BASICS: About Data
Relational Information Systems Chapter 1.1 (Revised 99/9)

October 31, 2006

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

1 Relations

Consider the data in figure 1. A considerable amount of data is represented, and it is not difficult to see what it means: a *Salesman* called `Hannah Trainman` has obtained an order from the `Pennsylvania Railroad` for `37 Car`s, which has been recorded on Order Number `4`; and so on. More inspection will reveal that Order Number `4` also includes `11 Toy Train`s sold by the same *Salesman* to the same *Customer*; that `Hannah Trainman` is involved with two other *Customer*s via two other orders concerning `13 Locomotive`s and `48` additional `Car`s; that a total of 55 `Toy Train`s have been ordered, etc.

Notice that we are hampered in making these additional observations because the data is not ordered in any convenient way. For instance, it might be handy if all rows for a given Order Number were together and the groups arranged by ascending Order Number. We might also get rid of some of the duplicated information, such as the fact that each Order Number involves exactly one *Customer* and exactly one *Salesman*. If all the Order Number `4`s were together, for example, we need not repeat the `Pennsylvania Railroad` or `Hannah Trainman`. The result of these considerations appears in figure 2.

Note that the representation of the data is now more complicated in that we have added horizontal lines to distinguish the items pertaining to different orders. These lines correspond to various implementation devices, such as trailer records, repeating groups, pointers, etc.

Orderbook

(*Ord#*	*Cust*	*Sales*	*Assembly*	*Qty*)
4	Pennsylvania Railroad	Hannah Trainman	Car	37
3	London & Southwestern	Eric Brakeman	Car	23
2	New York Central	Natacha Engineer	Locomotive	1
7	Grand Trunk Railway of Canada	Natacha Engineer	Locomotive	47
3	London & Southwestern	Eric Brakeman	Caboose	3
5	New York Central	Hannah Trainman	Locomotive	13
7	Grand Trunk Railway of Canada	Natacha Engineer	Caboose	43
8	Great North of Scotland	Eric Brakeman	Toy Train	37
1	Great North of Scotland	Eric Brakeman	Locomotive	2
5	New York Central	Hannah Trainman	Car	31
6	Baltimore & Ohio	Hannah Trainman	Car	17
4	Pennsylvania Railroad	Hannah Trainman	Toy Train	11
3	London & Southwestern	Eric Brakeman	Locomotive	5
1	Great North of Scotland	Eric Brakeman	Toy Train	7
7	Grand Trunk Railway of Canada	Natacha Engineer	Car	139

Figure 1: An Instance of the Relation *Orderbook* (*Ord#, Cust, Sales, Assembly, Qty*)

Orderbook

(*Ord#*	*Cust*	*Sales*	*Assembly*	*Qty*)
1	Great North of Scotland	Eric Brakeman	Locomotive	2
			Toy Train	7
2	New York Central	Natacha Engineer	Locomotive	1
3	London & Southwestern	Eric Brakeman	Car	23
			Caboose	3
			Locomotive	5
4	Pennsylvania Railroad	Hannah Trainman	Car	37
			Toy Train	11
5	New York Central	Hannah Trainman	Locomotive	13
			Car	31
6	Baltimore & Ohio	Hannah Trainman	Car	17
7	Grand Trunk Railway of Canada	Natacha Engineer	Locomotive	47
			Caboose	43
			Car	139
8	Great North of Scotland	Eric Brakeman	Toy Train	37

Figure 2: *Orderbook* re-ordered 1

Orderbook

(*Cust*	*Ord#*	*Sales*	*Assembly*	*Qty*)
Baltimore & Ohio	6	Hannah Trainman	Car	17
Great North of Scotland	1	Eric Brakeman	Locomotive	2
			Toy Train	7
	8	Eric Brakeman	Toy Train	37
Grand Trunk Railway of Canada	7	Natacha Engineer	Locomotive	47
			Caboose	43
			Car	139
London & Southwestern	3	Eric Brakeman	Car	23
			Caboose	3
			Locomotive	5
New York Central	2	Natacha Engineer	Locomotive	1
	5	Hannah Trainman	Locomotive	13
			Car	31
Pennsylvania Railroad	4	Hannah Trainman	Car	37
			Toy Train	11

Figure 3: *Orderbook* re-ordered 2

We could take the process further by grouping together all information for each *Customer*: figure 3.

Here there is a second type of horizontal line, separating subgroups for different Order Numbers within the group for each *Customer*. This further breakdown is not unique: we could also group by *Salesman* : figure 4.

Which of these modifiactions of the data of figure 1 we might choose depends on the application we have in mind. The information would probably be *recorded* in sequence of Order Number, since each *Salesman* would file heir orders as hey acquires a customer, and order numbers would be allocated in sequence. A summary of revenues by *Customer* would require grouping by *Customer*, but an analysis of *Salesman* activity could use a grouping by *Salesman*. None of the above would improve on figure 1 for an examination of what *Assembl*ies were selling well.

What is important is that none of the alternative representations of figure 2 through figure 4 adds information which is not already present in figure 1, and figure 1 is the simplest representation in that it uses no extraneous constructs (e.g., horizontal lines). The alternative representations give us handy ways of viewing the data for particular uses, but they do not augment the essential information of figure 1. In fact, they can cloud the issue for some applications by emphasizing inappropriate groupings and orderings.

Thus we take figure 1 to be the essential and simplest form of the data, and will represent all data in this or analogous forms. Figure 1 is an instance of an *m-ary relation* (with $m = 5$) satisfying the following properties.

1. All rows are distinct.
2. The ordering of rows is immaterial.
3. Each column is labelled, making the ordering of columns insignificant.
4. The value in each row under a given column is "simple".

Orderbook (*Sales*	*Ord#*	*Cust*	*Assembly*	*Qty*)
Eric Brakeman	1	Great North of Scotland	Locomotive	2
			Toy Train	7
	3	London & Southwestern	Car	23
			Caboose	3
			Locomotive	5
	8	Great North of Scotland	Toy Train	37
Hannah Trainman	4	Pennsylvania Railroad	Car	37
			Toy Train	11
	5	New York Central	Locomotive	13
			Car	31
	6	Baltimore & Ohio	Car	17
Natacha Engineer	2	New York Central	Locomotive	1
	7	Grand Trunk Railway of Canada	Locomotive	47
			Caboose	43
			Car	139

Figure 4: *Orderbook* re-ordered 3

The fourth of these properties is not essential to relations in general, and is vague: what do we mean by "simple"? If it were interpreted specifically to exclude relations as legal values, it would characterize relations in *first normal form*. If it were set aside, the resulting relations would be *non-first-normal-form* (*N1NF*, or NF^2), or *nested* relations. We will return to these later, but until then all relations will be 1NF, so we have included the fourth property. In practice, "simple" will mean limited to well-established types such as numeric, text, or Boolean.

Because we are going to view relations in forms other than tables, we will introduce terminology which not specific to tables. The rows of figure 1 are called the *tuples*, or "n-tuples" of the instance of the relation. This generalizes words such as "quintuples".

The columns of figure 1 are labelled by *attributes*. An attribute is associated with a set of values called a *domain*. That is, `Hannah Trainman, Eric Brakeman` and `Natacha Engineer` are all elements of the domain associated with the attribute *Salesman*. A domain may have several attributes associated with it, in different relations or in one relation. In this book, we will use `typewriter` font for all domain values, and *italic* font for attribute names and for relation names.

We can be more precise, if still informal, if we define a *domain* to be a set of values and an *attribute* to be a label. Then a *tuple* is *a mapping from attributes to domains*, and a *relation* is a set of tuples, more specifically, *a subset of the Cartesian product of its domains*.

2 Decomposition

You may still be bothered by figure 1 because it seems an awkward and verbose way to describe the information. If you agree that the various ways of grouping and sequencing the tuples force the data into restricted forms and obscure its general nature, you may nevertheless feel that these groupings and orderings express some aspects of the "meaning"

(*Ord#*	*Cust*	*Sales*)
4	Pennsylvania Railroad	Hannah Trainman
3	London & Southwestern	Eric Brakeman
2	New York Central	Natacha Engineer
7	Grand Truck Railway of Canada	Natacha Engineer
5	New York Central	Hannah Trainman
8	Great North of Scotland	Eric Brakeman
1	Great North of Scotland	Eric Brakeman
6	Baltimore & Ohio	Hannah Trainman

(*Ord#*	*Assembly*	*Qty*)
4	Car	37
3	Car	23
2	Locomotive	1
7	Locomotive	47
3	Caboose	3
5	Locomotive	13
7	Caboose	43
8	Toy Train	37
1	Locomotive	2
5	Car	31
6	Car	17
4	Toy Train	11
3	Locomotive	5
1	Toy Train	7
7	Car	139

Figure 5: An Instances of Relations *Orders* and *OrdLine*

of the data. I agree with you. But the advantages of the the very simple and symmetrical[1] form of figure 1 should not be thrown away. Let us see what we can do to capture the meaning of figure 1 better within the framework of properties 1–4.

We have observed that each Order Number involves one *Customer* and one *Salesman*. This is, in a sense, a unit of meaning in its own right: an order might be defined as the result of a deal between a *Customer* and a *Salesman*. Since this might be considered an independent fact, we might represent it separately, say in a 3-ary (ternary) relation, *Orders*(*Ord#, Customer, Salesman*).

The attributes *Assembly* and *Qty* must now be dealt with separately. The "meaning" that is significant here is that an assembly is part of an order—each order may be for several assemblies—and the quantity is a descriptive item associated with the assembly and the order. Thus, *Ord#* must be linked with *Assembly* and *Qty*, say in another 3-ary relation, *OrdLine*(*Ord#, Assembly, Qty*). The attributes *Customer* and *Saleman* are not directly relevant—or, if we need to know their connection with *Assembly* we have lost no information by splitting up figure 1 as long as we can somehow use *Ord#* as a link between *Orders* and *OrdLine*. The result of the split is shown fully in figure 5.

These two linked relations are an example of a *database*, which we can define in general as a set of relations. (Codd [1] originally modified "relations" with the adjective "time-varying". We will save this aspect until we come to updates, later.)

The process of breaking up the original relation into two is an example of *decomposition*, and this is an aspect of *database design*. Let us call the original relation

OrderBook(*Ord#, Customer, Salesman, Assembly, Qty*)

and the two that result from the decomposition

Orders(*Ord#, Customer, Salesman*)

OrderLine(*Ord#, Assembly, Qty*)

We will soon discuss relational algebra operators which allow us to decompose *OrderBook* into *Orders* and *OrderLine*, on one hand, and to reconstruct *OrderBook* from *Orders* and

[1]In the sense that no one attribute, such as *Ord#*, is favoured by the arrangement of tuples.

OrderLine, on the other. For the moment, we are content to observe that we can go both ways: the process of decomposition loses no information. How do we do it in general?

In the 1970's, database theorists spent considerable energy and ink seeking techniques for database design, including decomposition. This was a worthy endeavour, but it ended in a welter of NP-completeness, and even undecidability, results. In the process it also produced very arcane design problems, which are unlikely to arise in practice. There turned out to be no silver bullet, no crank one could turn algorithmically, to design a database from some simple set of basic facts. Here, we will advocate a seat-of-the-pants approach which requires that the designer *understand* the application and the data. There are not too many rules, but we can say some things.

The theoretical strands included normalization theory, decomposition theory, and dependency theory. These are all attempts to capture the semantics of data through formal, sometimes syntactical, analysis. The problem is that not all semantics can be tamed in this way. But some of the ideas are powerful and helpful. We look at the simplest aspects of dependency theory: the basic idea of functional dependence, and the consequent idea of a key of a relation.

A *key* of a relation is a *minimal subset of its attributes which can be used to identify each tuple uniquely*. That is, we can play a game: you tell me a value for the key, and I will look in the relation and find the one tuple that has that value (or no tuple at all, since there may be none). If there is more than one tuple, that attribute or set of attributes that you claimed was a key, is not.

In *Orders*, the key is the single attribute, *Ord#*. We can see this from the data: there are exactly as many tuples as there are different values of the key. More important, we can see it from the meaning: an order is understood to be a deal between a *Customer* and a *Salesman*, so it cannot involve a second *Customer–Salesman* pair. The first approach, inspecting the data, uses the "extension" of the relation. The second approach, relying on the meaning, uses the "intension". This latter shows how the notion of key captures an aspect of the semantics of the data. We could also speak of the "intention" of the relation; what was intended in designing it.

In *OrderLine*, the key is the pair of attributes, *Ord#, Assembly*. We can see this from the data, as before, but it is better to think about the intension. An order consists of various quantities of different assemblies. There is no point in putting any one assembly down more than once in any given order: were we to do so, we could achieve the same end better by adding up all the quantities associated with that same assembly in the order, and replacing all the tuples by one.

Note also, in *OrderLine*, that *Qty* is almost a key by extension. If one of the 37s were changed to a 38, the extension could lead us to believe that *Qty* is a key. But this makes no sense, and must be rejected by intension, because, of course, there is no reason why two of the values of *Qty* could not be the same in *OrderLine*. The data shown for *OrderLine* gives a counterexample to the supposition that *Qty* is a key. So also in the extension shown for *Orders* there are counterexamples to each of the suppositions that *Customer* is a key, that *Salesman* is a key, and that *Customer* and *Salesman* together is a key.

Finally on keys, a "superkey" of a relation is any set of its attributes which can be used to identify each tuple uniquely. Any superset of a key is a superkey. In particular, the set of all attributes of a relation is a superkey. Superkeys are not particularly interesting: we refer to them here to underline the importance of "minimal" in the definition of a key.

Going beyond keys, a *functional dependence* is *the relationship between two sets of attributes, in which, given a value for the first (determining) set, at most one value for the second (dependent) set can be found in the data.* A key is thus the determining set for which

the whole set of attributes in the relation is the dependent set. The adjective, "functional", comes from mathematics, where a *function* is defined to be a many-to-one relation. We can play the same game as before: you give me a value for the determining set, and I will find in the data at most one value for the dependent set.

In *OrderBook*, the key is the pair, *Ord#, Assembly*, the same as for *OrderLine*: this is not a coincidence, but the result of the decomposition. There are additional functional dependences not resulting from this key: *Ord#* $\longrightarrow$ *Customer* and *Ord#* $\longrightarrow$ *Salesman*, where $\longrightarrow$ means "functionally determines". These last can be written as a single functional dependence, using sets of attributes,

Ord# $\longrightarrow$ *Customer, Salesman*

It is also not a coincidence that this functional dependence becomes the key in *Orders*. In fact, noting that a *part* of the key, *Ord#, Assembly*, is the determining attribute of another functional dependence, is a trigger for one of the automated procedures for database design. (*Noticing* it is a result of checking the relation for "second normal form", part of normalization theory.) But we prefer to detect the need for decomposition in the way we originally did, by realizing, through understanding the data, that *Ord#, Customer* and *Salesman* have independent meaning which should be recorded separately.

It would be possible to split *Orders* further, say, into *OC*(*Ord#, Customer*) and *OS*(*Ord#, Salesman*), but we have no reason to do so. It would be wrong, however, to split *Orders* into *OC* and *CS*(*Customer, Salesman*), because *Customer* cannot correctly provide the link between the two components. In trying to reconstruct the original information, we would have no choice but to form, for example, four tuples involving `New York Central`, linking it to both *Ord#* `2` and `5`, and to both *Salesman* `Natacha Engineer` and `Hannah Trainman`. This would be two more tuples than the original and we would have *lost* information.

Now that we have talked *OrderBook* and its decompositions pretty well to death, it is best to turn to another example to illustrate the database design process. The problem we tackle is, given a set of attributes and their meaning, find a suitable decomposition. Since we can specify the "meaning" only loosely, unlike the situation of an analyst conducting extensive interviews with prospective users of a new database, there will be many correct answers depending on many valid interpretations and sets of assumptions. It will be important to elucidate and to demonstrate our assumptions and interpretations. To elucidate, we can look for keys and functional dependences. To demonstrate, we show sufficient *sample data* to eliminate any unintended dependences.

Here is an exercise [2], in which the database attributes are italicised. It is an interesting exercise, in that different decompositions and keys can reflect different assumptions about the market, retailer, and manufacturer. We will pursue only a few of many possible interpretations.

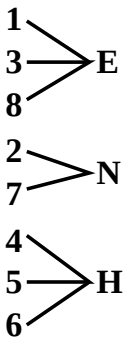
An *Agent* represents a *Product* with a given *Price*, made by a *Company*.

Suppose the *Agent* is a retailer who sets the *Price* for each *Product*, depending on the *Company* making it. Then there can be no further decomposition: everything is deeply intertwingled. The key is *Agent, Product, Company*. An example is

(*Agent*	*Product*	*Company*	*Price*)
`Joe`	`Widget`	`Acme`	`1.00`
`Joe`	`Widget`	`Star`	`1.00`
`Joe`	`Gizmo`	`Acme`	`1.00`
`Joe`	`Gizmo`	`Star`	`2.00`
`Sue`	`Widget`	`Acme`	`2.00`

Ord#	*Sales*
1	E
2	N
3	E
4	H
5	H
6	H
7	N
8	E

Table



Graph

	H	E	N
1		1	
2			1
3		1	
4	1		
5	1		
6	1		
7			1
8		1	

Matrix

Figure 6: Table, Graph and Matrix Forms of *OS*(*Ord#, Sales*)

(Note that we have shown enough tuples that no one attribute of *Agent, Product* or *Company* determines *Price*; nor do any pair of these attributes.)

Suppose, differently, that the *Agent* represents the *Company*, and does not control the price. Then one decomposition could separate *Agent* and *Company*, with *Agent* as key (because we assume hey does not work for more than one *Company*), from *Company, Product, Price*, with *Company, Product* as key.

(*Agent*	*Company*)	(*Company*	*Product*	*Price*)
`Ann`	`Acme`	`Acme`	`Widget`	`2.00`
`Joe`	`Acme`	`Acme`	`Gizmo`	`1.00`
`Sue`	`Star`	`Star`	`Widget`	`1.00`

A variation on this second supposition is that *Agent*s represent only certain products for their company, in which case the decomposition will be (*Agent, Company, Product*) and (*Company, Product, Price*), with *Agent, Product* as the key for the first.

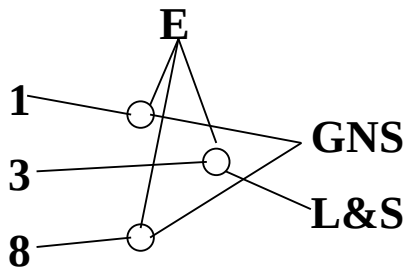
Design examples such as this are simplistic in that they assume that all the attributes are known and do not conflict. Often the analyst must tease out what the attributes are going to be. Often what is an attribute in one design is a relation in another, or a value in yet another. Design problems may be presented "bottom-up", in which different relations have been constructed by different units for different purposes, and must be integrated: then there are questions of "interoperability" and semantic heterogeneity.

3 Other Perspectives on Relations

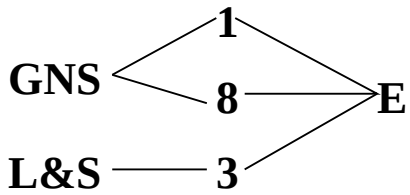
We have said that representing relations as tables is limiting because relations are an abstraction and tables only one concrete way of representing them. We get more flexibility by looking at relations in other ways. Hence we introduce two new forms, *graph form* and *matrix form*. Figure 6 shows all three forms for a simple binary relation (derived from the relations of figures 1 and 5, using obvious abbreviations of the data).

In the table form, each tuple is represented by a row, in the graph form by an edge, and in the matrix form by a `1` entry.

These alternative representations are sometimes much handier than the table form. For instance, the graph form makes plain the fact that *Ord#* is a key by the way that the edges converge from different order numbers to each *Salesman*. Divergence of any two edges from



1. Three tuples of *Orders*



2. Special case: revealing key

Figure 7: Exploiting the Graph Form

an order number would violate the supposition that *Ord#* is a key, and be immediately apparent. In the matrix form, this is again clear, since each row has only one entry.

What if the relation is not binary? For instance, suppose we must represent *Orders*(*Ord#, Sales, Cust*) in each of these forms. The table form is easy, which is why it is most commonly used: figure 5 gives it.

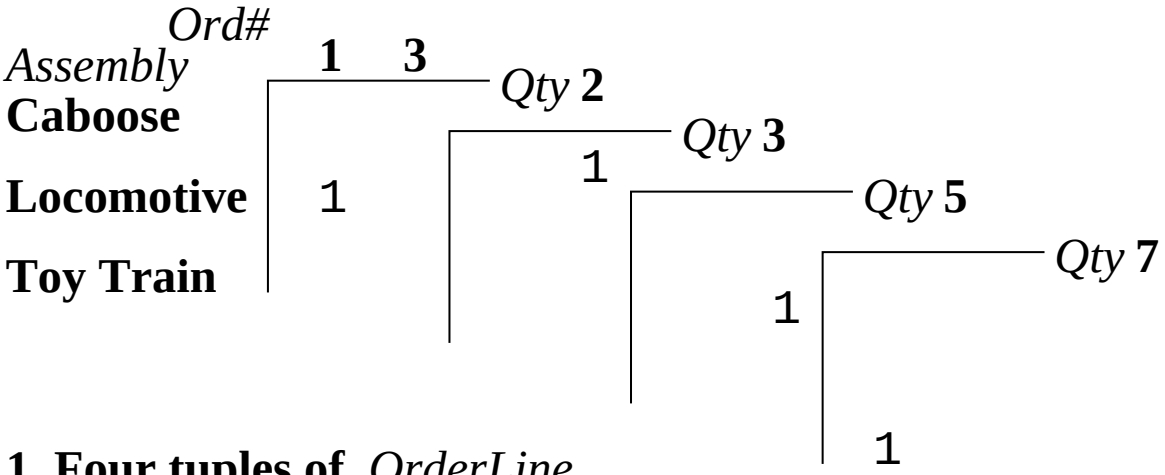
The graph form of a ternary relation would in general have "three-ended edges" and would be a "regular hypergraph". Figure 7 shows three of the tuples of *Orders* in this form. It also shows a much more revealing variant of the graph form for this case, which highlights that *Ord#* is a key: the tuples are represented by pairs of edges sharing a common value of the key, *Ord#*.

Similarly, the matrix form of a ternary relation is a three-dimensional array. Figure 8 shows four of the tuples of *OrderLine* in this form. It also shows a more revealing variant of the matrix form for this case, which highlights that *Ord#, Assembly* is a key: the tuples are represented by placing the value of the non-key attribute, *Qty*, where the 1s were shown in the general matrix representation.

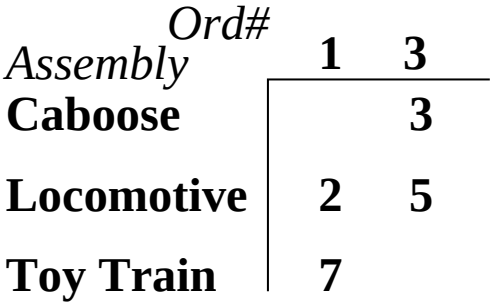
4 Some Applications

We now explore a variety of different relations to illustrate the flexibility of the data structure and some other variants of the three representations. Where the graph form is given, we invite you to formulate the table and matrix forms, and so on.

Figure 9 shows a project evaluation and review technique *PERT* network, often used by



1. Four tuples of *OrderLine*



2. Special case, revealing key

Figure 8: Exploiting the Matrix Form

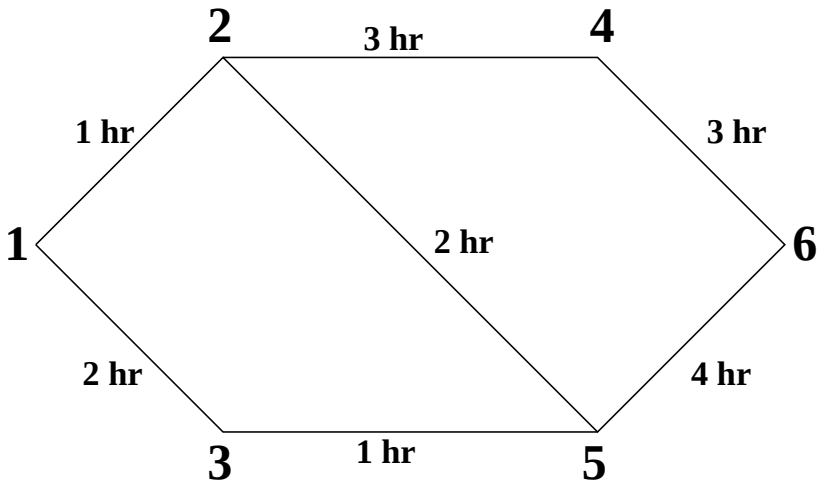


Figure 9: A PERT Network

civil and industrial engineers to plan projects, and particularly to determine the *critical path* (the path from start to finish on which any delay would delay the entire project).

Figure 10 is a "bill of materials", the structure used to represent the components of a manufactured item, and how many of each subassembly are put together to form the next level of component. Here, 2 `E`s make up a `C`, four `C`s go into an `A`, etc. (So 8 `E`s are needed for an `A`. How many `D`s are in an `A`?) This relation is similar to the PERT network, but it is one we will come back to.

If each assembly in the bill of materials had a cost associated with it (say the cost of buying it if it were a base component, or the cost of assembling it from its direct components), this would add another attribute. How should the database be constructed from the attributes from theoriginal bill of materials and these costs?

Figure 11 shows one of many posible relational representations of text. Other representations could reflect the hierarchy of letter within word within sentence within paragraph within .., or the syntax tree of the text after grammatical analysis. Note that order is important in a text, and so the order of the words must be induced on the tuples by showing a sequence number for each word. We do not suggest that this sequence number need be stored explicitly. Indeed, if the text were represented as a sequence of letters (useful, for instance, in cryptography), the sequence number could increase the amount of storage fivefold. A suggestive aspect of the representation shown is that, because order does not matter in relations, the relation is *both* the text and in a simplistic sense the index to the text: "noise words" would ordinarily be removed, and, for a proper index, the attentions of a human indexer would be needed.

Figure 12 shows a more difficult exercise, a diagram consisting of zero-, one-, and two-dimensional constructs (here a point, a two-piece line, and a hexagon). The challenge is to represent this whole diagram (and any other diagram containing similar elements) as a single relation.

A solution is to put one coordinate pair, X, Y, per tuple, with a *Seq*uence number to show the position of the point in the geometrical construct, a *Type* to distinguish 0-, 1-, and 2-dimensional features (the latter are closed lines), and some attribute(s) to group the tuples into subsets corresponding to each construct. This latter flies in the face of some

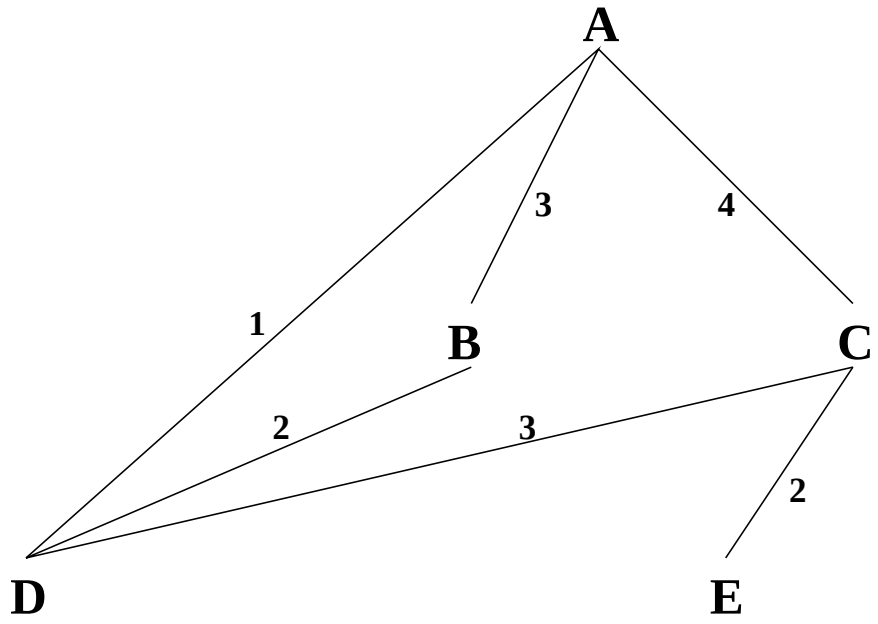


Figure 10: A Bill of Materials

Text (*Word*	*Seq*)
Algebraic	1
data	2
processing	3
techniques	4
can	5
enable	6
applications	7
programmers	8
to	9
work	10
with	11
units	12
of	13
data	14
larger	15
than	16
a	17
single	18
computer	19
word	20

Figure 11: A Text Representation

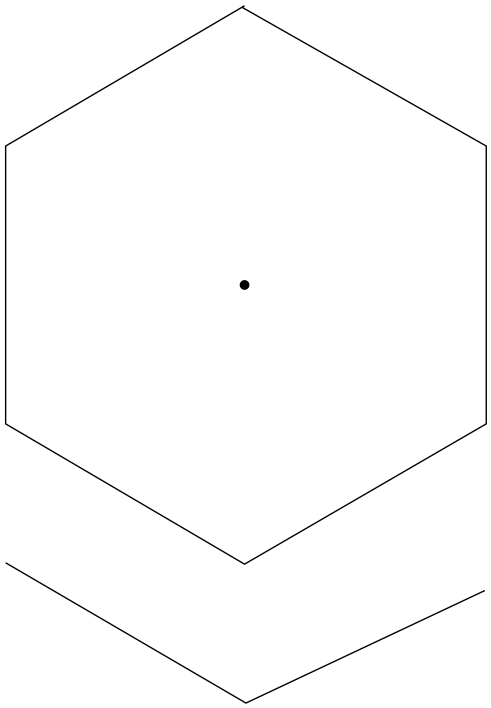


Figure 12: A Diagram

conventional wisdom about relations: that they are unable to represent complex objects, and that, when mapping from the relational to the object-oriented approach, a single tuple should correspond to an "object". As this example shows, it is better to represent "objects" by subrelations. Then we do not have to be pinned down as to what we might mean by "object", and we can remain flexible and mean different (and maybe incompatible) things. In this example, an "object" might be one of the basic constructs, a hexagon, a two-edge line, or a point; but it might also mean a single vertex in such a construct, or an edge.

References

[1] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–87, June 1970.

[2] T. Imielinski and W. Lipski, Jr. A systematic approach to relational database theory. In M. Schkolnick, editor, *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, pages 8–14, Orlando, June 1982.