Part 4: The Relational Model

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Edition, 1999.
  7.1 Relational Model Concepts
  7.2 Relational Constraints and Relational Database Schemas
  7.3 Update Operations and Dealing with Constraint Violations
- Kemper/Eickler: Datenbanksysteme (in German), 4th Edition, 2001.
  Section 3.1, "Definition des relationalen Modells" ("Definition of the Relational Model")
- Silberschatz/Korth/Sudarshan: Database System Concepts, Third Edition, 1999.
  Chap. 3: Relational Model. Section 6.2: "Referential Integrity".
- Heuer/Saake: Datenbanken, Konzepte und Sprachen (in German), Thomson, 1995.
- Date/Darwen: A Guide to the SQL Standard, Fourth Edition, Addison-Wesley, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Codd: A relational model of data for large shared data banks. Communications of the ACM, 13(6), 377–387, 1970. Reprinted in CACM 26(1), 64–69, 1983.
  See also: [http://www1.acm.org:81/classics/nov95/toc.html] (incomplete)

Objectives

After completing this chapter, you should be able to:

- explain the basic concepts of the relational model.

  What is a schema? What is a state for a given schema?
- explain what domains are and why they are useful.
- explain applications and problems of null values.
- explain the meaning of keys and foreign keys.
- read various notations for relational schemas.

Overview

1. Relational Model Concepts: Schema, State
2. Null Values
3. Key Constraints
4. Foreign Key Constraints

Relational Model: Importance

- Relational database management systems currently dominate the market.

  E.g. Oracle, IBM DB2, MS SQL Server, Sybase, Informix, CA Ingres.
- Most new database projects use an RDBMS.

  There remain "legacy systems" based on a network or hierarchical DBMS. E.g. the hierarchical system IMS from IBM is still in use.
- Object-oriented database systems are used mainly for "non-standard applications" (e.g. CAD data).

  OODBMS lost some of the advantages of RDBMS. The current trend goes to object-relational DBMS. All big vendors claim OR-features.
- XML DBMS are currently being developed.

Example Database (1)

STUDENTS			
SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS			
SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Example Database (2)

- `STUDENTS`: one row for each student in the course.
  - ◇ `SID`: "Student ID" (unique number).
  - ◇ `FIRST`, `LAST`: First and last name.
  - ◇ `EMAIL`: Email address (can be null).
- `EXERCISES`: one row for each exercise.
  - ◇ `CAT`: Exercise category.

    E.g. 'H': homework, 'M': midterm exam, 'F': final exam.
  - ◇ `ENO`: Exercise number (within category).
  - ◇ `TOPIC`: Topic of the exercise.
  - ◇ `MAXPT`: Max. no. of points (How many points is it worth?).

Example Database (3)

- `RESULTS`: one row for each submitted solution to an exercise.
  - ◇ `SID`: Student who wrote the solution.

    This references a row in `STUDENTS`.
  - ◇ `CAT`, `ENO`: Identification of the exercise.

    Together, this uniquely identifies a row in `EXERCISES`.
  - ◇ `POINTS`: Number of points the student got for the solution.
  - ◇ A missing row means that the student did not yet hand in a solution to the exercise.

Data Values (1)

- Table entries are data values taken from some given selection of data types.
- The possible data types are given by the RDBMS (or the SQL standard).

  DBMS differ in the exact collection of supported data types.
- E.g. strings, numbers (of different lengths and precisions), date and time, money, binary data.
- The relational model (RM) itself is independent from any specific selection of data types.

  The definition of the RM takes a set of data types as a parameter.

Data Values (2)

- Extensible DBMS allow the user to define new data types (e.g. multimedia and geometric data types).

  There are basically two ways to offer this extensibility: (1) One can link new procedures (written e.g. in C) to the DBMS. (2) The DBMS has a programming language built-in (for server-side stored procedures), and types and operations defined in this language can be used in table declarations and queries. Real extensibility should also permit to define new index structures and to extend the query optimizer.

- This extensibility is one important feature of modern object-relational systems.

  "Universal Server/DB": can store more than numbers and strings ("all kinds of electronic information").

Data Values (3)

- As explained in Chapter 2, the given data types are specified in form of a signature $\Sigma_\mathcal{D} = (\mathcal{S}_\mathcal{D}, \mathcal{P}_\mathcal{D}, \mathcal{F}_\mathcal{D})$ and a $\Sigma_\mathcal{D}$-interpretation $\mathcal{I}_\mathcal{D}$.
- In the following definitions we only need that
  - ◇ a set $\mathcal{S}_\mathcal{D}$ of data type names is given, and

    One often says simply "data type" instead of "data type name".
  - ◇ for each $D \in \mathcal{S}_\mathcal{D}$ a set $val(D)$ of possible values of that type ($val(D) := \mathcal{I}_\mathcal{D}[D]$).
- E.g. the data type "`NUMERIC(2)`" has values `-99..+99`.

Domains (1)

- The columns `ENO` in `RESULTS` and `ENO` in `EXERCISES` should have the same data type (both are exercise numbers). The same holds for `EXERCISES.MAXPT` and `RESULTS.POINTS`.
- One can define application-specific "domains" as names (abbreviations) for the standard data types:

```
CREATE DOMAIN EX_NUMBER AS NUMERIC(2)
```

- One could even add the constraint that the number must be positive.

```
CREATE DOMAIN EX_NUMBER AS NUMERIC(2) CHECK(VALUE > 0)
```


Domains (2)

- Then the column data type is defined indirectly via a domain:

Column "`EXERCISES.ENO`" → Domain "`EX_NUMBER`" → Data Type "`NUMERIC(2)`"

Column "`RESULTS.ENO`" → Domain "`EX_NUMBER`"

- If it should ever be necessary to extend the set of possible homework numbers, e.g. to `NUMERIC(3)`, this structure ensures that no column is forgotten.

Domains (3)

- Domains are also useful in order to document that the two columns contain the same kind of thing, so comparisons between them are meaningful.
- E.g. even if the column "`POINTS`" has the same data type "`NUMERIC(2)`", this query makes little sense:

  "Which homework has a number that is the same as its number of points?"
- However, SQL does not forbid comparisons between values of different domains.

  E.g. "subdomains" would be needed (which SQL does not have). Nevertheless, comparisons between different domains are suspicious.

Domains (4)

- The SQL-92 standard contains domain definitions, but until now few systems support them.

  Oracle 8i, IBM DB2 V5, and MS SQL Server 7 all do not support `CREATE DOMAIN`. But e.g. user-defined data types in SQL Server (`sp_addtype`) are quite similar.
- Domains are at least a useful comment to better understand the connections between columns.
- Even if the RDBMS does not support domains, they should be specified during DB design.

  Oracle Designer supports domains and replaces them by the defined types when it produces `CREATE TABLE` statements.

Domains (5)

- Often, domain names can be directly used as column names.

  E.g. in an old version of the example DB, the exercise number column was called "`NO`" in `EXERCISES` and "`ENO`" in `RESULTS`. The new version seems clearer. Column names will automatically be more uniform if one normally uses the domain name as column name.
- In summary, although domains are still a bit exotic in real systems, they are a useful tool for understanding the structure of a database, and ensuring uniformity and consistency during database design.

  Domains are a bit "higher level" than the given system data types.

Atomic Attribute Values (1)

- The relational model treats the single table entries as atomic.
- I.e. the classical relational model does not permit to introduce structured and multi-valued column values.

  Each cell in the table can only contain a single number, string, etc.
- In contrast, the NF^2 ("Non First Normal Form") data model allows table entries to be complete tables in themselves (example see next page).

Atomic Attribute Values (2)

- Example for an NF^2 table (not part of the classical relational model, not treated in this course):

HOMEWORKS				
NO	TOPIC	MAXPOINTS	SOLVED_BY	
			STUDENT	POINTS
1	Rel. Alg.	10	Ann Smith	10
			Michael Jones	9
2	SQL	10	Ann Smith	8
			Michael Jones	9
			Richard Turner	10

Atomic Attribute Values (3)

- Support for "complex values" (sets, lists, records, nested tables) is another typical feature of object-relational systems.

  Oracle8 (with "Objects" option) permits any PL/SQL type for the columns, including nested tables. PL/SQL is Oracle's language for stored procedures. Beginning with Oracle 8i, Java is supported as an alternative.

- Some systems permit an arbitrary nesting of the type constructors "set", "list", "array", "multiset" (set with duplicates) and "record".

  A relation/table is then simply the special case "set (or multiset) of records".

Atomic Attribute Values (4)

- Of course, even in a classical system, if e.g. `DATE` is one of the given data types, the data type operations can be used to extract day, month, year.

  And also, when data type operations are used, strings are not really atomic, but instead a sequence of characters.

- However, this happens on the level of the given data types, not on the level of the data model itself.

  E.g. one cannot introduce new structured types, and if one makes use of strings with an important inner structure, one will soon notice that there are meaningful queries that cannot be expressed in SQL with data type functions.

Relational DB Schemas (1)

- A relation schema ρ (schema of a single relation) defines
  - ◇ a (finite) sequence $A_1 \ldots A_n$ of attribute names,

    The names must be distinct, i.e. $A_i \neq A_j$ for $i \neq j$.

  - ◇ for each attribute A_i a data type (or domain) D_i.

    Let $dom(A_i) := val(D_i)$ (set of possible values of A_i).

- A relation schema can be written as

$$\rho = (A_1 \colon D_1,\ \ldots,\ A_n \colon D_n).$$

Relational DB Schemas (2)

- A relational database schema $\mathcal{R}$ defines
  - ◇ a finite set of relation names $\{R_1, \ldots, R_m\}$, and
  - ◇ for every relation R_i, a relation schema $sch(R_i)$.
  - ◇ A set $\mathcal{C}$ of integrity constraints (defined below).

    E.g. keys and foreign keys.
- I.e. $\mathcal{R} = (\{R_1, \ldots, R_m\},\ sch,\ \mathcal{C})$.

  There are many different notations for such schemas, see below.
- Compared to the definitions in Chapter 2, the attribute names are new.

  Otherwise, relations are nothing else than predicates.

Relational DB Schemas (3)

Consequences of the Definition:

- Column names must be unique within a table: no table can have two columns with the same name.
- However, different tables can have columns with the same name (e.g. `ENO` in the example).

  The two columns can even have different data types (bad style).
- For every column (identified by the combination of table name and column name) there is a unique data type.

  Of course, different columns can have the same data type.

Relational DB Schemas (4)

- The columns within a table are ordered, i.e. there is a first, second, etc. column.

  This is normally not very important, but e.g. `SELECT * FROM` R prints the table with the columns in the given sequence.
- Within a DB schema, table names must be unique: There cannot be two tables with the same name.
- A DBMS server can normally manage several database schemas.

  Then different schemas can contain tables with the same name. E.g. within an Oracle system (instance), tables are uniquely identified by the combination of schema (user) name and table name.

Schemas: Notation (1)

- Consider the example table:

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

- One way to specify the schema precisely is via an SQL statement (see Chapter 9):

```
CREATE TABLE EXERCISES(CAT   CHAR(1),
                       ENO   NUMERIC(2),
                       TOPIC VARCHAR(40),
                       MAXPT NUMERIC(2))
```

Schemas: Notation (2)

- Although in the end, a `CREATE TABLE` statement is needed for the DBMS, other notations can be used for communicating schemas between humans.
- When discussing the general database structure, the column data types are often not important.
- One concise notation is to write the table name followed by the list of attributes:

  `EXERCISES(CAT, ENO, TOPIC, MAXPT)`

- If necessary, column datatypes can be added:

  `EXERCISES(CAT: CHAR(1), ...)`

Schemas: Notation (3)

- One can also use the header (sketch) of the table:

EXERCISES			
CAT	ENO	TOPIC	MAXPT
⋮	⋮	⋮	⋮

- Or a table with a column definition per row:

EXERCISES	
Column	Type
CAT	CHAR(1)
ENO	NUMERIC(2)
TOPIC	VARCHAR(40)
MAXPT	NUMERIC(2)

Exercise

Define a relational database schema for a collection of recipes for cookies.

- For each recipe a unique number, the name of the cookie, a short explanation what to do, and the baking time and temperature must be stored.
- For each recipe, also a set of ingredients must be stored, and for each ingredient the amount.

Tuples (1)

- An n-tuple is a sequence of n values.

  One also says simply "tuple" for n-tuple if the n is not important or clear from the context. Tuples are used to formalize table rows, then n is the number of columns.

- E.g. XY-coordinates are pairs (X, Y) of real numbers. Pairs are tuples of length 2 ("2-tuples").

  3-tuples are also called triples, and 4-tuples quadruples.

- The cartesian product $\times$ constructs sets of tuples, e.g.:

$$\mathbb{R} \times \mathbb{R} := \{(X, Y) \mid X \in \mathbb{R},\ Y \in \mathbb{R}\}.$$

Tuples (2)

- A tuple t with respect to the relation schema
$$\rho = (A_1\colon D_1, \ldots, A_n\colon D_n)$$
is a sequence $(d_1, \ldots, d_n)$ of n values such that $d_i \in val(D_i)$. I.e. $t \in val(D_1) \times \cdots \times val(D_n)$.
- Given such a tuple, we write $t.A_i$ for the value d_i in the column A_i.

  Alternative notation: $t[A_i]$.
- E.g. one row in the example table "`EXERCISES`" is the tuple ('H', 1, 'Rel. Algeb.', 10).

Database States (1)

Let a database schema $(\{R_1, \ldots, R_m\},\ sch,\ \mathcal{C})$ be given.

- A database state $\mathcal{I}$ for this database schema defines for every relation R_i a finite set of tuples with respect to the relation schema $sch(R_i)$.
- I.e. if $sch(R_i) = (A_{i,1}\colon D_{i,1},\ \ldots,\ A_{i,n_i}\colon D_{i,n_i})$, then
$$\mathcal{I}[R_i] \subseteq val(D_{i,1}) \times \cdots \times val(D_{i,n_i}).$$
- I.e. a DB state interprets the symbols in the DB schema: It maps relation names to relations.

Database States (2)

- In mathematics, the term "relation" is defined as "a subset of a cartesian product".
- E.g. an order relation such as "$<$" on the natural numbers is formally: $\{(X, Y) \in \mathbb{N} \times \mathbb{N} \mid X < Y\}$.
- Exercise: What are the differences between relations in databases and relations like "$<$"?
  1. ______________________
  2. ______________________
  3. ______________________

Database States (3)

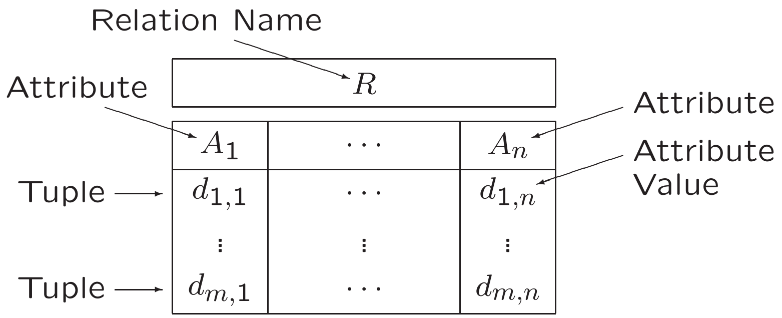
Relations are sets of tuples. Therefore:

- The sequence of the tuples is undefined.
  - ◇ The tabular representation is a bit misleading, there is no first, second, etc. row.

    The file space management strategy defines where a new row is inserted (e.g. reuse space freed by deleted rows).
  - ◇ Relations can be sorted on output.
- There are no duplicate tuples.
  - ◇ Most current systems allow duplicate tuples as long as no key is defined (see below).

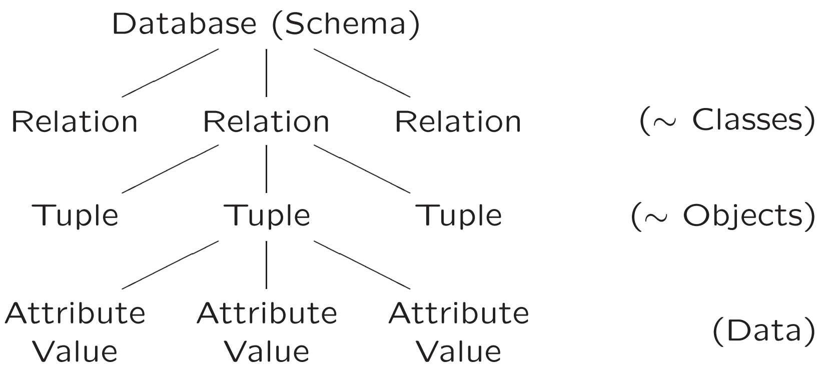
    So a formalization as a bag of tuples would be correct.

Summary (1)



Synonyms: Relation and Table.
Tuple, row, and record.
Attribute, column, field.
Attribute value, column value, table entry.

Summary (2)



Storage Structures

- Obviously, a relation can be stored as a file of records. But other data structures can offer a relational interface, too.

  The relational model does not require any specific storage structure. Tables are only the logical view. Other storage structures might allow one to answer certain queries more efficiently. E.g. the `V$*` tables in Oracle are an interface to data structures in the server.

- Exercise: Define a relational interface to

  ```
  month_names: array[1..12] of string;
  ```

  What are differences between this array and the standard "file of records" for the relation?

Update Operations (1)

- Updates transform a DB state $\mathcal{I}_{\text{old}}$ into a DB state $\mathcal{I}_{\text{new}}$. The basic update operations of the RM are:
  - ◇ Insertion (of a tuple into a relation):

    $$\mathcal{I}_{\text{new}}[R] := \mathcal{I}_{\text{old}}[R] \cup \{(d_1, \ldots, d_n)\}$$

  - ◇ Deletion (of a tuple from a relation):

    $$\mathcal{I}_{\text{new}}[R] := \mathcal{I}_{\text{old}}[R] - \{(d_1, \ldots, d_n)\}$$

  - ◇ Modification / Update (of a tuple):

    $$\mathcal{I}_{\text{new}}[R] := (\mathcal{I}_{\text{old}}[R] - \{(d_1, \ldots, d_i, \ldots, d_n)\}) \cup \{(d_1, \ldots, d_i', \ldots, d_n)\}$$

Update Operations (2)

- Modification corresponds to a deletion followed by an insertion, but without interrupting the existence of the tuple.

  It might be required by constraints that a tuple with certain values for the key attributes exists.

- SQL has commands for inserting, deleting, and modifying an entire set of tuples (of the same relation).
- Updates can also be combined to a transaction.

Overview

1. Relational Model Concepts: Schema, State
2. Null Values
3. Key Constraints
4. Foreign Key Constraints

Null Values (1)

- The relational model allows missing attribute values, i.e. table entries can be empty.
- Formally, the set of possible values for an attribute is extended by a new value "null".
- If R has the schema $(A_1: D_1, \ldots, A_n: D_n)$, then

  $$\mathcal{I}[R] \subseteq (val(D_1) \cup \{null\}) \times \cdots \times (val(D_n) \cup \{null\}).$$

- "Null" is not the number 0 or the empty string!

  It is different from all values of the data type.

Null Values (2)

- Null values are used in a variety of different situations, e.g.:
  - ◇ A value exists, but is not known.

    Suppose the university administration stores more information about students. E.g., their `STUDENTS` table might contain a column for the student's phone number, but they might not know every student's phone number, although probably most have one.

  - ◇ No value exists.

    Not every student has a second address for the duration of the term (distinct from his/her home address). Not every student has a university computer account. Yet, the `STUDENTS` table might contain columns for these data. Or consider a `COURSES` table: There might be a column `URL`, but not every course has a web page.

Null Values (3)

- Applications of null values, continued:
  - ◇ The attribute is not applicable to this tuple.

    E.g., only foreign students are required to take a Toefl test for measuring their knowledge of English. A column for the Toefl score in the `STUDENTS` table is not applicable to U.S. nationals, although they know English well. Even if they actually should have taken the test sometime in the past (e.g., because they are immigrants), the university is not interested in the result.
  - ◇ A value will be assigned later ("to be announced").
  - ◇ Any value will do.
- A comittee once found 13 different meanings for a null value.

Null Values (4)

Advantages of Null Values:

- Without null values, it would be necessary to split most relations in many relations ("subclasses"):
  - ◇ E.g. `STUDENT_WITH_EMAIL`, `STUDENT_WITHOUT_EMAIL`.
  - ◇ Or extra relation: `STUD_EMAIL(SID, EMAIL)`.
  - ◇ This complicates queries.

    One needs joins and unions (see Chapter 6).
- If null values are not allowed, users will invent fake values to fill the missing columns.

  This makes the database structure even more unclear.

Null Values (5)

Problems:

- Since the same null value is used for very different purposes, there can be no clear semantics.
- SQL uses a three-valued logic (true, false, unknown) for evaluating conditions with null values.

  For those accustomed to two-valued logic (most of us), there can be surprises — common equivalences do not hold.
- Most programming languages do not have null values. This complicates application programs.

  So when an attribute value is read into a program variable, it must be checked for a null value (→ indicator variables).

Excluding Null Values (1)

- Since null values lead to complications, it can be specified for each attribute whether or not a null value is allowed.
- It is important to invest careful thought as to where null values are needed.
- Declaring many attributes "not null" will result in simpler programs and fewer surprises with queries.
- However, flexibility is lost: Users are forced to enter values for all "not null" attributes.

Excluding Null Values (2)

- In SQL, one writes `NOT NULL` after the data type for an attribute which cannot be null.

  This is a kind of "column constraint" (see below). Between the data type and the "`NOT NULL`" one could write other column constraints (e.g. `CHECK`), but usually "`NOT NULL`" is specified first. In this way, it can also be seen as part of the data type.

- E.g. `EMAIL` in `STUDENTS` can be null:

```
CREATE TABLE STUDENTS(
             SID      NUMERIC(3)  NOT NULL,
             FIRST    VARCHAR(20) NOT NULL,
             LAST     VARCHAR(20) NOT NULL,
             EMAIL    VARCHAR(80)         )
```


Excluding Null Values (3)

- In SQL, null values are allowed by default, and one must explicitly request "`NOT NULL`".
- Often only a few columns can contain null values.
- Therefore, when using simplified schema notations, it might be better to use the opposite default:

  STUDENTS(SID, FIRST, LAST, EMAIL^o)

- In this notation, attributes which can take a null value must be explicitly marked with a small "o" (optional) in the exponent.

  This is not part of the column name. Alternative: "`EMAIL?`".

Excluding Null Values (4)

- For tabular notation, the possibility of null values can be indicated in one of these ways:

STUDENTS		
Column	Type	Null
SID	NUMERIC(3)	N
FIRST	VARCHAR(20)	N
LAST	VARCHAR(20)	N
EMAIL	VARCHAR(80)	Y

STUDENTS	SID	···	EMAIL
Type	NUMERIC(3)	···	VARCHAR(80)
Null	N	···	Y

Overview

1. Relational Model Concepts: Schema, State
2. Null Values
3. Key Constraints
4. Foreign Key Constraints

Constraints: Overview (1)

- (Integrity) Constraints are conditions that every (valid) database state must satisfy, see Chapter 3.
- E.g. in the SQL `CREATE TABLE` statement, the following types of constraints can be specified:
  - ◇ `NOT NULL`: A column cannot be null.
  - ◇ Keys: Each key value can appear only once.
  - ◇ Foreign keys: Values in a column must also appear as key values in another table.
  - ◇ `CHECK`: Column values must satisfy a condition.

    The condition can also refer to several columns of the same row.

Constraints: Overview (2)

- The SQL-92 standard contains a general `CREATE ASSERTION` statement, which is, however, not implemented in today's database systems.
- One can formalize general constraints as SQL queries that return the violations or as formulas, or one can at least document them in natural language.

  Of course, the DBMS cannot understand and thus cannot enforce constraints in natural language. But they can still be an important documentation for the later development of application programs (the checks can be programmed). If constraints are formulated as SQL queries that return violations, one can run them from time to time.

Keys: Unique Identification (1)

- A key of a relation R is an attribute/column A that uniquely identifies the tuples/rows in R.

  The key constraint is satisfied in the DB state $\mathcal{I}$ if and only if for all tuples $t, u \in \mathcal{I}[R]$ the following holds: if $t.A = u.A$ then $t = u$.

- E.g. if `SID` has been declared as key of `STUDENTS`, this database state is illegal:

STUDENTS			
SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
101	Michael	Jones	···
103	Richard	Turner	(null)
104	Maria	Brown	···

Keys: Unique Identification (2)

- If `SID` has been declared as key of `STUDENTS`, the DBMS will refuse the insertion of a second row with the same value for `SID` as an existing row.
- Note that keys are constraints: They refer to all possible DB states, not only the current one.
- Even though in the above database state (with only four students) the last name (`LAST`) could serve as a key, this would be too restrictive: E.g. the future insertion of "`John Smith`" would be impossible.

Keys: Unique Identification (3)

- A key can also consist of several attributes. Such a key is called a "composite key".

  If A and B together form a key, it is forbidden that there are two rows t and u which agree in both attributes (i.e. $t.A = u.A$ and $t.B = u.B$). They may agree in one attribute as long as they differ in the other.

- E.g. this relation satisfies the key FIRST, LAST:

STUDENTS			
SID	<u>FIRST</u>	<u>LAST</u>	EMAIL
101	Ann	Smith	···
102	John	Smith	···
103	John	Miller	···

Keys: Minimality (1)

- Let F be a formula specifying that LAST is a key.
- Let G be a formula that corresponds to the composed key consisting of FIRST and LAST.
- Then $F \vdash G$, i.e. every DB state that satisfies the key LAST also satisfies the key FIRST, LAST.

  In general, adding attributes makes keys weaker (statisfied by more states).

- Therefore, if LAST were declared as a key, it would be not interesting that FIRST, LAST also has the unique identification property.

Keys: Minimality (2)

- One will never specify two keys such that one is a subset of the other.

  Only minimal keys (with respect to "$\subseteq$") are interesting. Many authors even include this minimality condition in the definition of a key.

- However, the key "LAST" is not satisfied by the example state on Slide 4-53.
- If the database designer wants to permit this state, the key constraint "LAST" is too strong.

  This is not really a free decison: The database designer must look at possible situations in the real world to decide this question.

Keys: Minimality (3)

- Once the key "LAST" is excluded, the composed key "FIRST, LAST" becomes again interesting.

  One would also check that the key "FIRST" is not possible for the same reason.

- The database designer must now find out whether there could ever be two students in the course with the same first and last name.

  In the example state, there are no such students, but the integrity constraint must hold for all states.

Keys: Minimality (4)

- Natural keys nearly always could possibly have exceptions, yet, if exceptions are extremely seldom, one could still consider declaring the key:
  - ◇ Disadvantage: If the situation occurs, one will have to modify the name of one of the students in the database and to manually edit all official documents printed from the database.

    After I taught about 7 years, it did occur (in a course with about 150 students).
  - ◇ Advantage: Students can be identified in application programs by first name and last name.

Keys: Minimality (5)

- If the designer decides that the disadvantage of the key "FIRST, LAST" is greater than the advantage, i.e. that the key is still too strong, he/she could try to add further attributes.
- But the combination "SID, FIRST, LAST" is not interesting, because "SID" alone is already a key.
- If, however, the designer should decide that "FIRST, LAST" is "sufficiently unique", it would be minimal, even if "SID" is another key.

  The number of attributes in a key is not important for the minimality.

Multiple Keys

- A relation may have more than one key.
- E.g. SID is a key of STUDENTS, and FIRST, LAST might be another key of STUDENTS.
- One of the keys is designated as the "primary key".

  The primary key should be a key that consists only of a single, short attribute and is never updated (if available). The primary key is used in other tables that need to refer to rows in this table. In some systems, access via the primary key might be especially fast. Otherwise, the selection of the primary key is more or less arbitrary.
- Other keys are called "alternate/secondary keys".

  SQL uses the keyword UNIQUE for alternate keys.

Keys: Notation (1)

- The primary key attributes are often marked by underlining them in relational schema specifications:

$$R(\underline{A_1: D_1}, \ldots, \underline{A_k: D_k}, A_{k+1}: D_{k+1}, \ldots, A_n: D_n).$$

STUDENTS

<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

- Usually, the attributes of a relation are ordered such that the primary key consists of the first attributes.

Keys: Notation (2)

- In SQL, keys can be defined as follows:

```
CREATE TABLE STUDENTS(
             SID       NUMERIC(3)   NOT NULL,
             FIRST     VARCHAR(20)  NOT NULL,
             LAST      VARCHAR(20)  NOT NULL,
             EMAIL     VARCHAR(80),
             PRIMARY KEY(SID),
             UNIQUE(FIRST, LAST))
```

- The exact syntax details are treated in Chapter 9.

Keys and Null Values

- The primary key cannot be null, other keys should not be null.

  In SQL-89 and DB2, `NOT NULL` must be specified for every attribute in a `PRIMARY KEY` or `UNIQUE` constraint.
  In SQL-92 and Oracle, the "`PRIMARY KEY`" declaration automatically implies "`NOT NULL`", but "`UNIQUE`" (for alternate keys) does not.
  In Oracle, there can be several rows, all with a null value in a `UNIQUE` key. In SQL Server, only one row can be null. SQL-92 defines three different semantics for composite keys which have null values in only some of their attributes. One should avoid all this mess.

- It is as not acceptable if already the "object identity" of the tuple is not known.

Keys and Updates

- It is considered poor style if key attribute values are modified (updated).

  This would change the "object identity". Better: Delete the tuple first and then insert a tuple with the new values.

- But SQL does not enforce this constraint.

  The standard even contains specifications for what to do with foreign keys if the referenced key value is updated.

The Weakest Possible Key

- A key consisting of all attributes of a relation requires only that there can never exist two different tuples which agree in all attributes.

  Theoretically, relations are sets: Then this is no restriction. However, in practice, relations are multisets (bags), and this key ensures that there are no duplicates.

- Style Recommendation: Define at least one key for every relation in order to exclude duplicate tuples.

  If there is no other key, define the key consisting of all attributes of the relation.

Keys: Summary

- Declaring a set of attributes as a key is a bit more restrictive than the unique identification property:
  - ◇ Null values are excluded at least in the primary key.
  - ◇ One should avoid updates, at least of the primary key.
- However, the uniqueness is the main requirement for a key. Everything else is secondary.

Exercises (1)

- Select a key:

SOLVED		
STUDENT	HW	POINTS
John Smith	1	10
John Smith	2	12
Maria Brown	1	9

- Give an example for an insertion that would violate the key:

- Could "POINTS" also be declared as key?

Exercises (2)

- Consider an appointment calendar:

APPOINTMENTS				
DATE	START	END	ROOM	WHAT
Jan. 19	10:00	11:00	IS 726	Michael
Jan. 19	14:00	15:00	IS 726	Siripun
Jan. 19	18:00	20:50	IS 501	INFSCI 2710

- What would be correct keys?
- Give an example for a non-minimal key (superkey).
- Are additional constraints needed? I.e. can there be invalid database states, even if the key is satisfied?

Overview

Foreign Keys (1)

- The relational model has no explicit relationships, links or pointers.
- Values for the key attributes identify a tuple.

  They are "logical addresses" of the tuples.
- To refer to tuples of R in a relation S, include the primary key of R among the attributes of S.

  Such attribute values are "logical pointers" to tuples in R.
- E.g. the table `RESULTS` has the attribute `SID`, which contains primary key values of `STUDENTS`.

Foreign Keys (2)

`SID` in `RESULTS` is a foreign key referencing `STUDENTS`:

STUDENTS			
SID	FIRST	LAST	···
101	Ann	Smith	···
102	Michael	Jones	···
103	Richard	Turner	···
104	Maria	Brown	···

RESULTS			
SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
102	H	1	9
102	H	2	9
103	H	1	5
105	H	1	7

?
Error

The constraint that is needed here is that every `SID` value in `RESULTS` also appears in `STUDENTS`.

Foreign Keys (3)

- When `SID` in `RESULTS` is a foreign key that references `STUDENTS`, the DBMS will reject any attempt to insert a solution for a non-existing student.
- Thus, the set of `SID`-values that appear in `STUDENTS` are a kind of "dynamic domain" for the attribute `SID` in `RESULTS`.
- In relational algebra (see Chapter 6), the projection $\pi_{\texttt{SID}}$ returns the values of the column `SID`. Then the foreign key condition is:

$$\pi_{\texttt{SID}}(\texttt{RESULTS}) \subseteq \pi_{\texttt{SID}}(\texttt{STUDENTS}).$$

Foreign Keys (4)

- The foreign key constraint ensures that for every tuple t in `RESULTS` there is a tuple u in `STUDENTS` such that $t.\texttt{SID} = u.\texttt{SID}$.

  Pairs of such tuples t and u can be brought together by a relational algebra operation called "Join" (see Chapter 6). This corresponds to the dereferencing of pointers in other models. Without foreign key constraint, there could be "dangling pointers" that point to nowhere. However, an SQL query would not crash in this case: Tuples without "join partner" are silently eliminated in a query that does a join.
- The key constraint for `STUDENTS` ensures that there is at most one such tuple u.

  Together, it follows that every tuple t in `RESULTS` references exactly one tuple u in `STUDENTS`.

Foreign Keys (5)

- Enforcing foreign key constraints ensures the "referential integrity" of the database.

  I.e. foreign key constraint and referential integrity constraint are synonyms.

- A foreign key implements a "one-to-many" relationship: One student has solved many exercises.
- The table RESULTS which contains the foreign key is called the "child table" of the referential integrity constraint, and the referenced table STUDENTS is the "parent table".

Foreign Keys (6)

- The table RESULTS contains another foreign key that references the solved exercise.
- Exercises are identified by category (e.g. homework, midterm, final) and number (CAT and ENO):

RESULTS

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
⋮	⋮	⋮	⋮

EXERCISES

CAT	ENO	···	MAXPT
H	1	···	10
H	2	···	10
M	1	···	14

Foreign Keys (7)

- A table with a composed key (like EXERCISES) must be referenced with a composed foreign key that has the same number of columns.
- Corresponding columns must have the same data type.
- It is not required that corresponding columns have the same name.
- In the example, the composed foreign key requires that every combination of CAT and ENO which appears in RESULTS, must also appear in EXERCISES.

Foreign Keys (8)

- Columns are matched by their position in the declaration: E.g. if the key is (FIRST, LAST) and the foreign key is (LAST, FIRST) insertions will very probably give an error.

  If the data types of FIRST and LAST are sufficiently different, the error might be detected when the foreign key is declared. But some systems require only "compatible" data types and that would be satisfied even if FIRST and LAST are VARCHAR-types with different lengths.

- Only keys can be referenced: One cannot reference only part of a composite key or a non-key attribute.

  Normally, one should reference only the primary key, but SQL permits referencing alternate keys.

Foreign Keys: Notation (1)

- In the attribute list notation, foreign keys can be marked by an arrow and the referenced relation. Composed foreign keys need parentheses:

  RESULTS(<u>SID</u> → STUDENTS,
          (<u>CAT</u>, <u>ENO</u>) → EXERCISES, POINTS)
  STUDENTS(<u>SID</u>, FIRST, LAST, EMAIL)
  EXERCISES(<u>CAT</u>, <u>ENO</u>, TOPIC, MAXPT)

- Since normally only the primary key is referenced, it is not necessary to specify the corresponding attribute in the referenced relation.

Foreign Keys: Notation (2)

- The above example is untypical because all foreign keys are part of keys. This is not required, e.g.:

  COURSE_CATALOG(<u>NO</u>, TITLE, DESCRIPTION)
  COURSE_OFFER(<u>CRN</u>, CRSNO → COURSE_CATALOG, TERM,
          (INST_FIRST, INST_LAST) → FACULTY)
  FACULTY(<u>FIRST</u>, <u>LAST</u>, OFFICE, PHONE)

  In this example, also the names of the foreign key attributes and the referenced key attributes are not the same. That is legal.

- Some people mark foreign keys by dashed underlining or by overlining. This is not recommended because it does not specify the referenced table.

Keys: Notation (3)

- In SQL, foreign keys can be defined as follows:

```
CREATE TABLE RESULTS(
                SID      NUMERIC(3)   NOT NULL,
                CAT      CHAR(1)      NOT NULL,
                ENO      NUMERIC(2)   NOT NULL,
                POINTS   NUMERIC(4,1) NOT NULL,
                PRIMARY KEY(SID, CAT, ENO),
                FOREIGN KEY(SID)
                         REFERENCES STUDENTS,
                FOREIGN KEY(CAT,ENO)
                         REFERENCES EXERCISES)
```


Foreign Keys: Notation (4)

- In the tabular notation, foreign keys can be specified e.g. as follows:

RESULTS	SID	CAT	ENO	POINTS
Type	NUMERIC(3)	CHAR(1)	NUMERIC(2)	NUMERIC(2)
Null	N	N	N	N
References	STUDENTS	EXERCISES	EXERCISES	

- Composed foreign keys pose again a problem.

  If the above notation should be unclear or ambiguous specify the names of the referenced columns and/or distribute the foreign key information over several lines. In rare circumstances, foreign keys can also overlap, then certainly several lines are needed.

Foreign Keys: Notation (5)

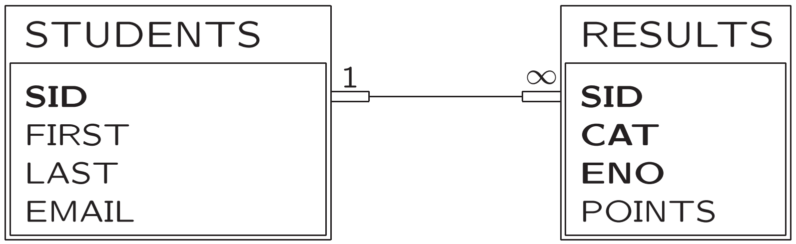
- In the Oracle DBA Exam, the following "Instance Chart" is used to describe a relation schema:

Instance Chart for Table MUSIC_PIECE			
Column Name:	PNO	PNAME	CNO
Key Type:	PK		FK
Nulls/Unique:	NN, U	NN	
FK Table:			COMPOSER
FK Column:			CNO
Datatype:	NUMBER	VARCHAR	NUMBER
Length:	4	40	2

- "FK" stands for "foreign key", "NN" for "not null", "PK" for "primary key", "U" for "unique".

Foreign Keys: Notation (6)

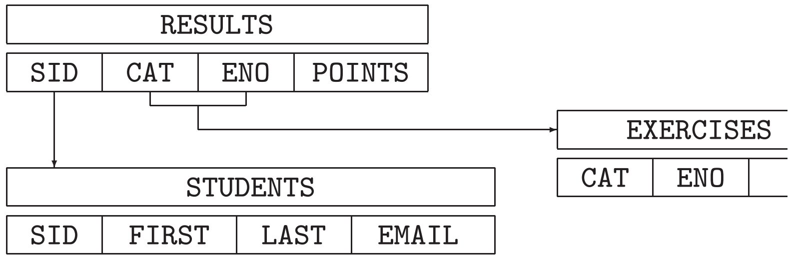
- MS Access displays foreign keys as "relationships" (primary key attributes are shown in boldface):



- "1" / "∞" symbolize the one-to-many relationship: "1" marks the side of the key, "∞" the foreign key.

Foreign Keys: Notation (7)

- Of course, also arrows between tables can be used:



- Some people draw the arrows not in the direction of the pointer, but in the opposite direction.

  E.g. in the Oracle DBA exam. Look carefully at the given database instance.

More about Foreign Keys (1)

Foreign Keys and Null Values:

- Unless a "not null" constraint is explicitly specified, foreign keys can be null.
- The foreign key constraint is satisfied even if the referencing attributes are "null". This corresponds to "nil" pointer.
- If a foreign key consists of more than one attribute, they should either all be null, or none should be null.

  But Oracle and SQL-92 allow partially defined foreign keys. In Oracle, if at least one attribute in the foreign key is null, the constraint counts as satisfied. The SQL-92 standard defines three different semantics.

More about Foreign Keys (2)

Mutual References:

- It is possible that parent and child are the same table, e.g.
  `EMP(EMPNO, ENAME, JOB, MGR`^{o}`→EMP, DEPTNO→DEPT)`
  `PERSON(NAME, MOTHER`^{o}`→PERSON, FATHER`^{o}`→PERSON)`
- Two relations can reference each other, e.g.
  `EMPLOYEES(EMPNO, ..., DEPT→DEPARTMENTS)`
  `DEPARTMENTS(DNO, ..., LEADER`^{o}`→EMPLOYEES).`
- Exercise/Puzzle: How can tuples be inserted?

Please Remember:

- Foreign keys are not themselves keys!

  The attributes which form a foreign key may be part of a key, but this is an exception, not the rule. The foreign key constraint has nothing to do with a key constraint.
  For some authors, however, a key is any attribute that identifies tuples, not necessary of the same relation. Then foreign keys would be keys, but normal keys need some adjective ("unique key/primary key").
- Only a key of a relation can be referenced, not arbitrary attributes.
- If the key of the referenced relation consists of two attributes, the foreign key must also consist of two attributes of the same data types in the same order.

Foreign Keys and Updates (1)

The following operations can violate a foreign key:

- Insertion into the child table `RESULTS` without a matching tuple in the parent table `STUDENTS`.
- Deletion from the parent table `STUDENTS` when the deleted tuple is still referenced.
- Update of the foreign key `SID` in the child table `RESULTS` to a value not in `STUDENTS`.

  This is usually treated like an insertion.
- Update of the key `SID` of the parent table `STUDENTS` when the old value is still referenced.

Foreign Keys and Updates (2)

Note:

- Deletions from `RESULTS` (child table) and insertions into `STUDENTS` (parent table) can never lead to a violation of the foreign key constraint.

  This means that the DBMS does not have to check the constraint for these operations.

Reactions on Insertions of Dangling References:

- The insertion is rejected. The DB state remains unchanged.

Foreign Keys and Updates (3)

Reactions on Deletions of Referenced Key Values:

- The deletion is rejected.
- The deletion cascades: All tuples from `RESULTS` that reference the deleted `STUDENTS` tuple are deleted, too.
- The foreign key is set to null.

  Contained in SQL-92, supported in DB2, not in Oracle.
- The foreign key is set to a declared default value.

  Contained in SQL-92, but not in Oracle or DB2.

Foreign Keys and Updates (4)

Reactions on Updates of Referenced Key Values:

- The update is rejected. The DB state remains unchanged.

  DB2 and Oracle support only this alternative of the SQL-92 standard. In any case, changing key attributes is bad style.
- The update cascades.

  I.e. the student ID is changed in the table `RESULTS` in the same way as it was changed in the table `STUDENTS`.
- The foreign key is set to null.
- The foreign key is set to a declared default value.

Foreign Keys and Updates (5)

- When specifying a foreign key, decide which reaction is best.
- The default is the first alternative ("No Action").
- At least for deletions from the parent table, all systems should support also the cascading deletion.

  This is a kind of active integrity enforcement: The system does not reject the update, but does some other updates in order to repair the database state.
- Other alternatives exist only in few systems at the moment.

Exercise (1)

Define a relational DB schema for hotel. It requires:

- Information about guests: First Name, Last Name, and Home Address.
- Information about rooms: Is it single or double? What is the official room rate? When was it last renovated?
- Information about stays: Which room was rented by which guest from which date to which date? And at what room rate was he/she charged?

  It might be less than the offical rate.

Exercise (2)

Please define the following:

- Table names and column names.
- Keys.
- Foreign keys.
- Null constraints.

Furthermore, describe any additional constraints which might be necessary.