

# Informatics 1 :: Data and Analysis



Lecture Notes



# Data and Analysis

## Lecture Notes

Frank Keller      Helen Pain      Stratis Viglas  
[keller@inf.ed.ac.uk](mailto:keller@inf.ed.ac.uk)    [helen@inf.ed.ac.uk](mailto:helen@inf.ed.ac.uk)    [sviglas@inf.ed.ac.uk](mailto:sviglas@inf.ed.ac.uk)

Revision 1.0  
January 2005



## CONTENTS

<b>I</b>	<b>Structured Data</b>	<b>1</b>
I.1	Overview . . . . .	1
I.2	The Entity/Relational Model . . . . .	2
I.2.1	Entities . . . . .	2
I.2.2	Relationships . . . . .	3
I.2.3	Additional Features . . . . .	5
I.3	Relational Databases . . . . .	9
I.3.1	The Relational Model . . . . .	9
I.3.2	Data Definition in SQL . . . . .	10
I.3.3	Mapping E/R Diagrams to Relational Schemata . . . . .	14
I.4	Querying and Manipulation . . . . .	18
I.4.1	Data Manipulation Through Relational Algebra . . . . .	18
I.4.2	Tuple Relational Calculus . . . . .	23
I.4.3	Examples . . . . .	25
I.5	Semi-structured Data and XML . . . . .	31
<b>II</b>	<b>Semi-structured Data</b>	<b>35</b>
II.1	Basic Concepts . . . . .	35
II.1.1	Corpus Data . . . . .	36
II.1.2	Questions Corpora Can Answer . . . . .	36
II.1.3	Obtaining Corpus Counts . . . . .	39
II.1.4	Building Applications Using Corpora . . . . .	40
II.2	Data Acquisition and Annotation . . . . .	41
II.2.1	Balancing and Sampling . . . . .	42
II.2.2	Pre-processing . . . . .	43
II.2.3	Markup Languages . . . . .	46
II.2.4	Corpus Annotation . . . . .	47
II.3	Querying Corpora . . . . .	51
II.3.1	Concordances . . . . .	52
II.3.2	Regular Expressions . . . . .	54
II.3.3	Collocations . . . . .	56
II.3.4	Statistical Tests . . . . .	58

II.4	Information Retrieval . . . . .	61
II.4.1	Information Retrieval Systems . . . . .	61
II.4.2	Indexing . . . . .	62
II.4.3	Vector Space Models . . . . .	65
II.4.4	Evaluation . . . . .	68

---

# CHAPTER I

## STRUCTURED DATA

---

*For some applications, data is inherently structured. In this chapter we shall look at different ways of representing structured data. In particular, we shall focus on data representation in the Entity/Relationship model and we shall see how this maps to the most dominant paradigm of structured data representation, namely the relational data model. We shall then look at different ways of manipulating data in this format and how these ways map to the paradigms introduced in other parts of the course.*

### I.1 Overview

When dealing with structured data, we essentially have all information regarding their properties at our disposal. As such, we can choose the way the data is represented so that we can facilitate its manipulation. The basic steps involved in structured data representation and manipulation are as follows:

1. *Requirements Analysis*: The data needs to be gathered and their structure identified. We also need to worry about what the users of the application expect, *i.e.*, what will the mode of interaction with the application be. This is the step where we will identify the structure of the data.
2. *Conceptual Design*: Once the data has been identified, the next step is to organise it in such a way so that the inherent semantic links between the data are maintained. The objective is to identify a representation of not only how the data interact, but also how the users interact with the data.
3. *Logical Design*: After the structure of the data and its semantic properties have been identified, we need to implement the conceptual design in terms of the actual application. This involves mapping the conceptual design to a logical data representation; the outcome of this process is what we call a *logical schema*.

Through the course of this chapter<sup>1</sup> we will focus on a particular conceptual data representation, namely the Entity/Relationship model and a particular data model, the relational model. These two models form the basis of *relational databases*, which are the most widely used data storage and manipulation paradigm. We shall focus on the conceptual and logical design for relational databases. We shall see how the

---

<sup>1</sup>The basic source for notes for this chapter is Ramakrishnan and Gehrke (2003).

Entity/Relationship model maps to the relational data model and what tools are available in relational databases to define and manipulate data, the latter in the form of relational algebra. We shall also see how the data manipulation primitives discussed map to data manipulation mechanisms that have been discussed in other parts of the course – namely logic.

## I.2 The Entity/Relational Model

The Entity/Relationship (ER) model Chen (1976) is a way of describing the objects involved in an organization and capturing the dependencies between them. It is a powerful abstraction that maps to numerous different logical designs, but it blends particularly well with the relational data model.

### I.2.1 Entities

**Entity:** *a distinguishable object in the real world.*

The first building block of the ER model is an *entity*, which is used to describe any distinguishable object in the real world. For instance, an Informatics student is an entity that can be distinguished by other entities. The term entity itself can be used to identify a set of entities with similar properties. Such a collection of entities is what is referred to as an *entity set*. In the previous example, the student in question belongs to the entity set of all Informatics students. Note that entity sets need not be disjoint; for instance a student can be both an Informatics student and an Engineering student if s/he is doing both degrees. We shall revisit this point in the sequence.

Each entity has some characteristic *attributes*. Entities belonging to the same entity set all share the same attributes. For example, all students have a matriculation number. The number of attributes we wish to assign to a given entity reflects upon the level of detail of our design. For instance, for students we may be interested in their matriculation number, their name and their degree. We may not be interested, however, in their address. The attributes we choose for an entity reflects upon the level of detail of the design. Different designs will have different requirements regarding the level of detail, even though they might represent the same entities.

For each attribute of an entity<sup>2</sup>, there is a *domain* associated with it. The domain is the set of possible values of the given attribute. For instance, the domain of a page number is the set of all positive integers. What distinguishes one entity from another in the same entity set is the values of their attributes. For instance, each student has a different matriculation number. The minimal set of attributes whose values allow us to uniquely identify an entity form the entity's *key*. Note that there may be more than one possibilities for a key for a given entity; these are all called *candidate keys*. When we choose to designate one as the key, this candidate key becomes the entity set's *primary key*. Each entity set *must* have a primary key to ensure set participation.

---

<sup>2</sup>And, hence, for all attributes of all entities in the same entity set.

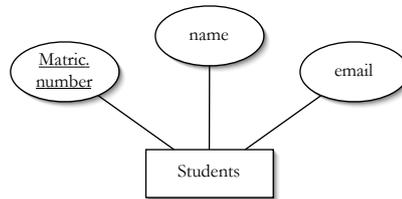


Figure I.1: Representation of the Students entity set

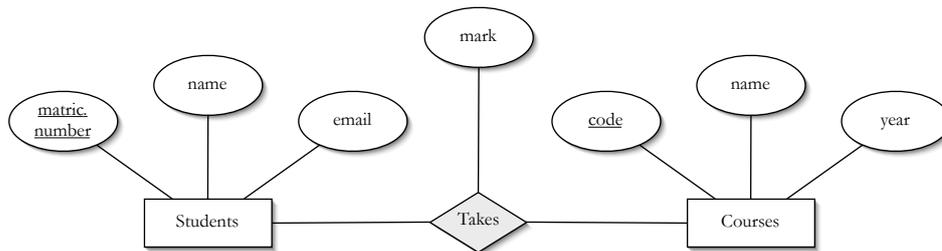


Figure I.2: The Takes relationship between Students and Courses

For each entity set, there is *always* at least one key: the collection of values across all attributes of the entity.

For example, consider the real world entity set of students. The representation of the entity set is shown in Figure I.1. The entity set is represented as a rectangle and each attribute is represented by an oval. The primary key attribute (in this case, the matriculation number) is underlined.

### I.2.2 Relationships

*Relationships* model associations between entities. For instance, Natassa may be an Informatics student taking a particular course. When describing entities, we saw that it is possible to group entities in an entity set. The same grouping principle applies for relationships too; relationships with similar properties can be grouped into a *relationship set*. The elements of a relationship set can be thought of as tuples consisting of  $n$  elements. For instance, the elements of the relationship set between the entity sets Students and Courses can be represented as

$$\{(s, c) | s \in \text{Students}, c \in \text{Courses}\}$$

*i.e.*, each element of the relationship set is a combination of an element from the Students entity set with an element from the Courses element set. The same relationship is shown in Figure I.2, where the relationship is represented as a diamond. Diagrams like the one shown in Figure I.2 are called ER *diagrams*.

Just as in the case of entities, relationships may have *descriptive attributes* that

**Relationship:** *an association between entities.*

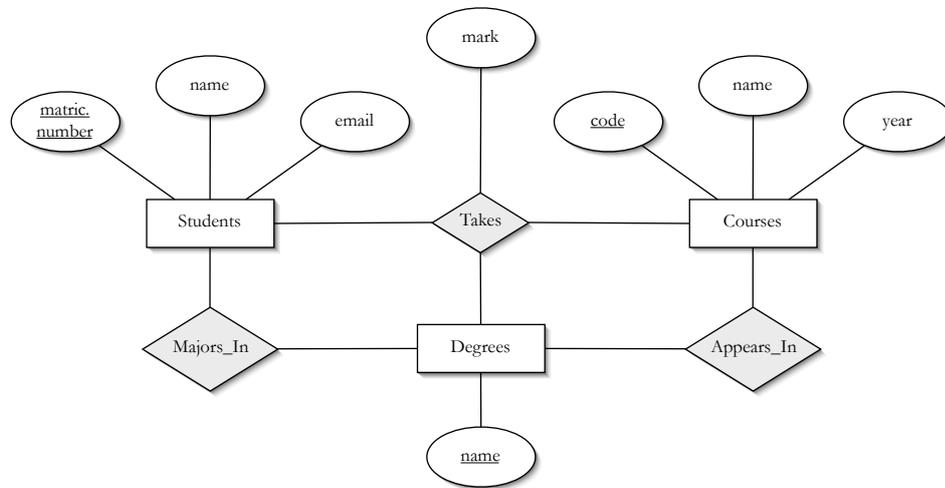


Figure I.3: More relationships between Students, Courses and Degrees

are relevant *only* to the relationship and not to any of the participating entities. In Figure I.2, for instance, there is an attribute called “mark” which is only relevant to the relationship, *i.e.*, it is independent of any of the participating entities.

Once values are given to the attributes of the participating entities of a relationship, and to the attributes of the relationship itself, we say that we have a *relationship instance*. For example, an instance of the Takes relationship is the following:

$$\left( \overbrace{(123, \text{Natassa}, \text{natassa@somewhere}, \underbrace{\text{infl}, \text{Informatics 1}, 1}_{\text{Courses entity}}}, \overbrace{88}_{\text{Takes relationship attribute}}) \right)$$

where two entities are combined, along with a value for the relationship-specific attribute, to form it.

A relationship does not have to be between only two entities. In fact, it is rather common that relationships are defined between more than two entities. Consider for example a third entity, Degrees. A student may be taking a course towards a particular degree; this means that the Takes relationship may be defined across all three participating entity sets, something which is shown in Figure I.3. On the other hand, the same entity can participate in more than one relationships. Consider for instance, the Majors\_In relationship between Students and Degrees. This relationship is independent of towards which degree a student takes a particular course. It is therefore modeled separately in the ER diagram.

Note how different facets of the data can be captured using the ER diagram:

- The entities can appear and be represented regardless of whether relationships actually exist between them.

- Different interactions between entities are captured by different relationships.

This allows for different aspects of the data to be modeled in a structured and clean manner. For instance, in the ER diagram of Figure I.3 a course may appear in multiple degrees; the particular degree for which a student is taking the course is decoupled from that relationship — it is a different relationship altogether.

### I.2.3 Additional Features

Up to this point, we have only been concerned with the declaration of the entities, the relationships among them and their properties. There are, however, a number of other issues we are interested in when modeling data using the ER model. In particular, we have not delved into any *semantic constraints* between the entities and relationships of the model. Semantic constraints are a way of capturing the meaning of the data we are representing.

**Semantic constraints** capture the meaning of the interactions among the data.

#### I.2.3.1 Key Constraints

The only instance of a semantic constraint we have seen so far, is the *key* of an entity set, *i.e.*, a way of uniquely identifying an entity in a given entity set. Semantic constraints in the form of keys, however, are possible when dealing with relationships. Consider for instance the interaction between Students and another entity set, Directors Of Studies — or simply DoSs, modeled as the relationship Directed\_By and shown in Figure I.4. Each student has a single DoS; in other words, given a student we can identify his or her DoS. On the other hand, a student has at most one DoS. This is an example of a *key constraint*. A more formal definition is the following: in a relationship R between  $n$  entity sets  $E_1 \dots E_n$  where a key constraint is on one of the entities  $E_k$ , then by instantiating  $E_k$ , *i.e.*, giving a value to the attributes of  $E_k$ , we can determine the instance of the relationship it participates in. In the example of Figure I.4, there is a key constraint on Students; instantiating a student, determines his or her DoS. In an ER diagram, this is denoted by an arrow from the entity set on which the key constraint is placed, to the relationship. In our example, the arrow means that given a Student entity, we can determine the relationship instance it participates in.

**Key constraint:** by instantiating one of the participating entities in a relationship, we can determine the others.

Another way of thinking stems of looking at the number of times an entity appears in relationship instances. In the example between DoSs and Students, we say that the relationship Directed\_By is a *one-to-many relationship* since it indicates that one DoS is associated with many Students. The Takes relationship of Figures I.2 and I.3, on the other hand, is an instance of what is called a *many-to-many relationship* since there is no constraint on how many courses a student can take. Note that if there is a key constraint then *by definition* the relationship is of the one-to-many kind.

**One-to-many and many-to-many relationships** capture how many entity instances per relationships may appear.

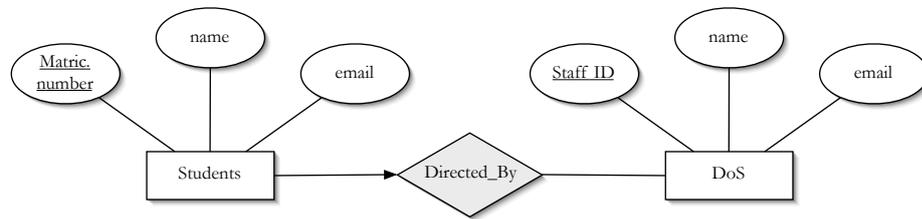


Figure I.4: A key constraint on the Students entity

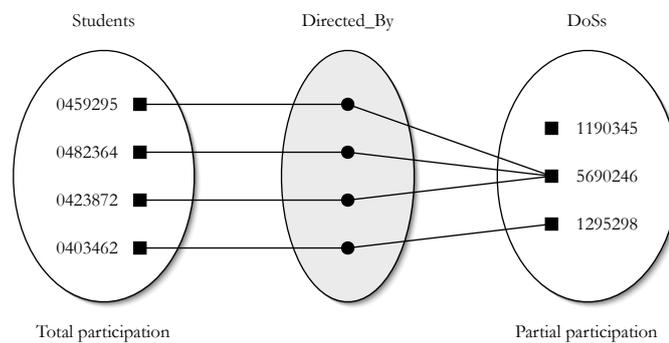


Figure I.5: Total and partial participation of entities

### 1.2.3.2 Participation Constraints

**Participation constraints** capture the mode in which an entity set participates in a relationship.

Finally, we can think in terms of participation of entities in relationships. Consider the Directed\_By relationship and the key constraint on it; it specifies that every student has a director of studies. How about the inverse? Does every DoS entity have a Student entity it is related to? The answer is no — there are DoSs that are not directors for any students. A *participation constraint* captures the mode in which an entity set participates in a relationship. We have two kinds of participation constraints: (i) a *total participation* on entity set E for relationship R is declared when every entity  $e \in E$  appears in a relationship instance of R; (ii) alternatively, a *partial participation* on entity set E for relationship R is declared when it is allowable for entities  $e \in E$  not to appear in instances of R.

Given the above formulation, the participation of the Students entity set in the Directed\_By relationship is total; the DoSs entity set, however, partially participates in the relationship. In terms of pictorial representation of participation, total participation is denoted as a thick line between the relationship and the participating entity. If there are key constraints on the totally participating entity, the arrow representation



Figure I.6: Representation of total participation; the attributes have been omitted for simplicity.

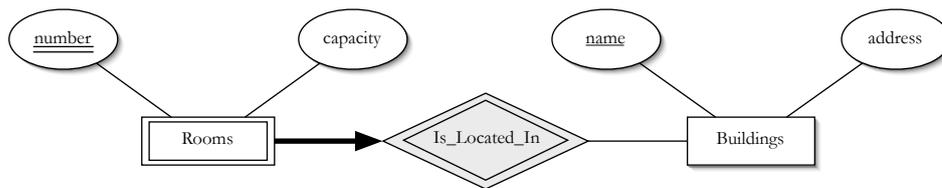


Figure I.7: An example of a weak entity set

is retained. (See also Figure I.6 for an example.)

### I.2.3.3 Weak Entities

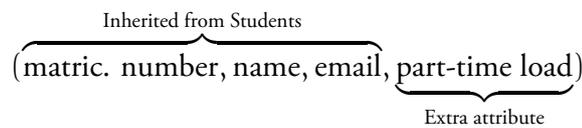
We have seen that a subset of the attributes of an entity set can be designated as a primary key. In certain cases, however, the key is formed not only from the attributes of a single entity set. It is formed by a combination of its own attributes and attributes from another entity set with which it has a relationship. In these cases, the entity set in question is called a *weak entity set*; the entity set from which attributes are borrowed in order to form the primary key is called the *identifying owner*. The relationship that connects the weak entity set and its identifying owner is called an *identifying relationship*.

Consider, for example, a scenario like the one shown in Figure I.7 where Buildings and Rooms are represented. A relationship between the two designates which building each room is located in. Each room has a number and a capacity; each building has a name and an address. Though the name of the building can be used as its primary key, the number of a room cannot be used as a primary key for Rooms; the reason is that across different buildings, the same room number may appear. For that reason, in order to form a primary key for the Rooms entity set, we need to “borrow” the key of the Buildings entity set in order to form a composite key. The part of the primary key that belongs to the weak entity is underlined with a double line. The weak entity set is drawn with a double box and the identifying relationship with a double diamond. Note also that a weak entity *must* have a total participation constraint with its identifying relationship.

**Weak entities cannot exist unless they have an identifying relationship with an identifying owner.**

### I.2.3.4 Hierarchical Entities and Inheritance

The final feature we will discuss has to do with the refinement of entity sets. For example, consider Students. They can either be Full-time Student, or Part-time Students. In both cases we say that they *refine* the Students entity sets and they become *subclasses*; the Students entity set is called the *superclass*. Subclasses provide additional information about their superclass, in the form of attributes, while at the same time they *inherit* all the superclass's attributes. For instance, the complete set of attributes for Part-time Students is



*i.e.*, all the Students attributes are still evident, but they are extended with information specific only to Part-time Students.

**A superclass is specialised through inheritance into subclasses; subclasses are generalised by the superclass.**

Note that this is a much cleaner way of thinking about the entity sets we are trying to model. The alternative would be to embed the extra full-time and part-time student attributes in the students entity and add an extra attribute for the Students entity set designating the type of student, *i.e.*, a “flag” denoting a student as full-time or part-time. However, this meant that although full-time and part-time students have different properties, we would treat them uniformly. By creating an inheritance, we have the following two advantages:

- The subclasses still retain the properties of their superclass, but they extend them. In this case, we say that the superclass is *specialised* in the subclasses. On the other hand, the subclasses are *generalised* by the superclass.
- Changing the values of the attributes of the superclass immediately has an effect on the attribute values of the subclasses. Changing the value for an attribute specific to a subclass does not reflect on the superclass's attribute values. For instance, the change of a student's name is a general one and takes place regardless of whether a student is a full-time or part-time student, *i.e.*, it takes place at the superclass level. Changing the load of a part-time student, however, is only specific to part-time students, *i.e.*, it takes place at the subclass level.

The meaning of these changes is elegantly captured by ER modeling through the forming hierarchies and the concept of inheritance.

The Students hierarchy is shown in Figure I.8; a hierarchy relationship between entity sets is designated as a triangle modeling what is known as an “Is-a” relationship (*i.e.*, a part-time student *is a* student).

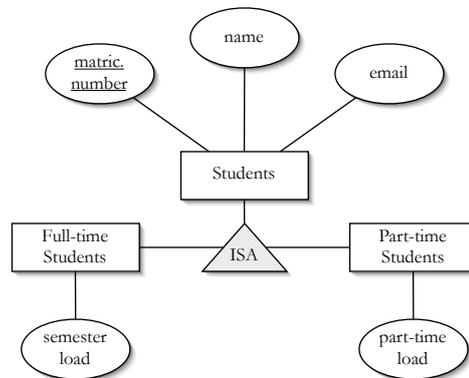


Figure I.8: An example of a hierarchy

### I.3 Relational Databases

The relational model was first introduced by E. F. Codd in 1970 Codd (1970) when most existing database system implementation were using a hierarchical or a network data model. The simplicity of the relational model immediately caught on and led to the first relational database systems being developed by industry Astrahan et al. (1979) (at IBM's Almaden Research Center) and by academia Stonebraker et al. (1976) (at the University of California at Berkeley). The first prototypes were built by the mid 1970's and the relational database industry has since become a multi-billion dollar industry. Nowadays, whenever one is doing anything involving structured data — ranging from bank account transactions to browsing the web — chances are that, at some point, a relational database server is going to be used. In this chapter we will introduce the relational data model, we will see how one can convert conceptual designs in the ER model to relational schemata and we will also see how one can manipulate relational data.

#### I.3.1 The Relational Model

The basic construct of representing data in the relational model is a *relation*. A relation consists of what is known as a *schema* and an *instance*. An instance of a relation is what is also known as a database *table*. The schema of the table is a description of the column headings and the type of each column of the table.

A schema is a set of *fields*. A field is a (attribute-name, attribute-domain) pair. In various texts, fields will be referred to as *columns* or *attributes*, while domains may be referred to as *types*. A domain is nothing more than the set of possible values that can be assigned to an attribute. For instance, if we are modeling a relation about Students, the relational schema might be:

```
Student(mn: string, name: string, age: integer, email: string)
```

*A relation consists of its schema; a schema is a collection of fields with domain information for each field. An instance of a relation is a table, consisting of a set of tuples that assign values to the fields of the schema.*

which means that the necessary information to describe a student is:

- his/her matriculation number is denoted by a field called *mn*, which is a `string`, *i.e.*, the domain of the matriculation number is alphanumeric character sets;
- the student's name is denoted by a field called *name*, which is a `string`;
- his/her *age*, which is an `integer` number;
- and, finally, the student's email address (denoted by a field called *email*) which again is a `string`.

An instance of the relation adhering to the specified schema is a table. Tables consist of *tuples*, *i.e.*, sets of values for the fields of the schema. Tuples are also referred to as *rows*, or *records*. These concepts are summarised in Figure I.9. There, we see a table with a schema specified as above and four tuples in it. Each tuple contains values for the four fields of the table's schema.

To formalise all of the above, let  $R$  be a relation,  $f_i, i = 1, \dots, n$  be the relation's fields and  $D_i$  the domain of field  $f_i$ . A relation instance is then a set of tuples, each containing  $n$  values,  $v_i, i = 1, \dots, n$  for the  $n$  fields of the schema such that:

$$\{\langle f_1 : v_1, \dots, f_n : v_n \rangle \mid v_1 \in D_1, \dots, v_n \in D_n\}$$

Given this notation, the first tuple of Figure I.9 can be written as:  $\langle mn : s0456782, name : John, age : 18, email : john@inf \rangle$ . Note that we treat a relation instance as a *set*; this means that the every tuple in the instance appears exactly

*Arity is the number of fields in a schema; cardinality is the number of tuples in an instance.*

once. Each relation is also characterised by its *arity*; each relation instance is characterised by its *cardinality*. The arity of a relation is the number of fields in its schema; the cardinality of a relation instance is its number of tuples. For instance, the arity of the Students relation is four; the cardinality of the relation instance if Figure I.9 is four as well.

### I.3.2 Data Definition in SQL

Before we move on to mapping ER diagrams to relational schemata, we will present the basic tool that allows us to define a relational schema in the first place. This is in the form of a subset of the *Structured Query Language* (SQL) used by relational database system, called the *Data Definition Language* (DDL) Boyce and Chamberlin (1973). In the next few sections, we will see how one can declare and modify a relation in SQL. SQL in its entirety allows us not only to define, but also query relational data; in this section we will confine ourselves to only data definition. For

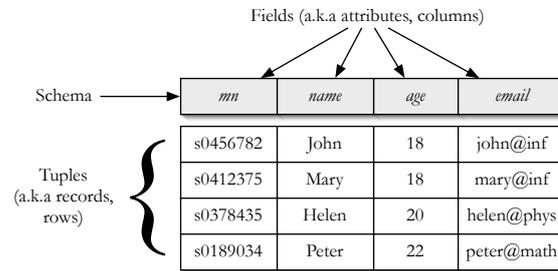


Figure I.9: An instance of the Students relation

simplicity, we assume that there are only three built-in types in SQL, with obvious semantics: `integer` for integer numbers, `real` for real numbers and `char( n )` for an alphanumeric sequence of maximum `n` characters.

The SQL language uses the keyword `table` to denote relations. In order to create a table<sup>3</sup>, one has to use the `create table` construct, which is of the form:

```
create table table name ( attribute name attribute type
                          [ , attribute name attribute type ] * )
```

*i.e.*, a `create table` command, followed by *at least one* definition of an attribute; more than one attributes are separated by commas. To properly define an attribute, we need to declare the attribute name and the attribute type. For instance, the Students relation can be defined as follows:

```
create table Students ( mn char(8),
                        name char(20),
                        age integer,
                        email char(15) )
```

The ER model, as we have described, allows one to specify semantic constraints over the entities and relationships described. All these constraints in the relational model are called *integrity constraints*. For a table to be acceptable, all integrity constraints relevant to the table need to be satisfied at all times. An implementation of the relational model — in all probability by using a relational database management system — will have to enforce these integrity constraints. In other words, only legal instances will be permitted to be stored in the database. The types of constraints we will talk about in the remainder of this section are *primary key constraints* and *foreign key constraints*

**Integrity constraints**  
model the semantic constraints in the relational model.

### I.3.2.1 Primary Key Constraints

We have seen when presenting the ER model, that it is possible to designate a subset of attributes of an entity set as the *minimal* set of attributes that allows one to

<sup>3</sup>In the sequence, we shall use the terms “table” and “relation” interchangeably.

decide on entity set participation. This translates almost verbatim to the relational model. A *primary key* for a relation means that each tuple in the relation will have different values for the attributes of the primary key. We also saw that it is possible for different subsets of an entity set's attributes to act as keys; each such subset is a *candidate key*. The relational model, through its implementation in SQL, allows primary keys to be declared.

Consider for example the Students relation. We know student matriculation numbers are unique. We also know that email addresses are unique. So, if one gives us a matriculation number, or an email address, we can tell whether there is a tuple in the Students relation containing those values. The matriculation number and the email address are candidate keys for the Students relation. However, in the relational model and, in extension, SQL there can be only *one* primary key. So at this point, we need to decide on which of the candidate keys we will use as the primary key for the relation. Let us assume that we choose the matriculation number as the primary key. We can make this choice explicit when defining the Students table, by employing the SQL primary key command:

```
create table Students (  mn          char(8) ,
                        name       char(20) ,
                        age        integer ,
                        email      char(15) ,
                        primary key ( mn ) )
```

Note at this point that we have declared the primary key and the SQL implementation itself will enforce this constraint at all times. For instance, if a user tries to modify the Students table by inserting a tuple that contains a matriculation number that already exists in the database, the insertion will *not* be permitted.

### I.3.2.2 Foreign Key Constraints

When discussing relationships and weak entities in the ER model, we saw that it is possible for “linking” between entity sets. This mechanism is available in the relational model as well, through the existence of foreign key constraints. Consider for example another relation that models which students take which classes, with the following schema:

```
Takes(mn: string, code: string, mark: integer)
```

*i.e.*, a table of three attributes, the first attribute being the matriculation number of the student, the second attribute being a symbolic code for the course and the third attribute being the mark that the student has achieved in the course (see also Figure I.2). We also know that the matriculation number of a student is unique (*i.e.*, it is a primary key for the Students table) and let us assume that the code of a course is unique as well (*i.e.*, if there is a Courses table, the *code* attribute has been declared as its primary key.)

Consider now the scenario in which we want to ensure that only existing students can enroll in courses. Or, the inverse of that, that students can enroll in courses that will certainly be offered. This information is already encoded in the Takes table, provided we can make it accessible. The only thing we need to declare is that there is a reference from the Takes relation to the Students relation and that reference is materialised by the *mn* field. We can declare what in relational model terms is called a *foreign key* constraint, which models a semantic linking between the two tables. We can declare as many foreign key constraints for a single relation as there are applicable. Foreign key declaration is achieved through the `foreign key` construct of SQL. In SQL, the definition for the Takes table becomes:

```
create table Takes (
    mn          char(8) ,
    code        char(20) ,
    mark        integer ,
    primary key ( mn, code ) ,
    foreign key ( mn )
                references Students ,
    foreign key ( code )
                references Courses )
```

The primary key for the relation is set to the combination of (*mn*, *code*), *i.e.*, every pair of values for these two columns in every Takes relation instance needs to be unique. The foreign key in the referencing relation (in this example, Takes) *must* match the primary key of the referenced relation(s) (in our example, Students and Courses). “Matching” in this context means that the two keys have the same number of columns and compatible data types; the actual column names may be different (*e.g.*, we could have named the *mn* column of the Takes table as *matric*). The definition of the Takes table essentially imposes two constraints:

1. whenever a tuple is inserted, the value for the *mn* field needs to be a value that appears in the matriculation number column of the Students table;
2. the value for the *code* field needs to be a value that appears in the corresponding column of the Courses table.

Consider for example the scenario depicted in Figure I.10 where we see instances of Students, Courses and Takes. Note that each tuple appearing in the Takes relation instance contains values in its *mn* and *code* fields that appear in the respective columns of the Students and Courses tables. The insertion of a tuple in the Takes table for a non-existing student, for instance  $\langle s0237367, adbs, 90 \rangle$ , is not allowed *regardless* of whether the code for the course exists; the same applies to the insertion of a tuple for a non-existing course, *e.g.*,  $\langle s0456782, phys1, 42 \rangle$ , is not permitted as well, again regardless of the fact that a corresponding student exists. *Both* foreign key constraints need to be satisfied for an insertion to be successful.

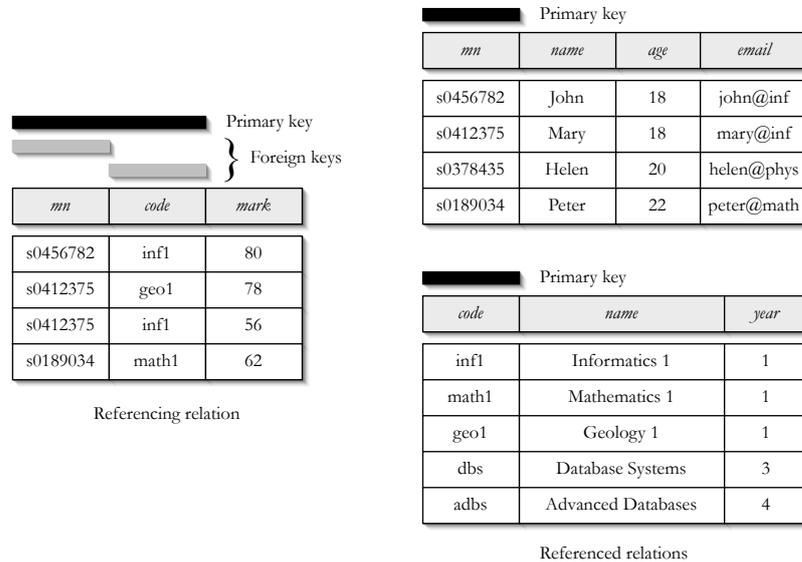


Figure I.10: Foreign key constraints

### I.3.3 Mapping E/R Diagrams to Relational Schemata

In Section I we saw how, after the requirements specification of an application are captured, they can be translated to a conceptual schema. The outcome of this process is an ER modeling of the data that captures the various entities, relationships among them and various semantics of those relationships. We will now talk about how it is possible to move from the conceptual design of an ER model to a logical design of a relation schema.

#### I.3.3.1 Mapping Entity Sets

An entity set is the most straightforward concept to map to a relation. An entity set becomes a table and each attribute of the entity set becomes an attribute of the table. Key constraints of the entity set are mapped to primary key constraints of the table.

Consider for example the Courses entity of Figure I.2; this is mapped straightforwardly to the following table definition:

```
create table Courses (
    code          char(6),
    name         char(20),
    year         integer,
    primary key  ( code ))
```

### I.3.3.2 Mapping Relationship Sets without Constraints

Each relationship set in the ER model is mapped to a relation in the relational model, by following these principles:

- the relation contains the primary keys of the participating entity sets;
- any descriptive attributes of the relationship are added to the relation;
- a composite primary key, consisting of the primary keys of the participating entity sets is declared on the relation;
- foreign key constraints are declared on the primary keys, referencing the primary keys of the participating entity sets.

Given these principles, then the relation for the Takes relationship shown in Figure I.3, between Students, Takes and Degrees can be defined as follows:

```
create table Takes (
    mn          char(8) ,
    code        char(20) ,
    name        char(20) ,
    mark        integer ,
    primary key ( mn, code, name ) ,
    foreign key ( mn )
                references Students ,
    foreign key ( code )
                references Courses ,
    foreign key ( name )
                references Degrees )
```

### I.3.3.3 Mapping Relationship Sets with Key Constraints

Whenever there is a key constraint, the situation is slightly altered. The key issue is that whenever we have a key constraint we cannot use a combination of the keys of the participating entity sets as the primary key of the relationship table. Consider the case of Students and DoSs first introduced in Figure I.4. Note that because each student has at most one DoS, no two tuples of the relationship table can have the same value for the matriculation number attribute but different values for the DoS's staff id. As a consequence, it suffices for the matriculation number of a student to be declared as the primary key of the relationship table. The Directed\_By relationship can therefore be mapped to the following table:

```

create table Directed_By (
    mn          char(8),
    staff_id   char(8),
    primary key ( mn ),
    foreign key ( mn )
                references Students ,
    foreign key ( staff_id )
                references DoSs )

```

The method to map a relationship to a relation for relationships with key constraints is therefore similar to the one used for mapping simple relationships to relations; the difference is that instead of declaring a composite primary key we only declare as primary key the *same* primary key of the entity sets that acts as the “source” of the directed edge in the ER diagram (in the previous case, Students). This is the minimal amount of information that allows us to decide participation in the relationship.

### I.3.3.4 Mapping Relationship Sets with Participation Constraints

The relational model allows for what are called `null` values, *i.e.*, values that are left undefined. For instance, one can insert a tuple in the Students table of the form  $\langle s0389483, \text{Joe}, \text{null} \rangle$  for a student named Joe but who does not have a designated email address. The value of the email address field is `null`. By allowing or disallowing `null` values in a relationship table we can implicitly declare total participation constraints. Total participation is achieved through designating a field as `not null`. For instance, total participation in the Directed\_By relationship table for Students can be declared by annotating the definition of the `mn` attribute with the `not null` keyword as follows:

```

[ ... ]
mn char(8) not null,
[ ... ]

```

In this specific example, the `not null` specification is redundant since by definition key fields in SQL are never allowed to have a `null` value. However, the general mechanism of ensuring total participation in a relationship table is by explicitly disallowing the fields relevant to the totally participating entity set to take `null` values.

### I.3.3.5 Mapping Weak Entity and Weak Relationship Sets

Recall from the discussion that weak entity sets cannot exist without an identifying relationship and an identifying owner in the relationship. This means that there should be no instances of a weak entity without a corresponding identifying owner. The steps to mapping a weak entity set to a relation are as follows:

- create a table for the weak entity set, incorporating all its attributes;

- add another attribute set, which is the primary key of the entity set's identifying owner;
- add a foreign key constraint on the identifying owner's primary key;
- instruct the system to automatically delete any tuples in the table for which there are no owners.

We have already seen how the first three steps can be accomplished in SQL. The final step is accomplished by another SQL construct called *cascading deletions*, implemented by the command `on delete cascade`. Consider for example the schema for the weak entity and weak relationship sets of Figure I.7. This will be translated in the following SQL command for defining the `Rooms` table:

```
create table Rooms (
    number          char(8),
    capacity        integer,
    name            char(20),
    primary key      ( number, name ),
    foreign key      ( name )
                    references Buildings ,
                    on delete cascade )
```

Note that the weak entity set and the weak relationship set have been collapsed into a single table. The primary key for this table has been set to  $(number, name)$  pairs. A foreign key constraint has been declared on the name of a building, with the additional constraint that whenever a building is deleted for any reason, all entries referring to this building in the table should be deleted as well; the latter is achieved through the `on delete cascade` command.

### I.3.3.6 Mapping Hierarchical Entities

Hierarchical relationships are special types of relationship. The approach we follow in mapping them to relations is the following:

- declare a relation for the superclass of the hierarchy, specifying its primary key;
- for each subclass specialisation, declare another relation, combining into it the primary key of the superclass and all the extra attributes of the subclass;
- for each subclass, declare as its primary key the same primary key as in its superclass;
- declare a foreign key constraint on the primary key of the subclass, to the primary key of the superclass.

Given this approach, the definition for Part-time Students from Figure I.8 is as follows:

```
create table PT_Students (  mn           char(8) ,
                          pt_load       integer ,
                          primary key   ( mn ) ,
                          foreign key   ( mn )
                                  references Students )
```

The definition for Full-time Students is similar. Note that by using this approach, if we want to retrieve information that is relevant to all students, we can do so by looking into the *Students* table. For information that is relevant to Part-time Students, we only need to look into the *PT\_Students* table. We can retrieve all extra information for Part-time Students not present in the *PT\_Students* table (for instance, their name or their email address) by taking advantage of the link between the *PT\_Students* and *Students* tables implemented by both tables' *mn* attribute.

## I.4 Querying and Manipulation

Once the data has been organised in a relational database, the natural next step is identifying ways of manipulating data conforming to the defined schema. In this section we shall present two such ways:

- *relational algebra*, is the dominant formal language for expressing queries over relationally represented data;
- *tuple-relational calculus* is another language, which is tightly coupled to first order predicate logic.

Relational algebra and tuple-relational calculus have the same expressive power, *i.e.*, any query that can be expressed in one can be expressed in the other. We will examine each mechanism in turn.

### I.4.1 Data Manipulation Through Relational Algebra

Relational algebra is a procedural abstraction of expressing queries over relational data. The key concept in relational algebra is an *operator*. Each operator has the fundamental property of accepting either a single relation instance or a pair of relations instances as input and producing a single relation instance as output. This means that relational algebra operator can be *composed* in order to form *relational algebra expressions* — in other words complicated queries. We shall first present the five basic operators of relational algebra (namely, *selection*, *projection*, *union*, *cross-product*, and *difference*) and we shall then present some additional operators that although they can be expressed in terms of basic operators, they are so frequently encountered that special attention is given to them.

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

(a)  $\sigma_{\text{age}>18}(\text{Students})$

<i>name</i>	<i>age</i>
John	18
Mary	18
Helen	20
Peter	22

(b)  $\pi_{\text{name,age}}(\text{Students})$

<i>name</i>	<i>age</i>
Helen	20
Peter	22

(c) Combination

Figure I.11: Examples of selection and projection operator

### I.4.1.1 Selection and Projection

Recall that relational data is organised in a tabular format; the selection and projection operators allow us to isolate any rectangular subset of a table. Selection operators are expressed as:

$$\sigma_{\text{predicate}}(\text{Relation Instance})$$

*i.e.*, the predicate in the subscript should be evaluated on the relation instance parameter. The predicate is a condition on one or more of the attribute values of the relation, that is evaluated on every tuple. Naturally, each predicate should evaluate to either *true* or *false*. The result of a selection is another table, with the same schema as its input but with fewer tuples (or, at most the same number of tuples if the selection predicate evaluates to true for each tuple). The predicate can be any Boolean combination *i.e.*, any combination  $\text{term}_1 \text{ bop } \text{term}_2 \text{ bop } \dots \text{ bop } \text{term}_i$  where  $\text{bop} \in \{\wedge, \vee\}$ . The  $\text{term}_i$  terms are of the form  $\text{attribute } \text{rop } \text{constant}$  or  $\text{attribute}_1 \text{ rop } \text{attribute}_2$  where  $\text{rop} \in \{>, <, =, \neq, \geq, \leq\}$ . Reference to an attribute is usually done by its name (*i.e.*, name) or by qualifying the attribute name with the name of the table it belongs to (*i.e.*, table-name.attribute-name).

*A selection filters rows of the input table.*

Consider, for example, the Students table of Figure I.9. The selection operator expressed as:

$$\sigma_{\text{age}>18}(\text{Students})$$

returns the relation instance shown in Figure I.11(a). Note how the tuples of the input that do not satisfy the selection predicate are eliminated from the output.

The projection operator extracts entire columns of its input relation instance. It is expressed as:

$$\pi_{\text{column list}}(\text{Relation instance})$$

*A projection filters columns of the input table.*

with the semantics that columns in the column list should be retained in the output; the rest of the columns are “projected out,” *i.e.*, dropped from the schema. Consider,

for example the evaluation of the projection operator:

$$\pi_{\text{name,age}}(\text{Students})$$

over the table of Figure I.9. The output of this operation is shown in Figure I.11(b). Note that only the attributes in the subscript list are retained in the output; the other attributes are not.

Given their properties, we can combine selection and projection operators in order to build more complicated expressions that allow us to select any rectangular region of the original table. For instance, the following expression:

$$\pi_{\text{name,age}}(\sigma_{\text{age}>18}(\text{Students}))$$

combines the two previous operations into a single one, giving the result shown in Figure I.11(c). This is a repercussion of selections and projections accepting relations as input and producing relations as output.

### I.4.1.2 Set Operations

**Set operations** have the expected semantics from algebra, but extended to the relational data model. There are four set operations in relational algebra: *union* ( $\cup$ ), *intersection* ( $\cap$ ), *difference* ( $-$ ) and *cross-product* ( $\times$ ). All four operations are binary, *i.e.*, they accept two relations as input. The first three assume that both input relations have the same number of fields and, in addition, corresponding fields taken in a left-to-right order have the same domains. Note that the actual field names are *not* used when testing compatibility.

**Union:** the union  $R \cup S$  of two relations  $R$  and  $S$  results in a new relation with the same schema as the two input relations; the output relation consists of tuples appearing in *either* of the input relations. For naming purposes, it is assumed that the fields of the output relation inherit the names of the relation appearing first in the intersection specification (*i.e.*,  $R$  in the previous case).

**Intersection:** the intersection  $R \cap S$  of two relations  $R$  and  $S$  results in a new relation with the same schema as the two input relations; the output relation consists of tuples appearing in *both* input relations. The field naming convention is the same as in the union case.

**Set difference:** the set difference  $R - S$  of two relations  $R$  and  $S$  results in a new relation with the same schema as the two input relations; the output relation consists of tuples appearing in  $R$  but not in  $S$ . The field naming convention is the same as in the union case.

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

$S_1$

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0489967	Basil	19	basil@inf
s0412375	Mary	18	mary@inf
s9989232	Ophelia	24	oph@bio
s0189034	Peter	22	peter@math
s0289125	Michael	21	mike@geo

$S_2$

<i>code</i>	<i>name</i>	<i>year</i>
inf1	Informatics 1	1
math1	Mathematics 1	1
geo1	Geology 1	1
dbs	Database Systems	3
adbs	Advanced Databases	4

$R$

Figure I.12: Three example relation instances

**Cross-product (also known as *Cartesian product*):** the cross-product  $R \times S$  of two relations  $R$  and  $S$  is a new relation. The schema of the new relation is the union of the two schemata. The resulting relation contains one tuple  $\langle r, s \rangle$  for each pair of tuples  $r \in R$  and  $s \in S$ .

Consider the three relation instances shown in Figure I.12. Two instances ( $S_1$  and  $S_2$ ) are over the Students; the third relation instance ( $R$ ) is over the Courses schema. The results of  $S_1 \cup S_2$ ,  $S_1 \cap S_2$ ,  $S_1 - S_2$ , and  $S \times R$  are shown in Figure I.13. Note that we can only apply union, intersection and set difference over  $S_1$  and  $S_2$ ; since  $S_1$  or  $S_2$  and  $R$  do not share compatible schemata, these operations are inapplicable. Cross-product, however, is perfectly permissible over relations with different schemata.

### I.4.1.3 Renaming

When discussing set operations, we had to resort to naming conventions in case of naming conflicts. For instance, when dealing with the cross-product of  $S_1$  and  $R$  in Figure I.13 we saw that there are two instances of a field named *name*. In that case, we “annotated” the conflicting field names with the name table they originated from. In order to avoid such conflicts we introduce a *renaming* operator of the form:

**Renaming** helps to give aliases to column and table names.

$$\rho_{\text{New relation name}(\text{renaming list})}(\text{Original relation name})$$

with the following semantics:

- the original relation is assigned the new relation name;

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math
s0489967	Basil	19	basil@inf
s9989232	Ophelia	24	oph@bio
s0289125	Michael	21	mike@geo

$S_1 \cup S_2$

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0412375	Mary	18	mary@inf
s0189034	Peter	22	peter@math

$S_1 \cap S_2$

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0456782	John	18	john@inf
s0378435	Helen	20	helen@phys

$S_1 - S_2$

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>	<i>code</i>	<i>name</i>	<i>year</i>
s0456782	John	18	john@inf	inf1	Informatics 1	1
s0456782	John	18	john@inf	math1	Mathematics 1	1
s0456782	John	18	john@inf	geo1	Geology 1	1
s0456782	John	18	john@inf	dbcs	Database Systems	3
s0456782	John	18	john@inf	adbs	Advanced Databases	4
s0412375	Mary	18	mary@inf	inf1	Informatics 1	1
s0412375	Mary	18	mary@inf	math1	Mathematics 1	1
s0412375	Mary	18	mary@inf	geo1	Geology 1	1
s0412375	Mary	18	mary@inf	dbcs	Database Systems	3
s0412375	Mary	18	mary@inf	adbs	Advanced Databases	4
s0378435	Helen	20	helen@phys	inf1	Informatics 1	1
s0378435	Helen	20	helen@phys	math1	Mathematics 1	1
s0378435	Helen	20	helen@phys	geo1	Geology 1	1
s0378435	Helen	20	helen@phys	dbcs	Database Systems	3
s0378435	Helen	20	helen@phys	adbs	Advanced Databases	4
s0189034	Peter	22	peter@math	inf1	Informatics 1	1
s0189034	Peter	22	peter@math	math1	Mathematics 1	1
s0189034	Peter	22	peter@math	geo1	Geology 1	1
s0189034	Peter	22	peter@math	dbcs	Database Systems	3
s0189034	Peter	22	peter@math	adbs	Advanced Databases	4

$S_1 \times R$

Figure I.13: Examples of set operations over the relation instances of Figure I.12

- the renaming list consists of terms of the form  $\text{oldname} \rightarrow \text{newname}$  which rename a field named  $\text{oldname}$  to  $\text{newname}$ ;
- for  $\rho$  to be well-defined there should not be any conflicts in the output.

In certain cases, either the new relation name, or the renaming list may be omitted.

Obviously, not both can be omitted at the same time — this is meaningless.

As an example, the expression:

$$\rho_{C(mn \rightarrow sid)}(S_1)$$

returns a new relation called C whose schema is:

$$C(sid: \text{string}, name: \text{string}, age: \text{integer}, email: \text{string})$$

*i.e.*, the same schema as  $S_1$ 's schema, but with the  $mn$  field renamed to  $sid$ .

#### I.4.1.4 Joins

Joins are some of the most commonly used operations in the relational model. However, there is no explicit need to define a special join operator, other than convenience. A join is essentially a selection with a predicate of the form  $col_1 \text{ rop } col_2$  where  $\text{rop} \in \{>, <, =, \neq, \geq, \leq\}$  over a cross-product of two relations. It can be defined as follows

*A join is a combination of the selection and cross-product operators.*

$$R \bowtie_p S = \sigma_p(R \times S)$$

where  $p$  is called the join predicate.

Consider, for example, the two relation instances of the Students and Takes relations, shown in Figure I.14(a) and Figure I.14(b). If we join the two relation instances by applying the operation

$$\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \text{Takes}$$

then the resulting relation instance will be the one shown in Figure I.14(c). Note that the result consists of the combination of the two participating tables' schemata, since a join necessarily implies a self-selection over a cross-product.

The *natural join* is a refinement of the join operator and it is helpful in situations like the one described in Figure I.14 where essentially there is a natural way of joining the two participating relations: they have fields of the same name ( $mn$  in this case). If this is the case, then the join predicate can be omitted and the join operation can be simply expressed as  $\text{Students} \bowtie \text{Takes}$ . Additionally, the joining attribute of one of the relations is dropped from the final schema so that there is no information duplication. In the previous example, one of the two  $mn$  columns would be omitted from the result schema.

#### I.4.2 Tuple Relational Calculus

Tuple relational calculus is another powerful mechanism allowing one to express queries over relationally organised data. The usefulness of tuple relational calculus

<i>mn</i>	<i>name</i>	<i>age</i>	<i>email</i>
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

<i>mn</i>	<i>code</i>	<i>mark</i>
s0456782	inf1	80
s0412375	geo1	78
s0412375	inf1	56
s0189034	math1	62

(a) Students                      (b) Takes

<i>(mn)</i>	<i>name</i>	<i>age</i>	<i>email</i>	<i>(mn)</i>	<i>code</i>	<i>mark</i>
s0456782	John	18	john@inf	s0456782	inf1	80
s0412375	Mary	18	mary@inf	s0412375	geo1	78
s0412375	Mary	18	mary@inf	s0412375	inf1	56
s0189034	Peter	22	peter@math	s0189034	math1	62

(c) Students  $\bowtie_{\text{Students.mn=Takes.mn}}$  Takes

Figure I.14: An example of a join operation

lies in the fact that it is entirely *declarative* in the sense that when expressing a query, one only needs to specify the properties that should hold in the result set.

The key concept in tuple relational calculus is a *tuple variable* that ranges over the tuples of a relation instance. Queries in tuple relational calculus are expressed as:

$$\{T \mid p(T)\}$$

where  $T$  is a tuple variable and  $p(T)$  is a first order logic formula that evaluates to either *true* or *false*. The result of the query is an instantiation of all tuples  $t \in T$  for which  $p(t)$  evaluates to *true*.

For instance, the following query identifies all students older than 18:

$$\{S \mid S \in \text{Students} \wedge S.\text{age} > 18\}$$

To evaluate this query, the tuple variable  $S$  is instantiated over all tuples in the Students table; the predicate  $S.\text{age} > \text{age}$  is then evaluated on the tuple. If the predicate evaluated to *true* the tuple is propagated to the output; otherwise it is dropped. Note that this tuple relational calculus query is equivalent to relational algebra query  $\sigma_{\text{age}>18}(\text{Students})$ .

### I.4.2.1 Formal Syntax of Tuple Relational Calculus Queries

Let  $\text{Rel}$  be a relation name,  $R$  and  $S$  be tuple variables,  $a$  an attribute of  $R$  and  $b$  an attribute of  $S$ . Let  $\text{op}$  denote a logical operator in the set  $\{>, <, =, \neq, \geq, \leq\}$ . The first key concept is an *atomic formula* which is one of the following:

- $R \in \text{Rel}$ ;

**Tuple relational calculus** is first order predicate logic extended for the relational model.

- $R.a \text{ op } S.b$ ;
- $R.a \text{ op constant}$ , or  $\text{constant op } R.a$ .

A *formula* is recursively defined to be a combination of formulae where  $p$  and  $q$  are themselves formulae and  $p(R)$  denotes a formula in which tuple variable  $R$  appears:

- any composite formula;
- $\neg p$ ,  $p \wedge q$ ,  $p \vee q$ , or  $p \Rightarrow q$ ;
- $\exists R(p(R))$ , where  $R$  is a tuple variable;
- $\forall R(p(R))$ , where  $R$  is a tuple variable.

The semantics of these should be obvious from logic. Sometimes, the following notational conventions are used:

- in the case of existential quantifiers, instead of denoting the domain of a tuple variable as a conjunct in a formula, we apply it as a separate step outside the formula, *i.e.*, instead of writing  $\exists R(R \in \text{Rel} \wedge p(R))$ , we write  $\exists R \in \text{Rel}(p(R))$ ;
- in the case of universal quantifiers, we follow a similar route if we have implication, *i.e.*, instead of writing  $\forall R(R \in \text{Rel} \Rightarrow p(R))$ , we write  $\forall R \in \text{Rel}(p(R))$ .

Finally, it is possible to implicitly declare resulting schemata by introducing tuple variables. For instance, the following query in tuple-relational calculus introduces the resulting relation  $P$  with two fields: name and age:

$$\{P \mid \exists S \in \text{Students}(S.\text{age} > 20 \wedge P.\text{name} = S.\text{name} \wedge P.\text{age} = S.\text{age})\}$$

*i.e.*, the atomic formulae  $P.\text{name} = S.\text{name}$  and  $P.\text{age} = S.\text{age}$  give values to the fields of the resulting tuples.

### I.4.3 Examples

In the sequence, we shall present examples queries expressed in both relational algebra and tuple relational calculus. If and when necessary, we shall provide a more detailed explanation of one can go about constructing the expressions.

**Example I.1:** *Find the names of students who are taking Informatics 1.*

A relational algebra expression for this query is:

$$\pi_{\text{Students.name}} \left( \text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \left( \text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} \left( \sigma_{\text{name}=\text{Informatics 1}}(\text{Courses}) \right) \right) \right)$$

It is sometimes easier to see the sequence of operations by drawing a tree diagram of the relational algebra expression; this is shown in Figure I.15(a). Note how the operators are actually applied:

- all three relations will need to be accessed:
  - Students, in order to retrieve the students' names;
  - Courses, in order to select only the course we are interested in; and
  - Takes because it stores information about which students are taking which courses;
- a selection on the Courses relation is applied so the information on only the relevant course (Informatics 1) is returned;
- the Students relation is joined with Takes (by comparing the key-foreign key values) so we know the codes of the courses the student is taking;
- the result (now with complete course code information) is joined with the Courses relation (after the irrelevant courses have been filtered out) again by comparing key-foreign key values) so we have information on students taking Informatics 1;
- finally, we are only interested in those students' names, so we project only this attribute in order to compute the output.

The equivalent tuple relational calculus expression for this query is the following:

$$\{P \mid \exists S \in \text{Students} \exists T \in \text{Takes} \exists C \in \text{Courses} \\ (C.\text{name} = \text{'Informatics 1'} \wedge C.\text{code} = T.\text{code} \wedge \\ S.\text{mn} = T.\text{mn} \wedge P.\text{name} = S.\text{name})\}$$

Again, note the correspondence with the steps we took for constructing the relational algebra expression: (i) we have introduced one tuple variable for each relation that needs to be accessed; (ii) a formula of atomic formulae is used to specify the properties of the output; (iii) the schema of the output is implicitly declared by a fourth tuple variable (P), with a single attribute (*name*); (iv) the values for P.name are assigned by adding the relevant atomic formula (P.name = S.name).  $\triangle$

**Example I.2:** *Find the names of all courses taken by Joe.*

A relational algebra expression for this query is the following:

$$\pi_{\text{Courses.name}}((\sigma_{\text{name}=\text{'Joe'}}(\text{Students})) \bowtie_{\text{Students.mn}=\text{Takes.mn}} \\ (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} \text{Courses}))$$

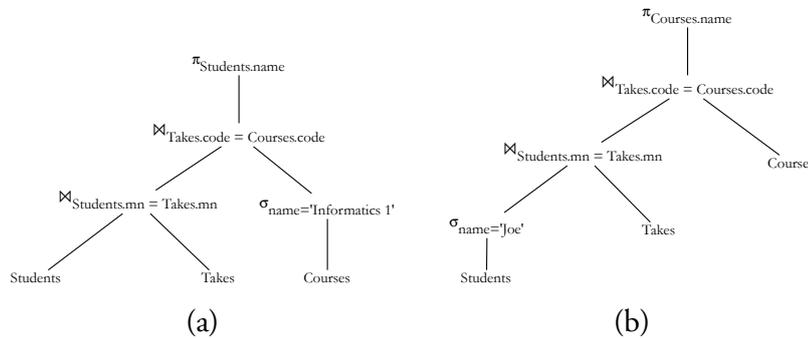


Figure I.15: Relational algebra expressions for (a) Example I.1 and (b) Example I.2

while a tree-like diagram of the expression is shown in Figure I.15(b). A tuple relational calculus for this query is:

$$\{P \mid \exists S \in \text{Students} \exists T \in \text{Takes} \exists C \in \text{Courses} \\ (S.\text{name} = \text{'Joe'} \wedge S.\text{mn} = T.\text{mn} \wedge \\ C.\text{code} = T.\text{code} \wedge P.\text{name} = C.\text{name})\}$$

Note how this query and the query of Example I.1 are quite similar. Their differences lie in which input relation is filtered — Courses in Example I.1, Students in this example — and information from which relation we are interested in the output — Students in Example I.1, Courses in this example. Intuitively, one would expect the mechanisms employed in answering the queries to be quite similar as well. This is indeed the case, as is evident from both the relational algebra and the tuple relational calculus expressions.  $\triangle$

**Example I.3:** Find the names of all students who are taking Informatics 1 or Geology 1.

One way of expressing this query in relational algebra is the following:

$$\pi_{\text{Students.name}} ( \\ (\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \\ (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} (\sigma_{\text{name}=\text{'Informatics 1'}}(\text{Courses})))) \cup \\ (\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \\ (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} (\sigma_{\text{name}=\text{'Geology 1'}}(\text{Courses}))))))$$

*i.e.*, “reproduce” the query used in Example I.1 that retrieves the students taking Informatics 1, compute the union of those students with students taking Geology 1 (so we now have students taking Informatics 1 or Geology 1 — or both) and then

making the final projection of those students' names. There is, however, a much better way of expressing this query using the union set operator by observing that we only need the union at the level of the Courses relation. So we only need compute the union of information relevant to Informatics 1 and Geology 1 and then go about forming the query as we did in the previous case. The relational algebra expression after this observation becomes:

$$\pi_{\text{Students.name}}(\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} (\sigma_{\text{name}=\text{'Informatics 1'}}(\text{Courses}) \cup \sigma_{\text{name}=\text{'Geology 1'}}(\text{Courses}))))$$

At this point we can improve the expression even more, by observing that we can combine the two selections and the union operation in the same selection operator by introducing a *disjunction* of the two predicates as follows:

$$\pi_{\text{Students.name}}(\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} (\sigma_{\text{name}=\text{'Informatics 1'}} \vee \sigma_{\text{name}=\text{'Geology 1'}}(\text{Courses}))))$$

Note how we started with a cumbersome and complicated expression and ended up in a much more elegant and readable relational algebra representation of the query. This example introduces certain concepts of *query optimisation*. If one aims at building a system employing these principles in order to answer user queries over relational schemata, there are ways of “re-writing” queries in equivalent forms. Some of these forms will be computationally cheaper than others. Query optimisation is the process by which one decides which equivalent form to use when answering a query by enumerating each alternative, estimating the computational cost of each alternative and picking the cheapest.

Moving on to tuple relational calculus, a quite straightforward way of expressing this query is the following:

$$\{P \mid \exists S \in \text{Students} \exists T \in \text{Takes} \exists C \in \text{Courses} ((C.\text{name} = \text{'Informatics 1'} \vee C.\text{name} = \text{'Geology 1'}) \wedge C.\text{code} = T.\text{code} \wedge S.\text{mn} = T.\text{mn} \wedge P.\text{name} = S.\text{name})\}$$

Note how this representation is much closer to the simplest relational algebra representation of the query. This is in essence the power of tuple relational calculus. It is *declarative* in the sense that we need not organise the order in which operations are applied. We only need to *declare* the properties of the results. In fact, tuple relational calculus is so elegant a representation of queries, that it forms the basis of SQL, *i.e.*, the standard query language used by commercial relational database management systems.  $\triangle$

**Example I.4:** *Find the names of students who are taking both Informatics 1 and Geology 1.*

One might be tempted to use the last relational algebra expression from Example I.3, but replace the disjunction in the selection predicate with a conjunction as follows:

$$\begin{aligned} & \pi_{\text{Students.name}} (\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \\ & \quad (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} \\ & \quad \quad (\sigma_{\text{name}=\text{'Informatics 1'} \wedge \text{name}=\text{'Geology 1'}}(\text{Courses})))) \end{aligned}$$

This, however, would be incorrect. The reason is that it tries to retrieve courses that are both named Informatics 1 and Geology 1 at the same time — which is not possible. Recall that a selection operator evaluates its predicate on every row of its input. This selection operation, although perfectly legal, would return the empty set as a result. The empty set would also be returned if instead of a conjunction in the predicate, one used a set intersection of the form:

$$\sigma_{\text{name}=\text{'Informatics 1'}}(\text{Courses}) \cap \sigma_{\text{name}=\text{'Geology 1'}}(\text{Courses})$$

for exactly the same reasons. The correct way of expressing this query in relational algebra is actually computing the intersection of students who are taking Informatics 1 and students who are taking Geology 1, in the same manner of the first approach taken in Example I.3:

$$\begin{aligned} & \pi_{\text{Students.name}} ( \\ & \quad (\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \\ & \quad \quad (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} (\sigma_{\text{name}=\text{'Informatics 1'}}(\text{Courses})))) \cap \\ & \quad (\text{Students} \bowtie_{\text{Students.mn}=\text{Takes.mn}} \\ & \quad \quad (\text{Takes} \bowtie_{\text{Takes.code}=\text{Courses.code}} (\sigma_{\text{name}=\text{'Geology 1'}}(\text{Courses})))))) \end{aligned}$$

The tuple relational calculus expression for this query is:

$$\begin{aligned} & \{P \mid \exists S \in \text{Students} \wedge \forall C \in \text{Courses} \\ & \quad ((\text{C.name} = \text{'Informatics 1'} \vee \text{C.name} = \text{'Geology 1'}) \Rightarrow \\ & \quad (\exists T \in \text{Takes} (\text{T.mn} = \text{S.mn} \wedge \text{T.code} = \text{S.code} \wedge \\ & \quad \quad \text{P.name} = \text{S.name})))\} \end{aligned}$$

where we make use of implication. This query can also be read as follows: retrieve all students for which if there is a course named Informatics 1 or Geology 1, then the student is taking that course. Again, note how more elegant it is to express this query

in tuple relational calculus as opposed to the cumbersome representation (through division) in relational algebra.  $\triangle$

**Example I.5:** *Find the names of students who are taking all courses.*

This query can be better expressed in relational algebra if we introduce the concept of *division*. In general all queries referring to all tuples of a relation can be readily expressed by using division.

Consider two relation instances  $A$  and  $B$  in which  $A$  the schema of  $A$  is a superset of the schema of  $B$ . Let  $s(A)$  be the schema of  $A$  and  $s(B)$  be the schema of  $B$ ;  $s(B) \subseteq s(A)$  holds. Division  $A/B$  is defined as the set of all tuples in  $A$  with the values for  $s(A) - s(B)$  fixed, such that for every tuple in  $B$  with the values for  $s(B)$  fixed there is a tuple in  $A$ .

It is helpful to consider the analogy to integer division. For integers  $A$  and  $B$ ,  $A/B$  is the largest integer  $Q$  such that  $Q \cdot B \leq A$ . For relation instances  $A$  and  $B$ ,  $A/B$  is the largest relation instance  $Q$  such that  $Q \times B \subseteq A$ . Let us now express division in terms of the basic relational operators. The basic idea is to compute the values for  $s(A) - s(B)$  in  $A$  that are not disqualified. A set of values for  $s(A) - s(B)$  if by filling in values for  $s(B)$  from  $B$  we obtain a value for  $s(A)$  that is not in  $A$ . The following relational algebra expression computed the disqualified values:

$$\pi_{s(A)-s(B)}((\pi_{s(A)-s(B)}(A) \times B) - A)$$

so division  $A/B$  can be defined as:

$$\pi_{s(A)-s(B)}(A) - \pi_{s(A)-s(B)}((\pi_{s(A)-s(B)}(A) \times B) - A)$$

The complete query then becomes:

$$\rho(\text{Tempnumbers}, \pi_{mn, code}(\text{Takes})/\pi_{mn}(\text{Courses})) \\ \pi_{name}(\text{Tempnumbers} \bowtie_{\text{Tempnumbers.mn} = \text{Students.mn}} \text{Students})$$

where we have introduced a renaming operation to avoid cluttering the expression.

Let us now “dissect” how this expression works:

- the division between Takes and Courses returns all matriculation numbers such that a combination of  $\langle$  matriculation number, course code  $\rangle$  exists in Courses; these are the matriculation number of all students taking all courses;
- this result is renamed to another relation instance, Tempnumbers;
- the Tempnumbers relation is then joined on matriculation numbers with the Students relation in order to retrieve name information for the relevant students; and

- finally, the names of the students are projected to form the output.

The query expressed in tuple relational calculus is as follows:

$$\{P \mid \exists S \in \text{Students} \forall C \in \text{Courses} \\ (\exists T \in \text{Takes} (C.\text{code} = T.\text{code} \wedge S.\text{mn} = T.\text{mn} \wedge \\ P.\text{name} = S.\text{name}))\}$$

Note how easy it is to express this query in tuple relational calculus. The reason is the existence of universal quantifiers. Another way of expressing this query is: find the names of students such that for each existing course, they are taking this course — which is exactly what the above formula expresses.  $\triangle$

## I.5 Semi-structured Data and XML

So far, we have dealt with completely structured data. In fact, a good part of the discussion has been about capturing the conceptual properties of the data and creating a schema based on these properties. Schema information is sometimes referred to as *metadata*, *i.e.*, data about the data.<sup>4</sup>

Sometimes, however, the data we wish to handle do not always have well-defined schemata, or there might be multiple ways of representing them. It may even be possible that we have free-flow data that we merely wish to annotate in a meaningful way. Consider for instance, these very lecture notes. When typesetting them, one is not interested in their meaning; they are only interested in the paragraph formatting, whether a piece of text is a section heading, how it is actually displayed on paper and so on. In that respect, the typesetting program only needs to know certain relevant typesetting information, such as, for instance, that the title of this section is “Semi-structured Data and XML” and that section headings should appear horizontally centered in the page. The actual piece of text used to create the section heading is:

```
\section{Semi-structured Data and XML}
```

which contains enough information to instruct its typesetting semantics within the text.

On the other hand, one performing some natural language analysis of this text, may be interested in its grammatical or syntax properties. For instance, in the following sentence: “This is a meaningful sentence”, they might be interested in the fact that *is* is the verb and *meaningful* is an adjective. So they may choose to write the sentence as:

---

<sup>4</sup>This is a well-known abstraction in relational databases; even metadata information is stored in collection of special tables of the system, called the *catalog*. The system can then query the catalog tables in the same way that it queries any data table in order to obtain information about the data that is stored.

This `<verb>is</verb>` a `<adjective>meaningful</adjective>` sentence.

Both are examples of what is called *marking up* the text, *i.e.*, a way of mixing both data (*e.g.*, the words) along with its semantics (*e.g.*, a section heading in the first case, or grammatical properties in the second case). The latter example (*i.e.*, surrounding text with certain “labels”) is what constitutes a general class of marking up technology, referred to as *markup languages*. The currently most dominant such language is XML, which stands for *eXtensible Markup Language*. In fact, XML is so widely spread that most of the relational modeling examples discussed may be expressed in XML.

Consider, for example, data about students, and in particular the table shown in Figure I.9. One way of expressing it in such a scheme is the following:

```
<students>
  <student>
    <mn> s0456782 </mn> <name> John </name>
    <age> 18 </age> <email> john@inf </email>
  </student>
  <student>
    <mn> s0412375 </mn> <name> Mary </name>
    <age> 18 </age> <email> mary@inf </email>
  </student>
  <student>
    <mn> s0378435 </mn> <name> Helen </name>
    <age> 20 </age> <email> helen@phys </email>
  </student>
  <student>
    <mn> s0189034 </mn> <name> Peter </name>
    <age> 22 </age> <email> peter@math </email>
  </student>
</students>
```

Note that we are essentially mixing both metadata and data in the same representation. Just by reading the text, we know that we are dealing with a collection of `students`, and each student has a certain amount of information associated with them — namely, a matriculation number (`mn`), a `name`, their `age`, and an email address (`email`). However, we do not have that descriptive information readily available in the sense of having a schema associated with all students. Rather, there is reasoning involved; we need to enumerate all students and figure out that they all share the same schema.

On the other hand, it is easier for the metadata to evolve — it is extensible in that sense. If we decide to make a note for one of the students, say Peter, we can easily add mark-up information. For instance, we may choose to make a note that Peter

has been granted an extension for submitting his first practical assignment. We can do so by modifying his entry as follows:

```
<students>
  ...
  <student>
    <mn> s0189034 </mn> <name> Peter </name>
    <age> 22 </age> <email> peter@math </email>
    <note> Has extension for first practical </note>
  </student>
</students>
```

This does not affect the rest of the student entries in any way; it also does not spoil the fact that all students continue to share a common core of information. It only adds to this particular student entry.

Constructs like XML and marking up are the best way of representing data for the problem at hand. These concepts will be presented in more detail in the next chapter.



---

## CHAPTER II

### SEMI-STRUCTURED DATA

---

*This chapter deals with unstructured and semi-structured data, focusing on linguistic data (corpora). We will discuss how such data can be acquired in a systematic way, and give an overview of annotation and metadata. We will then describe ways of querying collections of semi-structured data and introduce information retrieval as an application that relies on semi-structured data. Finally, we will deal with evaluation, i.e., with techniques that measure the performance of a system that processes data.*

#### II.1 Basic Concepts

The previous chapter dealt with data that are inherently structured. For *structured data*, information regarding the properties of the data is available at the outset, so that highly structured ways of describing, acquiring, and processing these data can be developed. The entity/relationship model and the relational model presented in the previous chapter are an important example for data models for handling structured data. Such models can be formalized in logical and algebraic terms, and these formalizations typically form the basis for contemporary database technology.

However, types of data exist that do not lend themselves readily to formalization and processing in a structured way. These data are *unstructured data*, i.e., data for which the properties relevant for their formalization and processing are not known at the outset. An example are data about bacteria that a biologist might encounter when they study a soil sample. It is not clear in advance which species they will encounter and what the properties of these species are; also the relationships between the species are probably not known initially; interaction between different types of bacteria might take place that cannot be anticipated. Also, the soil sample might be dynamic, the organisms present in can multiply or die. It is exceedingly difficult to come up with an a priori database scheme that captures this situation adequately.

A second example of unstructured data is *textual data*: Take a piece of text, e.g., extracted from a web site. The text does not have an inherent structure, except in a trivial sense: it is a linear sequence of words (probably interspersed with headings and links in the case of a text from a web site). It does not make a lot of sense to enter the text into a database; it is not even clear how this could be done (maybe each word would be an entity, with a relation *precedes* to link words?).

The text can be enriched with additional structure, but this structure is heavily dependent on the intended application, or on the scientific question we want to ask

“And the ark rested in the seventh month, on the seventh day of the month upon the mountains of Ararat.” Noah’s ark did not come to rest on Mount Ararat (Massis in Armenian) but on the mountains of Ararat or Armenia. It is very strongly believed that the first five books (Pentateuch) of the Bible were written by Moses and another Moses of Khorene stated in his history that Ararat was the central portion of Armenia.

Figure II.1: Excerpt from a corpus

about the text. For example we could identify all verbs in the text and give them a special label. Then a list of verbs can be compiled from the text, e.g., for the purpose of building a dictionary. This process of adding structure to unstructured data is called *annotation*; the result of enriching unstructured data with annotation will be referred to as *semi-structured data*.

*annotation*  
*semi-structured data*

In the remainder of this section, we will provide an overview of the most important concepts relating to semi-structured data, and sketch some of the applications in which semi-structured data play a role.

### II.1.1 Corpus Data

The only forms of semi-structured data that we will deal with in detail in this chapter are linguistic data. A collection of textual or spoken data is referred to as a *corpus* (plural corpora), if it meets a number of *criteria for corpora* (following (McEnery and Wilson, 2001, Ch. 2)):

*corpus*  
*criteria for corpora*

- it is sampled in a certain way;
- it is finite in size;
- it is available in machine-readable form;
- it typically serves as a standard reference.

**Example II.1:** We can therefore ask ourselves: Is a single email a corpus? Is a collection of 1,000 emails a corpus? Is the novel *Harry Potter* a corpus? The answers to these questions will be come clear by the end of this section. △

**Example II.2:** Let us look at excerpts from two different corpora, given in Figure II.1 and II.2. Where do you think these excerpts were taken from? △

### II.1.2 Questions Corpora Can Answer

*empirical questions*

Corpora can serve as a tool for answering *empirical questions* in linguistics and related fields: corpora are not simply sets of words, but can be analyzed using statistical

David Beckham is lending his voice to Vodafone's official voice-mail service, the first celebrity ever to allow such an extraordinary endorsement. Subscribers to the mobile phone company can now have a recording of Beckham's squeaky voice informing callers that "This is the voicemail service for X. Please leave a message after the tone". The move is part of a multi-million pound deal with the Manchester United star and opens up a whole new line of business for celebrities seeking extra cash while they are still in the lime-light.

Figure II.2: Excerpt from another corpus

"My dear fellow." said Sherlock **Holmes** as we sat on either a realistic effect," remarked **Holmes**. "This is wanting in the said **Holmes**, taking the paper and glancing his eye down "I have seen those symptoms before," said **Holmes**, throwing merchant-man behind a tiny pilot boat. Sherlock **Holmes** welcomed "You've heard about me, Mr. **Holmes**," she cried, "else how

Figure II.3: Lines containing the word *Holmes* in *A Case of Identity*

and other tools, and the results of these analyses can be used to test hypotheses about language use, or to discover new facts about language structure.

Corpora are also useful for addressing *engineering questions* : they represent the type of data that computer systems are exposed to if they process linguistic input. Engineers therefore rely on corpora to develop text-based or speech-based computer applications. Algorithms exist that extract regularities from corpus data, which is vital for building language processing systems that learn automatically from input data, that are robust (i.e., tolerant to errors and noise), and that are able to process a wide range of language input accurately.

Let us deal with the empirical aspects of corpus processing first. Assume we have a corpus that consists of the Sherlock Holmes story *A Case of Identity*.

**Example II.3:** Simple questions we could ask using this corpus are:

1. Find all lines containing the word *Holmes*.
2. Find all lines beginning with the word *Holmes*.
3. Find all lines starting with an upper case letter.

The results of these queries are depicted in Figures II.3, II.4, and II.5. △

In more general terms, there are certain important things that we want to find out about a corpus before we can process it effectively. This includes the *token count*

**Holmes**, when she married again so soon after father's death,  
**Holmes** alone, however, half asleep, with his long, thin form  
**Holmes**. "He has written to me to say that he would be here at  
**Holmes** had been talking, and he rose from his chair now with a

Figure II.4: Lines beginning with the word *Holmes* in *A Case of Identity*

**A** Case of Identity  
The husband was a teetotaler, there was no other woman  
**T**ake a pinch of snuff, Doctor, and acknowledge that I  
The larger crimes are apt to be the simpler, for the  
**A**nd yet even here we may discriminate.  
**W**hen a woman has a secret  
**E**therege, whose husband you found so easy when the

Figure II.5: Lines starting with an upper case letter in *A Case of Identity*

: how big is the corpus, i.e., how many tokens (words, punctuation marks, etc.) *token count*  
does it contain in total? This figure is often denoted by  $N$ . The token count needs  
to be distinguished from the *type count* which tells use how many different tokens  
there are in the corpus. For examples, in our example corpus there are many different  
tokens (occurrences) of the word type *Holmes*. Note that the type count of a corpus is  
always lower than the token count (or equal if all the tokens in the corpus only occur  
once). Tokens and types are typically words, but they can also be punctuation marks,  
sentence boundary markers, paragraph breaks, or other items that might occur in a  
corpus.

*type count*

The next important concept is the *absolute frequency* of a type, often denoted as  
 $f(t)$  (for frequency) or  $c(t)$  (for count). The frequency  $f(t)$  of a type  $t$  is defined  
simply the number of occurrences of  $t$  in a corpus. At the same time, the *relative*  
*frequency* of a type is the frequency of  $t$  normalized by the corpus size, denoted as  
 $f(t)/N$ . Using relative frequencies, we can compare the frequencies of types across  
different corpora.

*absolute frequency*

*relative frequency*

**Example II.4:** An important reference corpus for British English is the British  
National Corpus (BNC; Burnard (1995)). Table II.1 compares some counts from  
the BNC with counts from our sample corpus *A Case of Identity*. As we can see from  
this example, the absolute frequency of words such as *Sherlock* and *Holmes* is higher  
in the BNC. The relative frequency of these words is higher in *A Case of Identity*  
(which is after all a Sherlock Holmes story).  $\triangle$

	BNC	A Case of Identity
Token count N	100,000,000	7,006
Type count	636,397	1,621
$f(\textit{Holmes})$	890	46
$f(\textit{Sherlock})$	209	7
$f(\textit{Holmes})/N$	.0000089	.0066
$f(\textit{Sherlock})/N$	.00000209	.000999

Table II.1: Some absolute and relative frequencies in *A case of Identity*

BNC		A Case of Identity	
6184914	the	350	the
3997762	be	212	and
2941372	of	189	to
2125397	a	167	of
1812161	in	163	a
1372253	have	158	I
1088577	it	132	that
917292	to	117	it

Table II.2: The most frequent words in the BNC and in *A case of Identity*

### II.1.3 Obtaining Corpus Counts

Using the concept of absolute frequency, we can ask questions such as: what are the most frequent words in a given corpus? This question can be answered by simply counting all word types in the corpus, and tabulating them in an ordered list.

**Example II.5:** Table II.2 lists the ten most frequent words in the BNC and in *A Case of Identity*. As we can see, the two lists are remarkably similar: the article *the* is the most frequent word in both corpora, and prepositions like *of* and *to* and pronouns like *it* appear in both lists.  $\triangle$

The frequencies of single words are referred to as *unigram frequencies*. This notion can be generalized: we can talk about bigram frequencies (frequencies of pairs of words), trigram frequencies (frequencies of triples of words), or in the general case of *n-gram* frequencies (frequencies of n-tuples of words).  $\triangle$

**Example II.6:** Table II.3 lists the five most frequent n-grams in *A Case of Identity*, for  $n = 2 \dots 4$ . The larger the  $n$ , the more linguistically meaningful the units become.  $\triangle$

This example also illustrates another important concept. As can be seen from Table II.3, the n-gram frequencies get smaller with increasing  $n$ . This is of course

Bigrams	Trigrams	4-grams
40 of the	5 there was no	2 very morning of the
23 in the	5 Mr. Hosmer Angel	2 use of the money
21 to the	4 to say that	2 the very morning of
21 that I	4 that it was	2 the use of the
20 at the	4 that it is	2 the King of Bohemia

Table II.3: The most frequent  $n$ -grams in *A case of Identity**data sparseness*

due to the fact that there are more and more possible combinations of words the larger the  $n$  gets, and we are less and less likely to encounter each of these combinations. This phenomenon is known as *data sparseness*: the counts for larger and linguistic units become smaller and smaller. Even for trigrams and 4-grams, the data sparseness is so severe that many word combinations have a frequency of zero, even in a large corpus such as the BNC.

### II.1.4 Building Applications Using Corpora

*natural language processing**Speech processing*

Corpora are used extensively in two areas of informatics: *natural language processing* (NLP; also called computational linguistics) is the subdiscipline that has the aim of building computer systems that understand or produce text. *Speech processing* is the subdiscipline that develops systems that understand or produce spoken language. Both subdisciplines rely heavily on techniques developed in other disciplines to extract statistical regularities from text or speech corpora. These disciplines include probability theory, statistics, information theory, and machine learning.

*Information retrieval*

Three typical NLP applications that rely heavily on the use of corpus data are:

- *Information retrieval* (IR) is concerned with developing algorithms and models for retrieving information from document collections. For example, given a corpus of newspaper articles and a set of keywords (a query), the task is to return a ranked list of relevant newspaper articles. IR can also be applied to web data – search engines such as Google (see Figure II.6) are probably the most well-known IR systems. We will deal with information retrieval in more detail in Section II.4.

*Summarization*

- *Summarization* is the task of taking a text and compressing it, i.e., producing an abstract or summary that is considerably shorter than the original document. Again, a well-known example is provided by Google (see Figure II.6): with every document that the search engine returns, it also provides a short summary of the document's content. Corpora are an important tool for building summarizers: given a collection of texts and their (manually produced) summaries, we can develop algorithms that learn how to generate these summaries automati-

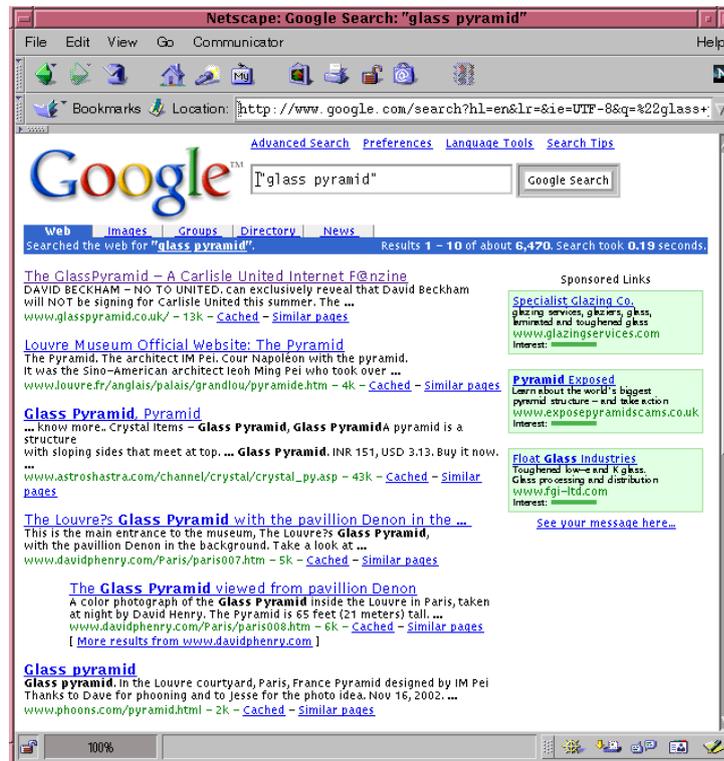


Figure II.6: A well-know example of an information retrieval system

cally.

- *Machine Translation* (MT) is considered by many people the ultimate goal of natural language processing, and it is correspondingly difficult. The task is to take a text in a source language and turn it into a well-formed text in the target language, while preserving its meaning. There are a number of commercial MT systems available, and free systems such as Babelfish (<http://world.altavista.com/>) exist on the web. Again, modern MT technology relies heavily on corpus data, typically aligned corpora, i.e., texts in which the sentences in the source language are manually aligned with the corresponding sentences in the target language. These aligned corpora can be used to learn translation correspondences, based on which the MT system can then translate new texts in the source language. *Machine Translation*

## II.2 Data Acquisition and Annotation

As we have seen in Section II.1.1, McEnery and Wilson (2001, Ch. 2) define a corpus as a collection of textual or speech data that is finite in size, machine readable,

and can serve as a reference. In this section we will discuss a number of standard corpora and explain their design. We will also deal with two important issues relating to corpus design: how to acquire corpus data in a systematic way (balancing and sampling) and how to add information to a corpus (annotation).

### II.2.1 Balancing and Sampling

*balanced corpus*

A *balanced corpus* is designed to contain material that is representative of the language, ideally reflecting the whole range of linguistic material that native speakers are exposed to. For example, for a balanced text corpus, we will want to include texts from books, newspapers, letters, etc. Furthermore, the material from each source should be subject to *sampling*, i.e., the steps should be taken to ensure that the material is representative of the source. If our material is newspaper text, then we will want to randomly select texts from a range of different newspapers, and from different issues of the same newspaper (within a pre-defined period of time, e.g., one year).

*sampling*

*language type*

More generally, a balanced corpus is typically designed to contain representative samples from several language types, and from a wide range of genres, domains, and media. The *language type* of a corpus indicates what kind of language the material is drawn from. This can be edited text (e.g., articles, books, newswire), spontaneous text (e.g., email, Usenet news), spontaneous speech (e.g., conversations, dialogs), or scripted speech (e.g., formal speeches). The *genre* of a text refers to a more fine-grained classification of the type of material. Examples for genres are 18th century novels, scientific articles, movie reviews, or parliamentary debates. Different genres typically differ in many linguistic parameters, such as writing style, level of formality, grammatical complexity. The *domain* of the material indicates what the material is about. Example domains are crime, travel, biology, or law. Finally, the *media* of a corpus are its physical realization; this can be text, audio, transcribed speech, video, etc.

*genre*

*domain*

*media*

*Brown Corpus*

We will illustrate these concepts using two well-known examples of standard reference corpora: the Brown Corpus and the British National Corpus. The *Brown Corpus* is a balanced corpus of written American English. It is famous for being one of the earliest machine-readable corpora, and was developed by Nelson Francis and Henry Kučera at Brown university in the 1960s (Francis et al., 1982). The corpus consists of one million words of American English texts printed in 1961. The texts for the corpus were sampled from 15 different genres and domain to make the corpus a good standard reference.

**Example II.7:** Table II.4 contains an overview of the genres and domains represented in the corpus. △

*British National Corpus*

Another important reference corpus is the *British National Corpus* (BNC;

Genre	Domain	Size
Press	Reportage	44 texts
Press	Editorial	27 texts
Press	Reviews	17 texts
–	Religion	17 texts
–	Skill and hobbies	36 texts
–	Popular lore	48 texts
–	Belles-lettres	75 texts
Miscellaneous	Government and house organs	30 texts
–	Learned	80 texts
Fiction	General	29 texts
Fiction	Mystery	24 texts
Fiction	Science	6 texts
Fiction	Adventure	29 texts
Fiction	Romance	29 texts
–	Humor	9 texts

Table II.4: Genres and domains in the Brown corpus

Burnard (1995)). The BNC is a large, synchronic corpus of British English, consisting of 90 million words of text and 10 million words of speech. Like the Brown corpus, the BNC is a balanced corpus, i.e., it was compiled so as to represent a wide range of present day British English. The written part includes samples from newspapers, magazines, books (both academic and fiction), letters, and school and university essays, among other kinds of text. The spoken part consists of spontaneous conversations, recorded from volunteers balanced by age, region, and social class. Other samples of spoken language are also included, ranging from business or government meetings to radio shows and phone-ins.

The fact that the BNC is a balanced corpus makes it particularly attractive for linguistic research: frequencies obtained from the BNC should be more representative of the language experience of native speakers than the ones obtained from unbalanced corpora. Unbalanced corpora, such as the Penn Treebank (which will be discussed later), typically only represent one genre. All the texts in the Penn Treebank are drawn from the Wall Street Journal, a financial newspaper.

In Table II.5, we compare the Brown Corpus, the BNC, and the Penn Treebank with other corpora in terms of size, genre, modality, and language.

### II.2.2 Pre-processing

Before the raw data in a corpus can be exploited for scientific or engineering tasks, it has to be processed various ways. Two types of processing can be distinguished:

Corpus	Size	Genre	Modality	Language
Brown Corpus	1M	balanced	text	American English
British National Corpus	100M	balanced	text/speech	British English
Penn Treebank	1M	news	text	American English
Broadcast News Corpus	300k	news	speech	7 languages
MapTask Corpus	147k	dialog	speech	British English
CallHome Corpus	50k	dialog	speech	6 languages

Table II.5: Comparison of some standard corpora

pre-processing, which identifies the basic units in a corpus (words, sentences), and annotation, which enriches the corpus with information that is useful for answering a given research question or solving a given engineering task.

*Tokenization*

We will discuss pre-processing first. *Tokenization* is a pre-processing step in which raw textual data is divided into units called tokens. Each token is either a word or a number or a punctuation mark. (Recall the type/token distinction from Section II.1.2.) A *word* is defined here as a continuous string of alphanumeric characters delineated by whitespace. Whitespaces can be space, tab, or newline.

*word*

**Example II.8:** The definition of a word seems straightforward enough. However, tokenization can present problems in some cases:

- Words that contain numbers, punctuation marks, or symbols: *b2b*, *amazon.com*, *Micro\$oft*.
- Words that include an apostrophe: *John's*, *isn't*, *rock'n'roll*.
- Hyphenated words: *child-as-required-yuppie-possession*.
- Non-English words: *cul de sac*, *Zeitgeist*.

△

While there are some problem cases, tokenization can be automated and highly accurate tokenizers are available for English. Note however that tokenization is much harder for other languages. In Chinese, for example, words often consist of several characters – while there is a white space between characters, the boundaries between words are not marked. This makes tokenization (i.e., word boundary detection) a difficult task for Chinese and other East Asian languages.

Even in English, word boundary detection is easy only for text; there are no overt boundaries (such as pauses, etc.) between the words in spoken English. The task of speech segmentation (tokenization of spoken input) is therefore much harder than the tokenization of textual input. (At least if done automatically; human beings can effortlessly and reliably segment the speech stream into words.)

sentence boundary de-  
tion

The next pre-processing step is typically *sentence boundary detection*, i.e., the task of identifying where sentences start and end in a text. As a first approximation, we can define a sentence as a string of words ending in a period, question mark or exclamation mark. This definition, however, is correct in only about 90% of the cases, i.e., one in ten sentences is identified incorrectly, which is an alarmingly high number.

**Example II.9:** Problematic cases for sentence boundary detection include the following:

- Dr. Foster went to Glasgow.
- He said “rubbish!”.
- He lost cash on lastminute.com.

△

A more accurate algorithm for sentence boundary detection therefore has to take into account periods at the end of abbreviations and multiple punctuation marks, and quotation marks.

**Example II.10:** Here is an example for a more sophisticated sentence boundary detection algorithm:

- Hypothesize a sentence boundary after: . ? !
- If a quotation mark follows the boundary, move the boundary past the quotation mark.
- Disqualify a period if:
  - it is preceded by a known abbreviation that is not usually sentence final, but is usually followed by a capitalized proper name: *Prof.* or *vs.*;
  - it is preceded by a known abbreviation and not followed by an uppercase word: *etc.* or *Jr.*
- Disqualify a boundary with a ? or ! if it is followed by a lowercase letter.
- Regard other hypothesized sentence boundaries all correct.

△

Note that this algorithm is not only more complicated, but it also requires additional resources, such as a dictionary of abbreviations.

```

<head type=MAIN>
<s n="233"><w NN2>Inspectors <w PRF>of <w NN2>schools <c PUQ>&#34;
<w AV0>poorly <w VVN>equipped <w PRP>for <w NN1>curriculum
<c PUQ>&#34;
</head>
<head type=BYLINE>
<s n="234"><w PRP>By <w NP0>PETER <w NP0>WILBY
</head>
<p>
<s n="235"><w NN1>SCHOOL <w NN2>INSPECTORS <w VVN>employed
<w PRP>by <w AJ0>local <w NN2>authorities <w VBB>are <w AV0>poorly
<w VVN>equipped <w PRP>for <w DPS>their <w AJ0-NN1>future
<w NN1>role <w PRF>of <w VVG>monitoring <w AT0>the <w AJ0>national
<w NN1>curriculum<c PUN>, <w AT0>a <w NN1>report <w PRP>from
<w AT0> the <w NN1>Audit <w NN1>Commission <w PRP>for
<w NN1-AJ0>Local <w NN2>Authorities <w VVZ>says <w AV0>today
<c PUN>.
</p>

```

Figure II.7: Extract of the BNC marked up in XML

### II.2.3 Markup Languages

*markup language*

*metadata*

*XML*

If we want to add additional information to a pre-processed corpus, then we need a *markup language* to do this. A markup language is basically a means of keeping different types of information in a corpus apart. In particular, it can be used to separate data and *metadata* (i.e., data about the data). In the case of corpora, the data would be words and sentences in the corpus, while the metadata would be data describing the words and sentences (e.g., indicating whether a word is a verb or an adjective).

The most commonly used markup language is *XML* (Extensible Markup Language). XML and metadata have already been discussed from a database perspective in Section I. Other, closely related markup languages are SGML (Standard Generalized Markup Language) and HTML (Hypertext Markup Language), the language that is used to mark up the formatting of web pages.

**Example II.11:** The concept of markup languages and metadata is best explained by way of an example. Figure II.7 contains a tiny section of the BNC marked up in XML.<sup>1</sup> △

*Entity tags*

The basic concepts of XML and related markup languages can be explained with respect to Figure II.7. First, we have to distinguish between entity tags and markup tags. *Entity tags* denote elements of the text, such as the `&#34;` which denotes an opening quotation mark. Entity tags have the purpose of keeping the content of

<sup>1</sup>Strictly speaking, the original BNC markup is in SGML, but the differences are not important in the present context.

the entity independent of its rendering (e.g., `&bquo;` can be rendered as " or " or ` ). Another purpose of entity tags is to encode characters that cannot easily be expressed in a standard character set (e.g., the tag `&uuml;` represents the accented character `ü` ).

*Markup tags* encode the meta data proper. Examples in Figure II.7 include the tags `<head>` , `<p>` , `<s>` , `<w>` , and `<c>` . The `<head>` separates the header from the body of the text. It takes an argument that denotes the type of header, and requires a closing tag `</head>` that indicates the end of the header. Figure II.7 includes two headers: one is the main header (the headline of this newspaper article), the other one is the byline of the article. The `<p>` indicates the beginning of a paragraph; it is followed by a `</p>` at the end of the paragraph.

The `<s>` tag indicates the beginning of a sentence; it takes as its argument a sentence number. The `<w>` tag marks the beginning of a word, and takes as its argument the *part of speech* (POS) of the word. The part of speech of a word is its grammatical category. For example, `NN2` stands for a plural noun, `PRF` stands for a preposition, and `AV0` indicates an adverb. The `<c>` tags indicates punctuation in the same way; its argument denotes the types of punctuation, e.g., `PUQ` mean quotation mark. *part of speech*

Part of speech annotation is only one example for linguistic annotation that can be applied to corpora. We will discuss this topic in more detail in the next section.

## II.2.4 Corpus Annotation

Annotation adds information to a corpus that is not explicitly there, and thus increases the utility of the corpus. The most common type of annotation is probably part of speech tags, but other types of annotation include syntactic structure (i.e., information about the grammatical makeup of sentences).

Before a corpus can be annotated, an *annotation scheme* has to be developed, typically consisting of a *tag set* and *annotation guidelines* . The tag set is an inventory of labels with which the entities in the corpus are to be marked up. The annotation guidelines tell the annotators (typically linguistically trained native speakers) how the tag set is to be applied. The guidelines ensure that the tag set is used consistently, even if more than one annotator carries out the annotation, and even in ambiguous or difficult cases. Only in very simple cases can corpus annotation succeed without extensive, explicit guidelines. *annotation scheme*  
*tag set*  
*annotation guidelines*

For example, a number of standard tag sets exist for part of speech tagging of English text. They include the Penn tag set, which is used for the Penn Treebank and consists of 45 tags. The CLAWS tag set includes 62 tags and has been used for POS annotating the BNC. It was later extended to 132 tags in the CLAWS2 tag set. Another classic tag set is the Brown tag set (87 tags), which has been used to POS

Category	Examples	CLAWS	Brown	Penn
Adjective	happy, bad	AJ0	JJ	JJ
Adverb	often, badly	PNI	CD	CD
Determiner	this, each	DT0	DT	DT
Noun	aircraft, data	NN0	NN	NN
Noun singular	woman, book	NN1	NN	NN
Noun plural	women, books	NN2	NN	NN
Noun proper singular	London, Michael	NP0	NP	NNP
Noun proper plural	Australians, Methodists	NP0	NPS	NNPS

Table II.6: Examples of part of speech tagsets for English

annotate the Brown corpus.

**Example II.12:** Table II.6 contains excerpts of the CLAWS, Brown, and Penn tag sets. △

One might wonder if POS tagging is a task that is hard to automate. Would it not be enough to simply look up the POS of every word in a dictionary? Unfortunately, many words in English exhibit *POS ambiguity*, i.e., they can have more than one part of speech. Consider the following example:

(1) Time flies like an arrow.

Here, *time* can be a singular noun or a verb, *flies* can be a plural noun or a verb, and *like an* can be a singular noun, a verb, or even a preposition. The correct part of speech of a given word only becomes clear only in the context of the other words of the sentence. Note that the POS ambiguity means that the sentence in (1) is highly ambiguous, it can be assigned  $2 \times 2 \times 3 = 12$  different POS sequences (assuming *an* and *arrow* are not ambiguous).

While many words can have more than one POS, most words appear most of the time in one POS. This means that we can perform *automatic POS annotation* of a text by simply assigning each word in the text its most common part of speech (we need manually annotated training data to work out the POS frequencies). This simple approach works remarkably well, and results in an accuracy of around 90%. However, this means that we still get about one word per sentence wrong (assuming that a sentence is on average ten words long). More sophisticated automatic POS tagger take the context into account in which a give word occurs to guess the correct part of speech. State-of-the-art performance is 96–98% accuracy for English.

Current POS taggers often use Hidden Markov Models (HMMs; see Informatics 1A) as the underlying technology. The POS tagging problem can be decomposed as follows: The sequence of words to be tagged is corresponds to the output sequence generated by the HMM. The tags to be assigned corresponds to the sequence of

*POS ambiguity*

*automatic POS annotation*

states that generates the output sequence. Then the problem of POS tagging reduces to finding the most probable state sequence for a given output sequence.

**Example II.13:** Here is an example of what an automatic part of speech tagger does. As an input it would take a sequence of words such as:

- (2) Our enemies are innovative and resourceful, and so are we. They never stop thinking about new ways to harm our country and our people, and neither do we.

The output would look like this (we are simply separating the POS tag from the word by a slash to save space, instead of using XML notation):

- (3) Our/PRP\$ enemies/NNS are/VBP innovative/JJ and/CC resourceful/JJ ,/, and/CC so/RB are/VB we/PRP ./ . They/PRP never/RB stop/VB thinking/VBG about/IN new/JJ ways/NNS to/TO harm/VB our/PRP\$ country/NN and/CC our/PRP\$ people/NN, and/CC neither/DT do/VB we/PRP ./ .

△

Part of speech annotation is useful for a wide range of tasks; for example lexicographers find POS tags useful when they compile lists of words (and their frequencies) for dictionaries, or when they monitor the development of new words that constantly enter the language.

POS information refers to the word level (each word in the sentence is assigned a part of speech). Other types of annotation, however, refer to the sentence level. The best example for this is *syntactic annotation*, which means that information about the structure of sentences is added to a corpus. Knowing the syntactic structure of a sentence is important as it is a prerequisite for computing its meaning, i.e., to figure out how does what to whom. The syntax of a sentence indicates which parts of a sentence belong together. For example, a verb and its objects are grouped together in a verb phrase (VP), and a noun and the words that belong to it (adjectives, determiners) are group together in a noun phrase (NP). A preposition and the NP that belongs to it form a prepositional phrase (PP). Verb phrases, noun phrases, and prepositional phrase can then combine into a sentence (S). The syntax of a sentence is typically represented as a tree that indicates how words are grouped into phrase.

*syntactic annotation*

**Example II.14:** Figure II.8 contains an example of a syntactically annotated sentence. This sentence is taken from the Penn Treebank (Marcus et al., 1993), a standard syntactically annotated corpus for English. A fragment of the XML representation of such a syntactic tree is given in Figure II.9. △

Again, syntactic annotation is a task that can be automated. Current syntactic annotation algorithms achieve an accuracy of around 90% for English (where accuracy

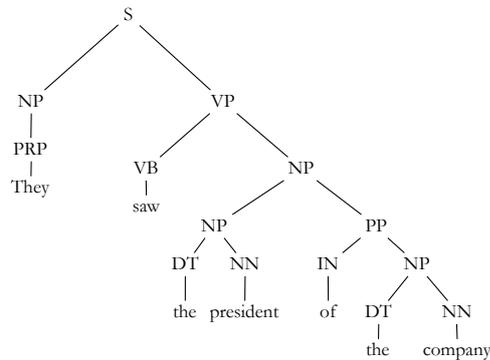


Figure II.8: Example for a syntactically annotated sentence from the Penn Treebank

```

<s>
  <np><w PRP>They</np>
  <vp><w VB>saw
    <np>
      <np><w DT>the <w NN>president</np>
      <pp><w NN>of
        <np><w DT>the <w NN>company</np>
      </pp>
    </np>
  </vp>
</s>

```

Figure II.9: XML version of the Penn Treebank tree

is computed on a phrase-by-phrase basis).

Most corpora are based on edited text or scripted speech, i.e., on material that is fairly free of noise and errors. However, most real-world communication takes place in the form of unscripted dialog. Attempts have been made to collect corpora of unscripted dialog, so as to make this type of data amenable to scientific study. A prominent example of such a corpus is the *MapTask Corpus*, developed at the Human Communication Research Centre at the University of Edinburgh (Anderson et al., 1991). This corpus consists of speech recorded from pairs of participants who collaborate in solving a route finding task. Each of the participants has a map which the other one cannot see. The instruction giver (IG) has a route marked on map. The instruction follower (IF) has no route on his or her map. The task is to reproduce IG's route on the IF's map.

The resulting speech is of course much less clean than newspaper text. It contains hesitations, false starts, slips of the tongue, and cross talk (when the two speakers

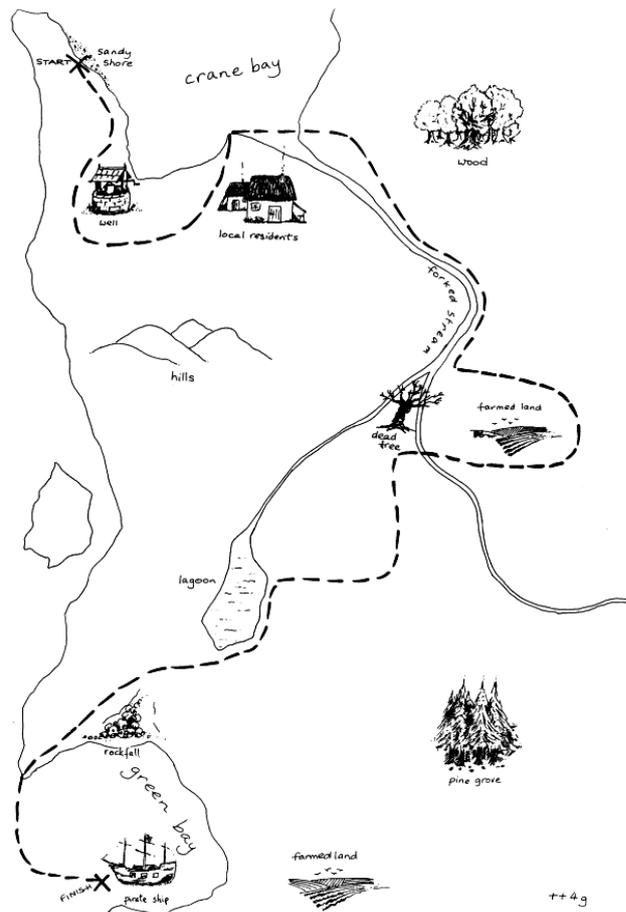


Figure II.10: Sample map from the MapTask corpus (instruction giver)

talk at the same time). The designers of the MapTask corpus have developed an annotation scheme that takes all of this into account, and annotates *dialog structure* : *dialog structure* speaker turns are marked up, the purpose of a given turn is labeled, etc.

**Example II.15:** Figures II.10 and II.11 depict a set of maps from the MapTask corpus; Figure II.12 contains a sample dialog. △

### II.3 Querying Corpora

In the previous section, we have seen how corpus data can be acquired systematically, pre-processed, and annotated with information such as parts of speech. The next step is to do something useful with the marked-up data, i.e., to use it to find linguistically interesting information, or to extract statistics that are useful for building NLP applications. This is what the present section is about: we will deal with concor-

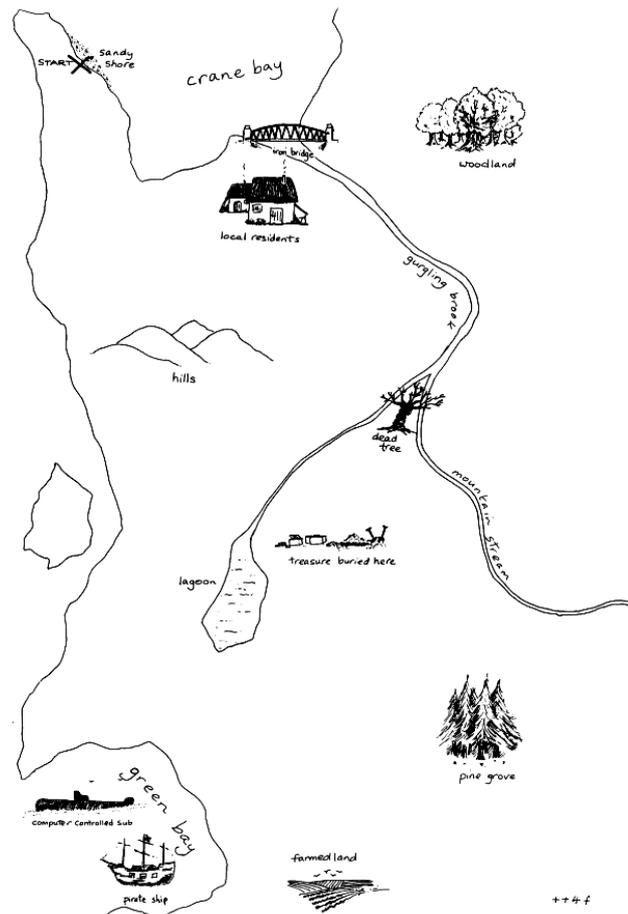


Figure II.11: Sample map from the MapTask corpus (instruction follower)

dances, regular expression queries, and discuss methods for discovering collocations, i.e., sequences of words that tend to occur together.

### II.3.1 Concordances

In Section II.1.2, we have already seen that it is sometimes useful to extract all occurrences of a given word from a corpus. If we do this, we typically want to see the word in its original context. An occurrence of a word within its context is called a *concordance*. Concordance programs allow us to specify a word, a set of words, a part-of-speech tag, or some other keyword, and then return a concordance for each corpus occurrence that matches the keyword. (Section II.3.2 will deal in more detail with constructing concordances.)

*concordance*

**Example II.16:** Suppose a preliminary corpus study suggests that the usage of

1. Neil:  
Right Start from the sandy shore.

2. Chris:  
Okay.

3. Neil:  
moving down ... straight down.

4. Chris:  
How far?

5. Neil:  
Down as far as the bottom of the well.

7. Neil:  
{fg Ah}. Right, {fg|eh}. Move down, {fg|eh}, vertically  
down about a quarter of the way down the page. Move to the  
right in ... Do you have local residents?

8. Chris:  
I do.

Figure II.12: Sample dialog from the MapTask corpus

```
's cellar . Scrooge then <remembered> to have heard that ghost
, for your own sake , you <remember> what has passed between
e-quarters more , when he <remembered> , on a sudden , that the
corroborated everything , <remembered> everything , enjoyed eve
urned from them , that he <remembered> the Ghost , and became c
ht be pleasant to them to <remember> upon Christmas Day , who
its festivities ; and had <remembered> those he cared for at a
wn that they delighted to <remember> him . It was a great sur
ke ceased to vibrate , he <remembered> the prediction of old Ja
as present myself , and I <remember> to have felt quite uncom
```

Figure II.13: Concordance for the word *remember*

the word *remember* is worth looking into further, e.g., to check whether it is used transitively (with an object) as well as intransitively (without an object). A good starting point for such a study is a concordance for *remember*. The typical way of displaying such a concordance is as in Figure II.13. △

In this example, the keyword (the word *remember*) appears roughly in the middle of each line, and each line has some predetermined number of characters, i.e., there is a fixed left and right context in which the word is displayed. This way of displaying a concordance is called a *keyword in context* index, or KWIC index. *keyword in context*

A KWIC index is not the only way of displaying a concordance. You could display the sentence or paragraph the keyword occurs in, rather than having a fixed context to the left and right of the keyword. But in what follows we will concentrate

on KWIC indexes.

### II.3.2 Regular Expressions

*Corpus Query Processor*

Specialized software is typically used to construct word concordances such the one in Figure II.13. In this course, we will learn how to use one such program, the *Corpus Query Processor* (CQP), which is part of the IMS Corpus Workbench, developed at IMS, the Institute for Natural Language Processing at Stuttgart University.<sup>2</sup> At the heart of CQP is a query engine that makes it possible to search corpora efficiently by specifying regular expressions over words, parts of speech, or other markup in the corpus.

*regular expressions*

Before we deal with CQP in detail, let us recap some basic facts about *regular expressions* (introduced in Informatics 1A). Here, we will adopt a slightly different perspective by talking about a string matching a regular expression under certain conditions. A regular expression can be defined as consisting of the following elements:

*Symbol*

- *Symbol* : A symbol  $a$  in a regular expression  $R$  matches the string  $a$ .

*Sequence*

- *Sequence* : A sequence  $R_1R_2$  of two regular expressions matches the concatenation of the string matched by  $R_1$  and the string matched by  $R_2$ .

*Choice*

- *Choice* : A choice  $R_1|R_2$  of two regular expressions matches the string matched by  $R_1$  or the string matched by  $R_2$ .

*Repeat*

*Kleene star*

- *Repeat* : A regular expression  $R^*$  matches a sequence of 0 or more instances of  $R$ . The star in  $R^*$  is also referred to as the *Kleene star*.

CQP's query language makes it possible to directly specify regular expressions in search expressions. An example for a very simple CQP query is:<sup>3</sup>

(4) `[word="remember"];`

*positional attribute*

Here, `word` is a *positional attribute*, i.e., something that is marked up at a certain position in the corpus (in this case it is simply the word form). The value of the attribute is matched against the right hand side of the query (here, the string `remember`). Thus, the query in (4) will return all the instances in the corpus that match the word *remember*. The right hand side of the query can contain a regular expression. For example, the choice operator `|` can be used to match multiple word forms:

<sup>2</sup>For more information on CQP and the Corpus Workbench, see <http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/>.

<sup>3</sup>The query `[word="remember"];` can be abbreviated to just `"remember";`.

(5) `[word="remember|remembers|remembered|remembering"];`

The query in (5) therefore returns all forms of the word *remember*, resulting in an output like the one in Figure II.13. The Kleene star can also be used in CQP regular expressions, as in the following example:

(6) `[word="blaa*"];`

(6) will match the words `bla`, `blaa`, `blaaa`, etc. (Note that the `*` binds only the previous letter, not the whole expression.) CQP also offers a number of additional regular expression operators, which do not add to the expressivity of the query language, but make it easier to formulate succinct queries. These include the dot operator, which matches any character, as in:

(7) `[word="s.ng"];`

This expression matches `sing`, `sang`, `sung`, but also `szng` and `s6ng`. The list operator `[...]` matches all characters in the square brackets. We could therefore formulate the previous query more accurately as:

(8) `[word="s[iaun]g"];`

Abbreviations for subsets of the alphabet are possible, as in `[a-d]` or `[1-6]`. The Kleene star is not the only repetition operator that CQP supports. While `*` matches zero or more repetitions of an string, `+` matches one or more repetitions, and `?` matches zero or one instance of a string.

Finally, round brackets `(` and `)` can be used to group characters together, and a subsequent operator will then apply to the whole string enclosed by the round brackets. For example:

(9) `[word="[Rr]remember(s|ed|ing)?"];`

The query in (9) matches the words `remember`, `remembers`, `remembered`, and `remembering`, and the corresponding capitalized forms.

The positional attribute `word` is available in every corpus (this is simply the word form); but many corpora contain more annotation, such as the *part of speech attribute* `pos`. Assuming we have a corpus that is POS tagged with the Penn Treebank tag set, *part of speech attribute* we can formulate queries like:

(10) `[pos="NN.*"];`

This returns all nouns in the corpus: the expression `NN.*` matches the tag `NN` for regular nouns and the tags `NNP` and `NNPS` for singular and plural proper nouns, see Table II.6. Several regular expressions can be combined in a single query using the

```

now , notwithstanding the <hot tea> they had given me before
.' ' Shall I put a little <more tea> in the pot afore I go ,
o moisten a box-full with <cold tea> , stir it up on a piece
tween eating , drinking , <hot tea> , devilled grill , muffi
e , handed round a little <stronger tea> . The harp was there ; t
e so repentant over their <early tea> , at home , that by eigh
rs. Sparsit took a little <more tea> ; and , as she bent her
s illness ! Dry toast and <warm tea> offered him every night
of robing , after which , <strong tea> and brandy were administ
rsty . You may give him a <little tea> , ma'am , and some dry t

```

Figure II.14: Concordance for the query `[pos="JJ.*"] [word="tea"];`

Boolean operators `&` (and), `|` (or), and `!` (not). For example, we can formulate a query that returns all word forms that start with `like` and are not tagged as a noun:

```
(11) [(word="like.*") & (pos!="NN.*")];
```

Finally, queries can refer to sequences of word, simply by concatenating several `[...]` expressions:

```
(12) [pos="JJ.*"] [word="tea"];
```

This query will match all instances of the word *tea* that are preceded by an adjective (coded as `JJ` in the Penn Treebank tag set). A sample output for this query is listed in Figure II.14.

### II.3.3 Collocations

#### Collocations

*Collocations* are sequences of words that occur together. For example the noun *amok* is almost always preceded by the verb *run*, hence *run amok* form a strong collocation in English. Another example for a collocation is *strong tea*; the noun *tea* does not appear to go well with adjectives such as *powerful*, even though *powerful tea* and *strong tea* have a very similar meaning. Hence *strong tea* is also classified as a collocation in English. Finally, phrasal verbs are collocations, too. For example, the verb *make* can occur with particles such as *up*, *off*, *out*, but not with the particle *in*. The verb *take*, on the other hand, can occur with *in* as in *He didn't take in the news*.

Given a large corpus, we can try to automatically identify collocations by searching the corpus for suitable occurrences. CQP can be used for this, for example the query in (12) should return the adjectival collocations of *tea*. The output of this query in Figure II.14 shows that *strong tea* is indeed a word combination that occurs in the corpus, while *powerful tea* fails to occur. However, there are other occurrences such as *more tea* which do not seem to form collocations. Intuitively, in order to make a word combination a collocation it has to occur frequently. For example, we would expect *strong tea* to have a higher frequency than *powerful tea* in a corpus.

strong	,	52	powerful	,	5
	and	31		effect	3
	enough	16		sight	3
	.	16		enough	3
	in	15		mind	3
	man	14		for	3
	emphasis	11		and	3
	desire	10		with	3
	upon	10		enchanter	2
	interest	8		displeasure	2
	a	8		motives	2
	as	8		impulse	2
	inclination	7		struggle	2
	tide	7		grasp	2
	beer	7		friends	2

Table II.7: Frequency distributions for all bigrams with *strong* and *powerful* as the first word

CQP can be used to compute frequency distributions. For instance, we can use it to obtain the *bigram frequencies* for all words that occur with *strong* and *powerful*. To do this, we first define two named queries, Q1 and Q2 (note that [ ] matches any word):

```
(13) Q1 = [word="strong"] [ ];
      Q2 = [word="powerful"] [ ];
```

Then we use the named queries together with CQP's `group` command to obtain frequency distributions:

```
(14) group Q1 matchend word by match word;
      group Q2 matchend word by match word;
```

These queries tell CQP to group together the values of the `word` attribute at the position `matchend` (the last word in the match, here: [ ]) and sort the result by `word` at position `match`, i.e., the beginning of the match (here: [word="strong"] or [word="powerful"]). The resulting bigram distribution is listed in Table II.7. We have only included the 15 most frequent bigrams in this table, in keeping with our hypothesis that a word combination has to be highly frequent in order to qualify as a collocation. Valid collocations should therefore be at the top of this table.

Unfortunately, neither *strong tea* nor *powerful tea* are frequent enough to make it into Table II.7 (this is an instance of sparse data, see Section II.1.3). However, we

managed to discover some interesting potential collocations for *strong*, such as *strong desire*, *strong inclination*, and *strong beer* (the later is similar to *strong tea*). Also for *powerful*, we find some potential collocations, such as *powerful effect*, *powerful motives*, and *powerful struggle*.

### II.3.4 Statistical Tests

In the previous section, we tried to use bigram frequencies to identify collocations involving the adjectives *strong* and *powerful*, and the result was displayed in Table II.7. However, there are a lots of bigrams in this table that are definitely not collocations, such as *strong and*, *strong enough*, *strong upon*, *powerful for*, and *powerful and* (in addition to bigrams involving commas and periods, which are also irrelevant). This means that pure bigram frequency is not sufficient to identify collocations; we need some way of filtering out “noise”, i.e., bigrams that are highly frequent, but do not qualify as collocations.

The reason for this problem is that some words like *for* or *and* or punctuation marks like comma or period are highly frequent in their own right. And this means that there is a high likelihood that they occur with our target words *strong* and *powerful*, resulting in highly frequent bigrams that show up in Table II.7. We therefore need a way of distinguishing collocations from words that co-occur a lot just by chance (such as *powerful for*). The answer to this problem is *hypothesis testing* using statistical techniques. We formulate a *null hypothesis*  $H_0$  that there is no association between the two words are interested in. Then we can compute the probability  $p$  that we see the two words together in a corpus, given that  $H_0$  is true. If  $p$  is sufficiently low, then we can reject  $H_0$ , and assume that the two words in question really form a valid collocation. Typically, the null hypothesis can be rejected if  $p < .05$  (sometimes the stricter criterion of  $p < .01$  is used). This threshold is referred to as the *significance level* of a statistical test.

Modern statistics makes available a large number of tests, many of which are applicable to the problem of discovering collocations. In what follows, we will only deal with one test that is particularly simple and has many applications beyond collocation discovery: the  $\chi^2$  (*chi-square*) test. This test compares  $n$  probability distributions, each with  $m$  values, and yields a significant results if the distributions are reliably different. The input for the  $\chi^2$  test is typically displayed as an  $n \times m$  *contingency table*.

**Example II.17:** For example, assume that we want to compare the performance of boys and girls in a given exam, which can yield the marks A, B, C, and D. We can therefore tabulate the data in a  $4 \times 2$  contingency table, with the marks on the x-axis and the distribution (boys or girls) on the y-axis. A fictitious example data set is displayed in Table II.8. △

*hypothesis testing*  
*null hypothesis*

*significance level*

$\chi^2$  (*chi-square*) test

*contingency table*

$O_{ij}$	A	B	C	D	$\sum_i O_{ij}$
Boys	3	23	43	10	79
Girls	6	34	31	4	75
$\sum_j O_{ij}$	9	57	74	14	154

Table II.8: Contingency table for fictitious exam data

$E_{ij}$	A	B	C	D
Boys	4.62	29.24	37.96	7.18
Girls	4.38	27.76	36.04	6.82

Table II.9: Expected frequencies for the exam data

The  $\chi^2$  statistic is now computed by comparing the *observed frequencies* in the contingency table with the *expected frequencies*. The expected frequencies are the frequencies that would be expected if the  $H_0$  hypothesis was true, i.e., if there was no difference in the distributions tabulated in the contingency table. Mathematically, this can be formulated as follows:

$$\chi^2 = \sum_{i,j} \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \quad (\text{II.1})$$

Here,  $i$  ranges over the rows of the contingency table, and  $j$  ranges over its columns.  $O_{ij}$  is the observed frequency for cell  $(i, j)$ , while  $E_{ij}$  is the expected frequency for cell  $(i, j)$ . The expected frequencies can be calculated as follows:

$$E_{ij} = \frac{\sum_j O_{ij} \sum_i O_{ij}}{N} \quad (\text{II.2})$$

Here,  $N$  is the overall number of observations. The sums  $\sum_j O_{ij}$  and  $\sum_i O_{ij}$  are also referred to as the *marginals* of the contingency table.

**Example II.18:** We use formula (II.2) to calculate the expected frequencies for the data in Table II.8. The result is tabulated in Table II.9. Using formula (II.1), we can then calculate the  $\chi^2$  value for this contingency table:

$$\begin{aligned} \chi^2 &= \sum_{i=1\dots 4, j=1\dots 2} \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \\ &= \frac{(3-4.62)^2}{4.62} + \frac{(23-29.24)^2}{29.24} + \frac{(43-37.96)^2}{37.96} + \frac{(10-7.18)^2}{7.18} + \\ &\quad \frac{(6-4.38)^2}{4.38} + \frac{(34-27.76)^2}{27.76} + \frac{(31-36.04)^2}{36.04} + \frac{(4-6.82)^2}{6.82} \\ &= .57 + 1.33 + .67 + 1.11 + .60 + 1.40 + .70 + 1.17 \\ &= 7.55 \end{aligned}$$

△

$O_{ij}$	$w_1$	$\neg w_1$
$w_2$	$f(w_1, w_2)$	$f(\neg w_1, w_2)$
$\neg w_2$	$f(w_1, \neg w_2)$	$f(\neg w_1, \neg w_2)$

Table II.10: Schematic contingency table for testing if a bigram occurs significantly more than chance

*degrees of freedom*

The  $\chi^2$  value computed using formula (II.1) then needs to be compared against *critical values*, i.e., values that  $\chi^2$  needs to exceed in order for the test to be significant. These critical values depend on the *degrees of freedom* of the test. Intuitively, the degrees of freedom correspond to the number of dimensions along which a test can vary. For the  $\chi^2$  test, the degrees of freedom ( $df$ ) are computed as follows:

*critical values*

$$df = (n - 1)(m - 1) \quad (\text{II.3})$$

Here,  $n$  is the number of rows in the contingency table, and  $m$  is the number of columns.

**Example II.19:** For our example in Tables II.8 and II.9, the degrees of freedom are:  $df = (n - 1)(m - 1) = 3 \times 1 = 3$ . The corresponding critical value for a significance level of  $p < .05$  is 7.82. The value  $\chi^2(3) = 7.55$  for our example data is below the critical value, which means that we can conclude that the exam performance of boys and girls is not significantly different.<sup>4</sup>  $\triangle$

*collocation filtering*

The  $\chi^2$  test can also be applied for *collocation filtering*, i.e., the task of identifying bigrams that are valid collocations. For each bigram  $w_1w_2$  we want to investigate, we compile a contingency table that tabulates the number of times  $w_1$  and  $w_2$  occur together, and compares it with the number of times  $w_1$  and  $w_2$  occur separately. This yields the contingency table in Table II.10, where  $f(w_1, w_2)$  refers to the frequency of  $w_1$  and  $w_2$  occurring together,  $f(w_1, \neg w_2)$  refers to the frequency of  $w_1$  occurring with a word other than  $w_2$ , etc. Applying the  $\chi^2$  test to this contingency table tests the hypothesis that  $w_1$  and  $w_2$  occur together more often than chance: the observed frequencies in the  $\chi^2$  test are the frequencies with which the words occur in the corpus, and the expected frequencies are the frequencies which we would expect the words to occur if their distribution was random.

**Example II.20:** Let us return to the collocations of *strong* in Table II.7. As an example take the two bigrams *strong desire* and *strong upon*, both of which occur 10 times in this corpus. We need to look up the number of occurrences for *strong*, *desire*, and *upon* (which is easily done using CQP), and we need to know the overall number

<sup>4</sup>In practice, the  $\chi^2$  is not suitable for contingency tables that contain frequencies smaller than 5, as it tends to overestimate the importance of rare events. Alternative tests can be applied that do not suffer from this shortcoming, such as the  $G^2$  test.

$O_{ij}$	<i>strong</i>	$\neg$ <i>strong</i>	$O_{ij}$	<i>strong</i>	$\neg$ <i>upon</i>
<i>desire</i>	10	214	<i>upon</i>	10	7107
$\neg$ <i>desire</i>	655	3407085	$\neg$ <i>upon</i>	655	3407085
	$\chi^2(1) = 46684423$			$\chi^2(1) = 6235$	

Table II.11: Contingency tables for the bigrams *strong desire* and *strong upon*

of words in the corpus. Then we can compute contingency tables for these words, which are displayed in Table II.11

Using formulas (II.1) and (II.2), we can now compute the  $\chi^2$  values for these two bigrams, which are also tabulated in Table II.11. The critical value for  $df = 1$  is 3.84, and both  $\chi^2$  values far exceed this. However, the valid collocation *strong desire* has a much higher  $\chi^2$  value than the invalid collocation *strong upon*, which indicates that the  $\chi^2$  can successfully applied as a filter: We simply remove all bigrams from Table II.7 whose  $\chi^2$  value falls below a certain threshold.  $\triangle$

## II.4 Information Retrieval

In the previous section, we discussed how linguistic information can be extracted from corpus data, using corpus query engines that support regular expression search over words. We also saw examples for the use of simple static tests to filter the information gleaned from corpora.

In the present section, we will broaden the scope of our study of corpora as semi-structured data. We will move from techniques for the retrieval of words or sentences to techniques that make it possible to retrieve whole documents from a document collection, based on a query posed by a user. This technology is called *information retrieval* (IR) and plays an increasingly important role in informatics.

*information retrieval*

### II.4.1 Information Retrieval Systems

We already briefly discussed information retrieval in Section II.1.4 as one of the applications of techniques for processing semi-structured data. Let us now define the IR task in a bit more detail.

The classical problem in IR is the *ad hoc retrieval problem*: Given a query, find the documents that are relevant to this query. Typically, the following assumptions are made in this scenario:

*ad hoc retrieval problem*

- We are searching a large, static document collection.
- The user accessing the document collection has an information need, which he or she formulates in terms of a query (typically in the form of keywords).
- The task is to find all and only the documents that relevant to the user's query.

Examples of widely used IR systems are web search engines, of which Google is a prominent example (see Figure II.6). In the case of search engines, the document collection to be searched is a large collection of web pages. The information need that the user has it typically to find pages on a particular topic, and he or she formulates a keyword-based query to express this information need. The search engine's task is to return a ranked list of web pages that match the user's queries.

*examples of IR systems*

Other *examples of IR systems* include bibliographic information systems such as the ones used by researchers to find scientific articles on a given topic; typically these systems search a collection of documents that list the titles, authors, and abstracts of scientific papers. Journalists use similar systems to retrieve newspaper articles from large, electronic newspaper archives.

In what follows, we will explore a number core techniques that will allow us to build a simple IR system. We will address the following scientific problems that are addressed by IR technology:

*Query type*

- *Query type* : How should we formulate the queries to an IR system?

*Indexing*

- *Indexing* : What is the best way of representing the documents searched by the system?

*Retrieval model*

- *Retrieval model* : How does the system find the best-matching document? How do we make sure that this happens efficiently?

*Output presentation*

- *Output presentation* : What is the best way of presented the results of the search? The results could be output as an unsorted list, a ranked list, or clusters of documents.

*Evaluation*

- *Evaluation* : How do we measure the performance of the system, i.e., how do we find out if the system does what it is supposed to do?

## II.4.2 Indexing

The purpose of an IR system is to search a document collection, and it has to have a representation of the documents in the collection that facilitates retrieval. This task is called *indexing* . Typically, indexing means finding *terms* , i.e., words or phrases that describe the documents well and thus make it possible to match the terms against a query to achieve successful retrieval. There are several ways of generating the terms for a document. *Manual indexing* means that human annotators manually choose a set of terms for a given document. Manual indexing typically employs large vocabularies, containing several thousand of index terms. Examples of such vocabularies are the Library of Congress Subject Headings, which are often used to index the holdings of scientific document collections, such as the books in a university library. Another

*indexing terms*

*Manual indexing*

Medical Subject Headings (MeSH)	
Eye Diseases	C11
Asthenopia	C11.93
Conjunctival Diseases	C11.187
Conjunctival Neoplasms	C11.187.169
Conjunctivitis	C11.187.183
Conjunctivitis, Allergic	C11.187.183.200
Conjunctivitis, Bacterial	C11.187.183.220
Conjunctivitis, Inclusion	C11.187.183.220.250
Ophthalmia Neonatorum	C11.187.183.220.538
Trachoma	C11.187.183.220.889
Conjunctivitis, Viral	C11.187.183.240
Conjunctivitis, Acute Hemorrhagic	C11.187.183.240.216
Keratoconjunctivitis	C11.187.183.394

Table II.12: Example for a vocabulary for manual indexing: Extract from the Medical Subject Headings

example is the classification of subfields of computer science developed by the Association for Computing Machinery (ACM), which is often used to index research articles appearing in scientific journals.

The advantages of manual indexing is that it works well for closed document collections (such as the books in a library), and it generally achieves high precision, because the manual index has been carefully selected to contain the correct terms. The disadvantages are that annotators need to be trained to achieve consistency across documents, and that many document collections are dynamic, which means that the indexing schemes changes constantly (examples for dynamic document collections are the web or a collection of newspaper articles). It is unlikely that a fixed vocabulary of terms is suitable for these dynamic IR tasks.

**Example II.21:** Table II.12 contains an example for a vocabulary used for manual indexing, viz., an extract from the Medical Subject Headings, which are used to index medical literature. Table II.13 contains another example, the ACM Computing Classification System used to index literature in computer science and related fields. Note that both vocabularies are organized in a hierarchical way (they contain classes and subclasses), i.e., they are *taxonomies*. △ *taxonomies*

Many IR systems perform *automatic indexing*, i.e., they automatically extract relevant words or phrases from the documents in the collection, instead of relying on manual annotation. Typically, automatic indexing also includes term weighting: certain terms are considered more important than others, for example based on their frequency. Another important technique is *term manipulation*, which includes pro- *automatic indexing*  
*term manipulation*

<b>Computing Classification System (1998)</b>	
B	Hardware
B.3	Memory structures
B.3.0	General
B.3.1	Semiconductor Memories (NEW) (was B.7.1)
	Dynamic memory (DRAM) (NEW)
	Read-only memory (ROM) (NEW)
	Static memory (SRAM) (NEW)
B.3.2	Design Styles (was D.4.2)
	Associative memories
	Cache memories
	Interleaved memories
	Mass storage (e.g., magnetic, optical, RAID)
	Primary memory
	Sequential-access memory

Table II.13: Example for a vocabulary for manual indexing: Extract from the ACM Computing Classification System

cesses that map certain words or phrases on the same term.

Automatic indexing does not use a predefined, artificially created set of index terms. Instead, natural language is used as the indexing language, under the assumption that the words in the document provide sufficient information about its content to allow successful retrieval. This has the advantage that the vocabulary of the index can change dynamically as the document collections change.

*inverted index*

An automatic index is typically implemented as an *inverted index*. This is a data structure that contains all the words that occur in the document collection, and for each word it specifies which document the word occurs in. Given an inverted index and a query, it's straightforward to retrieve the documents in the collection that contain the words in the query: we simply go through the inverted index and return the document listed there for a given query word. Searching the inverted index for a keyword is much faster than searching through the whole document collection, hence using the index speeds up the retrieval process considerably.

*position information*

An inverted index can also be enriched with *position information*: for each word, the index also lists which document it occurs in, and where in the document it can be found. This makes it possible for the IR system to specifically return part of a document in response to a query, or to highlight the query terms in the document.

**Example II.22:** Figure II.15 illustrates how an inverted index works. We start with a collection of documents (in this case, we only have six documents with one sentence each). Then we list all the words in the documents, and for each word

Document	Text
1	Pease porridge hot, pease porridge cold
2	Pease porridge in the pot
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot
6	Nine days old



Number	Text	Documents
1	cold	1, 4
2	days	3, 6
3	hot	1, 4
4	in	2, 5
5	it	4, 5
6	like	4, 5
7	nine	3, 6
8	old	3, 6
9	pease	1, 2
10	porridge	1, 2
11	pot	2, 5
12	some	4, 5
13	the	2, 5

Figure II.15: Creation of an inverted index

we index the document it occurs in the form of a document number. Figure II.16 illustrates the same process for an inverted index with position information.  $\triangle$

Often, index creation does not really include all the words in the documents. Typically *stop words* are excluded, i.e., words that are important for the structure of a sentence, but carry very little semantic information. Examples include determiners like *a, the*, prepositions like *of, with, at*, and pronouns like *he, her, our*. These words are highly frequent and tend to occur in all documents, so they add very little to the retrieval process.

### II.4.3 Vector Space Models

A very simple IR system could just use an inverted index, i.e., for a given query it could simply return all the documents that contain all the words (or any of the words) in the query. For a large document collection, and a sufficiently general query, this typically means that the system returns a large number of relevant documents, and the user has to trawl through thousands of hits. Modern IR system therefore provide a *document ranking* by relevance. A number of ranking methods have been developed, but in this section we will focus on one particularly intuitive approach, the *vector*

Document	Text
1	Pease porridge hot, pease porridge cold
2	Pease porridge in the pot
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot
6	Nine days old

Number	Text	(Document; Word)
1	cold	(1; 6), (4; 8)
2	days	(3; 2), (6; 2)
3	hot	(1; 3), (4; 4)
4	in	(2; 3), (5; 4)
5	it	(4; 3, 7), (5; 3)
6	like	(4; 2, 6), (5; 2)
7	nine	(3; 1), (6; 1)
8	old	(3; 3), (6; 3)
9	pease	(1; 1, 4), (2; 1)
10	porridge	(1; 2, 5), (2; 2)
11	pot	(2; 5), (5; 6)
12	some	(4; 1, 5), (5; 1)
13	the	(2; 4), (5; 5)

Figure II.16: Creation of an inverted index with position information

*vector space model**space model* of document ranking.

The core idea is that documents are treated as points in high-dimensional vector space, based on the words they contain. Queries are also represented in vector space. Then the IR system can select the documents with the highest document-query similarity and present these to the user (instead of just returning an unordered set of documents).

**Example II.23:** This can be illustrated schematically using Figure II.17. Assume the query is *car insurance*. Then we have a simple two dimensional vector space, with the terms *car* and *insurance* as the two dimensions (the x-axis and the y-axis). The vector  $q$  represents the query, and the vectors  $d_1$ ,  $d_2$ , and  $d_3$  represent three documents that contain the two terms. Document  $d_2$  is closest to  $q$  in vector space, so this would be the document returned by the system as the most relevant one.  $\triangle$

*insurance*

The document vectors for the vector space model can be created by tabulating the frequencies of the terms in a given document. This is an extension of the idea behind the inverted index: now we are using a table as our data structure, in which the columns represent the terms, and the rows represent the documents. In each cell of the matrix we specify how often a given term occurs in the document in question. The query is represented in the same way. The vectors required for comparing the

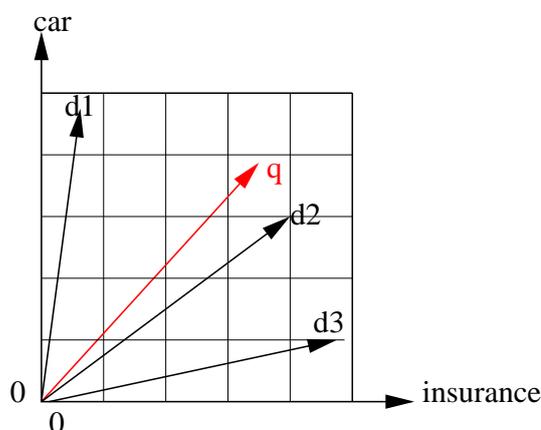


Figure II.17: Schematic illustration of the vector space model of IR

	Term <sub>1</sub>	Term <sub>2</sub>	Term <sub>3</sub>	...	Term <sub>n</sub>
Doc <sub>1</sub>	14	6	1	...	0
Doc <sub>2</sub>	0	1	3	...	1
Doc <sub>3</sub>	0	1	0	...	2
...	...	...	...	...	...
Doc <sub>N</sub>	4	7	0	...	5
Q	0	1	0	...	1

Table II.14: Example for the creation of the vectors for the vector space model

similarity of a query and the documents in the collection can be extracted straightforwardly from this data structure: they are simply the rows in the table.

**Example II.24:** Table II.14 is an example of a table containing the vectors for a document collection, with the documents Doc<sub>1</sub> . . . Doc<sub>N</sub> as rows and the terms Term<sub>1</sub> . . . Term<sub>N</sub> as columns. The query Q is tabulated in the last row.  $\triangle$

More formally, each document in the document collection is represented as a vector of  $n$  values, the term frequencies:

$$\vec{x} = (x_1, x_2, \dots, x_n) \quad (\text{II.4})$$

Now we need a way of measuring the similarity between the document vector  $\vec{x}$  and the query vector. A number of *vector similarity measures* have been proposed for this purpose, but here we will focus on the *cosine*, which measures the angle between two vectors  $\vec{x}$  and  $\vec{y}$ . This similarity measure has been applied successfully for a range of IR tasks.

The cosine between two vectors  $\vec{x}$  and  $\vec{y}$  is defined as:

$$\cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{|\vec{x}||\vec{y}|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \quad (\text{II.5})$$

*Term weighting*

As mentioned in Section II.4.2, there are a number of other things that an IR system can do to increase retrieval performance. *Term weighting* means that we give more weight to certain terms. The vector space model in the way it has been introduced here implicitly performs term weighting: the vectors are constructed based on term frequencies, which means that high frequency terms are more important than low frequency terms. More complicated term weighting schemes are often applied to improve retrieval performance, as explained in Manning and Schütze (1999, Ch. 15). These weighting schemes use term frequency, document frequency, collection frequency, or a combination of these frequencies. Typically, we also need *normalization*

*normalization*

to factor out the effect document length on the vectors, as of course longer documents are more likely to contain relevant terms, just because they are longer. For term frequency, this can be achieved by dividing the term frequency for a given document by the number of terms in that document. This corresponds to using relative frequencies instead of absolute frequencies (see Section II.1.2).

*term manipulation*

Another important technique is *term manipulation*, i.e., processes that modify the terms that end up in the vectors. Using a stop word list is one way of eliminating irrelevant terms that would distort retrieval. Another important process is *stemming*, which removes the endings from the words, e.g., the words *Constitution*, *constitution*, *constitutions* would all be mapped on the term *constitution*.

*stemming*

#### II.4.4 Evaluation

*Evaluation*

*Evaluation* is the process of systematically measuring how well a given system (e.g., an IR system) performs by comparing its output against pre-defined criteria. Formal evaluation makes it possible to demonstrate that a system successfully achieves the task it is designed for. In particular, we need evaluation if we want to compare the performance of several systems or algorithms on the same task. In what follows, we will discuss several basic evaluation techniques using information retrieval as an example. However, these techniques are general, and applicable in many areas of informatics.

*precision and recall*

Two fundamental ways of measuring the performance of an IR system is in terms of *precision and recall*. Intuitively, precision tells us how many of the documents that the systems retrieves are correct (i.e., relevant to the query), while recall tells us how many of the relevant documents in the document collection the system managed to find. More precisely, we can define precision and recall based on the following quantities:

*True positives*

- *True positives* (TP): number of relevant documents that the system retrieved.

	Relevant	Non-relevant
Retrieved	true positives	false positives
Not retrieved	false negatives	true negatives

Table II.15: Schematic confusion matrix

- *False positives* (FP): number of non-relevant document that the system re- *False positives*  
trieved.
- *True negatives* (TN): number of non-relevant documents that the system did *True negatives*  
not retrieve.
- *False negatives* (FN): number of relevant documents that the system did not *False negatives*  
retrieve.

The quantities are typically tabulated in the form of a *confusion matrix*, schemat- *confusion matrix*  
ically depicted in Table II.15. We can now define precision as the number of relevant  
documents retrieved over the total number of documents retrieved:

$$P = \frac{TP}{TP + FP} \quad (\text{II.6})$$

And recall is the number of relevant documents retrieved over the total number  
of relevant documents:

$$R = \frac{TP}{TP + FN} \quad (\text{II.7})$$

**Example II.25:** Assume we have a document collection with 130 documents in  
total, of which 28 documents are relevant for a given query. We now want to compare  
two IR systems. System 1 retrieves 25 documents in total, of which 16 are relevant.  
This means that the system achieves 16 true positives and  $25 - 16 = 9$  false positives.  
The number of false negatives is  $28 - 16 = 12$ . We therefore get:

$$P_1 = \frac{TP_1}{TP_1 + FP_1} = \frac{16}{16 + 9} = .64 \quad R_1 = \frac{TP_1}{TP_1 + FN_1} = \frac{16}{16 + 12} = .57$$

System 2 retrieves only 15 documents, of which 12 are relevant. Hence the sys-  
tems achieves 12 true positives and 3 false positives, and 16 false negatives:

$$P_2 = \frac{TP_2}{TP_2 + FP_2} = \frac{12}{12 + 3} = .80 \quad R_2 = \frac{TP_2}{TP_2 + FN_2} = \frac{12}{12 + 16} = .43$$

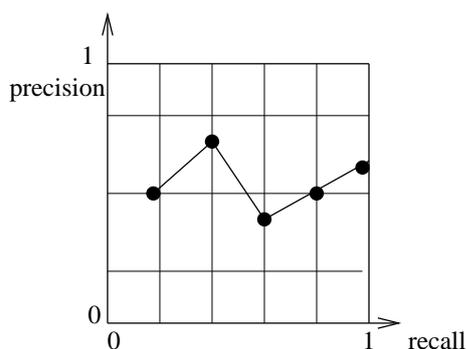


Figure II.18: Schematic precision-recall curve

This shows that system 2 beats system 1 in terms of precision, but system 1 outperforms system 2 in terms of recall.  $\triangle$

In order for a system to perform well, it has to achieve both high precision and recall: if both figures are 100%, then this means that the system has retrieved all and only relevant documents. It does not make sense to only look at one of the two figures. If a system simply returns all documents in the document collection it will achieve perfect recall (as it has returned all relevant documents), but precision will be low (as a lot of irrelevant documents have been returned, too). Conversely, if a system only returns one document, and this document is relevant, and it achieves perfect precision, but recall will be low (as most of the relevant documents were not retrieved).

*precision-recall tradeoff* More generally, a system is faced with a *precision-recall tradeoff*: It can try to optimize precision at the cost of recall, or it can try to increase recall at the cost of precision. Which of the two quantities is more important often depends on the particular application the system is designed for. In a formal evaluation of a system, different version the system are often compared with respect to this tradeoff, which can then be graphed in a precision-recall curve as in Figure II.18.

It is often useful for system evaluation to have a single measure that combines both precision and recall. A commonly used measure that achieves this is the *F-score*, which is defined as follows:

*F-score*

$$F_{\alpha} = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}} \quad (\text{II.8})$$

The parameter  $\alpha$  is a weighting factor: with a high  $\alpha$ , recall is more important, with a low  $\alpha$ , precision is more important. It is very common, however, to treat the two as equally important, i.e., to use  $\alpha = .5$ , in which case F-score is simply the

harmonic mean of P and R:

$$F_{.5} = \frac{2PR}{P + R} \quad (\text{II.9})$$

In Section II.4.3, we introduced the vector space model as a way of ranking the output of an IR system by comparing a document and a query in terms of their distance in a vector space over terms. An IR system that outputs a ranking of documents is significantly more useful than one that simply returns all documents that it considers relevant. However, the evaluation measures we have introduced so far do not take the ranking of the results into account, they apply to the whole set of results that the system retrieves.

A way of addressing this problem is to compute *precision at a cutoff*. For example, we can compute precision for the 5 or 10 most highly ranked documents. This figure is referred to as *precision at 5* and *precision at 10*. This figure captures the fact that a good system not only returns a lot of relevant documents, but also ranks them highly (here: among the top five or ten documents).

This approach can be generalized by computing *uninterpolated average precision*, which is the average of all precisions in a list of  $n$  documents: we compute precision at 1, precision at 2, etc., to precision at  $n$ , and then take the average. Two systems which both return the same number of relevant documents among the top  $n$  can still be distinguished by this measure: the system that ranks the relevant documents higher in the top  $n$  will have a higher uninterpolated average precision. (Interpolated average precision also exists, see (Manning and Schütze, 1999, Ch. 15).)

**Example II.26:** Table II.16 contains an example for ranking evaluations. We compare three rankings returned by three different IR systems. A tick indicates that a document is relevant, and a cross indicates that it is not relevant. With precision at 10, there is no difference between the systems, as they all return 5 relevant documents out of 10. With precision at 5, however, system 3 performs better than system 2, and system 1 performs even better than system 3, as they rank more of the relevant documents in the top 5. Uninterpolated average precision is also listed; this measure takes the complete ranking for positions 1 to 10 into account and therefore provides the most differentiated evaluation measure.  $\triangle$

Evaluation	Ranking 1	Ranking 2	Ranking 3
	d1: ✓	d10: ×	d6: ×
	d2: ✓	d9: ×	d1: ✓
	d3: ✓	d8: ×	d2: ✓
	d4: ✓	d7: ×	d10: ×
	d5: ✓	d6: ×	d9: ×
	d6: ×	d1: ✓	d3: ✓
	d7: ×	d2: ✓	d5: ✓
	d8: ×	d3: ✓	d4: ✓
	d9: ×	d4: ✓	d7: ×
	d10: ×	d5: ✓	d8: ×
Precision at 5	1	0	.40
Precision at 10	.50	.50	.50
Uninterpol. avg. prec.	1	.35	.57

Table II.16: Example for the evaluation of document rankings produced by an IR system

# Bibliography

- Anderson, A., Bader, M., Bard, E., Boyle, E., Doherty, G. M., Garrod, S., Isard, S., Kowtko, J., McAllister, J., Miller, J., Sotillo, C., Thompson, H. S., and Weinert, R. (1991). The HCRC map task corpus. *Language and Speech*, 34:351–366.
- Astrahan, Morton M., Blasgen, Mike W., Chamberlin, Donald D., Gray, Jim, III, W. Frank King, Lindsay, Bruce G., Lorie, Raymond A., Mehl, James W., Price, Thomas G., Putzolu, Gianfranco R., Schkolnick, Mario, Selinger, Patricia G., Slutz, Donald R., Strong, H. Raymond, Tiberio, Paolo, Traiger, Irving L., Wade, Bradford W., and Yost, Robert A. (1979). System R: A Relational Data Base Management System. *IEEE Computer*, 12(5):42–48.
- Boyce, Raymond F. and Chamberlin, Donald D. (1973). Using a structured english query language as a data definition facility. *IBM Research Report*, RJ1318.
- Burnard, Lou (1995). *Users Guide for the British National Corpus*. British National Corpus Consortium, Oxford University Computing Service.
- Chen, Peter Pin-Shan (1976). The Entity-Relationship Model-Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36.
- Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387.
- Francis, Nelson, Kucera, Henry, and Mackie, Andrew (1982). *Frequency Analysis of English Usage: Lexicon and Grammar*. Houghton Mifflin, Boston.
- Manning, Christopher D. and Schütze, Hinrich (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.
- Marcus, Mitchell P., Santorini, Beatrice, and Marcinkiewicz, Mary Ann (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.
- McEnery, Tony and Wilson, Andrew (2001). *Corpus Linguistics: An Introduction*. Edinburgh University Press, Edinburgh, 2 edition.

Ramakrishnan, Raghu and Gehrke, Johannes (2003). *Database Management Systems*. McGraw-Hill.

Stonebraker, Michael, Wong, Eugene, Kreps, Peter, and Held, Gerald (1976). The Design and Implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222.