GIS

A Computing Perspective

Third Edition

Edited by Matt Duckham, Qian (Chayn) Sun and
Michael F. Worboys

First published 1995

ISBN: 978-1-4665-8719-9 (hbk)
ISBN: 978-1-032-53973-7 (pbk)
ISBN: 978-0-429-16809-3 (ebk)

Chapter 2

Fundamental Database Concepts

DOI: 10.1201/9780429168093-2

The funder for this chapter is Matt Duckham

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

2

Fundamental Database Concepts

Sections

This chapter is all about databases, their structure and operations, and the role they play in a GIS. An understanding of databases is fundamental to an understanding of GIS, and indeed of many other types of information systems. Progressing through this chapter, you will learn to:

- discuss the key features of **databases** and the rationale for using them;
- contrast the characteristics and limitations of the **relational database** model and the **graph database** model, the two most important database paradigms;
- develop a database design using **entity-relationship modeling** and the standard diagrammatic language **UML**;
- build simple **queries** to a relational or graph database using a standard language, such as **SQL** or **Cypher**.

Databases are the foundation of GIS. A knowledge of the fundamental principles of databases is indispensable for understanding GIS technology. Most GIS are built upon general-purpose relational databases; certainly all GIS will connect with such systems in a distributed environment. This chapter introduces the reader to the main principles of databases. The general database approach is introduced in Section 2.1, with the principles of database development introduced in Section 2.2. The most common database model is the *relational model*, described in detail in Section 2.3. However, the graph database model is a markedly different approach to databases with growing significance for GIS, explored in Section 2.4.

2.1 *Introduction to databases*

From their earliest days, computers have always been used to convert one data set into another, accomplishing large and complex transformation processes that would be tedious for humans to complete manually. For example, we may wish to predict average annual traffic flows using a computer model of transportation. The input to this system might include data about city populations and about the road network. The transformation is encoded by the transport model, implemented in the computer as software. The output is the traffic flow prediction data, detailing expected flows on each road over time. This approach of treating the computer as a "giant calculator" is illustrated in Figure 2.1.

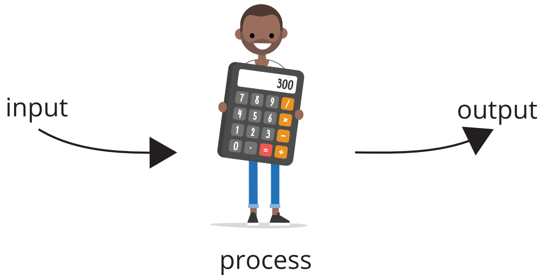


Figure 2.1: The "computer as a giant calculator" paradigm

Thinking of the computer as a giant calculator has several disadvantages, which are discussed in more detail below. In short, such an approach tends to lead to significant duplication of data and even of transformation processes. The alternative offered by the database approach is shown in Figure 2.2. In this case, the computer acts as a useful repository of data, allowing the deposit, storage, and retrieval of data. Data in the store can be accessed, modified, and analyzed in a standard way, ensuring that these and other basic functions are never duplicated.

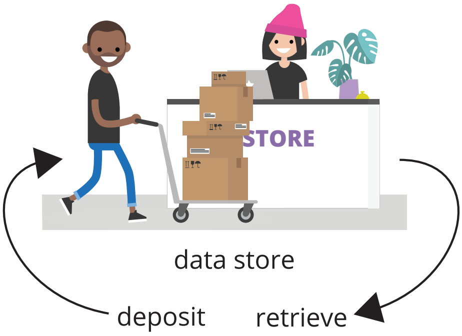


Figure 2.2: The "computer as data repository" paradigm

The remainder of this section delves into more detail on the database approach. A summary of the key features of the database approach can be found in Box 2.1 on the next page.

2.1.1 The database approach

The "computer as giant calculator" and "computer as data repository" are extreme positions. Most applications require a balance of calculation (processing) and a repository of data upon which the processes are to act. We illustrate this balance with a fictitious example.

"Nutty Nuggets" is a dinner kit company, established in 2018, delivering tasty vegetarian slow-food meal kits to customers to cook at home. Starting from just a handful of local customers, the business grew quickly by word-of-mouth. To help cope with their growing order book, the two owner-managers decided to apply their computer skills to help keep on top of business.

Box 2.1: Databases in a nutshell

In order to act effectively as a data store, a computer system must have the confidence of its users. A database must be dependable and continue to operate correctly even in the case of unexpected events, such as power or hardware failure (*reliability*). As far as possible, data in the database should be correct and consistent (*integrity*). A database must be able to prevent data being used in unauthorized ways (*security*), but it must offer sufficient flexibility to give different classes of users different types of access to the store (*user views*). Ideally, the database interface should be easy to use for casual or first-time users as well as offer more powerful functions for regular users (*user interface*). Most users will not be concerned with how the database works and should not be exposed to low-level database mechanisms (*data independence*). Users should be able to find out what is in the database (*self-describing*). Many users may wish to use the store, perhaps at the same time or accessing the same data (*concurrency*). Databases should be able communicate with each other in order to access remote data (*distributed database*). Finally, a database should be able to retrieve data rapidly (*performance*). All these functions are managed by a dedicated software application (*database management system*).

They started by populating a spreadsheet with details of their orders, including customer addresses, contact details, the meal type, and meal size in terms of how many people the meal kit should feed. Programs were then written to print out delivery schedules for each day, as well as meal kit contents sheets, helping to streamline the process of making up orders in the kitchen. Figure 2.3 shows on the left the order file being accessed by the two programs. On the right, Figure 2.3 shows the main constituents of order data. The order file is stored on a computer and accessed when required by the programs.

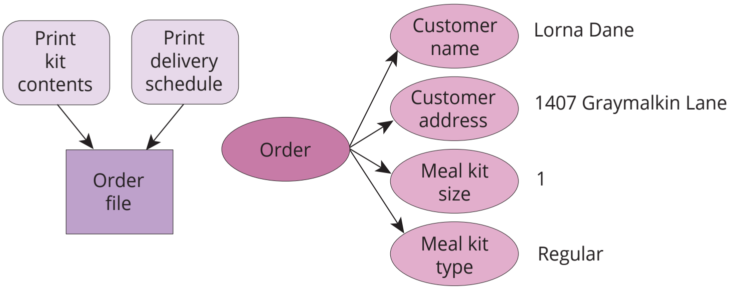


Figure 2.3: Nutty Nuggets stage one: Orders system

Time passed, the order system was successful, and the owners gained the confidence to extend the system to help with the growing issue of stock management. A stock extension was set up, consisting of a stock spreadsheet, and a program to print a daily stock report, highlighting low stock, shown in Figure 2.4. With stock and order details now together in one system, it became natural to link them. A program was written to use the data about customer orders to assist with restocking the kitchen. This program helped reduce costs and waste by helping ensure only enough perishable stock was kept to satisfy upcoming orders, shown at the bottom of Figure 2.4.

The system continued to grow, with new spreadsheets for supplier and customer details added. However, as the system became enlarged, some problems began to emerge, including:

Loss of integrity The system led to problems maintaining the structure and currency of the data. For example, when a staff member entered a new

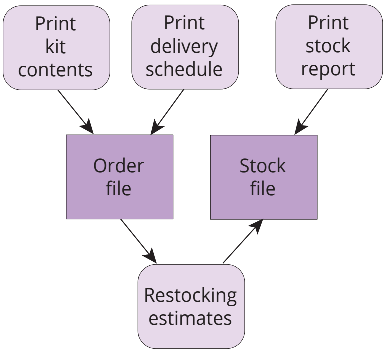


Figure 2.4: Nutty Nuggets stage two: Orders, stock, and restocking system

customer's address in the wrong format, the programs that accessed the data produced garbled output or crashed. Worse still, when one of the owners made a copy of the stock file to work on, the two versions rapidly became out of sync, requiring many hours' work later in the week to merge them back into a single authoritative version.

Loss of independence The programs encoded the relationships between the data items in the files. If the relationships changed, then the programs too had to be changed. This close linkage between program and data was becoming complex and costly to maintain, leading to errors. For example, when the option to pause orders was added to the order spreadsheet (popular with customers going on vacation, for example), the restocking program was not updated to incorporate the data associated with this new feature. As a result, some expensive perishable ingredients were ordered in error and went to waste.

Loss of security An unscrupulous kitchen staffmember working with the stock file also made a secret copy of the orders file, accessing customer names and personal details as the basis for an email scam.

The database philosophy is an attempt to solve these and other problems that occur in a traditional file processing system. Figure 2.5 shows a reorganization of the system, so that all data files are isolated from the rest of the system and accessible to the processes only through a controlled channel. The idea is to place as much of the structure of the information into the database as possible. For example, if there is a relationship between orders and the stock required by those orders, then this relationship should be stored with the data. Databases are *self-describing*, in the sense that they encode both data and the structure of that data. The means of expressing the structure and relationships in the data is provided by the data model.

self-describing

The data model also allows the designer to enter into the database any properties of the data that are expected always to be true, called *integrity constraints*. Integrity constraints are an aid to maintaining correctness of the data in the database, because they only allow modifications to the database that conform to these constraints. An example of an integrity constraint might be

integrity constraint

that the stored meal kit type comes from the list of available options, such as "Regular," "Lo carb," or "No gluten."

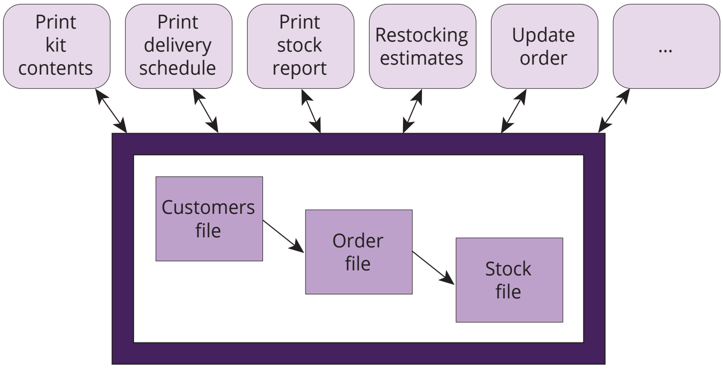


Figure 2.5: Nutty Nuggets stage three: Database approach

The data is collected in one logical, centralized location. A database is created and maintained using a general-purpose piece of software called a *database management system* (DBMS, see Box 2.2 on the following page). The DBMS manages the database by insulating the data from uncontrolled access. A DBMS allows the definition of the data model, supports the manipulation of the data, and provides controlled two-way access channels between the exterior and the database. Not all users have the same requirements of the database: each user group may require a particular window onto the data. This is the concept of a *view*. Views provide users with their own customizable data model, which is a subset of the entire data model, and the authorization to access the sectors of the database that fall within their domain.

DBMS

user view

A database also enables a designer to define the structure of the data in the database, providing levels of authorization that permit different groups of users secure access to data. Different users may access the database at the same time, termed *concurrency*. A DBMS also allows users to access the data in the database without precise knowledge of implementation details, termed *data independence*.

concurrency

data independence

A database can now be more precisely defined as a unified computer-based collection of data, shared by authorized users, with the capability for controlled definition, access, retrieval, manipulation, and presentation of data. Examples of common types of database application include:

Home/office database Home/office database systems are often relatively small and inexpensive, and they may not require concurrent multiuser access. An example is the Nutty Nuggets system.

Enterprise database Enterprise databases are widely used to manage the information of businesses, including financial and personnel data. Enterprise databases must be secure and reliable, run on many platforms, offer high performance, and allow concurrent access by different groups of users. An example might be the database used by a university to manage both staff and student information.

Box 2.2: Elements of a database management system

There are many different components in a DBMS that must work together to answer a query, such as a query to retrieve the address of a customer from the Nutty Nuggets database. The query, communicated to the DBMS using a database *query language*, must first be parsed and analyzed by the *query compiler*. Along the way, the compiler may call the *query optimizer* to optimize the code, so that performance on the retrieval is as good as possible. The *authorization controller* will also ensure the user has the correct privileges to access the requested data. Auxiliary units may be needed to handle constraint enforcement, transaction management, and concurrency control. The *stored data manager* then controls access to physical data storage devices, as well as back-up and recovery in the case of system failure. Throughout the process, stored data must be mapped to the high-level objects referred to in the query using the *metadata catalog*, also termed the *data dictionary*, which stores the information about the structure of the stored data and the data model.

Document store Document stores, also called document-oriented databases, enable storage and retrieval of a wide range of documents, such as web pages, catalogs, and social media and user-generated content. Document stores are highly flexible in the structure of stored data, and so they need specialized query capabilities to efficiently retrieve and query unstructured document contents.

Image database Image and multimedia databases allow the storage and retrieval of a wide range of media data types, such as images, audio, and video. Image and multimedia databases must offer specialized search functions, such as searching for (visually) similar images or for particular information that might appear part-way through a video.

Geodatabase Geodatabases store a combination of spatial and non-spatial data and require complex data structures and analyses. They are discussed throughout this book.

2.1.2 Transaction management

transaction

Databases exist to support their users. A *transaction* is the most basic unit of interaction between a user and a database. Transactions primarily involve one or more of the following operations:

- creation of data in the database,
- retrieval of data from the database,
- update of data in the database, or
- deletion of data from the database.

persistence

Data stored in such a way that it continues to exist beyond the session in which it was created—even after the DBMS and the physical storage media have been shut down—is said to be *persistent*. As we encountered in Chapter 1, data that is not persistent and that is only temporarily stored during active computing processes or sessions is termed *volatile*. Together, the four basic operations (creation, retrieval, update, and deletion) on persistent, stored data above are referred to as *CRUD*.

CRUD

A key issue that must be addressed by database transaction management is support for concurrency. Concurrent access to persistent data confronts us with this problem: if the same data item is involved in more than one concurrent transaction, the result may be a loss of database integrity.

T_1	T_2	B	X	Y
		−\$70		
$X \leftarrow B$		−\$70	−\$70	
$X \leftarrow X + \$70$		−\$70	\$0	
	$Y \leftarrow B$	−\$70	\$0	−\$70
	$Y \leftarrow Y - \$40$	−\$70	\$0	−\$110
$B \leftarrow X$		\$0	\$0	−\$110
	$B \leftarrow Y$	−\$110		−\$110

Table 2.1: Lost update for non-atomic interleaved transactions, T_1 and T_2, with variables X and Y and bank balance B

Imagine a Nutty Nuggets customer already owes $70 on their account. Now suppose that, by chance, two transactions are in progress together. In transaction T_1, the customer is paying their outstanding bill with $70 credit. Concurrently, a finance staff member is debiting a new invoice for $40 to the same account in transaction T_2. Table 2.1 shows a particular sequence of the constituent operations of each transaction, termed *interleaving*. Interleaving can improve database performance, because shorter operations may be executed while more lengthy operations are still in progress. However, interleaving must be controlled carefully to avoid potential problems.

interleaving

For example, transaction T_1 begins by reading the customer's balance B from the database into a volatile program variable X. Crediting the customer's payment of $70 then modifies X to $0. Before the update operation can conclude by writing this volatile data to persistent storage, however, concurrent transaction T_2 begins. Transaction T_2 likewise starts by reading the same customer's balance from the database, into volatile program variable Y. Transaction T_2 then debits the new invoice from that balance, modifying Y to −$110. T_1 then completes its update by writing to the database the new balance of $X = \$0$. When subsequently T_2 completes its update to the database, writing its new balance of $Y = -\$110$, it is as if transaction T_1 never occurred. This is a problem known as *lost update*.

lost update

To prevent lost updates and other problems that can occur with concurrent transactions, any DBMS must be able to guarantee four "ACID" properties:

ACID

Atomicity Atomicity dictates that constituent operations of a transaction must either *all* have their effect on the database or else make *no* change to the database. For example, in a transaction to transfer credit from one customer's account to another, debiting the first and crediting the second account, atomicity requires that either both or neither operations are completed, but never one without the other.

Consistency Consistency ensures that all transactions may only result in valid database states compliant with all the database constraints. For example, in a database where the customer balance is constrained to be a number, any transaction that attempts to update a customer's balance to a non-numerical value, such as "sixty dollars" or "overdrawn" should fail to be executed.

Isolation Isolation requires that concurrent transactions must leave the database in the same state as if the transactions were executed sequentially. For example, the lost update problem above is an example of a failure in isolation, where transactions T_1 and T_2 executed in sequence would result in a different outcome to the interleaved transaction in Table 2.1.

Durability Durability necessitates that transaction outcomes are persistent, even in the case of a serious system failure or power outage. One of the common protocols used by databases to support durability is discussed in more detail in Box 2.3 on the next page.

Databases are the engines of today's data-driven applications, including GIS and the WWW. Databases provide reliable, high-performance CRUD operations for authorized access to secure data, all underpinned by ACID guarantees for concurrent transactions. Most importantly, databases give users and developers confidence that data in the database will support transparent and standardized access methods to stored data, independent of the specific details of any data structures. The DBMS is the software component that is responsible for delivering these essential functions and features of any database.

2.2 Database design

A DBMS is a general-purpose information system that must be customized to meet the requirements of particular applications. In order to do this, we need to develop a precise idea of the way that information will be structured in the database, and the way data items will relate to one another. Here, we are not so much concerned with the actual data in the database, as with the *kinds* of data that we expect. For example, in a mapping application, we are not so much concerned with individual data items, "Addis Ababa," "Ethiopia," "Mount Entoto," as with data types, **city**, **country**, and **mountain**. We abstract from information system content to information structure.

As a result, this section focuses primarily on system analysis and design stages of the system development process introduced in Section 1.3.1. During these stages, the main task for the database analyst is the construction of the conceptual computational model for the database, termed a *conceptual data model*. A database designer will then tailor the conceptual data model to the particular kind of DBMS on which the system will be implemented, called a *logical data model*. We will come back to the database design when we look in more detail at the different DBMS models and structures we may need, in Sections 2.3 and 2.4.

Box 2.3: Two-phase commit (2PC) protocols

Durability and atomicity require DBMS have robust mechanism to deal with system failures. One of the most important ways that systems such as DBMS achieve durability and atomicity is the two-phase commit (2PC) protocol. The essence of 2PC is that each subsystem involved in a transaction must coordinate first in the *request* to make a change before actually *completing* any change. The 2PC begins with a transaction manager or coordinator making a request for a change to each subsystem responsible for any resources involved in that change. The subsystems then complete the transaction up to, but not including, finally committing that transaction. During the request phase, each subsystem also writes a persistent log of its own operations to aid in the recovery from any failure. Subsystems then respond directly to the transaction manager to indicate assent to the request, so completing the request phase of 2PC. If the transaction manager receives affirmative responses from all the requested resources, then a second round of messages from the transaction manager directs each subsystem to complete the pending transaction, and commit the change. However, if any subsystems did not respond correctly or in time to the request phase, the transaction manager can instead send a *rollback* or abort message, instructing subsystems to recover the state of the database immediately prior to the transaction from their persistent logs. As a result, only if all components are successful in making the update will the transaction be finally committed. A failure with any constituent operation or subsystem means that the transaction will not be committed and the database will roll back to its last valid state.

2.2.1 Conceptual data modeling

conceptual data model

A *conceptual data model* is a model of the proposed database system that is independent of any implementation details. A conceptual data model must express the structure of the information in the system: that is, the types of data and their interrelationships. The correctness (integrity) of the information in a system is often a critical factor in its success. Correctness is maintained as much as possible by the specification of integrity constraints that impose conditions on the static and dynamic structures of the system. A data model should allow the specification of a range of integrity constraints.

A good conceptual data model can act as an efficient means of communication between the analyst, designer, and potential users. This will aid the design and implementation of the system. Further, when the system is eventually implemented, the conceptual data model provides a basic reference for users who need to understand the structure of the data in the system (see Figure 2.6).

In summary, effective conceptual data models:

1. provide a framework that allows the expression of the structure of the system in a way that is clear and easy to communicate to non-specialists;

2. contain sufficient modeling constructs so that the complexity of the system may be captured as completely as possible; and

3. have the capability for translation into implementation-dependent models (i.e., logical and physical models), so that the system may be designed and built.

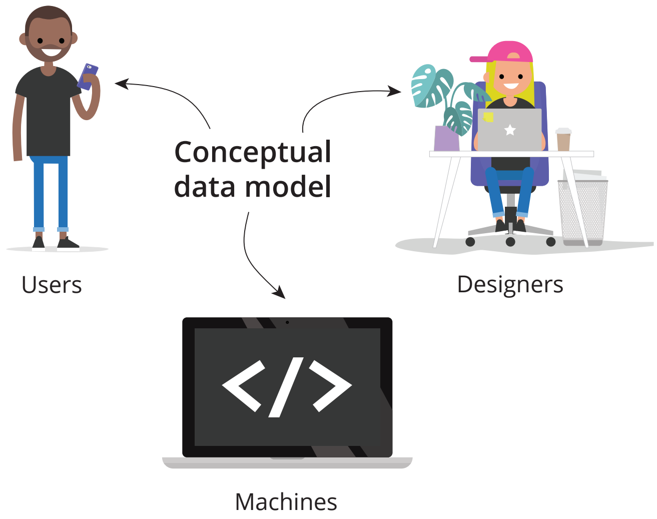


Figure 2.6: The conceptual data model as mediator between users, designers, and machines

2.2.2 *Entity-relationship models*

Imagine that we are responsible for designing the Nutty Nuggets database introduced above. This hypothetical system should contain data on customers (e.g., names, contact details), orders (e.g., meal types and sizes), delivery addresses (e.g., street addresses, zip codes); and meal kit contents (e.g., ingredients and contents). How would we make a start? Well, we have already started in that we have elicited some requirements of the system and expressed these requirements in the form of collections of entities and their relevant properties. This is the simple and powerful idea behind one of the most compelling and widely used approaches to forming a conceptual data model of an information system: the *entity-relationship model* (*E-R model*).

E-R model

entity type

An *entity type* is an abstraction that represents a collection of similar objects, about which the system is going to contain information. In our example, some of the entity types might be **customer**, **order**, and **address**. We make a distinction between the *type* of an entity and an *occurrence* or *instance* of an entity type. For example, we have entity type **address** and occurrences such as '1407 Graymalkin Lane' and '1128 Mission Street' or entity type **customer** and occurrences such as 'Lorna Dane' or 'Roberto Da Costa'. By convention for conceptual modeling, types are rendered in bold, and instance values in single quotes.

entity instance

For conciseness, we will start to use the term "entity" to refer to both "entity types" and "entity instances" as we move through the text as long as the meaning is clear from the context. For example, the caption in Figure 2.8 reads "Two entities ..." rather than "Two entity types ...". However, it is essential always to be precise in our own minds about this fundamental difference between a *type* and an *instance*, between *categories* and *individuals*.

Another aspect where precision is essential is in distinguishing between the entity instance itself and the *identifiers* of that instance. When we write about the customer Lorna Dane, the value 'Lorna Dane' is not the customer herself, of course, but a data item that serves to identify that customer. The data item 'Lorna Dane' is associated with a particular occurrence of the entity (type) **customer** as its name. Thus, entity types have properties, called *attribute types* that serve to describe them. For example, entity type **customer** has attribute types **title**, **given name**, **family name**, and telephone **contact number**. A particular occurrence of **customer** would have associated with it occurrences of these attributes, such as the value '+1-610-555-0195' assigned to the attribute (type) **contact number**.

attribute type

The attachment of attribute types to an entity type may be represented diagrammatically as an *entity-relationship diagram* (*E-R diagram*). As shown in Figure 2.7, entity and attribute types are shown in rectangular segmented boxes; entity types are labeled in the colored box header; attribute types are listed in the section below the header. The E-R diagrammatic notation presented here is common but not standard; there are many variations in use. However, this notation is based on UML (unified modeling language), a standard diagrammatic language widely used for describing many different aspects of information systems.

E-R diagram

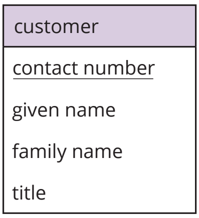


Figure 2.7: An entity type and its attribute types

An important characteristic of an entity is that any occurrence of it should be capable of unique identification. The means of unique identification is through the value of a subset of its attribute types. The name of a person, for example, is frequently insufficient to identify that person uniquely. Similarly, while there is only one place with the address '1128 Mission Street' (in South Pasadena, California), places with the address '11 Mission Street' appear in California, New York, South Carolina, and Connecticut in the US, for example, as well as in other countries around the world. An attribute type or combination of attribute types that serves to identify an entity type uniquely is termed an *identifier*. By definition, an entity type must have at least one identifier. The attributes comprising the chosen identifier are often underlined. In our example, attribute type **contact number** is assumed to be an identifier of **customer** (under the assumption that no two customers share the same telephone number).

identifier

So far, the model comprises a number of independent entity types, each with an associated set of attribute types, one or more of which serve to identify uniquely each occurrence of the type. The real power in this model comes with the next stage, which provides a means of describing connections between entity types. A question that we might want to ask of our finished system is, "Which customers placed which orders?" This question can only be

relationship
relationship type
relationship occurrence

answered by forming a link between customers and their orders. This connection is called a *relationship*. A *relationship type* connects one or more entity types. A *relationship occurrence* is a particular instance of a relationship type. Thus, the incidence of the customer 'Lorna Dane' with the order number '058' is an occurrence of the relationship **placed**. The relationship **placed** between entities **customer** and **order** may be captured in an E-R diagram, as shown in Figure 2.8. Relationships are shown with a labeled line connecting the two entities.

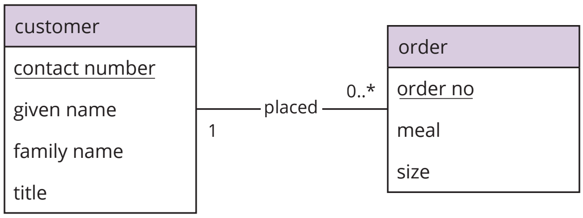


Figure 2.8: Two entities and a many-to-one relationship

Relationships may have their own attributes, which are independent of any of the attributes of the participant entities. In the above example, the relationship **placed** might have the attribute **date**, which gives the date on which the customer placed that order.

many-to-one

cardinality conditions

Relationship types are subdivided into many-to-one, many-to-many, and one-to-one relationships. The relationship **placed** is an example of a *many-to-one* relationship, because each customer may place several (potentially zero, one, or many) orders, but each order must be placed by exactly one customer. This constraint on the relationship is shown diagrammatically by the 0..* and 1 on each side of the relationship. Thus, E-R diagrams allow the modeler to express *cardinality conditions* upon entity occurrences entering into relationships. Cardinality conditions are a type of integrity constraint where the number of entities participating from each side of the relationship is restricted in some way.

Take a moment to note in particular how the location of each label in Figure 2.8 reflects the direction of the relationship. A customer can place zero or more orders so the "0..*" label appears on the right-hand side of the relationship, closest to the **order** entity type (i.e., reading the diagram as: **customer placed** 0..* **order**). An order must be placed by exactly one customer, so the "1" label appears on the left-hand side of the relationship, closest to the **customer** entity type (i.e., read: **order placed** by 1 **customer**). If these symbols were reversed, the diagram would have a very different interpretation (i.e., a rather strange business in which a single order might involve any number of customers, even none at all; but each customer would be required to place one and only one order).

Not all relationships are many-to-one. For example, consider the relationship **contained** between types **order** and **stock** shown as an E-R diagram in Figure 2.9 (top). Each stock item may appear in many different orders, and an order will naturally contain many different stock items (i.e., ingredients).

Such a relationship is called a *many-to-many* relationship. These cardinality conditions are again shown diagrammatically by the 0..* and 1..* on each side of the relationship. Note that these conditions also implicitly specify whether participation in the relationship is *optional* (i.e., 0..*, each ingredient may appear one, many or no orders currently in the system) or *mandatory* (i.e., 1..* each order may have many different ingredients but must have at least one).

many-to-many

The third and final relationship category is exemplified by the relationship **issued** between **staff member** and **staff card**. In this relationship, each staff member must have been issued a staff card, and each staff card is issued to that specific staff member. This is a *one-to-one* relationship. The form of the corresponding E-R diagram is shown in Figure 2.8. The positioning of the two 1s indicates the nature of the relationship.

one-to-one

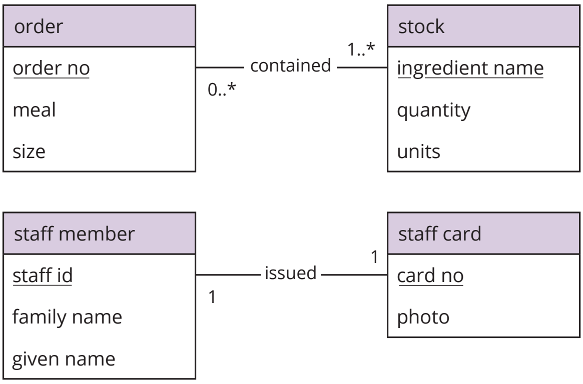


Figure 2.9: Many-to-many (top) and one-to-one (bottom) relationships

All the relationships given as examples so far have been *binary*, in that they have connected together precisely two entity types. A relationship that connects three entity types is called a *ternary* relationship. For example, Figure 2.10 shows a ternary **shipped** relationship, between the **order** that was delivered, the **address** it was delivered to, and the **staff member** who delivered it. Cardinality and participation conditions can apply as with the binary case, but with extra complexity. In the Nutty Nuggets example, we might assume: each order must be shipped to one address by one delivery staff member; but each staff member can deliver zero or many orders to multiple different addresses; and each address might receive zero, one, or many orders, in the latter case possibly delivered by different staff.

ternary relationship

2.2.3 Example E-R model

The E-R model can now be applied to the example of the Nutty Nuggets customer database. Imagine you are an owner-manager at the analysis stage of system development. There is a need to hold information about customers and orders, stock and addresses. Choosing entities and attributes for such a model is a matter of judgment, often with more than one acceptable solution. Sometimes it is difficult to decide whether to characterize something as an attribute or an entity. In the modeling of the Nutty Nuggets database, there

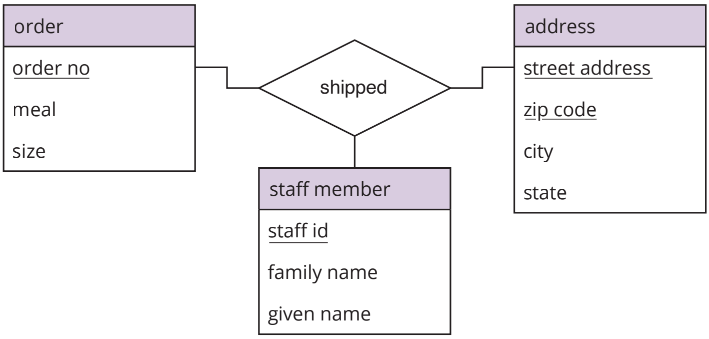


Figure 2.10: A ternary relationship, with cardinality conditions omitted

are several such choices. For example, should we make the city in which an address is located an entity or attribute? Some guidelines include:

- If the data type is relatively independent and identifiable, with its own attributes, then it is probably an entity; if it is just a property of something, then it is an attribute.
- If the data type enters into relationships with other entities (apart from being a property of something), then it is probably an entity.

In our system, the **city** is just a name and just one line of the address, so we choose to make it an attribute of **address**. If **city** had possessed its own attributes, such as **population**, then we would probably have made it an entity and constructed a relationship between entities **city** and **address**. Initial investigation reveals that the following entity types and their attributes are needed:

- **order** (**order no**, **meal**, **size**)
- **customer** (**given name**, **family name**, **contact number**)
- **address** (**street address**, **zip code**, **city**, **state**)
- **stock** (**ingredient name**, **quantity**, **units**)

Most of the attribute type names are self-explanatory, except that attribute types **meal** and **size** of entity type **order** capture the specific meal scheme ordered from the delivery menu (e.g., regular, low calorie, low carbohydrate, vegan) and the number of people the meal kit needs to feed (individual, couple, small or large family). The attributes **quantity** and **units** of **stock** denote the amount of that ingredient in stock (e.g., 10kg, 6l, or 250g). We have mentioned above that **contact number** is taken to be the identifier of the entity **customer**. By adopting this identifier, you will not be able to take business from customers who don't have a phone or who share their phone with other customers. Identifiers for the other entities require further analysis. For the purposes of this system, you decide that stock can be identified by its ingredient name; that each address can be uniquely identified by a combination of the street address (e.g., '1407 Graymalkin Lane') and zip code (e.g., '10573'); and that each order will require a unique identifier to be created for it, **order no**.

As we have already discussed, the entities **customer** and **order** are connected by the many-to-one **placed** relationship, with each customer optionally placing orders, but each order requiring (i.e., mandatory participation) exactly one customer to have placed that order (Figure 2.8). Similarly, each **order** is required to have at least one but possibly more ingredients from **stock**; while **stock** may optionally be included in zero, one, or many different orders (Figure 2.9). You decide to keep the **shipped** relationship to a simple binary relationship between an **order** and an **address**, rather than additionally capturing the delivery staff in a ternary relationship (cf. Figure 2.10). Each order is required to have exactly one delivery address. Each address in the database, however, may have zero, one, or many orders that need to be shipped there.

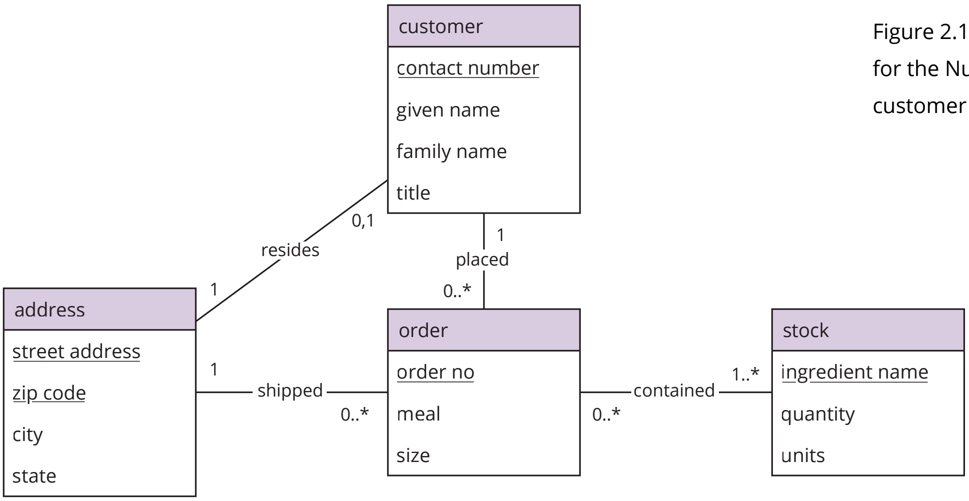


Figure 2.11: E-R diagram for the Nutty Nuggets customer orders database

Finally, you add a new one-to-one **resides** relationship to capture the home address of each customer. Each customer must nominate a single home address (mandatory participation); and if an address is a home address, you decide that there can be only one nominated customer at that address (i.e., optional participation). All this information can be modeled and represented by the E-R diagram shown in Figure 2.11.

Although originally developed in the 1970s, the E-R model remains an important and widely used modeling tool for database design. Measured against the three criteria of a modeling approach given at the beginning of Section 2.2.1, it scores highly. First, the method is based upon the intuitive notions of entity, attribute, and relationship, so it is easily grasped by non-specialists. E-R modeling therefore provides an excellent means of communication between systems analysts and users during the specification of the system. On the third criterion, we shall see later in the chapter how readily a conceptual data model summarized in our E-R diagram may be translated into a practical database implementation.

It is the second criterion, regarding the existence of sufficiently powerful modeling constructs, where the shortcomings of the basic E-R model are sometimes said to lie. This is particularly the case when the application domain to be modeled does not fit into the standard pattern and requires complex data types and relationships. Since it is precisely such systems that are the concern of this book, we will encounter later on more sophisticated modeling approaches that set out to remedy this deficiency. The price to be paid for any increase in modeling power that such extensions can provide is the slight loss of the natural feel and simple diagrams that the basic E-R model possesses.

2.3 Relational databases

The most widely used model for the overall structure of a database system is the *relational* model. The most important alternative model is the *graph* database model, covered in more detail in Section 2.4. The *object-oriented* model is less widely used in databases, but it involves some important concepts with direct relevance to modeling in GIS, returned to in Section 4.1.2. The focus of this section, however, is the relational model, introduced in a classic 1970 paper by Ted Codd.

2.3.1 The relational model

In the Nutty Nuggets example, we have seen that a database holds not only primary data about entities and their attributes, but also connections between entities as relationships. These connections are at the heart of the relational model. No single piece of data provides information on its own: it is the relationships between data items that provide much of the context for the data.

The structure of a relational database is very simple: this is what makes it so powerful. A relational database is a collection of tabular *relations*, often just called *tables*. It is unfortunate that the terms "relationship" (connection between entities in an E-R model) and "relation" (table in a relational database) are so similar. As we shall see, the concepts behind them are closely related, but it is still important not to confuse them.

Table 2.2 shows part of a relation called CUSTOMER (from the NUGGETS database given in full in Appendix A) containing data about some customers, their title, given and family names, and their contact phone number. A relation has associated with it a set of *attributes*: in Table 2.2 the attribute names are TITLE, GNAME, FNAME, and TEL, labeling the columns of CUSTOMER. The data in a relation is structured as a set of rows. A row, or *tuple*, consists of a list of values, one for each attribute. Each cell contains a single attribute occurrence, or *value*. The tuples of the relation are not assumed to have any particular order.

attribute

tuple

We make a distinction between *relation scheme*, which does not include the data but gives the structure of the relation, and *relation*, which includes the data. Data items in a relation are taken from *domains* (akin to data types

TITLE	GNAME	FNAME	TEL
Mr	Roberto	Da Costa	213-555-0506
Ms	Lorna	Dane	610-555-0195
Dr	Jane	Foster	939-555-0177
Mr	Bobby	Drake	757-555-0112
Mx	Loki	Laufeyson	785-555-0189
Ms	Joanna	Cargill	202-555-0125
…	…	…	…

Table 2.2: Part of the relation CUSTOMER

in programming). Each attribute of the relation scheme is associated with a particular domain. In basic database systems, the possible domains are often quite limited, comprising character strings, integers, floats, dates, etc. In our example, the attribute TEL might be associated with character strings of length 12. We may now give some definitions.

relation scheme

- A *relation scheme* is a set of attribute names and a mapping from each attribute name to a domain.

relation

- A *relation* is a finite set of tuples associated with a relation scheme in a relational database such that:
  - each tuple is a labeled list containing as many data items as there are attribute names in the relation scheme; and
  - each data item is drawn from the domain with which its attribute type is associated.

database scheme
relational database

- A *database scheme* is a set of relation schemes, and a *relational database* is a set of relations.

RDBMS

- The database software that manages a relational database model is termed a *relational database management system* (RDBMS).

Relations have the following properties:

- The ordering of tuples in the relation is not significant.
- Tuples in a relation are all distinct from one another.
- Columns are ordered so that data items correspond to the attribute in the relation scheme with which they are labeled.

1NF

relation degree

cardinality

Most relational systems also require that the data items are themselves *atomic*; i.e., they cannot be decomposed as lists of further data items. Thus a single cell cannot contain a set, list, or array. Such a relation, which contains only atomic attributes, is said to be in *first normal form* (1NF). In the example of the CUSTOMER relation, 1NF means that customers are not allowed to have multiple phone numbers or alternative names. The *degree* of the table is the number of its columns. The *cardinality* of the table is the number of its tuples. As tuples come, go, and are modified, the relation will change, but the relation scheme is relatively stable. The relation scheme is usually declared when the database is set up and then left unchanged through the lifetime of the system, although there are operations that will allow the addition, subtraction, and modification of attributes.

The theory of a relational database so far described has concerned the structuring of the data into relations. The other aspects of the relational model are the operations that may be performed on the relations (database manipulation) and the integrity constraints that the relations must satisfy. The manipulative aspects will be considered next, after we have described our working example.

2.3.2 Relational database design

We saw in Section 2.2 how E-R modeling can help in building a conceptual model of a database. An important criterion for the choice of E-R modeling as our conceptual modeling tool was its ability to translate from an E-R model into a practical database implementation. This section shows how an E-R model can aid in relational database design, and it considers some of the principles upon which a good design is based.

Database design, in the case of a relational database, concerns the construction of a relational database scheme. The central question is, "What characterizes a good set of relations for the target application?" Two advantageous features are:

redundancy

- lack of *redundant* data in relations (redundant data wastes space in the database and causes integrity problems); and
- fast access to data from relations.

relational join

These two features essentially trade off space (and integrity) against time. As we shall see later in this chapter, one of the most expensive of relational database operations involves combining information from multiple different relations, termed *relational join*. Fewer joins mean faster data access, which in turn implies fewer relations. Consequently, it is not usually efficient to have many small relations that will need to be joined in order to respond to common queries. On the other hand, while using fewer relations leads to fewer joins, it leads to other problems, shown by the following example from the NUGGETS database.

Suppose that we decide to have all the information about all the customers and their orders in a single relation in the database. The relation scheme could look something like:

CUSTOMER_ORDERS (TITLE, GNAME, FNAME, TEL, ORDER_NO, MEAL, SIZE)

This relation scheme of large degree might appear suitable if there were likely to be many retrievals requiring customer and order details together. The problem is that this scheme can result in redundant duplication of data, where a customer makes multiple orders. Table 2.3 shows an example of the problem. The relation includes redundant data. Each new order duplicates all of the customer data, including name and contact phone number. Such redundancy not only wastes space, but can also cause integrity problems (for example, if Lorna Dane's phone number changes, but only one of the three cells in the table in which it appears is updated).

TITLE	GNAME	FNAME	TEL	ORDER_NO	MEAL	SIZE
Mr	Roberto	Da Costa	213-555-0506	M315-22-06	Lo carb	4
Ms	Lorna	Dane	610-555-0195	M066-22-06	Regular	2
Ms	Lorna	Dane	610-555-0195	M066-22-07	Regular	1
Mr	Roberto	Da Costa	213-555-0506	M315-22-07	Lo carb	4
Dr	Jane	Blake	939-555-0177	M113-22-09	Vegan	1
Ms	Lorna	Dane	610-555-0195	M066-22-08	Regular	2
...	...	...	...	...	...	...

Table 2.3: Some rows and columns of the CUSTOMER_ORDER relation

The specific problems in the CUSTOMER_ORDERS relation above can be solved by splitting the scheme so that the customer data is held in one relation, and the order data is held in another. A first pass at this relation scheme is below:

CUSTOMER (TITLE, GNAME, FNAME, TEL)
ORDERS (ORDER_NO, MEAL, SIZE)

However, in designing this relation scheme we have lost the connection between the two tables, captured by the **placed** relation in our E-R model in Figure 2.11. Recovering this relationship requires that we first clarify the identifiers for each relation. In relational databases, a *candidate key* is an attribute or minimal set of attributes that will serve to uniquely identify each tuple of the relation. There may be several such candidate keys for a relation. The identifier in an E-R model will usually be a potential candidate key. One candidate key is chosen as the *primary key*.

candidate key

primary key

Following the E-R model in Figure 2.11, we can use TEL and ORDER_NO attributes as primary keys of the CUSTOMER and ORDERS relations, respectively. Armed with our primary keys, we could choose to add a new relation to our relation scheme called PLACED containing just two attributes TEL and ORDER_NO. This relation would then store the data about the relationship between customers and their orders. However, it is instead simpler to add the customer primary key TEL to the ORDERS relation. This construction is called *posting the foreign key*. In technical language, we have posted the identifier of CUSTOMER as a foreign key into ORDERS. Note that the converse—posting ORDER_NO as a foreign key to the CUSTOMER relation—will not work, as it will lead to the same duplication as in Table 2.3.

posting the foreign key

Posting a foreign key, rather than adding another relation, is usually an option if the relationship between entities is many-to-one. The resulting relation scheme is shown below. Just as identifiers for each entity type are underlined in our E-R model in Figure 2.11, so there is a convention that the set of attributes constituting the primary key of the relation is underlined in a relation scheme.

CUSTOMER (TITLE, GNAME, FNAME, TEL)
ORDERS (ORDER_NO, CUS_TEL, MEAL, SIZE)

The relations with the tuples from Table 2.3 appropriately distributed are shown in Table 2.2 above (CUSTOMER relation) and Table 2.4 below (ORDERS). There is now no redundant repetition of data.

Table 2.4: The ORDERS relation, together with the CUSTOMER relation in Table 2.2, capturing all the information in CUSTOMER_ORDER relation Table 2.3 without redundancy

ORDER NO	CUS_TEL	MEAL	SIZE
M315-22-06	213-555-0506	Lo carb	4
M066-22-06	610-555-0195	Regular	2
M066-22-07	610-555-0195	Regular	1
M315-22-07	213-555-0506	Lo carb	4
M113-22-09	939-555-0177	Vegan	1
M066-22-08	610-555-0195	Regular	2
...	...	...	...

Apart from avoiding redundant duplication of information, decomposition of relation schemes has the advantage that smaller relations are conceptually more manageable and allow separate components of information to be stored in separate relations. Of course, relations cannot be split arbitrarily. Relations form connections between data in the database and inappropriate decomposition can destroy these connections. One important guideline for appropriate decomposition has already been introduced in Section 2.3.1, with relations in 1NF having atomic attributes. In fact, there exists a hierarchy of normal forms for relational databases; higher normal forms require higher levels of decomposition of the constituent database relations (see Box 2.4 on page 58). The process of appropriately decomposing relations into normal form is termed *normalization*. Normal forms are useful guidelines for database design. However, in any logical data model, the level of normal form (i.e., degree of relation decomposition) must be balanced against the decrease in performance resulting from the need to reconstruct relationships by join operations.

normalization

In this way, an E-R model may be transformed into a set of relation schemes. As a first pass, the general principle is that to begin:

- each entity in the E-R model maps to a relation in the relation scheme;
- each attribute in the E-R model maps to an attribute in the corresponding relation;
- each identifier maps to a primary key;
- each relationship maps to a relation combining the identifiers/primary keys of the two entities/relations it connects.

In some cases it is also necessary to modify this first pass in order to maintain the information structure, while balancing the redundancy that accompanies too few relations and the inefficiency that arises from too many. The discussion above of the relations CUSTOMER and ORDERS provides one example of modifications needed to strike that balance. Posting the foreign key from CUSTOMER to ORDERS ensures the database scheme avoids the redundancy inherent in a single relation, Table 2.3, without the need for a third PLACED relation.

The database scheme below gives the final relational database scheme for our NUGGETS database.

CUSTOMER (TEL, HID, TITLE, GNAME, FNAME)
ORDERS (ORDER NO, CUS_TEL, AID, MEAL, SIZE)
STOCK (INGREDIENT, QUANT, UNITS)
CONTENT (OID, SID, WEIGHTG)
ADDRESS (ADDRESS ID, STREET_AD, ZIP, CITY, STATE)

With reference again to Figure 2.11, the following summarizes the decisions made in arriving at this database scheme:

- The CUSTOMER relation required the further addition of the foreign key HID as an attribute to identify the home address of each customer, necessitated by the **resides** relation.
- The ORDERS relation likewise includes the foreign key AID to capture the shipping address of each order (relation **shipped**).
- The STOCK relation with its attributes is mapped directly from the **stock** entity.
- The CONTENT relation captures the data associated with the **contained** relationship. Posting the foreign key (from ORDERS to STOCK or vice versa) is not an option for this many-to-many relationship without resulting in the kinds of redundancy already seen in Table 2.3. Hence, the primary key of the CONTENT relation becomes the combination of the foreign keys of ORDERS and STOCK: OID and SID, respectively. A primary key that is made up of two or more foreign keys is called a *compound key*.

compound key

- It was further decided that it would be useful to know the quantity of each ingredient in the order (and not simply in the stockroom, already stored in the STOCK relation). Although this feature was not captured in the design of the E-R model in Figure 2.11, it is not unusual to need to tweak and refine designs during the implementation process. Consequently the weight in grams of each ingredient in each order is captured through the attribute WEIGHTG.

Finally, the ADDRESS relation has acquired a new primary key ADDRESS ID. An alternative candidate key is suggested by the identifiers of the **address** entity in Figure 2.11: the combination of attributes STREET_AD and ZIP. A candidate key that is made up of two or more attributes is called a *composite key*. (A compound key, above, is therefore a special case of a composite key.) Three reasons led to this decision. First, posting a composite foreign key from ADDRESS to CUSTOMER and ORDERS is possible, but it is arguably less clear for database users than posting a singleton foreign key. Second, the STREET_AD attribute is expected to be a relatively long text string in many cases (e.g., '1407 Graymalkin Lane'). Using such attributes as keys, and especially foreign keys, increases the chances of errors, for example, where a mistype in one table breaks the connection between the tuples. Hence, creating a new, simplified address code in ADDRESS ID as the primary key in ADDRESS was

composite key

Box 2.4: Second normal form (2NF)

As introduced above, normalization is the process of decomposing relations so that they conform to certain standard principles used to reduce redundancy and increase integrity of stored data, called *normal forms*. As we have seen, *first normal form* (1NF) requires all relations in a relational database to have atomic attributes. Each higher normal form builds on the lower normal form, such that the first condition of *second normal form* (2NF) is that a relation is in 1NF. 2NF further requires that all non-key attributes are "fully functionally dependent" on the primary key alone. "Full functional dependency" means that an attribute is dependent on the whole primary key, and not on a subset of that key. For example, the original ADDRESS table with the relation scheme ADDRESS (STREET AD, ZIP, CITY, STATE) is arguably not in 2NF. A violation of 2NF arises if the CITY or STATE attributes are functionally dependent on the ZIP attribute (i.e., if I know the zip code, then I can deduce the city or state). ZIP is only a part of the primary key for ADDRESS, so in that case CITY and STATE are not fully functionally dependent on the primary key, only part of the primary key, and so the relation is not in 2NF. Regaining 2NF would require that the relation be split into two, with each part of the compound key becoming the primary key of a new, decomposed relation. So why "arguably"? While this partial dependency holds in most cases, some US zip codes do straddle multiple cities and even states (such as 42223 that covers addresses on both sides of the border between Kentucky and Tennessee). Consequently, it could be argued that in order to allow for this eventuality, the relation is in 2NF.

considered to be worth the small additional redundancy it entailed (and see also the inset on second normal form, Box 2.4).

The full NUGGETS relational database can be found in Appendix A.

2.3.3 Operations on relations

relational operator

A relation is nothing more than a structured table of data items. Each column of a relation is named by an attribute, and it has all its data items taken from the same domain. The basic operations supported by a relational database are therefore simple. There are five fundamental *relational operators*: *union*, *difference*, *product*, *project*, and *restrict*.

The first three of these are traditional set-based operators, introduced in the next chapter. The project and restrict operators are described below. Three further relational operators *intersection*, *divide*, and *join*, termed *derived* relational operators, can be expressed using different combinations of the fundamental five operators.[1] Of these, intersection is another set-based operator introduced in the next chapter, join is described below, and divide is a less commonly used operator not discussed further here. The structure of these operations and the way that they can be combined is called *relational algebra*.

relational algebra

The relational model is *closed* with respect to all the above relational operations, because they each take one or more relations as input and return a relation as a result. The set operations union, intersection, product, and difference work on the relations as sets of tuples. Thus, if we have two relations, one holding all Californian and one all Texan addresses, then their union will hold the addresses in both states and their intersection will be empty. For all the set operations except product, the relations must be *compatible*, in that they must have the same attributes; otherwise, the new relation will not be well formed.

[1] Can you deduce how to define the three derived relational operators in terms of the five fundamental operators? Which two fundamental operators when combined can result in the join operator, described below, for example?

project operator

Project operator The *project* operation is unary, applying to a single relation. It returns a new relation that has a subset of attributes of the original. The relation is then modified so that any duplicate tuples formed are coalesced. The project operator π has the following syntax:

$$\pi_{<attribute\ list>}(relation)$$

For example, $\pi_{\text{CITY,STATE}}(\text{ADDRESS})$ returns the relation shown in Table 2.5a, and $\pi_{\text{STATE}}(\text{ADDRESS})$ returns the relation shown in Table 2.5b. Note that in the second case the four identical tuples containing the value “NY” have been coalesced into a single tuple.

Table 2.5: Results of relational projections and restrictions

(a)

CITY	STATE
New York	NY
North Salem	NY
Philadelphia	PA
Washington	DC
Fort Washington	NY
Broxton	OK
Brooklyn	NY
San Francisco	CA

(b)

STATE
NY
PA
DC
OK
CA

(c)

ORDER_ID	CUS_TEL	AID	MEAL	SIZE
M113-22-09	939-555-0177	74012-RA01	Vegan	1
M315-22-06	213-555-0506	94110-MS01	Lo carb	4
M315-22-07	213-555-0506	94110-MS01	Lo carb	4

(d)

ORDER_ID
M113-22-09
M315-22-06
M315-22-07

restrict operator

Restrict operator The *restrict* operation is also unary. The restrict operator works on the tuples of the table rather than the columns, and it returns a new relation that has a subset of tuples of the original. A condition specifies those tuples required. The restrict operator is often referred to as the *select* operator, and consequently it is denoted with the Greek symbol σ (sigma). The syntax used here is:

$$\sigma_{<condition>}(relation)$$

For example, the list of meal orders other than ‘Regular’ can be retrieved from the database using the expression $\sigma_{\text{NOT(MEAL='Regular')}}(\text{ORDERS})$. This will return the relation shown in Table 2.5c. Operations can be combined, for example:

$$\pi_{\text{ORDER\_ID}}(\sigma_{\text{NOT(MEAL='Regular')}}(\text{ORDERS}))$$

returns the order numbers of non-‘Regular’ meal orders, as shown in Table 2.5d.

join operator

natural join

Join operator With the *join* operation, the relational database begins to merit the term "relational." Join is a binary operator that takes two relations as input and returns a single relation. The join operation allows connections to be made between relations. There are several different kinds of relational join but we describe only the *natural join* of two relations, defined as the relation formed from all combinations of their tuples that agree on a specified common attribute or attributes. The join operator $\bowtie$ has the following syntax:

$$\bowtie_{attribute_1 = attribute_2}(relation_1, relation_2)$$

to indicate that $relation_1$ and $relation_2$ are joined on attribute combinations $attribute_1$ of $relation_1$ and $attribute_2$ of $relation_2$. For example, to relate details of orders to the customers who placed each order on which they are showing, relations CUSTOMER and ORDERS are joined on the film title attribute in each relation. The expression is:

$$\bowtie_{\text{TEL=CUS\_TEL}}(\text{CUSTOMER, ORDERS})$$

The resulting relation shown in Table 2.6 combines tuples of CUSTOMER with tuples of ORDERS, provided that the tuples have the same customer phone number. Notice that the join has not repeated the duplicate attribute.

TEL	GNAME	FNAME	HID	TITLE	ORDER_ID	AID	MEAL	SIZE
213-555-0506	Roberto	Da Costa	94110-MS01	Mr	M315-22-06	94110-MS01	Lo carb	4
213-555-0506	Roberto	Da Costa	94110-MS01	Mr	M315-22-07	94110-MS01	Lo carb	4
610-555-0195	Lorna	Dane	10560-GL01	Ms	M066-22-06	19104-CS01	Regular	2
610-555-0195	Lorna	Dane	10560-GL01	Ms	M066-22-07	19104-CS01	Regular	1
610-555-0195	Lorna	Dane	10560-GL01	Ms	M066-22-08	19104-CS01	Regular	2
939-555-0177	Jane	Foster	74012-RA01	Dr	M113-22-09	74012-RA01	Vegan	1

Table 2.6: Result of relational join

If we only require the customer contact numbers, given names, and order numbers of non-'Regular' meal orders, we may again restrict and project the relation in Table 2.6 to only those desired attributes and rows. The combined relational algebra statement is below, with the resulting table shown in Table 2.7:

$$\pi_{\text{TEL,ORDER\_ID,GNAME}}(\sigma_{\text{NOT(MEAL='Regular')}}(\bowtie_{\text{TEL=CUS\_TEL}}(\text{CUSTOMER, ORDERS})))$$

Table 2.7: Result of combined relational join, restriction, and projection

TEL	GNAME	ORDER_ID
213-555-0506	Roberto	M315-22-06
213-555-0506	Roberto	M315-22-07
939-555-0177	Jane	M113-22-09

The last example may be used to demonstrate an important property of relation operations: the order in which operations are performed will affect performance. The join operation is the most time-consuming of all relational operations, because it needs to compare every tuple of one relation with every tuple of another. To extract data for Table 2.7 we performed operations join, project, and restrict. In fact, it would have been more efficient to have first done a restrict operation on the ORDERS table, then joined the resulting smaller table to CUSTOMERS, and then projected. The result would be the same, but the retrieval would perform better, because the join involves smaller tables.

In general, reordering the elements of a relational algebra expression may not lead to an equivalent expression. For example, the relational algebra expression $\pi_{\text{ORDER\_ID}}(\sigma_{\text{NOT(MEAL='Regular')}}(\text{ORDERS}))$ given above is not equivalent to $\sigma_{\text{NOT(MEAL='Regular')}}(\pi_{\text{ORDER\_ID}}(\text{ORDERS}))$, because MEAL is not a valid attribute of the relation $\pi_{\text{ORDER\_ID}}(\text{ORDERS})$. The topic of *query optimization* is a critical study for high-performance databases, concerned with processing queries as efficiently as possible. An important component of query optimization involves performing transformations (such as reordering) upon queries, to produce equivalent queries that can be processed more efficiently.

query optimization

2.3.4 *Structured query language*

The *structured query language* (SQL) provides users of relational databases with the facility to define the database scheme (data definition), and then insert, modify, and retrieve data from the database (data manipulation). The language may either be used on its own, as a means of direct interaction with the database, or embedded in a general-purpose programming language. The aim of this section is to provide an introduction to SQL, highlighting its close relationship with the relational algebra, without making any attempt to be a complete SQL reference.

SQL

Data definition using SQL The *data definition language* (DDL) component of SQL allows the creation, alteration, and deletion of relation schemes. Normally, a relation scheme is altered only rarely once the database is operational. A relation scheme provides a set of attributes, each with an associated data domain. SQL allows the definition of a domain by means of the expression below (square brackets indicate an optional part of the expression).

data definition language

```
CREATE DOMAIN name datatype
 [DEFAULT definition]
 [CHECK constraint]
```

The user specifies the name of the domain and associates that name with a predefined data type, such as a character string (VARCHAR), or an integer, float, date, time. The DEFAULT definition allows the user to specify a default value for a tuple; a common default value is NULL. The domain CHECK defines integrity constraints by restricting the domain to a set of specified values. An example of the definition of a domain for the attribute MEAL is as follows:

```
CREATE DOMAIN MEALTYPE VARCHAR(10)
   CHECK (VALUE IN ('Regular', 'Vegan', 'Lo carb', 'Lo fat', 'No gluten'));
```

Box 2.5: SQL and CRUD

SELECT statements covered in the main text are at the heart of SQL, and they provide access to the relational database retrieval operations. In addition, basic data creation, update, and deletion operations (found in CRUD) are provided by INSERT, UPDATE, and DELETE statements. For example:

```
INSERT INTO STOCK VALUES ('Flour', 37, 'kg');
```

will record a new entry in the STOCK table. In addition, the SELECT INTO statement makes a new table that is a copy of part or all of an existing table, such as:

```
SELECT * INTO STOCKBACKUP FROM STOCK;
```

The INSERT INTO can similarly copy data across from one table into another, suitably structured existing table. UPDATE and DELETE statements have a WHERE clause to specify which records to alter. For example:

```
DELETE FROM STOCK WHERE INGREDIENT='Oil';
```

deletes the oil from the STOCK table, and:

```
UPDATE STOCK SET QUANT=8 WHERE INGREDIENT='Corn';
```

updates the stored corn stock quantity.

A relation scheme is created as a set of attributes, each associated with a domain, with additional properties relating to keys and integrity constraints. For example, the relation scheme ORDERS can be created by the command:

```
CREATE TABLE ORDERS
  ORDER_NO CHAR(10),
  CUS_TEL CHAR(12),
  AID CHAR(10),
  MEAL MEALTYPE,
  SIZE INT(1),
  PRIMARY KEY (ORDER_NO),
  FOREIGN KEY (CUS_TEL) REFERENCES CUSTOMER(TEL),
  FOREIGN KEY (AID) REFERENCES ADDRESS(ADDRESS_ID),
  CHECK (SIZE < 7 AND SIZE > 0);
```

This statement begins by naming the relation scheme (called a table in SQL) as ORDERS. The attributes are then defined by giving their name and associated domain. It is optionally possible to identify to the database any keys, including the foreign keys posted from CUSTOMER and ADDRESS. Doing so enables the database to maintain referential integrity, for example, ensuring that if an address is deleted (or updated) from the ADDRESS relation, then any reference to it is also deleted (or updated) in ORDERS. Finally, one further integrity check is added to limit the meal size of an order to between 1 and 6 people, inclusive. Any attempt to insert a row with an order for 8 people, say, will be disallowed.

Data manipulation using SQL Having defined the relation schemes, the next step is to insert data into the relations. These SQL commands are quite straightforward, allowing insertion of single or multiple tuples, update of tuples in tables, and deletion of tuples (see Box 2.5). Data retrieval forms the most complex aspect of SQL: a large book could be written on this topic alone. Our treatment is highly selective, giving the reader a feel for SQL in this respect. The general form of the retrieval command is:

```
SELECT item−list
  FROM reference−list
  [WHERE condition]
  [GROUP BY attribute−list]
  [HAVING condition]
```

A simple example of data retrieval, already considered in the relational algebra section, is to find the order number of all orders for meals other than 'Regular'. The corresponding SQL expression is:

```
SELECT ORDER_NO
  FROM ORDERS
  WHERE NOT(MEAL='Regular');
```

The SELECT clause serves to project (how confusing!) on the required ORDER_NO attribute. The FROM clause tells us from which table the data is coming, in this case ORDERS. The WHERE clause provides the restrict condition.

Relational joins are effected by allowing more than one relation (or even the same relation called twice with different names) in the FROM clause. For example, to find details of orders and where they are shipping to, we could give the following SQL command:

```
SELECT ORDER_NO, MEAL, SIZE, STREET_AD, ZIP, CITY, STATE
  FROM ORDERS, ADDRESS
  WHERE ORDERS.AID = ADDRESS.ADDRESS_ID;
```

In this case, the WHERE clause provides the *join condition* by specifying that tuples from the two tables are to be combined only when the values of the attributes AID in ORDERS (indicated by ORDERS.AID) and ADDRESS_ID in ADDRESS (ADDRESS.ADDRESS_ID) are equal. A more complex case, using all the clauses of the SELECT expression, is the following expression, which retrieves the total weight of ingredients in all orders with more than three items.

```
SELECT ORDER_NO, SUM(WEIGHTG)/1000 AS 'TOTAL (kg)'
  FROM ORDERS, CONTENT
  WHERE ORDERS.ORDER_NO = CONTENT.OID
  GROUP BY ORDER_NO
  HAVING COUNT(*) > 3;
```

With the exception of the term SUM(WEIGHTG)/1000 AS 'TOTAL (kg)', the first three lines of code act to retrieve the orders and associated ingredients, using the join of ORDERS and CONTENT. The GROUP BY clause serves to logically construct a table where the tuples are in groups, one for each order. The HAVING clause comes into play to operate as a filter condition on the groups in the grouped relation. It selects only groups that have a tuple count of at least three. This intermediate table (Table 2.8) is not a legal first normal form relation because values in most cells are not atomic (i.e., there are mini-tables within most cells). But we are concerned with the total weight for each order and will eventually project out unused attributes.

ORDER_ID	CUS_TEL	AID	MEAL	SIZE	SID	WEIGHTG
M066-22-06	610-555-0195	19104-CS01	Regular	2	Chickpeas	400
	610-555-0195	19104-CS01	Regular	2	Chili	10
	610-555-0195	19104-CS01	Regular	2	Rice	400
	610-555-0195	19104-CS01	Regular	2	Tomato	400
M066-22-08	610-555-0195	19104-CS01	Regular	2	Carrot	300
	610-555-0195	19104-CS01	Regular	2	Corn	200
	610-555-0195	19104-CS01	Regular	2	Potato	300
	610-555-0195	19104-CS01	Regular	2	Vegetable stock	500
M113-22-09	939-555-0177	74012-RA01	Vegan	1	Chickpea	400
	939-555-0177	74012-RA01	Vegan	1	Garlic	20
	939-555-0177	74012-RA01	Vegan	1	Lemon	200
	939-555-0177	74012-RA01	Vegan	1	Mushroom	600
M315-22-06	213-555-0506	94110-MS01	Lo carb	4	Chickpea	800
	213-555-0506	94110-MS01	Lo carb	4	Chili	20
	213-555-0506	94110-MS01	Lo carb	4	Corn	600
	213-555-0506	94110-MS01	Lo carb	4	Tomato	600

Table 2.8: Evaluation of an SQL query to the NUGGETS database: intermediate stage

The final stage of the retrieval, after projecting out the unwanted attributes, is shown in Table 2.9. The weights in the WEIGHTG column in Table 2.8 are aggregated by summation. SUM and COUNT are built-in SQL functions, along with a range of other aggregation functions including AVG, MAX, and MIN. Failing to specify any aggregate function for any attributes in the final projection (except the already grouped-by ORDER_ID) will result in an SQL error. In constructing the final result, dividing the SUM(WEIGHTG) attribute by 1000 in the projection converts the attribute values from grams to kilograms. The SQL AS keyword renames the column in the final output to make it more easily understandable.

Table 2.9: Evaluation of an SQL query to the NUGGETS database: final result

ORDER_ID	TOTAL (kg)
M066-22-06	1.210
M066-22-08	1.300
M113-22-09	1.220
M315-22-06	2.020

There is much more to SQL than could be covered in a text such as this, but a wealth of books and online resources are available to assist the interested reader in learning SQL. In addition to giving a flavor of the main capabilities of language, the brief introduction to SQL provided here is also designed to highlight the close relationship between the SELECT statement at the heart of SQL and the relational algebra operations that underpin it. Other database query languages, such as queries to the graph database model, cannot boast such rigorous formal foundations, as we shall discover in the following section.

2.4 Graph databases

The second main model of the overall structure of a database is the *graph* model. Where the relational model is founded on the table as its fundamental data structure, the graph model uses *abstract graphs*. Abstract graphs—referred to simply as "graphs" where there is no possibility of confusion—offer much greater flexibility in how data is structured than the more rigid tables we have seen in a relational databases. This additional flexibility comes at the cost of additional technical challenges in querying stored data efficiently. As a result, although graph DBMS have existed since the 1960s, it is only since around the mid-2000s that scalable graph DBMS have become widely available.

2.4.1 Abstract graphs

graph

A *graph* is a highly abstracted model of connectivity between elements. Graphs are important models underpinning many spatial concepts, not only graph databases. So in this section we will take the time to introduce abstract graphs systematically.

node
edge
adjacency
incident

A graph is made up of a set of distinct *nodes* together with a set of *edges* that connect pairs of nodes. It is usual to summarize a graph by means of a diagram. The graph in Figure 2.12 consists of the five nodes *a*, *b*, *c*, *d*, and *e* and five edges. A node is said to be *adjacent* to another node if both nodes are connected by an edge. In Figure 2.12, nodes *a* and *b* are adjacent, but nodes *a* and *c* are not. An edge is said to *join* or to be *incident on* the nodes it connects. Likewise, a node is said to be *incident* with an edge that connects to it.

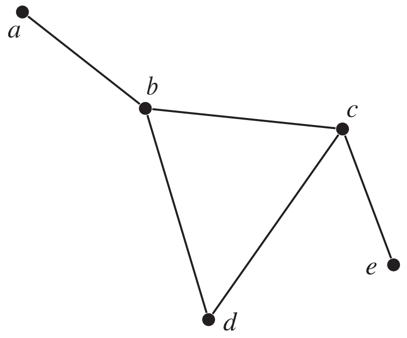


Figure 2.12: A connected, undirected graph with order 5

order
degree
path
connected graph

The *order* of a graph is the number of nodes it contains. The *degree* of a node is the number of edges with which it is incident. For example, the degree of nodes *b* and *c* in Figure 2.12 is three; the degree of node *d* is two; and the degree of nodes *a* and *e* is one. A *path* through a graph is a sequence of nodes in the graph where each consecutive pair of nodes is adjacent, i.e., connected by an edge in the graph. Examples of paths between nodes *a* and *e* in Figure 2.12 are *abce*, *abdce*. In contrast, the sequence *ace* is not a path in the graph because nodes *a* and *c* are not connected by an edge. A *connected graph* is one such that there exists a path between any two of its nodes. The graph in Figure 2.12 is connected.

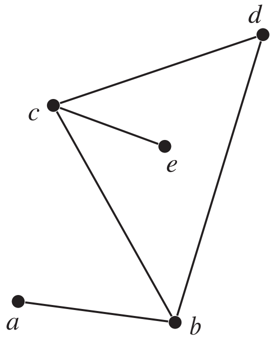
cycle

A path from a node to itself traversing at least one edge is called a *cycle*. There are two cycles in Figure 2.12: $bcdb$ (or equivalently $cdbc$ or $dbcd$) and $bdcb$ (or $cbdc$ or $dcbd$). A graph that has no cycles is called an *acyclic graph*.

acyclic graph

Two graphs may show exactly the same connectivity relationships. Such graphs are said to be *isomorphic*. Thus, graphs Figure 2.12 and Figure 2.13 are isomorphic, since both have precisely the same nodes and edges. In this case, we have made the case clearer by labeling the nodes to show the isomorphism.

Figure 2.13: A graph isomorphic to that in Figure 2.12



undirected graph

The abstract graphs above are *undirected*, meaning that the direction of the edges is not significant. Thus, an undirected edge from a to b, written $\{a, b\}$, is the same as that from b to a (i.e., $\{a, b\}$ is equivalent to $\{b, a\}$). A *directed graph* or *digraph* is a graph in which each edge is assigned a direction. Thus in a directed graph an edge from a to b, written ab, is distinct from an edge from b to a, written ba. In diagrams, directed edges are usually indicated by arrowed lines.

directed graph

Figure 2.14 shows a directed graph of order 5. Because the direction of edges is significant, edge ba is an edge of the graph in Figure 2.14, but ab is not. Hence, in Figure 2.14 node b is adjacent to node a, but a is not adjacent to b (since there exists no directed edge *from* a *to* b). The asymmetry introduced by directed graphs means we have to refine our undirected definition of connected graphs above. A directed graph that satisfies the definition of a connected graph above—that there exists a path between every pair of nodes—is termed *strongly* connected. The graph in Figure 2.14 is not strongly connected. Replacing all the directed edges with undirected edges, as in Figure 2.12, does lead to a connected graph though. As a result, the graph in Figure 2.14 is said to be *weakly* connected. Closely related, a directed graph where there exists a path either from x to y *or* from y to x for any pair of nodes x and y is termed *semiconnected*.

strongly connected

weakly connected

semiconnected

The graph in Figure 2.14 is both weakly connected and semiconnected. Indeed, every semiconnected graph is also weakly connected; not every weakly connected graph is semiconnected, however.[2]

Four further types of graph are especially useful in representing data:

labeled graph

- A *labeled graph* is a graph in which each edge and/or node is assigned

[2] Can you construct your own example of a weakly connected, but not semiconnected graph? Hint: it is possible to achieve this by reversing the direction of just one of the directed edges in Figure 2.14.

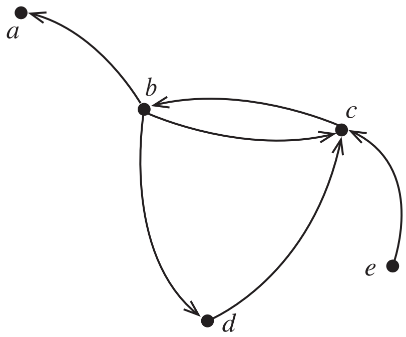


Figure 2.14: A directed, semiconnected graph of order 5, not isomorphic to that in Figure 2.12

a label (maybe a number or string). Labels are usually indicated on a diagram near to the corresponding edges or nodes.

- A *weighted graph* is a labeled graph in which each label is a (usually non-negative) number, or weight. Weighted graphs are of particular importance in GIS, as we shall see in the next chapter.
- A *multigraph* is a graph that supports multiple distinct edges between a pair of nodes, whereas ordinarily there can be at most one edge between any pair or nodes.
- A *hypergraph* is a graph where a single edge may connect more than two nodes, whereas ordinarily an edge connects exactly two nodes. Hypergraphs are mentioned here purely for completeness, but are not encountered again in this book.

weighted graph

multigraph

hypergraph

We will return to the topic of graphs in more detail in later chapters, as they are also important tools for representing spatial networks and spatial relationships. For example, one can imagine the usefulness of a graph in modeling the road network in a city center. In that case, edges can represent roads, with nodes representing road intersections. A directed graph could be used to additionally capture information about one-way streets. A labeled graph can be useful for storing road names, road lengths, or travel times.

However, for now we have enough knowledge of abstract graphs to introduce graph databases.

2.4.2 The graph database model

The most common graph databases are founded on three basic structures:

- Nodes are used to represent entities in a domain. Nodes are approximately equivalent to a record or a tuple in a relational database.
- Edges are used to represent relationships between entities. Edges are central to the difference between graph and relational databases. The relational model contains no comparable structure.
- *Properties* are data associated with nodes or edges. Properties are akin to attribute values in relational databases.

graph property

The most common graph databases structure their data as a directed, labeled multigraph of nodes, edges, and associated properties, known as the property graph model

property graph model. Another closely related graph database model which adopts a slightly different structure is called the *triple store*. Triple stores are particularly important in the context of linked open data and the Semantic Web, which we return to later in this book in Chapter 9.

The property graph model is an incredibly flexible way to structure and store an enormous range of different data. Indeed, this description of the property graph model may sound already familiar to you: the simpler E-R diagrams encountered earlier in this chapter can themselves be thought of as property graphs.[3]

Figure 2.15 shows the Nutty Nuggets customer data within a graph database, displaying the same data encountered earlier in Table 2.2. Each tuple in the relational database table corresponds to a node in the graph database. Each data item in the relational database table can be found as a property of the corresponding node.

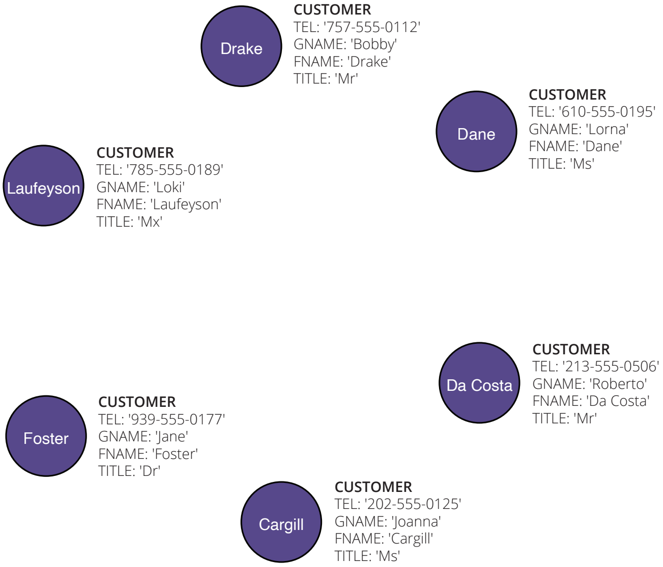


Figure 2.15: A view of the Nutty Nuggets graph database showing the **customer** data from Table 2.2

The graph in Figure 2.15 contains only nodes and properties, no edges. Adding in more data from the database into the view, we can see our first edges in Figure 2.16. As highlighted above, the relational model has no direct correspondent to edges; edges are a distinguishing feature of graph databases. However, an important consequence of this difference is that the Nutty Nuggets graph database needs no foreign keys. The customers in Figure 2.16 need no HID property, unlike the CUSTOMER relation scheme developed in Section 2.3.2. Instead, a RESIDES edge connects each customer with a home address. Additionally that edge is directed, indicating that it is the customer that resides at the address, not the address that resides in the customer. This support for relationship directionality is absent from the relational data model.

[3] Indeed, edges in a graph database are often also called *relationships*, just like the corresponding structure in the E-R model. However, to avoid confusion with E-R model "relationships" (not to mention relational database "relations"!), we will continue to use the term *edge* in this book, highlighting the underlying connection to abstract graphs.

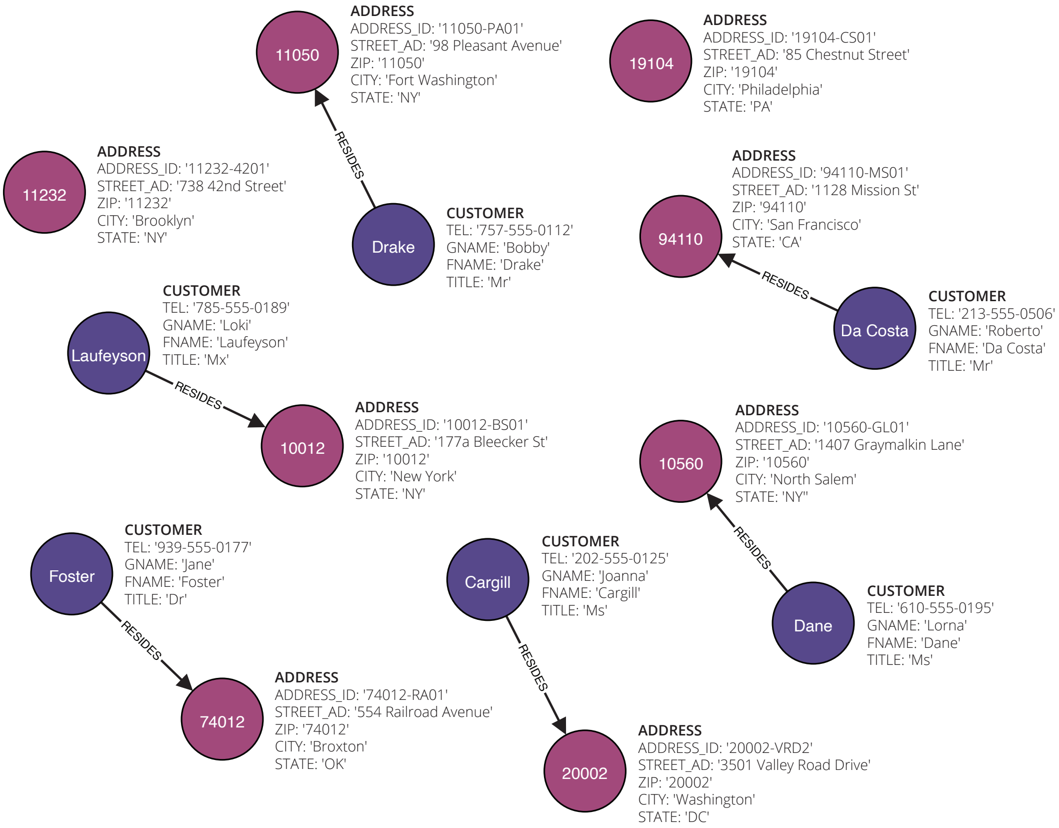


Figure 2.16: A view of the Nutty Nuggets graph database showing both the **customer** and **address** data

The resulting graph database view in Figure 2.16 also contains two isolated address nodes with no incident edges. These nodes relate to delivery addresses in the database that are not customer home addresses.

2.4.3 *Graph database design*

The process of database design begins in exactly the same way for a graph database as we have already seen for a relational database. Indeed, here we see emphasized again the most important property of any conceptual data modeling: it must be independent of any specific implementation. Hence, the E-R conceptual model developed in Section 2.2.1 and summarized in Figure 2.11 concerns the entities and relationships of interest in our application, but it is agnostic on the structure of the data itself.

Consequently, we have already completed the first and most important step in designing our graph database: designing the E-R model in Section 2.2.1. The next step is to again translate that model into a database scheme, but this time a graph database rather than relational database scheme. This step highlights an appealing feature of graph databases that has contributed greatly to their popularity. Whereas relational database design is a careful and methodical process with important implications for subsequent database performance, graph database design typically involves little more than identifying what will be represented as a node, as an edge, and as a property.

Some natural mappings immediately suggest themselves as a first pass, namely:

- each entity in the E-R model maps to a set of nodes in the graph database;
- each attribute in the E-R model maps a property found on each corresponding node;
- each relationship maps to a set of edges connecting corresponding nodes.

Unlike a relational database, a graph database does not usually require primary keys to be identified from the conceptual model. Most graph DBMS will instead automatically assign a unique system identifier to every node and edge, but they may also allow an identifying attribute from the data to be used in place of the system identifier. Without primary keys, a graph database also needs no foreign keys, as we have seen. Instead, we must be careful to capture explicitly any relationships in the E-R model as edges connecting pairs of nodes.

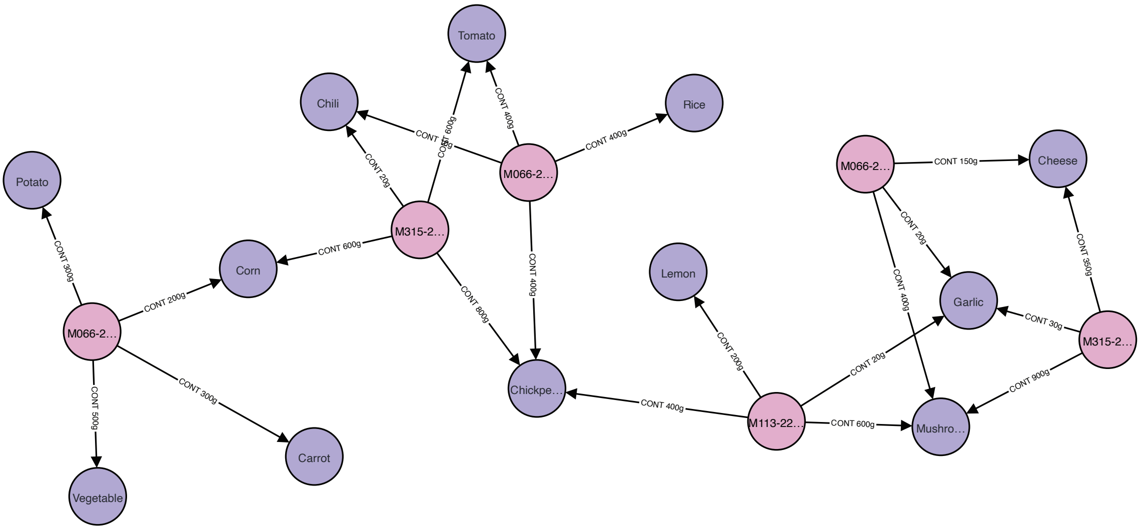


Figure 2.17: A view of the Nutty Nuggets graph database showing the WEIGHTG property of the CONT edge connecting STOCK and ORDERS nodes, node properties omitted

In a graph database, edges can have properties just as nodes can. These properties can be used to capture directly any attributes of relationships in the E-R model. For example, the view of the Nutty Nuggets graph database in Figure 2.17 shows the edges that capture the **contained** relationship from our E-R model in Figure 2.6. Where our relational database scheme required a CONTENT relation with a WEIGHTG attribute, as well foreign keys posted from STOCK and ORDERS, the relationship is captured in the graph database in Figure 2.17 by the CONT edges labeled with their WEIGHTG property.

The remarkable flexibility of the graph database model does come with some drawbacks, particularly for those database designers used to working within the structures provided the relational database model. At its most flexible, the graph database places no constraints upon the structure of each node and edge. There is no explicit requirement for any two nodes to share any particular properties in common, unlike that requirement enforced on tuples by the tabular structure of relations. The database developer is free to add a HAIR_COLOR: GREEN property only to the Lorna Dane customer node in Figure 2.15 or a CANS: 2 property only to the edge from order node 'M315-22-06' to stock node 'Chickpeas' Figure 2.17 without reference to the E-R model. Such *ad hoc* changes are not possible in the relational model, requiring first an update to the relation scheme and second the inclusion of corresponding hair color or cans data to other tuples in the relation (even if only NULL values).

Indeed, the different "types" of nodes in Figures 2.15–2.17, also shown in the complete database view in Figure 2.18, are colored for the purposes of graphical clarity only. The basic property graph model makes no such distinctions between different node or edge types. An extension to the property graph model, called the *labeled property graph* model, does however require nodes and edges to be grouped into labeled "types," but it still makes no restrictions on whether two nodes with the same label share any properties in common. The ability to have at least the minimal structure provided by node and edge labels is still practically useful. Accordingly, many graph DBMS today do adopt the labeled property graph model.

labeled property graph

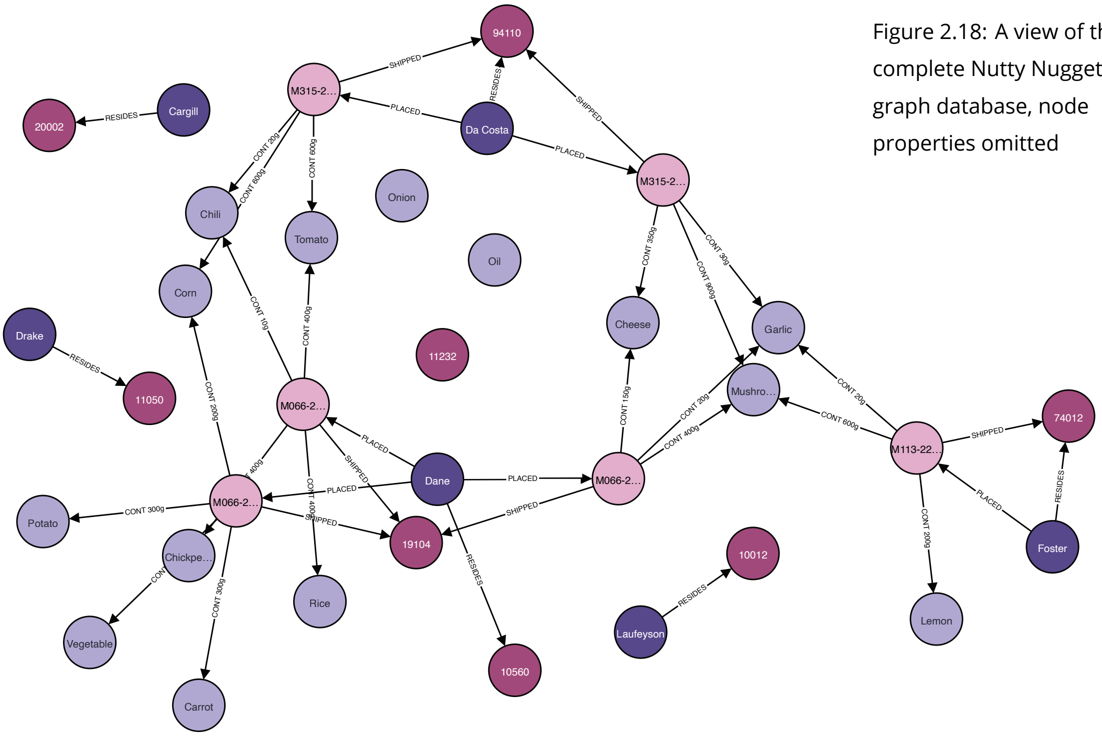


Figure 2.18: A view of the complete Nutty Nuggets graph database, node properties omitted

This extreme flexibility to not only add data but evolve structure has led to graph databases sometimes being referred to as "NoSQL" databases. The

NoSQL

term is somewhat misleading, suggesting that graph databases do need query languages or never accept SQL, neither of which is the case. Instead, a more accurate encoding of the idea would be a "not only SQL." As we shall see, a number of different languages are used to query graph databases, sometimes including extensions to SQL.

2.4.4 Graph query languages

Whereas Section 2.3.4 was entitled "Structured query *language*," this corresponding section for graph databases is intentionally titled "Graph query *languages*." Relational databases, founded on Codd's relational algebra introduced in Section 2.3.3, enjoy a standard query language, SQL, supported universally with only minor syntactic variations across different relational DBMS. Unfortunately, at the time of writing there exists no comparable consensus on a standard query language for graph databases.

The establishment in 2019 of an ISO (International Standards Organization) initiative to develop a standard GQL (Graph Query Language) makes it more likely one may exist when you read this. From the graph query languages available today[4], Cypher (and its open-source sibling openCypher) is used here, as it is amongst the most popular languages today and certain to be a major influence on any future standards.

Cypher is an SQL-like language for querying labeled property graph databases. As a result, Cypher is built around nodes and edges and their associated properties and labels. As for SQL, the basic Cypher commands for creating, deleting, and updating data in the graph database are quite straightforward and not covered further here. Unlike SQL, querying involves searching for matching patterns rather than combinations of relational restrictions, projections, and joins. For example:

```
MATCH (n:CUSTOMER) RETURN n
```

retrieves all nodes with the label ("type") CUSTOMER (as shown in Figure 2.15), whereas:

```
MATCH e=()–[r:RESIDES]–>() RETURN e
```

retrieves all edges with the label RESIDES together with their incident nodes (as shown in Figure 2.16). Many Cypher queries are similarly constructed and involve matching data in the database against various forms of the basic (node)–[directed edge]–>(node) pattern.

While the two queries above reflect the underlying node and edge storage structures in a graph database, it is often still convenient to organize query results in the form of a table. For example, the query:

```
MATCH (c: CUSTOMER)–[p:PLACED]–>(o: ORDERS)
  RETURN c.TEL, c.GNAME, c.FNAME, c.TITLE, o.ORDER_ID, o.MEAL, o.SIZE
```

[4] Common graph database query languages today include *Cypher* and *openCypher*; *GraphQL* and *GSQL*, two entirely different languages that should not be confused with the proposed standard *GQL*; *PGQL* (property graph query language); and *SPARQL*, which is a de facto standard for triple stores, the second major class of graph database discussed further in later chapters.

will return the table of customer orders as per the relational join $\bowtie_{\text{TEL=CUS\_TEL}}$ (CUSTOMER, ORDERS) (already seen in Table 2.6) or equivalently from the SQL join query:

```
SELECT * FROM CUSTOMER, ORDERS WHERE CUSTOMER.TEL = ORDERS.CUS_TEL
```

Note that the explicit link between customers and their orders, in the form of an edge in the graph database, negates any need to specify the implicit connection between the primary keys of the corresponding tables, found in the relational database query.

The power and efficiency of graph pattern-matching starts to become most evident when queries responses require searching for paths through the graph. For example, the Cypher query:

```
MATCH (a)<−[:RESIDES]−(c)−[:PLACED]−>(o)−[:SHIPPED]−>(b)
  WHERE b<>a RETURN DISTINCT c.GNAME, c.FNAME,
    a.STREET_AD AS HOME, b.STREET_AD AS SHIP
```

identifies the customers who have a different home address to their shipping address. The query searches for non-cyclic paths from address nodes, through customers' home addresses and orders, and finally back to shipping addresses, using the pattern (a)<−[:RESIDES]−(c)−[:PLACED]−>(o)−[:SHIPPED]−>(b). The equivalent SQL query, below, requires a more complex four-way join across three tables to achieve the same result.

```
SELECT CUSTOMER.GNAME, CUSTOMER.HNAME,
    HADDRESS.STREET_AD AS HOME, SADDRESS.STREET_AD AS SHIP
  FROM ADDRESS AS HADDRESS, ADDRESS AS SADDRESS, CUSTOMERS, ORDERS
  WHERE HADDRESS.ADDRESS_ID = CUSTOMERS.HID AND
    CUSTOMER.TEL = ORDERS.CUS_TEL AND
    ORDERS.ORDER_ID = SADDRESS.ADDRESS_ID AND
    NOT(HADDRESS.ADDRESS_ID = SADDRESS.ADDRESS_ID)
```

Table 2.10 gives the final result for both Cypher and SQL versions of the query.

GNAME	FNAME	HOME	SHIP
Lorna	Dane	1407 Graymalkin Lane	85 Chestnut Street

Table 2.10: Evaluation of a graph-based path matching query or four-way relational join

Cypher also needs no GROUP BY statement, simplifying many aggregate queries. For example, the Cypher query below generates the total weight of orders containing more than three items, equivalent to the SQL query used to generate Table 2.9:

```
MATCH (o:ORDERS)−[c:CONTAINED]−>()
  WITH o.ORDER_ID AS ORDN, COUNT(o.ORDER_ID) AS ORDC, SUM(c.WEIGHTG) AS WG
  WHERE ORDC > 3
  RETURN ORDN, ORDC, WG
```

Cypher also requires no comparable data definition commands to SQL, as there is no comparable need to define the schema structure in a graph database. Hence, SQL commands such as CREATE TABLE have no analog within Cypher. The one exception is that Cypher does allow the creation of integrity constraints to mandate the existence or uniqueness of properties on nodes or edges. For example, to require that the contact telephone number property is not null for customer nodes, we can issue the Cypher query:

```
CREATE CONSTRAINT ON (c:CUSTOMER) ASSERT c.TEL IS NOT NULL
```

This balance between structure and flexibility is at the root of the practical implications of choosing a relational versus a graph database. Graph databases are without question more flexible than relational databases, making it easy to change or adapt stored data structures in response to evolving needs. The rigid tabular structure of relational databases, however, provides in-built constraints that can improve consistency, for example, between the conceptual model and the implemented database.

2.4.5 Databases for spatial data handling

In an unmodified state, neither relational nor graph databases can claim to be well suited to spatial data management. In both cases, there can be difficulties when the technology is applied to spatial data. The main issues are:

Structure of spatial data Spatial data has a structure that does not naturally fit with tabular structures. Vector areal data is typically structured as boundaries composed of sequences of line segments, each of which is a sequence of points. Such sequences of arbitrary length violate first normal form. We shall see that some spatial data, in particular data about networks, does fit well with graph database structures. However, other spatial data—such as vectors and rasters—does not fit neatly into a graph structure either. So the need for special structures to store spatial data remains for graph databases too.

Performance The structure of relational databases can ensure simple and efficient queries about logically related data, in particular data contained within a single table. However, as queries become more complex, requiring joins across multiple tables, so the benefits of graph databases come to the fore, both in terms of performance and ease of formulation of queries. Reconstructing spatial objects and relationships is typically complex, usually requiring joining multiple tables stored in a relational database, with resulting performance overheads. Graph databases tend to be better adapted for more complex queries, and so they might be expected to be better suited to spatial queries. However, the underlying mismatch between graphs and some spatial data structures means the relationship is more complex than it might appear at first glance.

Indexes Indexing questions are the focus of Chapter 6. An index aims to increase the speed of queries by storing some additional data in order to aid access and retrieval. Both relational and graph databases need specialized spatial indexes if they are to perform well with spatial data structures. Happily, in many cases these index structures work well for spatial data stored in both relational and graph databases, as we shall see.

Reflections

Databases are a core technology for GIS; equally, databases are a core expertise for geographic information scientists. A solid grounding in databases often serves as a marker of advanced technical skills for geographic information scientists in the GIS industry. For those wishing to deepen their knowledge in databases, Elmasri & Navathe (2016) and Connolly & Begg (2014) are two venerable and trusted database texts with recently updated editions. Garcia-Molina, Ullman, & Widom (2013) is also recommended as a shorter yet respected, authoritative text.

All three classic database texts are firmly founded in relational databases, and less oriented towards graph databases specifically. In writing this third edition, it has been exciting to reflect on how much graph databases (mentioned only in passing in the second edition) have changed the landscape in databases and in GIS over recent years. Of the introductory database texts with an explicit graph database focus, Sullivan (2015) is recommended. Zhang, Song, & Liu (2014) and Baralis, Dalla Valle, Garza, Rossi, & Scullino (2017) provide some brief comparisons of the technical aspects of relational and NoSQL databases for spatial data, but no other introductory texts explicitly on the topic existed at the time of writing this book.

Practical mastery of databases, whether relational or graph, is best achieved by mastering a database query language.[5] Database query languages are among the most rewarding of computing languages to learn and use. There is great satisfaction to be gained from finding compact and elegant expression to capture a complex query using SQL, Cypher, or other graph database languages. While there are many excellent online resources for learning database query languages, Date (2015) is highly recommended as a resource for truly mastering SQL.

Finally, in introducing abstract graphs, set operations, and (relational) algebra, this chapter has taken a first step into the topic of discrete mathematics—an important topic in its own right that later chapters will build upon. As for databases, a foundation in discrete mathematics can be rewarding intellectually and provide the foundations of a deeper understanding of the GIS, and computing more generally. Lipschutz & Lipson (2021) and Epp (2018) are

[5] The website to accompany this book contains downloadable relational and graph versions of the NUGGETS database in Appendix A for readers who would like to start learning to query there.

both excellent, accessible, and thorough introductions to the topic, replete with practical questions and exercises to support the learner. A brief discrete mathematics primer is also included in Appendix B of this book for those who only need a refresher.