# Chapter 2 Relational Model of Data

## 2.1 Overview of Data Models

***Data model***: notation for describing data or information.

The description generally consists of three parts:

1. ***Structure of the data***

Data structures used to implement data in the computer are sometimes referred to as a physical data model, although in fact they are far removed from the gates and electrons that truly serve as the physical implementation of the data. In the database world, data models are at a somewhat higher level than data structures, and are sometimes referred to as a conceptual model to emphasize the difference in level.

2. ***Operations on the data***

Operations on the data are generally anything that can be programmed.

In database data models, operations: limited set of operations that can be performed.

We are generally allowed to perform a limited set of queries (operations that retrieve information) and modifications (operations that change the database). This limitation is not a weakness, but a strength. By limiting operations, it is possible for programmers to describe database operations at a very high level, yet have the database management system implement the operations efficiently.

3. ***Constraints on the data***

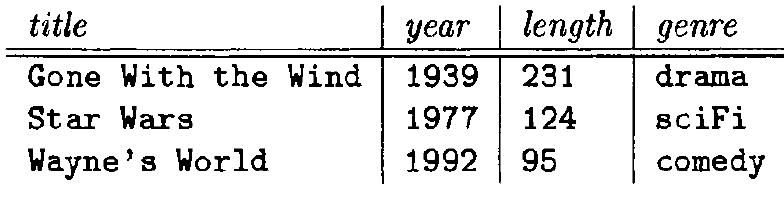
Database data models usually have a way to describe limitations on what the data can be.

2 data models of preeminent importance for database systems are:

1. The ***relational model***, including object-relational extensions.

2. The ***semistructured-data model***, including XML and related standards

|  |
| --- |
| ***Relational Model in Brief***  The relational model is based on tables.  The structure portion of the relational model might appear to resemble an array of structs in C, where the column headers are the field names, and each of the rows represent the values of one struct in the array. However, it must be emphasized that this physical implementation is only one possible way the table could be implemented in physical data structures. In fact, they are not normally implemented as main-memory structures, and their proper physical implementation must take into account the need to access relations of very large size that are resident on disk. |



|  |  |
| --- | --- |
| ***Semistructured Model in Brief***  Semistructured data resembles trees or graphs, rather than tables or arrays.  XML  The tags, similar to those used in HTML, define the role played by different pieces of data much as the column headers do in the relational model.  The operations on semistructured data usually involve following paths in the implied tree from an element to one or more of its nested subelements, then to subelements nested within those, and so on.  Constraints on the structure of data in this model often involve the data type of values associated with a tag.  Other constraints determine which tags can appear nested within which other tags. | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 4.48.24 PM.png |

|  |
| --- |
| ***Comparison of modelling approach***  It appears that semistructured models have more flexibility than relations.  This difference becomes even more apparent when we discuss how full graph structures are embedded into tree-like, semistructured models.  The relational model is still preferred in DBMS's.  Because databases are large, efficiency of access to data and efficiency of modifications to that data are of great importance. Also very important is ease of use - the productivity of programmers who use the data.  Surprisingly, both goals can be achieved with a model, particularly the relational model:  1. Provides a simple, limited approach to structuring data, yet is reasonably versatile, so anything can be modelled.  2. Provides a limited, yet useful, collection of operations on data.  Together, these limitations turn into features. They allow us to implement languages, such as SQL, that enable the programmer to express their wishes at a very high level. |

## 2.2 Basic of Relational Model

The relational model gives us a single way to represent data: as a two-dimensional table called a relation.

|  |
| --- |
| **Attributes**  The ***columns*** of a relation are named by ***attributes***. ***Attributes*** appear at the tops of the columns and they ***describe the meaning of entries in the column*** below. |

|  |
| --- |
| **Schemas**  The ***name of a relation and the set of attributes*** for a relation is called the ***schema*** for that relation.  *Ex of notation of schema*: Movies(title, year, length, genre)  The ***attributes in a relation schema*** are a ***set***, not a list.  In the relational model, a ***database consists of one or more relations***.  The ***set of schemas for the relations of a database*** is called a ***relational database schema***, or just a database schema. |

|  |
| --- |
| **Tuples**  The ***rows*** ***of a relation***, other than the header row containing the attribute names, are called ***tuples***. A tuple has one component for each attribute of the relation.  *Ex of notation of a tuple* : (Gone With the Wind, 1939, 231,drama) |

|  |
| --- |
| **Domains**  The relational model r***equires that each component of each tuple be atomic***; that is, it must be of some ***elementary type*** such as integer or string. It is ***not permitted*** for a value ***to be a record structure, set, list, array***, or any other type that reasonably can have its values broken into smaller components.  It is further assumed that associated with each attribute of a relation is a domain, that is, a particular elementary type.  The components of any tuple of the relation must have, in each component, a value that belongs to the domain of the corresponding column.  *Ex schema*: Movies(title:string, year:integer, length:integer, genre:string) |

|  |
| --- |
| **Equivalent Representation of a Relation** |
| Relations are sets of tuples which means the order in which the tuples of a relation are presented is immaterial. |
| Moreover, we can reorder the attributes of the relation as we choose, without changing the relation. However, when we reorder the relation schema, we must be careful to remember that the attributes are column headers. Thus, when we change the order of the attributes, we also change the order of their columns. |

|  |
| --- |
| **Relation Instances**  We expect ***relations to change over time but not the schema of a relation***.  However, there are situations where we might want to add or delete attribute. Schema changes, while possible in commercial database systems, ***can be very expensive***, because each of perhaps millions of tuples needs to be rewritten to add or delete components.  ***Instance of a relation***: set of tuples for a given relation.  This instance of the relation is called the current instance. |

|  |
| --- |
| **Key of Relations**  There are many constraints on relations that the relational model allows us to place on database schemas. However, one kind of constraint is so fundamental that we shall introduce it here: ***key constraints***. A ***set of attributes forms a key*** for a relation if we do ***not allow two tuples in a relation instance to have the same values in all the attributes of the key***.  *Ex*: We indicate the attribute or attributes that form a key for a relation by underlining the key attribute(s).  Movies(title, year, length, genre)  Remember that the statement that a set of attributes forms a key for a relation is a statement about all possible instances of the relation, not a statement about a single instance. |

## 2.3 Defining a Relation Schema in SQL

SQL (pronounced "sequel") is the principal language used to describe and manipulate relational databases.

There are two aspects to SQL:

1. The Data-Definition sublanguage for declaring database schemas (declares data)

2. The Data-Manipulation sublanguage for querying databases and for modifying the database (executes data)

|  |
| --- |
| **Relations in SQL**  SQL makes a distinction between three kinds of relations:  1. Stored relations, which are called *tables*. These are the kind of relation we deal with ordinarily - a relation that exists in the database and that can be modified by changing its tuples, as well as queried.  2. *Views*, which are relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed.  3. Temporary tables, which are constructed by the SQL language processor when it performs its job of executing queries and data modifications. These relations are then thrown away and not stored.  The **SQL CREATE TABLE** statement declares the schema for a stored relation. It gives a name for the table, its attributes, and their data types. It also allows us to declare a key, or even several keys, for a relation. |

|  |
| --- |
| **Data Types**  All attributes must have a data type.  1. Character strings of fixed or varying length.  The type **CHAR**(n) denotes a fixed-length string of up to n characters.  **VARCHAR**(n) also denotes a string of up to n characters.  The difference is implementation-dependent; typically CHAR implies that short strings are padded to make n characters, while VARCHAR implies that an end marker or string-length is used.  2. Bit strings of fixed or varying  The type **BIT**(n) denotes bit strings of length n, while **BIT VARYING**(n) denotes bit strings of length up to n.  3. The type **BOOLEAN** denotes an attribute whose value is logical.  The possible values of such an attribute are TRUE, FALSE, and UNKNOWN.  4. The type **INT** or **INTEGER** denotes typical integer values.  The type **SHORTINT** also denotes integers, but the number of bits permitted may be less, depending on the implementation.  5. Floating-point numbers can be represented in a variety of ways. We may  use the type **FLOAT** or **REAL** Higher precision can be obtained with the type **DOUBLE PRECISION**; **DECIMAL**(n, d) for real numbers with a fixed decimal point, n for decimal digits and d for positions from the right.  6. Dates and times can be represented by the data types **DATE** and **TIME**.  Essentially character string of special form. |

|  |
| --- |
| **Simple Table Declarations**  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 5.38.54 PM.png |

|  |
| --- |
| **Modifying Relation Schemas**  We can remove the entire table, including all of its current tuples, or we could change the schema by adding or deleting attributes.  We can delete a relation R by the SQL statement:  DROP TABLE R;  These modifications are done by a statement that begins with the keywords ALTER TABLE and the name of the relation.  1. ADD followed by an attribute name and its data type.  ALTER TABLE MovieStar ADD phone CHAR(16);  2. DROP followed by an attribute name  ALTER TABLE MovieStar DROP birthdate; |
| **Default Values**  When we create or modify tuples, we sometimes do not have values for all components.  In general, any place we declare an attribute and its data type, we may add the keyword DEFAULT and an appropriate value. That value is either NULL or a constant.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 5.48.18 PM.png |

|  |
| --- |
| **Declaring Keys**  There are ***two ways to declare an attribute or set of attributes to be a key*** in the CREATE TABLE statement that defines a stored relation.  1. We may declare one attribute to be a key when that attribute is listed in the relation schema.  2. We may add to the list of items declared in the schema (which so far have only been attributes) an additional declaration that says a particular attribute or set of attributes forms the key.  If the ***key consists of more than one attribute, we have to use method (2).*** If the key is a single attribute, either method may be used.  There are ***two declarations*** that may be used ***to indicate keyness***:  a) **PRIMARY KEY**, or  b) **UNIQUE**.  The effect of declaring a set of attributes S to be a key for relation R either using PRIMARY KEY or UNIQUE is the following:  • Two tuples in R cannot agree on all of the attributes in set S, unless one of them is NULL. Any attempt to insert or update a tuple that violates this rule causes the DBMS to reject the action that caused the violation.  In addition, if **PRIMARY KEY** is used, then attributes in S are ***not allowed to have NULL*** as a value for their components. Again, any attempt to violate this rule is rejected by the system. NULL is permitted if the set S is declared UNIQUE. However, a DBMS makes other distinctions between the two terms, if it wishes. |

## 2.4 An Algebraic Query Language

To begin our study of operations on relations, we shall learn about a special algebra, called ***relational algebra*** that consists of some ***simple but powerful ways to construct new relations from given relations***. When the given relations are stored data, then the constructed relations can be answers to queries about this data.

|  |
| --- |
| **Why Do We Need a Special Query Language**  The surprising answer is that relational algebra is useful because it is less powerful than C or Java. That is, there are computations one can perform in any conventional language that one cannot perform in relational algebra. |

|  |
| --- |
| **What is an Algebra**  ***Algebra***: consists of operators and atomic operands.  In the algebra of arithmetic, the atomic operands are variables like x and constants like 15.  The operators are the usual arithmetic ones: addition, subtraction, multiplication, and division.  Any algebra allows us to build expressions by applying operators to atomic operands and/or other expressions of the algebra. Parentheses are needed to group operators and their operands. For instance, in arithmetic we have expressions such as (x + y) \* z or ((x + 7)/(y- 3)) + x.  Relational algebra is another example of an algebra. ***Its atomic operands*** are:  1. ***Variables that stand for relations***.  2. ***Constants, which are finite relations***. |

|  |
| --- |
| **Overview of Relational Algebra**  The operations of relational algebra fall into four broad classes:  a) The ***usual set operations*** - ***union, intersection, and difference*** – applied to relations.  b) ***Operations that remove parts of a relation***: "***selection***" eliminates some rows (tuples), and "***projection***" eliminates some columns.  c) ***Operations that combine the tuples of two relations***, including "***Cartesian product***," which pairs the tuples of two relations in all possible ways, and various kinds of "***join***" operations, which selectively pair tuples from two relations.  d) An operation called "***renaming***" that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.  We generally shall refer to ***expressions of relational algebra as queries***. |

|  |
| --- |
| **Set Operations on Relations**  3 most common operations on sets are ***union***, ***intersection***, and ***difference***.  • R∪S, the ***union*** of R and S, is the ***set of elements that are in R or S or both***. An element appears only once in the union even if it is present in both R and S.  • R∩S, the ***intersection*** of R and S, is the ***set of elements that are in both R and S***.  • R−S, the ***difference*** of R and S, is the ***set of elements that are in R but not in S***. ***Note that R−S is different from S−R***; the latter is the set of elements that are in S but not in R.  When we apply these operations to relations, we need to put some conditions on R and S:  1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.  2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of Rand S must be ordered so that the order of attributes is the same for both relations. |

|  |
| --- |
| **Projection**  The ***projection operator, π,*** is ***used to produce from a relation R a new relation that has only some of R's columns***. The value of expression πA1, A2,...,An(R) is a relation that has only the columns for attributes {A1, A2,...,An} of R.  *Example:*  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.02.24 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.02.29 PM.png |

|  |
| --- |
| **Selection**  The ***selection operator produces a new relation with a subset of R's tuples***. The tuples of resulting relation are those that satisfy some condition C that involves the attributes of R. We denote this operation σC(R).  The ***schema for the resulting relation*** is the ***same as R's schema***, and we conventionally show the ***attributes in the same order*** as we use for R.  ***C is a conditional expression*** of the type with which we are familiar from conventional programming languages. The only difference is that the operands in condition C are either constants or attributes of R.  *Example:*  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.06.13 PM.png  *Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.06.19 PM.png*  *Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.06.24 PM.png* |

|  |
| --- |
| **Cartesian Product**  The ***Cartesian product*** (or *cross-product*, or *just product*) ***of two sets R and S*** is the ***set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S***.  This product is denoted R x S.  When R and S are relations, the product is essentially the same. However, since the members of R and S are tuples, usually consisting of more than one component, the result of pairing a tuple from R with a tuple from S is a longer tuple, with one component for each of the components of the constituent tuples.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.09.34 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.09.30 PM.pngMacintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.09.24 PM.png*Example:* |

|  |
| --- |
| **Natural Joins**  More often than we want to take the product of two relations, we find a need ***to join them by pairing only those tuples that match in some way***.  The simplest sort of match is the natural join of two relations R and S, ***denoted R***⋈***S***, in which ***we pair only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S***.  If the tuples r and s are successfully paired in the join R⋈S, then the result of the pairing is a tuple, called the ***joined tuple***.  The joined tuple agrees with tuple r in each attribute in the schema of R, and it agrees with s in each attribute in the schema of S. Since r and s are successfully paired, the joined tuple is able to agree with both these tuples on the attributes they have in common.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.12.41 PM.png |

A tuple that fails to pair with any tuple of the other relation in a join is said to be a ***dangling tuple***.

|  |
| --- |
| **Theta-Joins**  The natural join forces us to pair tuples using one specific condition. While this way, equating shared attributes, is the most common basis on which relations are joined, it is sometimes ***desirable to pair tuples from two relations on some other basis***. For that purpose, we have a related notation called the ***theta-join***.  The notation for a theta-join of relations R and S based on condition C is R⋈CS. The ***result of this operation is constructed*** as follows:  1. Take the product of R and S.  2. Select from the product only those tuples that satisfy the condition C.  *Example:*  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.23.00 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.23.06 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.23.11 PM.png  **Naming and Renaming**  In order to control the names of the attributes used for relations that are constructed by applying relational-algebra operations, it is often convenient to use an operator that explicitly renames relations.  We shall use the ***operator ρS(A1, A2,...,An)(R) to rename a relation R.***  The resulting relation has exactly the same tuples as R, but the name of the relation is S. Moreover, the attributes of the result relation S are named A1, A2, ..., An, in order from the left. If we only want to change the name of the relation to S and leave the attributes as they are in R, we can just say ***ρS***(R).  *Example:*  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.29.49 PM.png |

|  |
| --- |
| **Relationships Among Operations**  Some of the operations can be expressed in terms of other relational-algebra operations.  For example, ***intersection can be expressed in terms of set difference***:  R∩S = R−(R−S)  The two forms of join are also expressible in terms of other operations. ***Theta-join can be expressed by product and selection***:  R⋈CS = πL(σC (R x S) ) |

|  |
| --- |
| **A Linear Notation for Algebraic Expressions**  An alternative is to ***invent names for*** the ***temporary relations*** that correspond to the interior nodes of the tree ***and write a sequence of assignments*** that create a value for each.  The ***order of the assignments is flexible***, as long as the children of a node N have had their values created before we attempt to create the value for N itself.  The notation for assignment statements is:  1. A ***relation name and parenthesized list of attributes for that relation***.  The name Answer will be used conventionally for the result of the final step; i.e., the name of the relation at the root of the expression tree.  2. The ***assignment symbol : =.***  3. ***Any algebraic expression on the right***.  We can choose to use only one operator per assignment, in which case each interior node of the tree gets its own assignment statement.  *Example:*  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-18 at 7.38.45 PM.png |

## 2.5 Constraints on Relations

We now take up the third important aspect of a data model: the ***ability to restrict the data that may be stored in a database***. In this section, we show how to express both key constraints and "referential-integrity" constraints; the latter require that a values appearing in one column of one relation also appear in some other column of the same or a different relation.

|  |
| --- |
| **Relational Algebra as a Constraint Language**  There are ***two ways in which we can use expressions of relational algebra to express constraints***:  1. If R is an expression of relational algebra, then R = Ø is a constraint that says "The value of R must be empty," or "There are no tuples in the result of R."  2. If R and S are expressions of relational algebra, then R ⊆ S is a constraint that says, "Every tuple in the result of R must also be in the result of S."  These are actually equivalent in what they can express, but sometimes one or the other is clearer or more succinct. That is, the constraint R⊆S could just as well have been written R−S = Ø. To see why, notice that if every tuple in R is also in S, then surely R- Sis empty.  Conversely, if R−S contains no tuples, then every tuple in R must be in S (or else it would be in R−S). On the other hand, a constraint of the first form, R = Ø, could just as well have been written R⊆Ø. Technically, 0 is not an expression of relational algebra, but since there are expressions that evaluate to 0, such as R−R, there is no harm in using Ø as a relational-algebra expression. |

|  |
| --- |
| **Referential Integrity Constraints**  A common kind of constraint, called a ***referential integrity constrain***t, ***asserts that a value appearing in one context also appears in another***, related context.  In general, if we have any value v as the component in attribute A of some tuple in one relation R, then because of our design intentions we may expect that v will appear in a particular component (say for attribute B) of some tuple of another relation S.  We can express ***this integrity constraint in relational algebra*** as πA(R) ⊆ πB(S), or equivalently, πA(R) − πB(S) = Ø.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 10.49.32 AM.png  Assume that *producer of every movie would have to appear in the MovieExec* relation. *If not*, there is something wrong, and we would at least want a s*ystem implementing a relational database to inform us that we had a movie with a producer of which the database had no knowledge*. To be more precise, the producerC# component of each Movies tuple must also appear in the cert# component of some MovieExec tuple. |

|  |
| --- |
| **Key Constraints**  See how we can express algebraically the constraint that a certain attribute or set of attributes is a key for a relation.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 10.51.51 AM.png  No two tuples agree on the name component. If ***two tuples agree on name***, then they must also agree on address, which means ***must be the same tuple*** and therefore ***certainly agree in all attributes***.  The idea is that if we construct all pairs of MovieStar tuples (t1 ,t2), we must ***not find a pair that agree in the name component and disagree in the address component***. To construct the pairs we ***use a Cartesian product***, and ***to search for pairs that violate the condition we use a selection***. We then assert the constraint by equating the result to Ø. Then the requirement can be expressed by the algebraic constraint:  σMS1.name=MS2.name AND MS1.addresso≠MS2.address (MS1 x MS2) = Ø  MS1 in the product MS1 x MS2 is shorthand for the renaming:  ***ρ***MS1( name ,address ,gender, birthdate) (MovieStar) |