[ISHI91] Ishida, T. "Parallel Rule Firing in Production Systems" *IEEE Transactions on Knowledge and Data Engineering, 3(1), 1991, pp. 11-17*

[JENS91] Jensen, C.S. and Mark, L. "Queries on Change in an Extended Relational Model". *IEEE Transactions on Data and Knowledge Engineering 4(2), pp.192-200, 1992*

[JENS92] Jensen, C.S. "Incremental Implementation Model for RElational Databases with Transaction Time" *IEEE Transactions on Data and Knowledge Engineering 3(4), pp. 461-473, 1991*

[KHOS86] Khoshafian, S.N. & Copeland, G.D. "Object Identity" *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR. September, 1986*

[KIMW90] Kim, W. "Object-Oriented Databases: Definition and Research Directions" *IEEE Transactions on Knowledge and Data Engineering, 2(3), 1990*

[KMAN91] "Application of Ingres Rules" *Ingres Knowledge Manager Course Notes, ASK Ingres, 1991*

[LECL88] Lecluse, C. "O_2, an Object-Oriented Data Model" *ACM International Conference on the Management of Data, Chicago, May, 1988*

[LOHM91] Lohman, G.M., Lindsay, B., Pirahesh, H., Schiefer, K.B. "Extensions to Starburst : Objects, Types, Functions, and Rules" *Communications of the ACM, October 1991, 34,(10),1991, pp. 94-109*

[LYNG84] Lyngbeck, P and McLeod, D. "Object Management in Distributed Information Systems" *ACM Transactions on Office Information Systems, 2(2), 1984, pp. 96-122*

[MAIE86] Maier, D. and Stein, J. "Development of an Object-Oriented DBMS" *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, September, 1986, pp. 472-482*

[MAIE87] Maier, D. & Stein, J. "Development and Implementation of an Object-Oriented DBMS". In B. Shriver and P. Werner eds. *Research Directions in Object-Oriented Programming, pp. 355-392, MIT Press, Cambridge, MA, 1987*

[MCCA89] McCarthy, D. and Dayal, U. "The Architecture of an Active Object-Oriented Database System" *Proc. 1989 ACM-SIGMOD Conf. on Management of Data, Portland, OR, June 1989*

[MCMA92] McManus, C. "A Comparison of Conventional Query Optimisation and Semantic Query Optimisation in a Relational Database" *M.Sc Thesis, Dublin City University, August 1992*

[MELT93] Melton, J. & Simon, A.R. "Understanding the New SQL: A Complete Guide" *Morgan Kaufmann, San Mateo, California.*

[PECK88] Peckman, J. & Maryanski, F. "Semantic Data Models" *ACM Computing Surveys, 20(3), 1988, pp. 153-189*

[RMON92] "RoboMon Rule Writing" *Course Notes for RoboMon Version 5.0, Computer Information Software Ltd., July 1992.*

[ROGE88] Rogers, T.R. & Cattell, R.G.G. "Entity-Relationship Database User

Interfaces". *Proceedings of the ER Institute (1988)*

[ROWE87] Rowe, L.A. & Stonebraker, M.R. "The Postgres Data Model" *Proceedings of the International Conference on Very Large Databases, September, 1987, pp. 83-96*

[SHEK88] Shekar, S., "A formal model of trade-off between optimization and execution costs in semantic query optimization" *Proceedings of the 14th International Conference on Very Large Databases, August, 1988, pp. 457-467*

[SHEN89] Shenoy, S., T. & Ozsoyoglu, Z.M. "Design and Implementation of a Semantic Query Optimizer" *IEEE Transactions on Knowledge and Data Engineering, 1(3), 1989, pp. 344-361*

[SHIP81] Shipman, D.W. "The Functional Data Model and the Data Language DAPLEX" *ACM Transactions on Database Systems, 6(1), 1981, pp. 140-173*

[SILB91] Silberschatz, A. et al. "Database Systems: Achievements and Opportunities" *Communications of the ACM, 34(10), 1991, pp. 110-120*

[SKAR87] Skarra A.H. & Zdonik, S.B. "Type Evolution in an Object-Oriented Database" in *Research Directions in Object-Oriented Programming, B. Shriver & P. Wegner (eds.), MIT Press 1987, pp. 1-15*

[SNYD86] Snyder, A. "Encapsulation and Inheritance in Object-Oriented Programming Languages" *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications Portland, OR, September, 1986*

[STON76] Stonebraker et al. "The Design & Implementation of INGRES" *ACM Transactions on Database Systems, 1(3), 1976, pp. 189-222*

[STON83] Stonebraker, M et al. "Implementation of Rules in Relational Database Systems" *University of California Berkeley CSD, UCB/CSD 83/151, November 1983*

[STON85] Stonebraker, M. "Triggers and Inference in Data Base Systems" *Memorandum No. UCB/ERL M85/46, 2 May 1985*

[STON86] Stonebraker, M. et al. "An Analysis of Rule Indexing Implementations in Data Base Systems"

Memorandum No. UCB/ERL M86/6, 16 January 1986

[STON87a] Stonebraker, M. "The Design of the Postgres Storage System" *Proceedings of the 13th International Conference on Very Large Databases, Brighton, England, 1987*

[STON87b] Stonebraker M. et. al. "The Design of the Postgres Rules System" *IEEE International Conference on Data Engineering 1987, pp. 356-374*

[STON88] Stonebraker, M. "Readings in Database Systems" *Morgan Kaufman Publishers, Inc. San Matio, California, 1988*

[STON90a] Stonebraker, M. et al. "The Implementation of Postgres" *IEEE Transactions on Knowledge and Data Engineering, 2(1), 1990, pp. 125-142*

[STON90b] Stonebraker, M. et al. "The Implementation of Postgres" *Memorandum No. UCB/ERL M90/34, 27 April 1990*

[STON91] Stonebraker, M. & Kemnitz, G. "The Postgres Next Generation Database Management System" *Communications of the ACM, 34(10), 1991, pp. 78-92*

[STON92] Stonebraker, M. "The Integration of Rule Systems and Database Systems" *IEEE Transactions on Knowledge and Data Engineering, 4(5), 1992, pp. 415-423*

[WENS88] Wensel, S. (Ed.) "The Postgres reference Manual" *Memorandum No. UCB/ERL M88/20, 25 March 1988*

[WIDO89] Widom, J. & Finkelstein, S.J. "A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems" *Research Report - IBM Research Division RJ 6880 (65706) 6/12/89*

[ZANI83] Zaniolo, C "The Database Language GEM" *Proceedings SIGMOD Conference 1983, pp. 297-218*

[ZDON90] Zdonik S.B. & Maier, D. "Readings in Object-Oriented Database Systems" *S.B. Zdonik & D. Maier (eds.), Morgan Kaufman, 1990*