1 Introduction

Everything in the universe, except God, exists in the context of an hierarchy, denial of that existential fact when modelling data leads to horrendous problems. The *Relational Model* supports hierarchies completely. That is to say, the definition and storage done correctly (Relationally), such that the presentation (display) is straight-forward. Logic; mathematics, is top-down [1]. The methods herein have been available from the days of the first SQL platforms in the 1980's, it is pure SQL, it does not need CTEs or temporary tables or the `HIERARCHYID` datatype.

Hierarchies occur naturally in the world, they are everywhere. That results in hierarchies being implemented in many databases. While the logic in the *Relational Model* is founded on First Order Predicate Calculus, it is a progression of the Hierarchical Model (as well as the Network Model [2]). It supports hierarchies brilliantly. Unfortunately the academics and authors of "textbooks":

- do not understand the *Relational Model*. They understand and teach only 1960's Record Filing Systems, characterised by `RecordIds`, falsely marketed as "relational". Such "databases" are devoid of Relational Integrity (as distinct from Referential Integrity which is physical); Relational Power; and Relational Speed. Such primitive systems are placed in an SQL container, for convenience (access; backup; etc).
- do not understand the hierarchies that exist in the data, let alone the hierarchies in the *Relational Model*. The result is, the hierarchies that exist in the data are not recognised as such, and thus they are implemented in a grossly incorrect and massively inefficient manner.

The academics suppress both. Those methods are bottom-up, devoid of logic, extremely slow, and cannot scale. Explained in §1.2.

Conversely, if the hierarchy that occurs in the data, is modelled correctly, and implemented using genuine Relational methods (Relational Keys, etc) the result is an easy-to-use and easy-to-code database, as well as being devoid of data duplication in any form (full Normalisation). It is completely unlimited (eg. no limit to levels or scale), moving a single node (branch) updates just one row, and it is extremely fast [3]. It is literally *Relational* at its best.

1.1 Relational Hierarchy

There are three types of Hierarchies that occur in data, which is logical, that need to be modelled correctly. Because the context is SQL, which is physical, it is given first.

0 Physical • Primary Concept
For understanding, in the first instance, it needs to be appreciated that every `FOREIGN KEY` relation is an hierarchy: a single parent row is referenced by multiple child rows; the multiple referring rows refer to a single referred row. That is to say, the child exists in dependence of, in subordination of, the parent. This is the physical declaration only, any `FOREIGN KEY` reference, including the three logical types have to use it.

1 Logical • Hierarchy Formed in Sequence of Tables
The Data Hierarchy is the first principle of the *Relational Model* [4], and the **Relational Key** is the method [5]. They occur in every database.

- Conversely, the lack of it cripples the modelling exercise, and produces a 1960's Record Filing System, characterised by physical `RecordIds`.

The parent and child rows are in discrete tables. The hierarchy is plainly visible in the *form* (composition) of the Relational Key, which progresses in each compounded step, in the sequence of tables: father, son, grandson, etc. It is essential for ordinary Relational data Integrity, the kind that 95% of the database implementations do not have, due to following the false and vociferous prophets.

I have written about Relational Keys extensively elsewhere, this type of hierarchy is not expanded in this document.

2 Logical • Hierarchy of Rows within One Table
Wherein each row has a single parent in the same table. This is articulated in §2.

3 Logical • Hierarchy of Rows within One Table, via an Associative Table
Wherein each row has multiple parents in the same table, and resolution requires an Associative table. The problem is explained in §3, and the solution is articulated in §4.

1.2 Ignorance of the Relational Hierarchy

The main problem with 95% of the "databases" is that they are not logical (data rows with Keys formed from the data), they are primitive Record Filing systems (physical records with `RecordIds` as "keys"). As such they are slow, and obtaining data recursively or traversing the trees, is very slow. Continuing in the dark trench of ignorance, instead of obtaining education about the *Relational Model*, they declare that "relational databases do not support hierarchies", oblivious to the fact that their "database" is neither a database nor Relational, and devise methods to deal with their primitive structures. Such methods are ridiculous and ham-fisted. Note the focus on the display requirement rather than on the data, the total absence of genuine Relational or logical data modelling.

Adjacency List

- The suppressors hilariously state that "the *Relational Model* does not support hierarchies", in denial that it is founded on the Hierarchical Model (each of which provides plain evidence that they are ignorant of the basic concepts in the *Relational Model*, which they allege to be postulating about). So they can't use the word *Hierarchy* in the name, it would give the game away. This is the stupid name they use.
- Generally, the data model will have recognised that there is an hierarchy in the data, but the implementation will be very poor, limited by physical `RecordIds`, etc, and absent Relational Integrity, etc.
- They are clueless as to how to traverse the tree, or to find members of a branch, that it needs recursion.

Nested Sets

- An abortion, straight from hell. A Record Filing System within a Record Filing system. Not only does this generate masses of duplication and break Normalisation, this fixes the records in the filing system in concrete. The inner RFS is a linked-list of physical records based on their *position* in the *displayed tree*. I was not aware that the fixation of the display (as opposed to storage) could be materialised to this degree, to pathology Surely treatment with drugs would provide more relief.
- Moving a single node requires the entire affected branch of the tree to be re-written. Beloved of the Date; Darwen; Fagin; and Celko types, and the OO/ORM groupies: the addiction to endless manual labour.

`HIERARCHYID`

- The MS SQL `HIERARCHYID` Datatype does the same thing. Using it gives you a mass of concrete that has to be jack-hammered and poured again, every time a node changes.

Common Table Expression

- The recent MS SQL feature, that provides exposition of an hierarchy, that exists in the data, which is stored in primitive `RecordId` based files. It is laborious and slow, both to code (complexity) and to execute (temporary tables; etc). Completely unnecessary if one implements the ordinary Relational hierarchy that preceded CTE by decades, as documented herein.

---

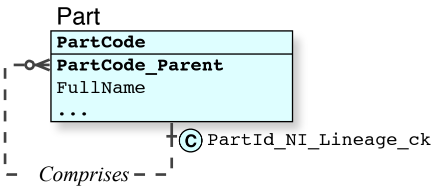
1 Whereas the anti-Relational mob (Date; Darwen; Fagin; the OO/ORM crowd) work backwards, or bottom-up, from the desired display, to the storage required for such.
2 While the hierarchic features in the *Relational Model* are a progression of the Hierarchical Model, the Independent Access feature is a progression of the Network Model.
3 The solution is pure Relational; pure SQL, which means a genuine SQL compliant platform (the freeware is not SQL), that supports recursion.
4 Dr E F Codd, ***A Relational Model of Data for Large Shared Data Banks***. §1.4 Normal Form. First the pre-requisite is given in *Fig 3(a) Unnormalised Set*: the data must be arranged in Trees, i.e.. Directed Acyclic Graphs. That prohibits circular references. A Tree is an hierarchy. The concept was familiar because the predecessor was Hierarchical DBMS, well-known and understood.
5 Next, the definition of the Relational Key, the **Relational Normal Form**, is given in *Fig 3(b) Normalised Set*.

SOFTWARE GEMS

2 Single Parent • Normalised

- Good for any single tree structure: for each row that is a parent, a tree is possible. Strangely called "one way" or "one tree" by non-technical publishers, it means a single tree "down" for each parent, which in Foreign Key terms means the child references "up" to the parent.
- For each row, **single parent** in the same table (for each row, multiple children via the `FOREIGN KEY` relation, is of course ordinary)
  - the term *self-reference* is false and confusing: the table cannot refer to itself, it is a row that makes a reference, and then to another row, not itself
- Exposition of **Lineage** (ancestry, the generations of parents), which may be a computed column (eg.. *`Path`*), requires a recursive **Function** (single column, a scalar). The level of recursion is simple: one for each generation.
- Circular References are stupid, they are explicitly prohibited in the *Relational Model* [6]. It is enforced by a Constraint that calls the Function.
- Do not store a Level: level is relative to the branch that is queried, and thus derived, storing it is exceedingly stupid because the trees are then physicalised, and changes to the tree would demand changes to many rows [7].

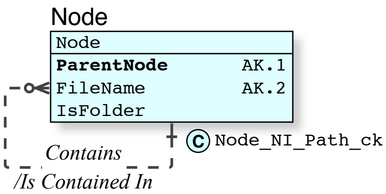
2.1 Example Inventory



- `PartCode [parent] comprises 0-to-n Parts[children]`
- `PartCode [child]  is a constituent of 1 Part[parent]`

Ⓒ `Part.PartCode_NI_Lineage_ck`
`CHECK @PartCode NOT IN Part_Lineage_fn( @PartCode_Parent )`

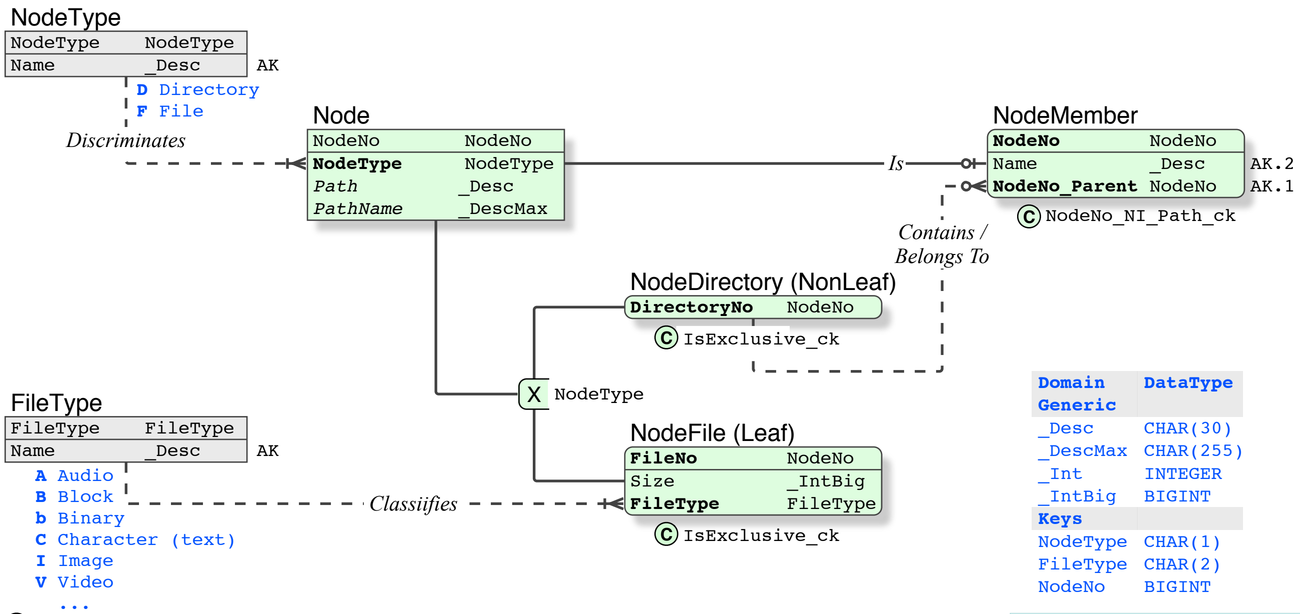
2.2 Example Directory • Simple



- `Node` is either a File or a Folder, based on `IsFolder`
- `Node[parent] contains 0-to-n Nodes[children]`
- `Node[child]  is contained in 1 Node[parent]`
- AK prevents duplicate `FileNames` under a single `ParentNode` (a File of the same Name in different Folders is permitted)
- `Nodes` cannot be duplicated

Ⓒ `Node.Node_NI_Path_ck`
`CHECK @NodeNo NOT IN Node_Path_fn( @ParentNode )`

2.3 Example Directory • Full



Domain	DataType
Generic	
_Desc	CHAR(30)
_DescMax	CHAR(255)
_Int	INTEGER
_IntBig	BIGINT
Keys	
NodeType	CHAR(1)
FileType	CHAR(2)
NodeNo	BIGINT

Ⓒ `NodeMember.NodeNo_NI_Path_ck`
`CHECK @NodeNo NOT IN Node_GetPath_fn( @NodeNo_Parent )`
Check child not in the parent tree (not the child tree)

Ⓒ `NodeDirectory.IsExclusive_ck`
`CHECK ValidateExclusive_fn ( NodeNo, "D") = 1`

Ⓒ `NodeFile.IsExclusive_ck`
`CHECK ValidateExclusive_fn ( NodeNo, "F") = 1`

- A column in italics (IDEF1X) is a derived or computed column, it is not stored.
  `Path` is the list of `NodeNos` from the root of the data tree, obtained via the Function
  `PathName` is the list of `Names`
- The true logical Key is:
  `( Path, Name )`
  which is (a) huge, and (b) impossible because `Path` is multivalued and fails 1NF; 2NF, thus a proper surrogate NodeNo is implemented as the Primary Key
- `NodeNo` is either a File or a Directory, based on Discriminator `NodeType`
- `NodeNo` that is a member (File or Directory) has 1 `NodeNo_Parent`
- A Directory `NodeNo` contains 0-to-n member `NodeNos`, wherein it is the `NodeNo_Parent`
- prevents duplicate `Names` within a single `NodeNo_Parent`
  - `Name` alone is **not** unique
  - `NodeNo` alone **is** unique (cannot be duplicated)

Subtype
- Refer to the **Subtype** document for a full explanation of the concept, and the associated Constraints.

Anchor
- An anchor or zero row, is required, in order to:
  - allow multiple data trees
  - inform the recursive Function to terminate
- This is not data. It is safe because the data rows will be accessed with joins to the subordinate tables.
- It is not a contrived row, it is an anchor.
- After the `CREATE TABLE` commands, before applying the Foreign Key Constraints, eg:
  `NodeDirectory.Node_Is_Directory_fk`
  insert the single anchor row with a zero `NodeNo` value:
  `INSERT Node ( 0, "D", 0, "[Anchor]" )`
  `INSERT NodeDirectory ( 0 )`

IDEF1X Notation

6 Whereas the anti-Relational mob purposefully implement circular references, and demand that SQL (the data sublanguage defined in the *Relational Model*) be changed to allow the insanity. Note well, that circular references do not exist in reality.
7 Those who teach Record Filing Systems, and physicalised methods for the implementation of hierarchies (page 1), such as Date; Darwen; Fagin; Celko; etc, are not only ignorant of the *Relational Model* that they allege to explain, but exceedingly stupid, as evidenced by their propositions.

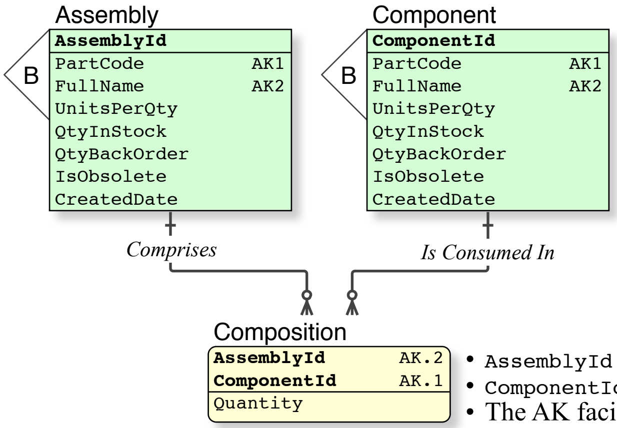
Multiple Parent • Problem

3 Pre-Relational

For understanding and comparison only.

This is how the **Bill of Materials** structure was actually implemented in DBMS platforms prior to the *Relational Model*. In the 1960's & 70's DBMS world, this was known as the **Bill of Materials Problem**, it was famous, because it signified the limit of implementation in HDBMS, and the problem that needed to be solved. It was a specific problem that IBM tasked Dr E F Codd [7] to overcome. Which he did, brilliantly (solution, next page).

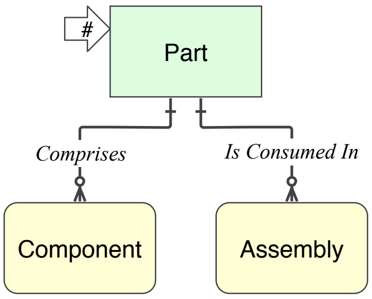
3.1 Hierarchic Model



- This is a simplified model of the **Hierarchical DBMS** implementation.
- The IDEF1X model shows the logical data as usual.
  - The physical structure, a B-Tree for initial access by Key, followed by a linked-list of pointers for the records, is not shown.
- Massive duplication: many Parts are both an Assembly and a Component.

- `AssemblyId` comprises many `ComponentIds`
- `ComponentId` is consumed in many `AssemblyIds`
- The AK facilitates search by Component

3.2 Network Model



- This is the **Network DBMS** implementation of the Bill of Materials Structure.
  - Hashed or Randomised access by Key for the Master file, followed by a linked-list of pointers to the records in the Variable files.
- In addition to being superior to HDBMS for OLTP due to the absence of the B-Tree, it is considerably superior for the Bill of Materials implementation because the duplication is deployed to the child files, the records of which are much smaller.

IDEF1X Notation

8 In those days, in database science, all theory and practice was produced by engineers within the big five DBMS firms, including patents and internal academic papers, with very few published. See the References in the *Relational Model*. An independent academia was virtually non-existent. When it did start, it was insanity, solutions to problems in total isolation from reality, such as MVCC and Ingres, the "dbms" that never worked. Same with its bastard son PostGres**NON**sql.

It is because Codd was a theoretical engineer, not a pure academic divorced from reality, that academics hate him; suppress the *Relational Model*; undermine and sabotage Relational theory and practice, at every opportunity.

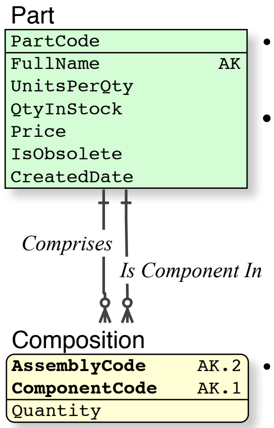
Hierarchy

Multiple Parent • Solution

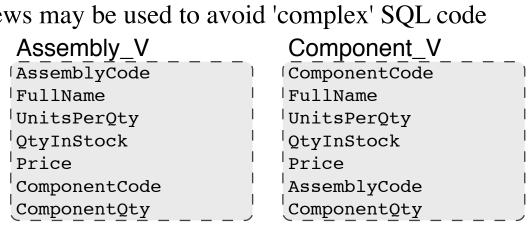
4 Multiple Parent • Normalised

- Sometimes called "two way" meaning two trees for a given row (one "up"; and one "down"): a row has **multiple parents** in the same table
- This is known as the **Bill of Materials** structure, available since 1970, on genuine SQL Platforms since 1981.
  - No duplication due to genuine Relational perspective, and Normalisation.
  - it overcomes the *Bill of Materials Problem* [§3] beautifully
  - put another way, it resolves a many-to-many relation between rows in the same table [Part as Assembly; Part as Component]
- Bill of Material **Explosion** (full exposition of the hierarchy; any branch of the tree; either ancestry or progeny) requires a **stored proc** (multiple rows, a vector)
- Exposition of **Lineage** is not possible due to multiple parents
  - Since parents are multiple, `Part::Path` is no longer 1::1, thus it cannot be deployed in `Part`, it may be deployed in the View.
- Whether circular references are to be prevented or not depends on the data (permitted in §4.1; prevented in §4.2)
  - If it is to be prevented, it must be validated in the Transaction [9] that adds the row to the associative table.

4.1 Example Inventory

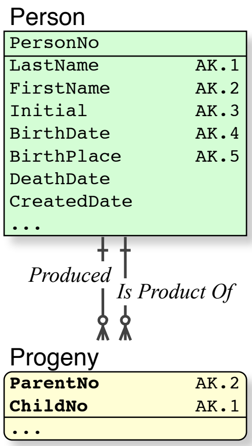


- `Assembly` and `Component` [§3] are Normalised into `Part`
  - `Each Part [Assembly] comprises 0-to-n Composition[Component Parts]: many child Parts`
  - `Each Part [Component] is component in 0-to-n Composition[Assembly Parts]: many parent Parts`
- Views may be used to avoid 'complex' SQL code



- `AssemblyCode` & `ComponentCode` are RoleNames for `PartCode` (Relational concept)
  - `Each Composition [AssemblyCode] is constituent of 1 Assembly [Part]`
  - `Each Composition [ComponentCode] composes 1 Part[Component]`
- The AK is not *required*, it is provided for performance: search by `ComponentCode`

4.2 Example Progeny



- Parent and Child are Normalised into `Person`
  - `Person produced 0-to-n [child] Persons`
  - `Person is product of 0-to-2 [parent] Persons`
- `ParentNo` & `ChildNo` are RoleNames for `PersonNo`
  - `Each Progeny[ParentNo] is 1 [producer] Person`
  - `Each Progeny[ChildNo]  is 1 [product] Person`

```
Ⓒ Progeny.Parent_LE_2_ck
  CHECK Person_ParentCount_fn( @ChildNo ) <= 1
Ⓒ Progeny.Parent_NE_Child_ck
  CHECK @ParentNo != @ChildNo
```

IDEF1X Notation

9 In **Open Architecture** databases, direct writes to the tables are prohibited, and all changes to the database are made via a set of OLTP Transactions, which is the Database API.

