A Paraphrase of *The Third Manifesto*

David M. Bennett FACS

The Third Manifesto written in language intended for a general audience.

1. Background

1.1.1. The Third Manifesto

The Third Manifesto by C. J. Date and Hugh Darwen is the culmination of more than two decades of work on defining a foundation for the future of managing data. The authors view SQL and the relational database management systems that use it as deeply flawed, and have set down their views on a language to replace it. The title reflects two prior attempts (manifestos) by other authors.

The *Manifesto* is intentionally an academic publication written in academic language for an academic audience. Parts of it can only be read in conjunction with other books by the same authors, and parts use highly specialised language. It is not easily accessible to a general audience.

The intention here is to paraphrase, generalise and shorten the wording, using terminology intended for a general IT audience, while retaining the original intent, meaning and numbering. This has involved some reorganisation and some inclusion of background material from other writings to ensure that the all the terms used can be understood within one document.

This is not an easy thing to do while keeping true to the original and there may well be errors and omissions, for which I apologise in advance. Those who seek further clarity or authority should consult the primary references.

1. The original: http://www.dcs.warwick.ac.uk/~hugh/TTM/TTM-2013-02-07.pdf.
2. The book: Database Explorations by C. J. Date and Hugh Darwen.
3. The web site: http://www.dcs.warwick.ac.uk/~hugh/TTM.

The *Manifest* mentions the Inheritance Model, which may be found via the above links but is not covered here.

1.1.2. This Document

The document broadly follows the structure of the *Manifesto*, including an initial section with definitions and explanations of terms.

The *Manifesto* expresses many of its requirements as references to a hypothetical language called D. In this document they are replaced by references to the requirements of an implementation.

The *Manifesto* makes use of the syntax of **Tutorial D**, a teaching implementation of D. This material has been omitted on the assumption that readers will not be familiar with it.

References to "the system" have been replaced by references to "built-in" features. Additional definitions have been included as needed, mainly drawn from other books by the same authors. Minor changes have been made in some terminology. Some notes have been added to clarify intent or draw attention to specific consequences.

1.2. Preamble and Definitions

This document is written as a series of requirements on an implementation that is expected to include a programming language, its libraries and an underlying relational database manager.

Nothing here should be taken to mandate any particular design or exclude any approach or prohibit any additional capabilities beyond what is explicitly stated.

Terms that have particular meanings are underlined at the point where they are defined, or where first used if not defined.

A requirement for something in the singular or plural should not be taken as mandatory as to quantity unless this is made explicit, to avoid having to add such qualifications repeatedly in the text.

1.2.1. Definitions

Prescriptions or Pre are requirements that an implementation must or should or may implement, support or allow. The individual language will make it clear which.

Proscriptions or Pro are requirements that an implementation must or should not implement, support or allow. The individual language will make it clear which.

Very Strong Suggestions or VSS are recommendations that fall short of being requirements.

Relational Model or RM means requirements that arise from the Relational Model.

Other Orthogonal or OO means requirements that arise from other than the Relational Model.

The special relation values **`DUM`** and **`DEE`**, with no attributes and zero and one tuples respectively, are mentioned but not by those names.

A literal is a symbol or an expression composed of literals that has a fixed known meaning at compile time. It denotes a constant value.

A type is a named and non-empty set of values. Types with different names are different types. Any type may form the basis for defining any variable, attribute, component, argument or result as set out below.

Every type is a scalar type or a non-scalar type. A non-scalar type is a tuple or relation type. A scalar type is any other type.

An ordered type is one for which a total ordering is defined.

Built-in means that something is provided as a standard part of the implementation. User-defined means that it is defined using means provided by the implementation.

An expression is a syntactic form that includes operators and values and when evaluated produces a value as a result.

An operator is a syntactic form that may be invoked with arguments, may update a variable and may return a value as a result. The term operator is intended to include both operators and functions as ordinarily understood in other languages.

An argument is a value or a reference to a variable that is provided to an operator when it is invoked. It takes the syntactical form of an expression or a symbol denoting a variable.

A variable means a portion of the state of a program or database that is given a name and a declared type and always contains a value of that type. Every access to the value of a variable returns a value of its type. A value is assigned to a variable by means of an update operator.

A statement is some minimal unit of program execution that results in a change in program or database state and can execute atomically. Atomic means that it either executes in its entirety or not at all.

The relational algebra is a family of operations with well-known semantics used to manipulate relational data.

1.3. Relational Model Prescriptions

These requirements of an implementation arise from the Relational Model.

1.3.1. RM Pre 1 – Scalar type

A scalar type is a named set of scalar values. A scalar type reveals no visible internal structure, but may provide possible representations (see below).

An implementation must provide built-in scalar types, and a way to create and destroy user-defined scalar types. It must ensure that every scalar type definition is accompanied by an example value of that type.

The built-in scalar types must include a logical type with the values of true and false and the logical operators and, or and not. It should provide concise additional logical operators that are combinations of these.

Note: In practice the built-in scalar types must also include an ordered numeric type (for at least positive integer values) and a text type (for names).

1.3.2. RM Pre 2 – Scalar values

A scalar value is a value that is a member of a scalar type. Every scalar value belongs to exactly one scalar type. Values that belong to different types are different values.

1.3.3. RM Pre 3 – Scalar operators

A read-only operator takes one or more arguments, each of which is a value of a declared type, and returns a result that is a value of a declared type. It updates no variables.

An update operator has the effect of assigning a value to a variable, or perhaps more than one. It takes one or more arguments, each of which is either a value of a declared type or a reference to a variable of a declared type that it updates. It does not return a result.

Note: In this context variable refers to the visible program or database state and not merely to local or temporary variables used during execution.

An implementation must provide built-in operators and means to create and destroy user-defined operators of all kinds. Operator definitions must allow recursion.

Note: The above applies to both scalar and non-scalar operators.

A scalar operator is one that returns a scalar result or causes a scalar assignment.

An implementation must provide built-in scalar operators that include:

- Selector, comparisons and assignment as specified elsewhere
- Additional operators required to give reasonable effect to the built-in scalar types.

An implementation must provide means to create and destroy user-defined scalar operators, both read-only and update.

1.3.4. RM Pre 4 – Representation of scalar types and values

Values are conceived to be immutable and pre-existing, so that particular values are chosen by a selector (rather than a constructor). A selector takes the form of a read-only operator that has arguments and returns a value of the type.

Values of a scalar type are assumed to have a concrete or physical representation, which is not made visible. Instead values may be represented by the components of one or more abstract or possible representations or possreps.

An implementation must provide means to create user-defined types by specifying one or more possreps, each of which is made up of one or more components, each with a name and a declared type. Each possrep and its components is required to correspond one-to-one with a selector and its arguments, and to have an invocation with some argument values that will produce any value of that type.

Note: an implementation may provide additional operators that resemble selectors and are not bound by this requirement.

For a built-in type components are optional, but if omitted then an implementation must ensure that there is some selector invocation with literal arguments that will produce any value of that type.

Note: an implementation is not prohibited from providing a built-in type solely by not being able to meet this last requirement.

1.3.5. RM Pre 5 – Components are exposed

For each value of a type that has a possible representation, an implementation must provide read-only operators by which the value of any component of that value may be retrieved as a value of its declared type. These are informally referred to as getters.

For each variable of a type that has a possible representation, an implementation must provide update operators by which a new value is assigned to the variable so that any given component of its value will have a different value before and after the update.

An implementation may provide means for assignment to individual components. These are informally referred to as setters.

Note: an implementation may provide additional operators that resemble getters and setters and are not bound by these requirements.

1.3.6. RM Pre 6 – Tuple types

A heading comprises a set of attributes, each of which has a distinct name and is of a declared type. The degree of a heading is the number of those attributes. The attributes are unordered, and accessed only by name. The empty heading has a degree of zero.

Note: An implementation is not required to provide means to define a heading type.

A tuple type is a named set of tuple values, for which the type name is generated by reference to its unique heading. The effect is that two tuple values are of the same type if they have the same heading, and thereby the same type name. The degree of a tuple type is the degree of its heading. The empty tuple type has a degree of zero.

Attribute extraction means an operator that takes a tuple value and an attribute name as arguments and returns the value of that attribute as its result.

Tuple nesting means an operator that takes tuple values as arguments and returns a tuple value with a tuple-valued attribute as its result. Tuple unnesting means an operation that achieves the reverse.

An implementation must provide means to define a tuple type by specifying the name and type of each attribute that comprise its heading.

An implementation must provide built-in tuple operators that include:

- selector, comparisons and assignment as specified elsewhere
- the relational algebra
- attribute extraction
- nesting and unnesting.

1.3.7. RM Pre 7 – Relation types

A relation type is a named set of relation values, for which the type name is generated by reference to its unique heading. The effect is that two relation values are of the same type if they have the same heading, and thereby the same type name. The degree of a relation type is the degree of its heading. The empty relation type has a degree of zero.

Tuple extraction means an operator that takes a relation of cardinality one as an argument and returns its sole tuple as its result.

Relation nesting means an operator that takes a relation and tuple values as arguments and returns a relation value with those tuples constituting a relation-valued attribute as its result. Relation unnesting means an operation that achieves the reverse.

An implementation must provide means to define a relation type by specifying the name and type of each attribute that comprise its heading.

An implementation must provide built-in relation operators that include:

- selector, comparisons and assignment as specified elsewhere
- the relational algebra
- tuple extraction
- nesting and unnesting.

1.3.8. RM Pre 8 – Equality operator

An equality operator is one that takes two values as its arguments and returns true if and only if they are the same value. If this is so, then of course they must be the same type.

Two values that compare equal must be indistinguishable in their behaviour for all other operators, and if they behave differently in any way then they are different values and must not compare equal.

An implementation must provide or require an equality operator for every type, whether built-in or user-defined, scalar or non-scalar.

1.3.9. RM Pre 9 – Tuple values

A tuple value comprises a heading (that of its type) and a set of values, one corresponding to each of its attributes. The attributes are not ordered, and its attribute values are accessible only by name.

The degree of a tuple value is the degree of its heading. The empty tuple has a degree of zero.

For each tuple type, an implementation must provide a selector with parameters that correspond one-to-one with its attributes. It must ensure that there is some selector invocation with literal arguments that will produce any value of that type.

1.3.10. RM Pre 10 – Relation values

A relation value comprises a heading (that of its type) and a body. The body consists of a set of tuples, all with that same heading. The tuples are not ordered, and cannot be accessed individually.

The degree of a relation value is the degree of its heading. Its cardinality is the number of tuples in its body. An empty relation has a cardinality of zero.

For each relation type, an implementation must provide a selector with parameters that comprise a set of tuples of that same heading. It must ensure that there is some selector invocation with literal arguments that will produce any value of that type.

Note: there are two relation values with an empty heading, one that is empty and one with a body consisting of a single empty tuple.

1.3.11. RM Pre 11 – Scalar variables

A scalar variable means a variable of a scalar type. It must always hold a value of that same type.

An implementation must provide means to create scalar variables.

1.3.12. RM Pre 12 – Tuple variables

A tuple variable means a variable of a tuple type. It must always hold a value of that same type. The heading, attributes and degree of a tuple variable are as defined for its type and value.

An implementation must provide means to create tuple variables.

1.3.13. RM Pre 13 – Relation variables

A relation variable or relvar means a variable of a relation type. It must always hold a value of that same type. The heading, attributes, degree and cardinality of a relvar are as defined for its type and value.

Relation variables comprise database relvars and application relvars. Database relvars are contained in a database. Application relvars are local to an application.

An implementation must provide means to create application relvars, and to create and destroy database relvars.

Note: the lifetime and visibility of application variables (which together comprise program state) including the extent to which they may be shared amongst programs that constitute an application, is not specified. Database relvars do not form part of program state.

1.3.14. RM Pre 14 – Real, virtual, public and private relvars

Database relvars may be real or virtual. Application relvars may be public or private.

A real relvar or base relvar is a database relvar that is not virtual. It exists in the database and is updatable.

A virtual relvar is a database relvar that is not a variable, but rather a name for the result of evaluating a relational expression that in turn refers to at least one database relvar other than itself.

Note: a virtual relvar is roughly equivalent to a view. Whether it is updateable is not specified.

A public relvar is one that an application will perceive as a real relvar, but may actually be virtual. The intent is to provide a measure of isolation between the application's requirements and possible changes in the database. If it is a view, it should be updateable.

A private relvar is one that is known and accessible only to the application. Its lifetime and visibility are not specified.

An implementation must provide means to create each of the above.

1.3.15. RM Pre 15 – Candidate keys

A candidate key is a set of attributes of a relation for which the values in every tuple are different, and that has no subset with that property.

An implementation must ensure that at least one candidate key is defined when any relvar is created.

Note: it is possible for a candidate key to be the set of all attributes, or to be empty (in which case the relation can contain only a single value or be empty).

1.3.16. RM Pre 16 – Database

A database is a named container for database relvars. Databases need not be disjoint.

An implementation should not provide means to define, create or destroy a database, but rely on means provided elsewhere.

1.3.17. RM Pre 17 – Transactions

A transaction is a grouping of statement executions that interacts with a single database. Distinct transactions interact with distinct databases (which need not themselves be disjoint).

An implementation must ensure that every statement is executed within the context of some transaction. Statements must be executed atomically, except that there may be specific exceptions such as nested statements and user-defined update operators.

1.3.18. RM Pre 18 – Relational Algebra

In this context the Relational Algebra means a set of operators that take at least one relation value as an argument and produce a relation value as a result.

An implementation must provide a concise set of such operators from which others can be constructed. A sufficient set would be monadic Restrict, Rename, Project, Extend and dyadic Join, Antijoin, Union but the precise set is not specified.

Although each relational operator is generic in that it can apply to any relational type as input, the output type for any input can be inferred. An implementation must use type inference to determine the result of using relational operators and avoid type errors during execution.

The above should apply to tuple types as well, as far as is possible.

1.3.19. RM Pre 19 – References

An implementation must provide means to use a reference to a variable or a constant of any type as a value in any expression.

1.3.20. RM Pre 20 – Tuple and Relational Operators

An implementation should provide built-in relational operators in addition to those of the relational algebra, and must provide a way to create and destroy user-defined read-only and update relational operators.

The above applies equally to tuple operators.

1.3.21. RM Pre 21 – Assignment

Assignment means replacing the value of a variable by a different value. It is the intended effect of all update operators, including those that seek to improve performance by altering rather than copying large values.

Multiple assignment means replacing the values of two or more variables by new values in such a way that they constitute a single atomically-executed statement.

An implementation must provide or require assignment for every type by means of update operators (which may or may not include the form of assignment found in other languages). If a value is assigned to a variable, then subsequently the value and that of the variable will compare equal.

An implementation must provide means for multiple assignment.

1.3.22. RM Pre 22 – Comparison Operators

A comparison operator is one that takes two arguments of declared types and returns a logical result. The minimum set of comparison operators that an implementation must provide is:

- For all types: is equal to (see RM Pre 8).
- For ordered types: is less than.
- For relational types: is a subset of.
- For a tuple and a relational argument: is a member of.

For all but the last the arguments must be of the same type; for the last they must have the same heading.

An implementation should provide concise forms of the operators that result from combining and/or reordering the above together with the logical operators previously mentioned, and may provide others.

1.3.23. RM Pre 23 – Integrity Constraints

A constraint or integrity constraint is the equivalent of a logical assertion as to some part of the program or database state that is satisfied if it is true and is violated if it is false. The assertion may take the syntactic form of a logical expression, or some other form.

A type constraint specifies the set of values that constitute a type. The appearance of any other value whether as an intermediate result or as the value of a variable is a constraint violation.

A database constraint specifies the values allowed for a given set of database relvars taken in combination. The appearance of any combination of values that does not satisfy the constraint at the end of any statement execution (which may of course include multiple assignment) is a constraint violation.

An implementation must provide means to define and destroy type and database constraints. It must also where possible provide constraint inference, including detecting constraint incompatibility.

An automatic update is one that may be performed where a database constraint applies to a set of database relvars and an update to one then requires that a specific update to another be performed to avoid a constraint violation. An implementation should where possible and not prohibited determine and perform such automatic updates.

Note: an implementation may provide constraints on application relvars in similar form to those on database relvars.

1.3.24. RM Pre 24 – Total Database Constraint

The total database constraint is the logical **AND** of all of the currently defined database constraints.

An implementation must ensure that no update leaves a database in a state that violates its own total constraint.

1.3.25. RM Pre 25 – Catalog

The catalog for a database is a set of database relvars in that database, including the catalog itself. Operations such as defining and destroying types, operators, variables, constraints and so on are regarded as assignments to the catalog and are subject to the rules of assignment, including multiple assignment.

1.3.26. RM Pre 26 – Good language design

An implementation must conform to the principles of good language design. Some of the principles to be considered include generality, parsimony, completeness, similarity, extensibility, openness, orthogonality and conceptual integrity.

1.4. Relational Model Proscriptions

These requirements on what an implementation should avoid arise from the Relational Model.

1.4.1. RM Pro 1 – Attributes are not ordered

An implementation must not distinguish attributes of a relation or tuple by order, but only by name.

1.4.2. RM Pro 2 – Tuples are not ordered

An implementation must not distinguish tuples of a relation by order, but only by value.

1.4.3. RM Pro 3 – Duplicates are not allowed

An implementation must not permit duplicate tuples in a relation. No pair of tuples in a relation may compare equal.

1.4.4. RM Pro 4 – No nulls

An implementation must ensure that every attribute in every tuple has a value of its type, which then of course cannot be null.

1.4.5. RM Pro 5 – Empty relations, tuples, headings and keys not excluded

An implementation must allow empty relations, tuples, headings and keys, as they may be useful.

1.4.6. RM Pro 6 – No internal access

An implementation must not allow access to physical, storage or internal levels of the system. It must rely on such access being provided by other means.

1.4.7. RM Pro 7 – No tuple-at-a-time operations

An implementation must not allow individual tuple operations or tuple-at-a-time operations on relational values.

1.4.8. RM Pro 8 – No composite attributes

A composite or compound attribute is one that can hold more than a single value, or can hold other attributes. Common examples include records, structures and arrays. The scalar and non-scalar values described above are single values for this purpose.

An implementation must not support composite or compound attributes.

1.4.9. RM Pro 9 – No domain check override

Domain check override refers to means to defeat or override or bypass the type system.

An implementation must not include domain check override operators, but must ensure that all operations are in accordance with the type system.

1.4.10. RM Pro 10 – Not SQL

An implementation must not be SQL (but see OO VSS 7).

2. Other Orthogonal Prescriptions

These requirements of an implementation arise outside the Relational Model.

2.1. OO Prescriptions

2.1.1. OO Pre 1 - Compile time type checking

An implementation should check the usage of all types at compile time to avoid type errors that might otherwise occur during execution. See also RM Pre 18.

2.1.2. OO Pre 2 - Type inheritance

If an implementation supports type inheritance then it must be in accordance with the *Manifesto* Inheritance Model (see preamble).

2.1.3. OO Pre 3 - Computational completeness

A language is considered computationally complete if entire applications can be written in it, and that in particular it does not depend on another language for defining new types and operators.

An implementation must be computationally complete, for at least some useful set of applications, types and operators. An implementation that is a data sub-language would not satisfy this prescription.

An implementation may also co-exist with other languages, including being called by a host program and/or calling upon another language for the implementation of user-defined operators and types where the requirements make it reasonable to do so.

2.1.4. OO Pre 4 – Explicit transaction support

Transactions begin with start transaction and end with commit or rollback. Commit means that the updates in that transaction are made permanent in the database. Rollback means that they are discarded and the database state is unaffected by the transaction. These are the transaction boundaries.

Implicit commit means that each statement is treated as if it were preceded by start transaction and followed by commit, that is, as if there were transaction boundaries around each statement.

An implementation must provide explicit support for start transaction and commit, and automatic rollback if a commit fails. It may also provide explicit support for rollback, and implicit commit.

2.1.5. OO Pre 5 – Nested transactions

A nested transaction is one that has boundaries that lie entirely within those of another transaction. Overlapped transactions are not allowed.

An implementation must support nested transactions, which must interact with the same database (see RM Pre 17). A rollback of the outer transaction must cause a rollback of the inner, even if it has already performed a commit. The order in which operations are carried out is not specified.

2.1.6. OO Pre 6 – Aggregation operators

An aggregation operator is one that takes as its arguments an attribute value (or values) from each of the tuples in a relation and returns a single value.

If an implementation provides such an operator which is dyadic, specifies an identity value and is allowed to return any value of its type then it must return a result equivalent to the successive application of the operator to the identity value and each of the values in the tuple, regardless of order.

Note: the result for an empty relation will be the identity value.

2.2. OO Proscriptions

These are requirements on what an implementation should avoid that arise outside the Relational Model.

2.2.1. OO Pro 1 – Relvars are not domains

A domain means a relational type.

An implementation must not use individual relvars as if they were relational types.

2.2.2. OO Pro 2 – No pointers

A pointer is a machine address or a physical representation for one.

An implementation must not allow any attribute of any database relvar to be of type pointer.

Note: whether pointer types are allowed for scalar variables, attributes of tuple types and attributes of application relvars is not specified.

2.3. Relational Model Very Strong Suggestions

These are requirements an implementation should consider that arise from the Relational Model.

2.3.1. RM VSS 1 – Built-in key value generator

An implementation should provide a built-in mechanism by which the values of certain attributes may be generated, and which are guaranteed to be unique (within limits). The intended purpose is to generate unique key values.

2.3.2. RM VSS 2 – Inferred candidate keys

Where possible an implementation should infer the candidate keys for a relvar from its definition. In particular it should do so for a virtual relvar based on an analysis of its defining expression and any base or other relvars it references.

2.3.3. RM VSS 3 – Transition constraints

A transition constraint is one that serves to limit the transitions that a database can make from one value to another.

An implementation should support transition constraints, and may associate them with specific update operators.

2.3.4. RM VSS 4 – Quota queries

A quota query is one that specifies a limit on the cardinality of the relational value that is the result of an expression.

An implementation should support quota queries, which may either be absolute or relative to the rank of some attribute value. In the latter case the actual result may contain more or less tuples than the specified limit.

2.3.5. RM VSS 5 – Generalised transitive closure

A transitive closure is a relational operation that takes as an argument a relation that represents a directed graph as individual arcs and returns one that lists all paths through the graph.

A generalised transitive closure extends that to one that can perform calculations such as concatenation and aggregation along those paths.

An implementation should provide concise generalised transitive closure operators.

2.3.6. RM VSS 6 – Generic operators

A generic operator is one that has arguments and results that are drawn from a specified set of types and are not just individual declared types. All built-in operators of the relational calculus are generic, in that they apply to all relational types.

An implementation should provide means for creating user-defined generic operators, particularly for relational types.

2.3.7. RM VSS 7 – Implementation of SQL

An implementation should be sufficiently capable that it may be used to implement a dialect of SQL (as a transition aid), or that existing SQL programs and databases may be converted to it (for migration).

2.4. Other Orthogonal Very Strong Suggestions

These are requirements an implementation should consider that arise outside the Relational Model.

2.4.1. OO VSS 1 – Type inheritance

An implementation should support type inheritance as described in OO Pre 2.

2.4.2. OO VSS 2 – Operator and types unbundled

Bundling means the common object-oriented practice of defining operators inside a particular class, and thereby having a privileged *receiver* or *self* or *this* object for the operator.

An implementation should avoid bundling and instead provide operators that are separate from the types they act on, and that have no privileged receiver.

2.4.3. OO VSS 3 – Single level store

Single level store means the principle that database and application variables be interchangeable and indistinguishable for most programming purposes.

An implementation should adopt this principle and ensure that database and application relvars are interchangeable to the maximum extent possible, except as is necessary for their specific purpose or for performance considerations.