Principles of Data- and Knowledge-based Systems

Torsten Schaub
University of Potsdam
torsten@cs.uni-potsdam.de

Rough Roadmap

- Relational data model
- First-order logic
- Relational algebra
- Database design

Description logics

Web reasoning

Stream processing

Bibliography

Literature

Primer Stefan Brass, MLU Halle-Wittenberg: Datenbanken I
`http://users.informatik.uni-halle.de/~brass`

Books [1], [3], [4], [8], [9], [10], [14], [15], [23], [24], [28], [29], [21], [6], [22],

Surveys [7], [12], [20],

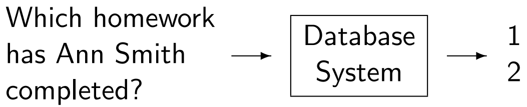
Articles [5], [11], [16], [17], [18], [19], [25], [26], [27], etc.

Introduction: Overview

Outline

Task of a Database (1)

- What is a database? Difficult question. There is no precise and generally accepted definition.
- Naive approach: The main task of a database system (DBS) is to answer certain questions about a subset of the real world, e.g.



Task of a Database (2)

- The DBS acts only as storage for information.
  The information must first be entered and then kept up tp date.

$$\begin{tikzcd}
\begin{array}{l}\text{Ann Smith has}\\ \text{done Homework 3}\\ \text{and received}\\ \text{10 points for it.}\end{array} \arrow[r] & \boxed{\begin{array}{c}\text{Database}\\ \text{System}\end{array}} \arrow[r] & \text{ok.}
\end{tikzcd}$$

- A database system is a computerized version of a card-index box/filing cabinet (but more powerful).
- A spreadsheet could be considered a restricted, small DBS.

Task of a Database (3)

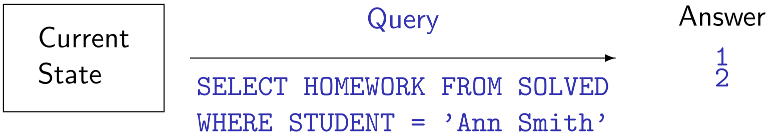
- Normal database systems do not perform very complicated computations on the stored data in order to answer questions.
- However, they can find/retrieve the requested data quickly in/from a large set of data
  (Terabytes or Petabytes — larger than main memory).
- They can also aggregate/combine several pieces of stored data for one answer, e.g. compute the average points for Homework 3.

Task of a Database (4)

- Above, the question "Which homeworks has Ann Smith completed?" was shown in natural language.
- Making computers understand natural language is not easy (large potential for misunderstandings).
- Therefore, questions ("queries") are normally written in a formal language, today typically in SQL.
- But there are natural language interfaces for DBS.

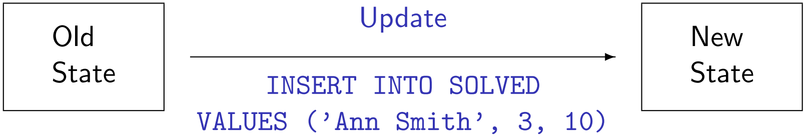
State, Query, Update

- The set of stored data is called the database state:



```
SELECT HOMEWORK FROM SOLVED
WHERE STUDENT = 'Ann Smith'
```

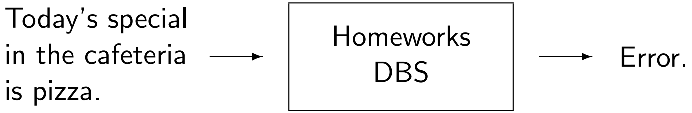
- Entering, modifying, or deleting information changes the state:



```
INSERT INTO SOLVED
VALUES ('Ann Smith', 3, 10)
```

Structured Information (1)

- Each database can store only information of a predeclared structure (a limited domain of discourse):



- Because the data are structured, not simply text, more complex evaluations are possible, e.g.:

How many homeworks has each student done?

Structured Information (2)

- Actually, a database system stores only data (character strings, numbers), and not information.
- Data become information by interpretation.
- Therefore, concepts like students and exercises must be defined/declared before the database can be used.

State vs. Schema (1)

Database Schema

- Formal definition of the structure of the database contents.
- Determines the possible database states.
- Defined only once (when the DB is created).
- Corresponds to variable declaration (type information).

State vs. Schema (1)

Database State (Instance of the Schema)

- Contains the actual data, structured according to the schema.
- Changes often (whenever database-information is updated).
- Corresponds to current contents/value of a variable.

State vs. Schema (2)

- In the relational data model, the data is structured in form of tables (relations).
- Each table has a name, sequence of named columns (attributes) and a set of rows (tuples).

SOLVED		
STUDENT	HOMEWORK	POINTS
Ann Smith	1	10
Ann Smith	2	8
Mike Jones	1	9
Mike Jones	2	9

DB Schema

DB State (Instance)

Data Model (1)

- A data model defines
  - a set $\mathcal{SCH}$ of possible database schemas,
  - for each database schema $\mathcal{S} \in \mathcal{SCH}$ a set $\mathcal{ST}(\mathcal{S})$ of possible database states.
- Often (but not always), a data model is parameterized in a set of basic data types.
- E.g., for the relational model, it is not important whether the table cells can contain only strings, or also numbers, date and time values, and so on.

Data Model (2)

Languages

- Data Definition Language (DDL)
  Language that is used to define DB schemas (write them down).
- Data Manipulation Language (DML)
  Language that is used to write queries and updates.
- Some people use the term “Data Model” for “Database Schema”.

Data Model (3)

Examples

- Relational Model
- Entity-Relationship-Model
- Object-Oriented Data Models
- Object-Relational Data Models
- XML
- etc.

- Network Model (historical)
- Hierarchical Model (historical)

Outline

Database Management Systems (1)

- A Database Management System (DBMS) is an application-independent software that implements a data model, i.e. allows
  - definition of a DB schema for some concrete application,
  - storage of an instance of this schema on e.g. a disk,
  - querying the current instance (database state),
  - changing the database state.

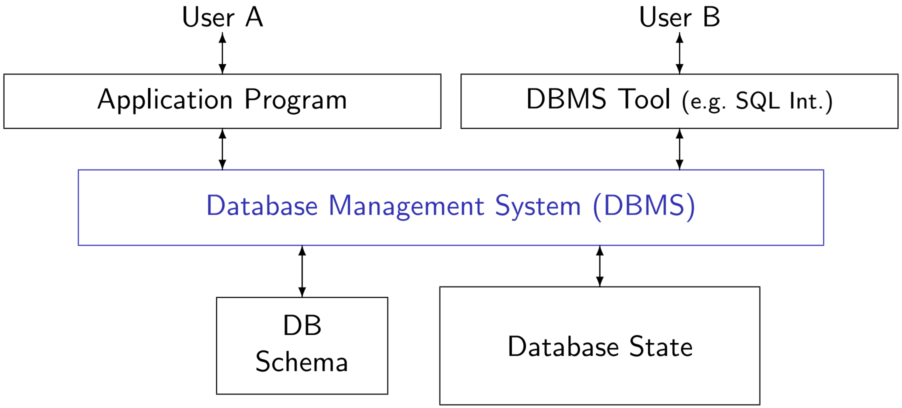
Database Management Systems (2)

- Of course, normal users do not need to use SQL for their daily tasks of data entry or lookup.
- They use application programs that have been developed especially for this task and offer a nicer user interface.
- However, internally, the application program contains SQL statements (queries, updates) in order to communicate with the DBMS.

Database Management Systems (3)

- Often, several different application programs are used to access the same centralized database.
- E.g. the homeworks DB might have:
  - A web interface for students.
  - A program used by the graduate student assistant to load homework and exam points.
  - A program that prints a report for the professor used to assign grades.
- The interactive SQL interface that comes with the DBMS is simply another way to access the DB.

Database Management Systems (4)

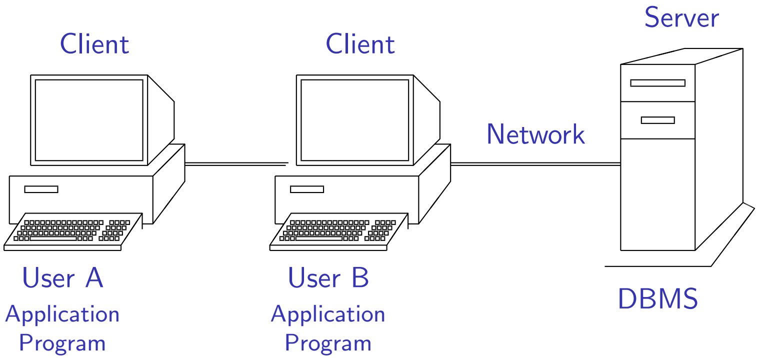


DB Application Systems (1)

- Often, different users access the same database concurrently (i.e. at the same time).
- The DBMS is usually a background server process (or set of such processes) that is accessed over the network by application programs (clients).
- One can also view the DBMS as an extension of the operating system (more powerful file system).

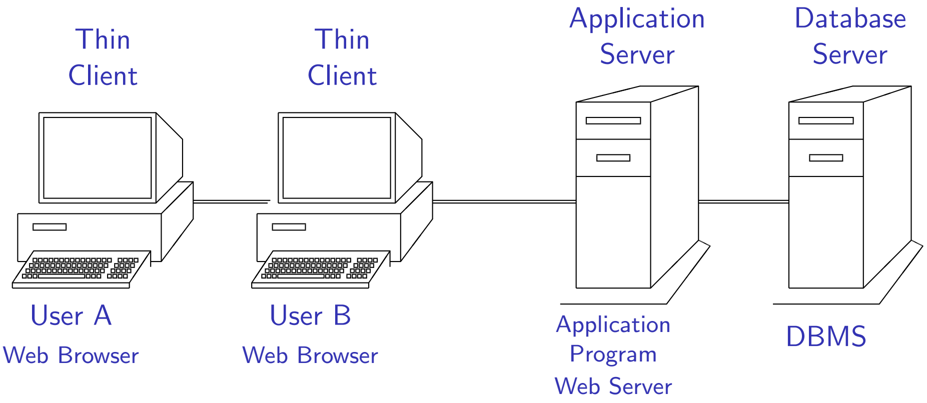
DB Application Systems (2)

Client-Server Architecture



DB Application Systems (3)

Three-Tier Architecture



DB Application Systems (4)

Some Database Vocabulary

- A database consists of DB schema and DB state.
- A database system (DBS) consists of a DBMS and a database.
- A database application system consists of a DBS and a set of application programs.

Outline

Persistent Storage (1)

Today

$$\begin{tikzcd} 5 \arrow[r] & \boxed{\text{factorial}} \arrow[r] & 120 \end{tikzcd}$$

Tomorrow

$$\begin{tikzcd} 5 \arrow[r] & \boxed{\text{factorial}} \arrow[r] & 120 \end{tikzcd}$$

$\Rightarrow$ No persistent storage necessary.
The output is a function of the input only.

Persistent Storage (2)

Today

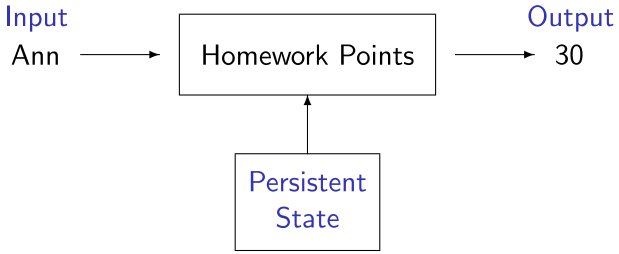
$$\begin{tikzcd}\text{Ann} \arrow[r] & \boxed{\text{Homework Points}} \arrow[r] & 20\end{tikzcd}$$

Tomorrow

$$\begin{tikzcd}\text{Ann} \arrow[r] & \boxed{\text{Homework Points}} \arrow[r] & 30\end{tikzcd}$$

$\Rightarrow$ Output is a function of the input
and a persistent state.

Persistent Storage (3)



- Persistent Information
  - Information that lives longer than a single process (e.g. program execution).
  - Survives power outage and a reboot of the operating system.

Typed Persistent Data (1)

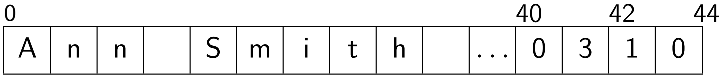
Classical Way to Implement Persistence

- Information needed in other program invocations is saved in a file.
- The operating system (OS) stores the file on a disk.
- Disk is persistent memory: The contents is not lost if the computer is switched off or the operating system is rebooted.
- File systems are predecessors of modern database management systems.

Typed Persistent Data (2)

Implementing Persistence with Files

- OS files are usually only sequences of bytes.
- A record structure must be defined as in Assembler languages.



- File structure information is contained only in the programmers' heads.
- The system cannot prevent errors because it does not know the file structure.

Typed Persistent Data (3)

Implementing Persistence with a DBMS

- The structure of the information to be stored must be defined in a way the system understands:

```
CREATE TABLE SOLVED(STUDENT  VARCHAR(40),
                    HOMEWORK NUMERIC(2),
                    POINTS   NUMERIC(2))
```

- Thus, the file structure is formally documented.
- The system can detect type errors in application programs.
- Simplified programming (higher abstraction level).

A Subprogram Library (1)

- Most DBMS use OS files to store the data.
- One can view a DBMS as a subprogram library that can be used for file accesses.
- Compared with the direct OS calls for file accesses, the DBMS offers higher level operations.
- I.e. it contains already many algorithms that one would otherwise have to program.

A Subprogram Library (2)

- For instance, a DBMS contains routines for
  - Sorting (e.g. Mergesort)
  - Searching (e.g. B-trees)
  - File Space Management, Buffer Management
  - Aggregation, Statistical Evaluation
- Optimized for large data sets (that do not fit into main memory).
- It also has multi-user support (automatic locking) and safety measures to protect the data in case of system crashes (see below).

Data Independence (1)

- The DBMS is a layer of software above the OS files.
  The files can be accessed only via the DBMS.
- Indirection gives the possibility of hiding internal changes.
- Idea of abstract data types:
  Change the implementation, but keep the interface.
- Here the implementation is the file structure, which has to be changed for performance reasons.
  The application program interface is kept stable.

Data Independence (2)

Typical Example

- At the beginning, a professor used the homeworks database only for his courses in the current term.
- Since the database was small, and there were relatively few accesses, it was sufficient to store the data as a "heap file".

- Later the entire university used the database, and information of previous courses had to be kept for some time.
- Thus, the DB became much bigger and was also accessed more frequently.
- An index (e.g. B-tree) is needed for faster access.

Data Independence (3)

Without DBMS (or with a Pre-Relational DBMS)

- Using the index to access the file must be explicitly mentioned in the query command.
- Thus, application programs must be changed if the file structure is changed.
- If one forgets to change a seldom used application program, and it does not update the index when the table is changed, the DB becomes inconsistent.

Data Independence (4)
With Relational DBMS

- The system hides the existence of indexes at the interface.
- Queries and updates do not have to (and cannot) refer to the index.
- The system automatically
  - modifies the index in case of updates,
  - uses the index to evaluate queries when advantageous.

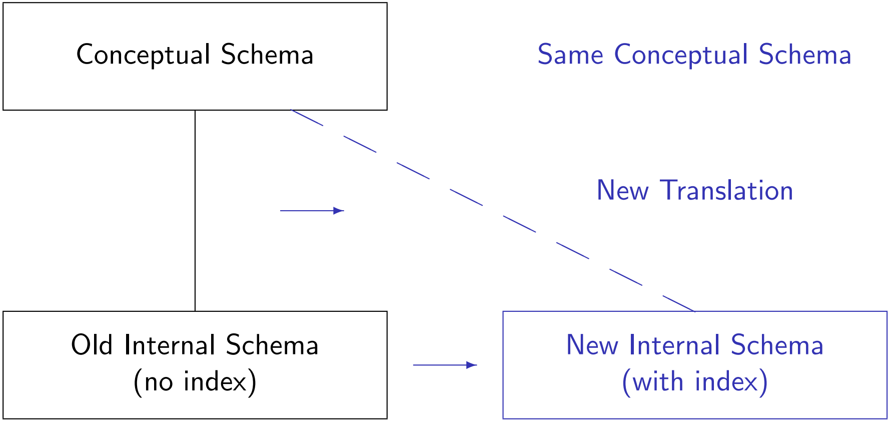
Data Independence (5)

- Conceptual Schema ("Interface")
  - Only logical information content of the database
  - Simplified View: Storage details hidden.
- Internal/Physical Schema ("Implementation")
  - Indexes
  - Division of tables among disks
  - Storage management if tables grow or shrink
  - Physical placement of new rows in a table.

Data Independence (6)

1. The user enters a query (e.g. in SQL) that refers to the conceptual schema.
2. The DBMS translates this into a query/program ("execution plan") which refers to the internal schema (this is done by the "query optimizer").
3. The DBMS executes the translated query on the stored instance of the internal schema.
4. The DBMS translates the result back to the conceptual level.

Data Independence (7)


Conceptual Schema
Same Conceptual Schema
New Translation
Old Internal Schema (no index)
New Internal Schema (with index)

Declarative Languages (1)

- Physical data independence requires that the query language cannot refer to indexes.
- Declarative query languages go one step further:
  - Queries should describe only what information is sought,
  - but should not prescribe any particular method how to compute this information.
- Algorithm = Logic + Control (Kowalski)

Declarative Languages (2)

- SQL is a declarative query language.
  The user specifies only conditions for the requested data:

```
SELECT X.POINTS
FROM   SOLVED X
WHERE  X.STUDENT = 'Ann Smith'
AND    X.HOMEWORK = 3
```

- Often simpler formulations: The user does not have to think about efficient execution.
- Much shorter than imperative programming:
  Less expensive program development/maintainance.

Declarative Languages (3)

- Declarative query languages
  - allow powerful optimizers
  - need powerful optimizers
- Larger independence of current hardware/software technology:
  - Simpler Parallelization
  - Today's queries will use tomorrow's algorithms when a new version of the DBMS is released.

"Data Independence"

- Decoupling between programs and data.
- Data is an independent resource by itself.
- Physical data independence:
  - Programs should not depend on data storage methods.
  - Vice versa, the file structures are not determined by the programs.

Logical Data Independence (1)

- Logical data independence allows changes to the logical information content of the database.
- Of course, information can only be added, e.g. add a column "`SUBMISSION_DATE`" to the table `SOLVED`.
- This may be required for new applications.
- It should not be necessary to change old applications, although the records are now longer.

Logical Data Independence (2)

- Logical data independence is only important when there are application programs with distinct, but overlapping information needs.
- Logical data independence also helps to integrate previously distinct databases.
  - In earlier times, every department of a company had its own database / data files.
  - Now, businesses generally aim at one central DB.

Logical Data Independence (3)

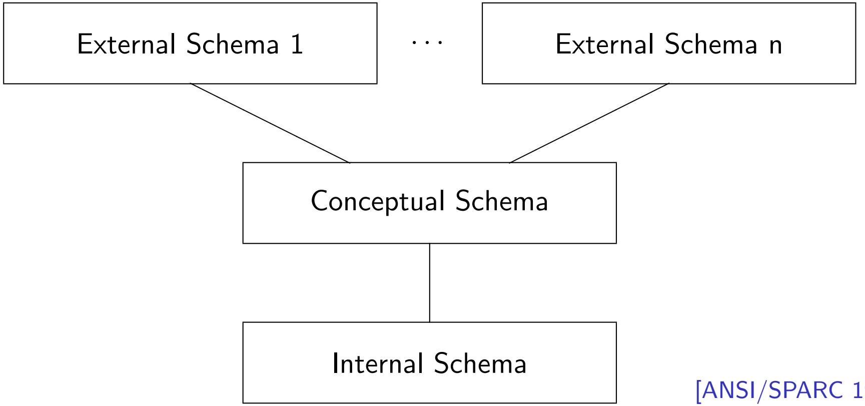
- If a company has more than one DB, the information in these databases will normally overlap, i.e. some pieces of information are stored several times.
- Data is called redundant if it can be derived from other data and knowledge about the application.
- Problems:
  - Duplicates data entry and update efforts.
  - Sooner or later one will forget to modify one copy (data becomes inconsistent).
  - Wastes storage space, also on backup tapes.

Logical Data Independence (4)

External Schemas/Views

- Logical data independence requires a third level of database schemas, the external schemas/views.
- Each user can have an individual view of the data.
- An external view contains a subset of the information in the database, maybe slightly restructured.
- In contrast, the conceptual schema describes the complete information content of the database.

Three-Schema Architecture



[ANSI/SPARC 1978]

More DBMS Functions (1)

Transactions

- Sequence of DB commands that are executed as an atomic unit ("all or nothing").
- Support for Backup and Recovery
- Support of concurrent users

More DBMS Functions (2)

- Security
  - Access rights: Who may do what on which table.
  - Auditing: The DBMS may remember who did what.
- Integrity
  - It is possible to let the DBMS check that the entered data are plausible.
  - The DBMS can also reject updates that would violate defined business rules.

More DBMS Functions (3)

Data Dictionary

- Information about the data (e.g. schema, user list, access rights) is available in system tables, e.g.:

SYS_TABLES	
TABLE_NAME	OWNER
SOLVED	BRASS
SYS_TABLES	SYS
SYS_COLUMNS	SYS

SYS_COLUMNS		
TABLE_NAME	SEQ	COL_NAME
SOLVED	1	STUDENT
SOLVED	2	HOMEWORK
SOLVED	3	POINTS
SYS_TABLES	1	TABLE_NAME
SYS_TABLES	2	OWNER
SYS_COLUMNS	1	TABLE_NAME
SYS_COLUMNS	2	SEQ
SYS_COLUMNS	3	COL_NAME

Outline

Database Users (1)

Database Administrator (DBA)

- Should know the complete database schema.
- Gives access rights to users. Ensures security.
- Monitors system performance.
- Monitors available disk space and installs new disks.
- Ensures that backup copies of the data are made.
  Does the recovery after disk failures etc.
- Installs new versions of the DBMS software.
- Tries to ensure data correctness.
- Responsible for licence agreement.
- Contact for support / DBMS vendor.
- Expert on the DBMS software.
- Can damage everything.

Database Users (2)

Application Programmer

- Writes programs for standard tasks.
- Knows SQL well, plus some programming languages and development tools.
- Usually supervised by the database administrator.
- Might do DB design (i.e. develop the DB schema).

Database Users (3)

- Sophisticated User (one kind of “End User”)
  - Knows SQL and/or some query tools.
  - Does non-standard evaluations of the data without help from application programmers.
- Naive User (the other kind of “End User”)
  - Uses the database only via application programs.
  - Does the real work of entering data.

Database Tools

- Interactive SQL interpreter
- Graphical/Menu-based query tools
- Interface for DB access from standard programming languages (C, Pascal, Java, . . .)
- Tools for form-based DB applications
- Report generator
- WWW interface
- Tools for import/export of data, backup&recovery, performance monitoring, . . .

Outline

Summary (1)

Functions of Database Systems

- Persistence (data lives longer than a single program execution)
- Integration / No Redundancy (duplicate storage)
- Data Independence
- Less programming effort: Many algorithms built-in, especially for external memory (disks)
- Ad-hoc Queries
- High data security (Backup & Recovery)
- Combinition of updates into atomic transactions
- Multi-User: synchronization of concurrent accesses
- Integrity Enforcement
- Views for different users (user groups)
- Data Access Control
- System Catalog (Data Dictionary)

Summary (2)

- The main goal of the DBMS is to give the user a simplified view on the persistant storage
- The user does not have to worry about:
  - Physical storage details
  - Different information needs of different users
  - Efficient query formulation
  - Possibility of system crashes / disk failures
  - Possibility of concurrent access by other users

Logic: Overview

Outline

Introduction, Motivation (1)

Important goals of mathematical logic are

- to formalize the notion of a statement about a certain domain of discourse (logical formula),
- to precisely define the notions of logical implication and proof,
- to find ways to mechanically check whether a statement is logically implied by given statements.

Introduction, Motivation (2)

Mathematical logic is applied in databases I

- In general, the purpose of both, mathematical logic and databases, is to
  - formalize knowledge,
  - work with this knowledge (process it).
- For instance, in order to talk about a domain of discourse, symbols are needed.
  - In logic, these are defined in a signature.
  - In databases, they are defined in a DB schema.

Introduction, Motivation (2)

Mathematical logic is applied in databases II

- In order to formalize logical implication, mathematical logic had to study possible interpretations of the symbols,

  i.e. possible situations in the domain of discourse about which the logical formulas make statements.
- Database states also describe possible situations in a certain part of the real world.
- Basically, logical interpretations and DB states are the same (at least in the "model-theoretic view").

Introduction, Motivation (3)

Mathematical logic is applied in databases III

- SQL queries are quite similar to formulas in mathematical logic, and there are theoretical query languages that are simply a version of logic.
- The idea is that
  - a query is a logical formula with placeholders ("free variables"),
  - the database system then determines values for these placeholders that make the formula true in the given database state.

Introduction, Motivation (4)

Why it makes sense to learn mathematical logic I

- Logical formulas are simpler than SQL, and can easily be formally studied.
- Important concepts of database queries can already be learned in this simpler, purer environment.
- Experience has shown that students often make logical errors in SQL queries.

Introduction, Motivation (5)

Why it makes sense to learn mathematical logic II

- SQL changes, and becomes more and more complicated (standards: 1986, 1989, 1992, 1999, 2003).
- There are new data models (e.g., XML) with new query languages, and faster changes than SQL.
- At least some part of this course should still be valid and useful in 30 years.

History of the Field (1)

∼322 BC	Syllogisms [Aristoteles]
∼300 BC	Axioms of Geometry [Euklid]
∼1700	Plan of Mathematical Logic [Leibniz]
1847	"Algebra of Logic" [Boole]
1879	"Begriffsschrift" (Early Logical Formulas) [Frege]
∼1900	More natural formula syntax [Peano]
1910/13	Principia Mathematica (Collection of formal proofs) [Whitehead/Russell]
1930	Completeness Theorem [Gödel/Herbrand]
1936	Undecidability [Church/Turing]

History of the Field (2)

1960	First Theorem Prover [Gilmore/Davis/Putnam]
1963	Resolution-Method for Theorem proving [Robinson]
∼1969	Question Answering Systems [Green et.al.]
1970	Linear Resolution [Loveland/Luckham]
1970	Relational Data Model [Codd]
∼1973	Prolog [Colmerauer, Roussel, et.al.] (Started as Theorem Prover for Natural Language Understanding) (Compare with: Fortran 1954, Lisp 1962, Pascal 1970, Ada 1979)
1977	Conference “Logic and Databases” [Gallaire, Minker]

Outline

Alphabet (1)

Definition

- Let *ALPH* be some infinite, but enumerable set, the elements which are called symbols.
- *ALPH* must contain at least the logical symbols, i.e. $LOG \subseteq ALPH$, where

$$LOG = \{(,\ ),\ ,\ ,\ \top,\ \bot,\ =,\ \neg,\ \wedge,\ \vee,\ \leftarrow,\ \rightarrow,\ \leftrightarrow,\ \forall,\ \exists\}.$$

- In addition, *ALPH* must contain an infinite subset $VARS \subseteq ALPH$, the set of variables.
  This must be disjoint to *LOG* (i.e. $VARS \cap LOG = \emptyset$).

Alphabet (2)

- E.g., the alphabet might consist of
  - the special logical symbols LOG,
  - variables starting with an uppercase letter and consisting otherwise of letters, digits, and "_",
  - identifiers starting with a lowercase letter and consisting otherwise of letters, digits, and "_".
- Note that words like "`father`" are considered as symbols (elements of the alphabet).
- In theory, the exact symbols are not important.

Alphabet (3)

- If the special logical symbols are not available, use:

Symbol	Alternative	Another	Name
$\top$	`true`	`T`	
$\bot$	`false`	`F`	
$\neg$	`not`	$\sim$	Negation
$\wedge$	`and`	`&`	Conjunction
$\vee$	`or`	`|`	Disjunction
$\leftarrow$	`if`	`<-`	
$\rightarrow$	`then`	`->`	
$\leftrightarrow$	`iff`	`<->`	
$\exists$	`exists`	`E`	Existential Quantifier
$\forall$	`forall`	`A`	Universal Quantifier

Signatures (1)

Definition

- A signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ consists of:
  - A non-empty and finite set $\mathcal{S}$, the elements which are called sorts (data type names).
  - For each $\alpha = s_1 \dots s_n \in \mathcal{S}^*$, a finite set (of predicate symbols) $\mathcal{P}_\alpha \subseteq \mathit{ALPH} \setminus (\mathit{LOG} \cup \mathit{VARS})$.
  - For each $\alpha \in \mathcal{S}^*$ and $s \in \mathcal{S}$, a set (of function symbols) $\mathcal{F}_{\alpha,s} \subseteq \mathit{ALPH} \setminus (\mathit{LOG} \cup \mathit{VARS})$.
- For each $\alpha \in \mathcal{S}^*$ and $s_1, s_2 \in \mathcal{S}$, $s_1 \neq s_2$, it must hold that $\mathcal{F}_{\alpha,s_1} \cap \mathcal{F}_{\alpha,s_2} = \emptyset$.

Signatures (2)

- A sort is a data type name, e.g. `int`, `string`, `person`.
- A predicate is something that can be true or false for given input values, e.g. <, `substring_of`, `female`.
- If $p \in \mathcal{P}_\alpha$, then $\alpha = s_1, \dots, s_n$ are called the argument sorts of p.
- For example:
  - $< \in \mathcal{P}_{\texttt{int int}}$, also written as `<(int, int)`.
  - `female` $\in \mathcal{P}_{\texttt{person}}$, also written as `female(person)`.

Signatures (3)

- The number of argument sorts (length of α) is called the arity of a predicate symbol, e.g.:
  - $<$ is a predicate symbol of arity 2.
  - `female` is a predicate symbol of arity 1.
- Predicates of arity 0 are called propositional constants, or simply propositions. E.g.:
  - `the_sun_is_shining`,
  - `i_am_working`.
- The symbol ϵ is used to denote the empty sequence.
  The set $\mathcal{P}_\epsilon$ contains the propositional constants.

Signatures (4)

- The same symbol p can be element of several $\mathcal{P}_\alpha$ (overloaded predicate), e.g.
  - $< \in \mathcal{P}_{\mathtt{int}\,\mathtt{int}}$.
  - $< \in \mathcal{P}_{\mathtt{string}\,\mathtt{string}}$ (lexicographic order).
- This means that there are actually two different predicates that have the same name.

Signatures (5)

- A function is something that returns a value for given input values, e.g. `+`, `age`, `first_name`.
- A function symbol in $\mathcal{F}_{\alpha,s}$ has argument sorts α and result sort s, e.g.
  - $+ \in \mathcal{F}_{\texttt{int int}, \texttt{int}}$, also written as `+(int, int): int`.
  - $\texttt{age} \in \mathcal{F}_{\texttt{person}, \texttt{int}}$, also written as `age(person): int`.

Signatures (6)

- A function with 0 arguments is called a constant.
- Examples of constants:
  - $1 \in \mathcal{F}_{\epsilon,\texttt{int}}$, also written as `1: int`.
  - $'\texttt{Ann}' \in \mathcal{F}_{\epsilon,\texttt{string}}$, also written as `'Ann': string`.
- For data types (e.g., `int`, `string`), it is usual that every possible value can be denoted by a constant.

Signatures (7)

- A signature specifies the application-specific symbols that are used to talk about the domain of discourse (a part of the real world that is to be modeled in the database).
- The above definition is for a multi-sorted (typed) logic.
  One can also use an unsorted logic.

Signatures (8)
Example

- $\mathcal{S} = \{\texttt{person}, \texttt{string}\}$.
- $\mathcal{F}$ consists of
  - constants of sort `person`, e.g. `arno`, `birgit`, `chris`.
  - infinitely many constants of sort `string`, e.g. $''$, $'\texttt{a}'$, $'\texttt{b}'$, ..., $'\texttt{Arno}'$, ...
  - function symbols $\texttt{first\_name}(\texttt{person})\colon \texttt{string}$ and $\texttt{last\_name}(\texttt{person})\colon \texttt{string}$.
- $\mathcal{P}$ consists of
  - a predicate $\texttt{married\_to}(\texttt{person}, \texttt{person})$.
  - predicates $\texttt{male}(\texttt{person})$ and $\texttt{female}(\texttt{person})$.

Signatures (9)
Definition

- A signature $\Sigma' = (\mathcal{S}', \mathcal{P}', \mathcal{F}')$ is an extension of a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ iff
  - $\mathcal{S} \subseteq \mathcal{S}'$,
  - for every $\alpha \in \mathcal{S}^*$: $\mathcal{P}_\alpha \subseteq \mathcal{P}'_\alpha$,
  - for every $\alpha \in \mathcal{S}^*$ and $s \in \mathcal{S}$: $\mathcal{F}_{\alpha,s} \subseteq \mathcal{F}'_{\alpha,s}$.
- I.e. an extension of Σ' adds new symbols to Σ.

Interpretations (1)

Definition

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be given.
- A Σ-interpretation $\mathcal{I}$ defines:
  - a set $\mathcal{I}(s)$ for every $s \in \mathcal{S}$ (domain),
  - a relation $\mathcal{I}(p, \alpha) \subseteq \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n)$ for every $p \in \mathcal{P}_\alpha$, and $\alpha = s_1 \dots s_n \in \mathcal{S}^*$.
  - a function $\mathcal{I}(f, \alpha)\colon\ \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n) \to \mathcal{I}(s)$ for every $f \in \mathcal{F}_{\alpha,s}$, $s \in \mathcal{S}$, and $\alpha = s_1 \dots s_n \in \mathcal{S}^*$.
- In the following, we write $\mathcal{I}[\dots]$ instead of $\mathcal{I}(\dots)$.

Interpretations (2)

Note

- Empty domains cause certain problems, therefore it is usual to exclude them.
- But in databases, domains can be empty (e.g. a set of persons when the database was just created).

Interpretations (3)

- The relation $\mathcal{I}[p]$ is also called the extension of p (in $\mathcal{I}$).
- Formally, predicate and relation are not the same, but isomorphic notions.
- For instance, $\texttt{married\_to}(\texttt{X}, \texttt{Y})$ is true in $\mathcal{I}$ if and only if $(\texttt{X}, \texttt{Y}) \in \mathcal{I}[\texttt{married\_to}]$.
- Another Example: $(3, 5) \in \mathcal{I}[<]$ means simply $3 < 5$.

Interpretations (4)

Example interpretation for signature on Slide 85

- $\mathcal{I}[\texttt{person}]$ is the set of Arno, Birgit, and Chris.
- $\mathcal{I}[\texttt{string}]$ is the set of all strings, e.g. $'\text{a}'$.
- $\mathcal{I}[\texttt{arno}]$ is Arno.
- For the string constants, $\mathcal{I}$ is the identity mapping.
- $\mathcal{I}[\texttt{first\_name}]$ maps e.g. Arno to $'\texttt{Arno}'$.
- $\mathcal{I}[\texttt{last\_name}]$ maps all three persons to $'\texttt{Schmidt}'$.
- $\mathcal{I}[\texttt{married\_to}] = \{(\text{Birgit}, \text{Chris}),\ (\text{Chris}, \text{Birgit})\}$.
- $\mathcal{I}[\texttt{male}] = \{(\text{Arno}),\ (\text{Chris})\}$, $\mathcal{I}[\texttt{female}] = \{(\text{Birgit})\}$.

Relational Databases (1)

- A DBMS defines a set of data types, such as strings and numbers, together with constants, data type functions (e.g. $+$) and predicates (e.g. $<$).
- For these, the DBMS defines names (in a signature Σ) and their meaning (in an interpretation $\mathcal{I}$).
- For every value $d \in \mathcal{I}[s]$, there is at least one constant c with $\mathcal{I}[c] = d$.

- The DB schema in the relational model then adds further predicate symbols (relation symbols).
- The DB state interprets these by finite relations.

Relational Databases (2)
Example

- In a relational database for storing homework results, there might be three predicates/relations:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`

- Here, we treat the "domain calculus" version of the relational model.

Outline

Variable Declaration (1)

- Definition
  - Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be a signature.
  - A variable declaration for Σ is a partial mapping $\nu\colon \mathit{VARS} \to \mathcal{S}$
- Remark
  - The variable declaration is not part of the signature because it is locally modified by quantifiers (see below).
  - The signature is fixed for the entire application, the variable declaration changes even within a formula.

Variable Declaration (2)

Example

- A variable declaration simply defines which variables are available and what are their sorts, e.g.
  $\nu = \{\texttt{SID}/\texttt{int},\ \texttt{Points}/\texttt{int},\ \texttt{E}/\texttt{exercise}\}$.
- Of course, each variable must have a unique sort.

Variable Declaration (3)

- Definition
  - Let ν be a variable declaration, $X \in \mathit{VARS}$, and $s \in \mathcal{S}$.
  - Then we write $\nu\langle X/s\rangle$ for the modified variable declaration ν' with

$$\nu'(V) := \begin{cases} s & \text{if } V{=}X \\ \nu(V) & \text{otherwise.} \end{cases}$$

- Remark
  - Both is possible: ν might have been defined before for X or it might be undefined.

Terms (1)

- Terms are syntactic constructs that can be evaluated to a value (a number, a string, an exercise).
- There are three kinds of terms:
  - constants, e.g. `1`, `'abc'`, `arno`,
  - variables, e.g. `X`,
  - composed terms, consisting of a function symbol applied to argument terms, e.g. `last_name(arno)`.
- In programming languages, terms are also called expressions.

Terms (2)
Definition

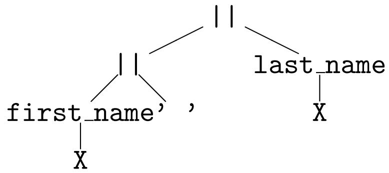
- Let $\Sigma=(\mathcal{S},\mathcal{P},\mathcal{F})$ be a signature and ν a variable declaration for Σ.
- The set $TE_{\Sigma,\nu}(s)$ of terms of sort s is recursively defined as follows:
  - Every variable $V \in VARS$ with $\nu(V) = s$ is a term of sort s.
  - Every constant $c \in \mathcal{F}_{\epsilon,s}$ is a term of sort s.
  - If t_1 is a term of sort s_1, ..., t_n is a term of sort s_n, and $f \in \mathcal{F}_{\alpha,s}$ with $\alpha = s_1 \dots s_n$, $n \geq 1$, then $f(t_1, \dots, t_n)$ is a term of sort s.
  - Nothing else is a term of sort s.
- Each term can be constructed by a finite number of applications of the above rules.
- Let $TE_{\Sigma,\nu} := \bigcup_{s\in\mathcal{S}} TE_{\Sigma,\nu}(s)$ be the set of all terms.

Terms (3)

- Certain functions are also written as infix operators, e.g. $\mathtt{X}+\mathtt{1}$ instead of the official notation $+(\mathtt{X}, \mathtt{1})$.
- Functions of arity 1 can be written in dot-notation, e.g. "`X.first_name`" instead of "`first_name(X)`".
- Such "syntactic sugar" is useful in practice, but not important for the theory of logic.
- In the following, the above abbreviations are used.

Terms (4)

- Terms can be visualized as operator trees
  (“||” is in SQL the function for string concatenation):



Terms (5)

Exercise

- Which of the following are legal terms (given the signature on slide 85 and a variable declaration ν with $\nu(\mathtt{X}) = \mathtt{string}$)?
  - ☐ `arno`
  - ☐ `first_name`
  - ☐ `first_name(X)`
  - ☐ `firstname(arno, birgit)`
  - ☐ `married_to(birgit, chris)`
  - ☐ `X`

Atomic Formulas (1)

- Formulas are syntactic expressions that can be evaluated to a truth value (true or false), e.g.

$$1 \leq X \wedge X \leq 10.$$

- Atomic formulas are the basic building blocks of such formulas (comparisons etc.).
- Atomic formulas can have the following forms:
  - A predicate symbol applied to terms, e.g. `married_to(birgit, X)`.
  - An equation, e.g. `X = chris`.
  - The logical constants $\top$ (true) and $\bot$ (false).

Atomic Formulas (2)

Definition

- Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be a signature and ν a variable declaration for Σ .
- An atomic formula is an expression of one of the following forms:
  - $p(t_1, \dots, t_n)$ with $p \in \mathcal{P}_\alpha$, $\alpha = s_1 \dots s_n \in \mathcal{S}^*$, and $t_i \in TE_{\Sigma,\nu}(s_i)$ for $i = 1 \dots n$.
  - $t_1 = t_2$ with $t_1, t_2 \in TE_{\Sigma,\nu}(s)$, $s \in \mathcal{S}$.
  - $\top$ and $\bot$.
- Let $AT_{\Sigma,\nu}$ be the set of atomic formulas for Σ,ν.

Atomic Formulas (3)

Remarks

- For some predicates, one traditionally uses infix notation, e.g. $X > 1$ instead of $>(X, 1)$.
- For propositional constants, the parentheses can be skipped, e.g. one can write p instead of $p()$.
- Of course, it would be possible to treat "$=$" as a normal predicate, and some authors do that.

Formulas (1)

Definition

- Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be a signature and ν a variable declaration for Σ.
- The sets $FO_{\Sigma,\nu}$ of (Σ, ν)-formulas are defined recursively as follows:
  - Every atomic formula $F \in AT_{\Sigma,\nu}$ is a formula.
  - If F and G are (Σ, ν)-formulas, so are $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
  - $(\forall\, s\, X\colon F)$ and $(\exists\, s\, X\colon F)$ are in $FO_{\Sigma,\nu}$ if $s \in \mathcal{S}$, $X \in VARS$, and F is a $(\,\Sigma,\ \nu\langle X/s\rangle\,)$-formula.
  - Nothing else is a (Σ, ν)-formula.

Formulas (2)

- The intuitive meaning of the formulas is as follows:
  - $p(t_1 \dots t_n)$: The predicate p is true for the values of the terms $t_1, \dots, t_n$.
  - $\neg F$: "Not F" (F is false).
  - $F \wedge G$: "F and G" (F and G are both true).
  - $F \vee G$: "F or G" (at least one of F and G is true).
  - $F \leftarrow G$: "F if G" (if G is true, F must be true).
  - $F \rightarrow G$: "if F, then G"
  - $F \leftrightarrow G$: "F if and only if G".
  - $\forall\, s\, X : F$: "for all X (of sort s), F is true".
  - $\exists\, s\, X : F$: "there is an X (of sort s) such that F".

Formulas (3)

- Above, many parentheses are used in order to ensure that formulas have a unique syntactic structure.
- One uses the following rules to save parentheses:
  - The outermost parentheses are never needed.
  - $\neg$ binds strongest, then $\wedge$, then $\vee$, then $\leftarrow$, $\rightarrow$, $\leftrightarrow$ (same binding strength), and last $\forall$, $\exists$.
  - Since $\wedge$ and $\vee$ are associative, no parentheses are required for e.g. $F_1 \wedge F_2 \wedge F_3$.

Formulas (4)

Abbreviations for Quantifiers

- When there is only one possible sort of a quantified variable, one can leave it out, i.e. write $\forall X \colon F$ instead of $\forall\, s\, X \colon F$ (and the same for $\exists$).
- If one quantifier immediately follows another quantifier, one can leave out the colon.
- Instead of a sequence of quantifiers of the same type, e.g. $\forall X_1 \ldots \forall X_n \colon F$, one can write $\forall X_1 \ldots X_n \colon F$.

Formulas (5)

- Abbreviation for Inequality
  - $t_1 \neq t_2$ can be used as an abbreviation for $\neg(t_1 = t_2)$.
- Note
  - Some people say "formulae" instead of "formulas".
- Exercise
  - Given a signature with $\leq \in \mathcal{P}_{\mathsf{int}\ \mathsf{int}}$ and $1, 10 \in \mathcal{F}_{\epsilon,\mathsf{int}}$, and a variable declaration with $\nu(X) = \mathtt{int}$.
  - Is $1 \leq X \leq 10$ a syntactically correct formula?

Formulas (6)
Exercise

- Which of the following are syntactically correct formulas (given the signature on Slide 85)?
  - ☐ $\forall\,\mathtt{X},\mathtt{Y}\colon \mathtt{married\_to}(\mathtt{X},\mathtt{Y}) \rightarrow \mathtt{married\_to}(\mathtt{Y},\mathtt{X})$
  - ☐ $\forall\,\mathtt{person}\ \mathtt{P}\colon\ \vee\ \mathtt{male}(\mathtt{P}) \vee \mathtt{female}(\mathtt{P})$
  - ☐ $\forall\,\mathtt{person}\ \mathtt{P}\colon\ \mathtt{arno} \vee \mathtt{birgit} \vee \mathtt{chris}$
  - ☐ $\mathtt{male}(\mathtt{chris})$
  - ☐ $\forall\,\mathtt{string}\ \mathtt{X}\colon\ \exists\,\mathtt{person}\ \mathtt{X}\colon\ \mathtt{married\_to}(\mathtt{birgit},\mathtt{X})$
  - ☐ $\mathtt{married\_to}(\mathtt{birgit},\mathtt{chris}) \wedge \vee\,\mathtt{married\_to}(\mathtt{chris},\mathtt{birgit})$

Closed Formulas

- Definition
  - Let Σ be a signature.
  - A closed formula (for Σ) is a (Σ,ν)-formula for the empty variable declaration ν.
- Exercise
  - Which of the following are closed formulas?
    - ☐ $\texttt{female}(\texttt{X}) \wedge \exists\,\texttt{X}\colon \texttt{married\_to}(\texttt{chris}, \texttt{X})$
    - ☐ $\texttt{female}(\texttt{birgit}) \wedge \texttt{married\_to}(\texttt{chris}, \texttt{birgit})$
    - ☐ $\exists\,\texttt{X}\colon \texttt{married\_to}(\texttt{X}, \texttt{Y})$

Variables in a Term
Definition

- The function *vars* computes the set of variables that occur in a given term t.
  - If t is a constant c: $\mathit{vars}(t) := \emptyset$.
  - If t is a variable V: $\mathit{vars}(t) := \{V\}$.
  - If t has the form $f(t_1 \dots t_n)$: $\mathit{vars}(t) := \bigcup_{i=1}^{n} \mathit{vars}(t_i)$.

Free Variables in a Formula
Definition

- The function *free* computes the set of free variables (not bound by a quantifier) in a formula F:
  - If F is an atomic formula $p(t_1 \ldots t_n)$ or $t_1 = t_2$: $\mathit{free}(F) := \bigcup_{i=1}^{n} \mathit{vars}(t_i)$.
  - If F is $\top$ or $\bot$: $\mathit{free}(F) := \emptyset$.
  - If F has the form $(\neg G)$: $\mathit{free}(F) := \mathit{free}(G)$.
  - If F has the form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.: $\mathit{free}(F) := \mathit{free}(G_1) \cup \mathit{free}(G_2)$.
  - If F has the form $(\forall\, s\, X \colon G)$ or $(\exists\, s\, X \colon G)$: $\mathit{free}(F) := \mathit{free}(G) \setminus \{X\}$.

Variable Assignment (1)

- Definition
  - A variable assignment $\mathcal{A}$ for $\mathcal{I}$ and ν is a partial mapping $\nu\colon \mathit{VARS} \to \bigcup_{s \in \mathcal{S}} \mathcal{I}[s]$.
  - It maps every variable V, for which ν is defined, to a value from $\mathcal{I}[s]$, where $s := \nu(V)$.
- Remark
  - I.e. a variable assignment for $\mathcal{I}$ and ν defines values from $\mathcal{I}$ for the variables that are declared in ν.

Variable Assignment (2)

Example

- Consider the following variable declaration ν:
  $\nu = \{\mathtt{X}/\mathtt{string}, \mathtt{Y}/\mathtt{person}\}$.
- One possible variable assignment is
  $\mathcal{A} = \{\mathtt{X}/\mathit{abc}, \mathtt{Y}/\mathit{Chris}\}$.

Variable Assignment (3)

- Definition
  - $\mathcal{A}\langle X/d\rangle$ denotes a variable assignment $\mathcal{A}'$ that agrees with $\mathcal{A}$ except that $\mathcal{A}'(X) = d$.
- Example
  - Given the variable declaration on the last slide, $\mathcal{A}\langle \mathtt{Y}/\text{Birgit}\rangle$ is: $\mathcal{A}\langle \mathtt{Y}/\text{Birgit}\rangle = \{\mathtt{X}/abc, \mathtt{Y}/\text{Birgit}\}$.

Value of a Term

Definition

- Let Σ be a signature, ν a variable declaration for Σ, $\mathcal{I}$ a Σ-interpretation, and $\mathcal{A}$ a variable assignment for $(\mathcal{I}, \nu)$.
- The value $\langle\mathcal{I}, \mathcal{A}\rangle[t]$ of a term $t \in TE_{\Sigma,\nu}$ is defined recursively as follows:
  - If t is a constant c, then $\langle\mathcal{I}, \mathcal{A}\rangle[t] := \mathcal{I}[c]$.
  - If t is a variable V, then $\langle\mathcal{I}, \mathcal{A}\rangle[t] := \mathcal{A}(V)$.
  - If t has the form $f(t_1 \dots t_n)$, with t_i of sort s_i:

$$\langle\mathcal{I}, \mathcal{A}\rangle[t] := \mathcal{I}[f, s_1 \dots s_n]\big(\langle\mathcal{I}, \mathcal{A}\rangle[t_1], \dots, \langle\mathcal{I}, \mathcal{A}\rangle[t_n]\big).$$

Truth of a Formula (1)
Definition

- Let Σ be a signature, ν a variable declaration for Σ, $\mathcal{I}$ a Σ-interpretation, and $\mathcal{A}$ a variable assignment for $(\mathcal{I}, \nu)$.
- The truth value $\langle\mathcal{I}, \mathcal{A}\rangle[F] \in \{\mathrm{f}, \mathrm{t}\}$ of a formula F in $(\mathcal{I}, \mathcal{A})$ is defined as follows (f means false, t true):
  - If F is an atomic formula $p(t_1 \dots t_{\mathrm{n}})$ with terms t_i of sort s_i:

$$\langle\mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} \mathrm{t} & \text{if } \big(\langle\mathcal{I}, \mathcal{A}\rangle[t_1], \dots, \langle\mathcal{I}, \mathcal{A}\rangle[t_n]\big) \in \mathcal{I}[p, s_1 \dots s_n] \\ \mathrm{f} & \text{otherwise.} \end{cases}$$

  - (continued on next three slides)

Truth of a Formula (2)

Definition, continued

- Truth value of a formula, continued:
  - If F is an atomic formula $t_1 = t_2$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle[t_1] = \langle \mathcal{I}, \mathcal{A} \rangle[t_2] \\ \mathrm{f} & \text{else.} \end{cases}$$

  - If F is $\top$: $\langle \mathcal{I}, \mathcal{A} \rangle[F] := \mathrm{t}$.
  - If F is $\bot$: $\langle \mathcal{I}, \mathcal{A} \rangle[F] := \mathrm{f}$.
  - If F is of the from $(\neg G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle[G] = 0 \\ \mathrm{f} & \text{else.} \end{cases}$$

Truth of a Formula (3)

Definition, continued

- Truth value of a formula, continued:
  - If F is of the from $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.:

G_1	G_2	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
f	f	f	f	t	t	t
f	t	f	t	f	t	f
t	f	f	t	t	f	f
t	t	t	t	t	t	t

  - E.g. if $\langle \mathcal{I}, \mathcal{A} \rangle[G_1] = \mathrm{t}$ and $\langle \mathcal{I}, \mathcal{A} \rangle[G_2] = \mathrm{f}$ then $\langle \mathcal{I}, \mathcal{A} \rangle[(G_1 \wedge G_2)] = \mathrm{f}$.

Truth of a Formula (4)

Definition, continued

- Truth value of a formula, continued:
  - If F has the form $(\forall s\, X\colon G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathrm{t} \\ & \quad \text{for all } d \in \mathcal{I}[s] \\ \mathrm{f} & \text{otherwise.} \end{cases}$$

  - If F has the form $(\exists s\, X\colon G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathrm{t} \\ & \quad \text{for at least one } d \in \mathcal{I}[s] \\ \mathrm{f} & \text{otherwise.} \end{cases}$$

Model (1)

Definition

- If $\langle \mathcal{I}, \mathcal{A} \rangle[F] = \mathrm{t}$, one also writes $\langle \mathcal{I}, \mathcal{A} \rangle \models F$.
- Let F be a (Σ, ν)-formula.

  A Σ-interpretation $\mathcal{I}$ is a model of the formula F (written $\mathcal{I} \models F$) iff $\langle \mathcal{I}, \mathcal{A} \rangle[F] = \mathrm{t}$ for all variable declarations $\mathcal{A}$.
- If $\mathcal{I} \models F$, one says that $\mathcal{I}$ satisfies F.
- A Σ-interpretation $\mathcal{I}$ is a model of a set Φ of Σ-formulas, written $\mathcal{I} \models \Phi$, iff $\mathcal{I} \models F$ for all $F \in \Phi$.

Model (2)
Definition

- A formula F or set of formulas Φ is called consistent iff there is an interpretation $\mathcal{I}$ and a variable assignment $\mathcal{A}$ such that $(\mathcal{I}, \mathcal{A}) \models F$ (it has a model).

  Otherwise it is called inconsistent.
- A (Σ, ν)-formula F is called a tautology iff for all Σ-interpretations $\mathcal{I}$ and (Σ, ν)-variable assignments $\mathcal{A}$, we have $(\mathcal{I}, \mathcal{A}) \models F$.

Model (3)
Exercise

- Consider the interpretation on Slide 90:
  - $\mathcal{I}[\texttt{person}] = \{\text{Arno}, \text{Birgit}, \text{Chris}\}$.
  - $\mathcal{I}[\texttt{married\_to}] = \{(\text{Birgit}, \text{Chris}),\ (\text{Chris}, \text{Birgit})\}$.
  - $\mathcal{I}[\texttt{male}] = \{(\text{Arno}),\ (\text{Chris})\}$,
    $\mathcal{I}[\texttt{female}] = \{(\text{Birgit})\}$.
- Which of the following formulas are true in $\mathcal{I}$?
  - ☐ $\forall\, \texttt{person X}: \texttt{male(X)} \leftrightarrow \neg\texttt{female(X)}$
  - ☐ $\forall\, \texttt{person X}: \texttt{male(X)} \vee \neg\texttt{male(X)}$
  - ☐ $\exists\, \texttt{person X}: \texttt{female(X)} \wedge \neg\exists\, \texttt{person Y}: \texttt{married\_to(X,Y)}$
  - ☐ $\exists\, \texttt{person X}, \texttt{person Y}, \texttt{person Z}:\ \texttt{X} = \texttt{Y} \wedge \texttt{Y} = \texttt{Z} \wedge \texttt{X} \neq \texttt{Z}$

Outline

Databases and Logic (1)

Data values

- The DBMS defines a datatype signature $\Sigma_{\mathcal{D}}$ together with an interpretation $\mathcal{I}_{\mathcal{D}}$, to stipulate the following:
  - For each data type (sort), name and a (non-empty) domain of admissible values.
  - Names of constants (e.g. `123`, `'abc'`) interpreted by a corresponding elements in their data type domain.
  - Names of functions on data types (e.g. `+`, `strlen`) together with their domain and range sorts, interpreted by corresponding functions on domain and range.
  - Names of predicates on data types (e.g. `<`, `odd`), interpreted by corresponding relations on domain and range.

Databases and Logic (2)

- Two formal query languages for the relational model:
  - tuple calculus (with variables for whole tuples)
  - domain calculus (with variables for data values)
- Both are rooted in mathematical logic and portray formal perspectives on relational models.
- Both are equivalent in expressive power but use different logical constructs.
- We consider tuple and domain calculus as restricted variants of first order logic.
- The relational model is formally embedded in first order logic.
- This embedding determines the required restrictions on signatures and interpretations.

Relational Databases (1)

- In relational databases, data is stored as tables, e.g.

Student		
SID	FirstName	LastName
101	Lisa	Weiss
102	Michael	Schmidt
103	Daniel	Sommer
104	Iris	Meier

- Rows are often seen formally as “tuples”.

Relational Databases (2)

- In logic, we can formally define the access to table rows in two different ways:
  - Domain Calculus (DC)
    A table with n rows corresponds to an n-ary predicate:
    $p(t_1, \ldots, t_n)$ is true iff

t_1	$\cdots$	t_n

    is a row in the table.
  - Tuple Calculus (TC)
    A table with n rows corresponds to a sort with n (access) functions that map to the values of the columns.

Relational Databases (3)

Example

- DC would use a predicate `student`:
  - `student(101, 'Lisa', 'Weiss')` would be true
  - `student(200, 'Martin', 'Mueller')` false
- TC would use a sort `student` accompanied by (access) functions `sid`, `first_name`, `last_name`.
  - For an `X` of sort `student`, we then have that: `sid(X)=101`, `first_name(X)='Lisa'`, and `last_name(X)='Weiss'`.

Relational Databases (1)

Domain Calculus

- A DBMS defines a set of data types, such as strings and numbers, together with constants, data type functions (e.g. $+$) and predicates (e.g. $<$).
- For these, the DBMS defines names (in a signature Σ) and their meaning (in an interpretation $\mathcal{I}$).
- For every value $d \in \mathcal{I}[s]$, there is at least one constant c with $\mathcal{I}[c] = d$.

Relational Databases (2)

Domain Calculus

- The DB schema in the relational model then adds further predicate symbols (relation symbols).
- The DB state interprets these by finite relations.

- Thus, the main restrictions of the relational model are:
  - No new sorts (types),
  - No new function symbols and constants,
  - New predicate symbols can only be interpreted by finite relations.

Relational Databases (3)

Domain Calculus, Example

- In a relational database for storing homework results, there might be three predicates/relations:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`

Relational Databases (4)
Domain Calculus

Student

SID	FirstName	LastName
101	Lisa	Weiss
102	Michael	Schmidt
103	Daniel	Sommer
104	Iris	Meier

Exercise

ENO	MaxPt
1	10
2	10

Result

SID	ENO	Points
101	1	10
101	2	8
102	1	9
102	2	9
103	1	5

Relational Database (1)
Tuple Calculus

- In TC, a DBMS also defines a set of data types (with constants, functions, predicates).
- The DB schema adds the following:
  - sorts, one per relation (table)
  - unary functions, each mapping from a sort to a data type, one function per column

Relational Databases (2)

Tuple Calculus, Example

- E.g, in the exercise-results DB there is sort `student` with the functions
  - `sid(student): int`
  - `first_name(student): string`
  - `last_name(student): string`

Relational Databases (3)

Tuple Calculus

- E.g. $\mathcal{I}[\texttt{student}]$ contains tuple

$$t = (101, ''Lisa'', ''Weiss'')$$

- Then, we have $\mathcal{I}[\texttt{sid}](t) = 101$.
- As usual, these new sorts are also finite (possibly empty) sets.

Formulas in Databases

- The DBMS defines a signature $\Sigma_{\mathcal{D}}$ and an interpretation $\mathcal{I}_{\mathcal{D}}$ for the built-in data types (`string`, `int`, ...).
- Then the database schema extends $\Sigma_{\mathcal{D}}$ to the signature Σ of all symbols that can be used in, e.g., queries.
- A database state is then an interpretation $\mathcal{I}$ for the extended signature Σ.
- Formulas are used in databases as:
  - Integrity constraints
  - Queries
  - Definitions of derived symbols (views).

Integrity Constraints (1)

- Not all interpretations are reasonable DB states.
- For instance, in the old world, a person could only be male or female, but not both.
  Therefore, the following two formulas must be satisfied:
  - $\forall\,\mathtt{person}\;\mathtt{X}\colon\; \mathtt{male}(\mathtt{X}) \lor \mathtt{female}(\mathtt{X})$
  - $\forall\,\mathtt{person}\;\mathtt{X}\colon\; \neg\,\mathtt{male}(\mathtt{X}) \lor \neg\,\mathtt{female}(\mathtt{X})$
- These are examples of integrity constraints.

Integrity Constraints (2)

- An integrity constraint is a closed formula.
- A set of integrity constraints is specified as part of the database schema.
- A database state (an interpretation) is called valid iff it satisfies all integrity constraints.

Integrity Constraints (3)

Keys I

- Objects are often identified by unique data values (numbers, names).
- For example, there should never be two different objects of type `student` with the same `sid` (in TC):

$$\forall\, \texttt{student X},\ \texttt{student Y}\colon\ \texttt{sid(X)} = \texttt{sid(Y)} \ \rightarrow\ \texttt{X} = \texttt{Y}$$

- Alternative, equivalent formulation:

$$\neg\,\exists\, \texttt{student X},\ \texttt{student Y}\colon\ \texttt{sid(X)} = \texttt{sid(Y)} \ \wedge\ \texttt{X} \neq \texttt{Y}$$

Integrity Constraints (4)
Keys II

- In the relational schema (in DC on Slide 92) a predicate of arity 3 is used to store the student data.
- The first argument (SID) uniquely identifies the values of the other arguments (first name, last name):

$$\begin{array}{l}\forall\, \texttt{int ID},\ \texttt{string F1},\ \texttt{string F2},\ \texttt{string L1},\ \texttt{string L2}:\\ \qquad \texttt{student}(\texttt{ID},\texttt{F1},\texttt{L1}) \wedge \texttt{student}(\texttt{ID},\texttt{F2},\texttt{L2}) \ \rightarrow\\ \qquad\qquad \texttt{F1} = \texttt{F2}\ \wedge\ \texttt{L1} = \texttt{L2}\end{array}$$

- Since keys are so common, each data model has a special notation for them (one does not actually have to write such formulas).

Queries (1)
Domain Calculus

- In DC, a query is an expression of the form

  $$\{s_1\,X_1, \ldots, s_n\,X_n \mid F\},$$

  where F is a formula for the given DB signature Σ and the variable declaration $\{X_1/s_1, \ldots, X_n/s_n\}$.
- The query asks for all variable assignments $\mathcal{A}$ for the result variables $X_1, \ldots, X_n$ that make the formula F true in the given database state $\mathcal{I}$.

Queries (2)
Domain Calculus, Examples I

- Consider the schema on Slide 92:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`
- Who got at least 8 points for Homework 1?

$$\{\texttt{string FName}, \texttt{string LName} \mid \exists\, \texttt{int SID}, \texttt{int P}: \\ \texttt{student}(\texttt{SID}, \texttt{FName}, \texttt{LName}) \wedge \\ \texttt{result}(\texttt{SID}, 1, \texttt{P}) \;\wedge\; \texttt{P} \geq 8\}$$

Queries (1)

Domain Calculus, Examples I

- The formulas `student(S, FirstName, LastName)` and `result(S, 1, P)` correspond to the table lines:

Student		
SID	FirstName	LastName
S	FirstName	LastName

Result		
SID	ENO	Points
S	1	P

- By the same variable `S` the entries are “joined” in the two tables. They must refer to the same student.

Queries (3)
Domain Calculus, Examples II

- Print all results for Ann Smith:

$$\{\texttt{int ENO}, \texttt{ int Points} \mid \exists\, \texttt{int SID}: \\ \texttt{student}(\texttt{SID}, '\texttt{Ann}', '\texttt{Smith}') \land \\ \texttt{result}(\texttt{SID}, \texttt{ENO}, \texttt{Points})\}$$

- Who has not yet submitted Exercise 2?

$$\{\texttt{string FName}, \texttt{ string LName} \mid \\ \exists\, \texttt{int SID}: \texttt{student}(\texttt{SID}, \texttt{FName}, \texttt{LName}) \land \\ \neg \exists\, \texttt{int P}: \texttt{result}(\texttt{SID}, 2, \texttt{P})\}$$

Queries (1)
Tuple Calculus

- In TC, a query is an expression of the form

$$\{t_1, \ldots, t_k\ [s_1\ X_1, \ldots, s_n\ X_n] \mid F\},$$

  where F is a formula and the t_i are terms for the given DB signature Σ and the variable declaration $\{X_1/s_1, \ldots, X_n/s_n\}$.
- The DBMS will print the values $\langle \mathcal{I}, \mathcal{A} \rangle[t_i]$ of the terms t_i for every variable assignments $\mathcal{A}$ for the result variables $X_1, \ldots, X_n$ such that $\langle \mathcal{I}, \mathcal{A} \rangle \models F$.

Queries (2)

Tuple Calculus, Example

- Consider the schema on Slide 136 in TC:
  - Sort `student` with access functions for rows:
    `sid(student): int`,
    `first_name(student): string`,
    `last_name(student): string`.
  - Sort `result` with functions `sid`, `eno`, `points`.
  - Sort `exercise` with functions `eno`, `maxpt`.

Queries (3)
Tuple Calculus

- Who has at least 8 points for homework 1?

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\ \exists\,\texttt{result R}:\ \texttt{R.eno} = 1 \wedge \\ \texttt{R.sid} = \texttt{S.sid} \wedge \texttt{R.points} \geq 8\}$$

- Variables run in tuple calculus over table rows (tuples).
- Equations are typically used to link table rows.

Queries (4)

Tuple Calculus

- We could have formulated the question with a variable for the task itself (who has at least 8 points for homework 1):

$$
\begin{aligned}
&\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\
&\quad \exists\, \texttt{result R}, \texttt{exercise E}: \\
&\qquad \texttt{E.eno} = 1 \land \texttt{R.eno} = \texttt{E.eno} \land \\
&\qquad \texttt{R.sid} = \texttt{S.sid} \land \texttt{R.points} \geq 8\}
\end{aligned}
$$

- This is logically equivalent.

Queries (5)
Tuple Calculus, Example II

- Who hasn't submitted exercise 2 yet?

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\ \neg\exists\,\texttt{result R}:\ \texttt{R.sid} = \texttt{S.sid} \land \texttt{R.eno} = 2\}$$

- Other possible solution:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\ \forall\,\texttt{result R}:\ \texttt{R.sid} = \texttt{S.sid} \rightarrow \texttt{R.eno} \neq 2\}$$

- Another solution:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\ \forall\,\texttt{result R}:\ \texttt{R.eno} = 2 \rightarrow \texttt{R.sid} \neq \texttt{S.sid}\}$$

Queries (6)
Tuple Calculus

- The tuple calculus is very close to SQL.
  E.g. who has ≥ 8 points for homework 1?

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S},\ \texttt{result R}] \mid \\ \texttt{R.eno} = 1 \land \\ \texttt{R.sid} = \texttt{S.sid} \land \texttt{R.points} \geq 8\}$$

- Same query in SQL:

```
SELECT S.FirstName, S.LastName
FROM   Student S, Result R
WHERE  R.ENO = 1
AND    R.SID = S.SID
AND    R.Points >= 8
```

Queries (7)
Tuple Calculus

- Variant with explicit existential quantifier:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\ \exists\, \texttt{result R}:\ \texttt{R.eno} = 1 \land \\ \texttt{R.sid} = \texttt{S.sid} \land \texttt{R.points} \geq 8\}$$

- A subquery corresponds to this in SQL:

```
SELECT S.FirstName, S.LastName
FROM   Student S
WHERE  EXISTS (SELECT *
               FROM   Result R
               WHERE  R.ENO = 1
               AND    R.SID = S.SID
               AND    R.Points >= 8)
```

Boolean Queries

- A Boolean query is a closed formula F.
- The system prints "yes" if $\mathcal{I} \models F$ and "no" otherwise.

Outline

Implication

Definition/Notation

- A formula or set of formulas Φ (logically) implies a formula or set of formulas G iff every model $\langle \mathcal{I}, \mathcal{A} \rangle$ of Φ is also a model of G.
- In this case we write $\Phi \vdash G$.

Equivalence (1)

Definition

- Two (sets of) (Σ, ν)-formulas F_1 and F_2 are (logically) equivalent iff for every Σ-interpretation $\mathcal{I}$ and every $(\mathcal{I}, \nu)$-variable assignment $\mathcal{A}$

$$(\mathcal{I}, \mathcal{A}) \models F_1 \iff (\mathcal{I}, \mathcal{A}) \models F_2.$$

- In this case we write $F_1 \equiv F_2$.

Equivalence (2)

- F_1 and F_2 are equivalent iff $F_1 \vdash F_2$ and $F_2 \vdash F_1$.
- "Equivalence" of formulas is an equivalence relation, i.e. it is reflexive, symmetric, and transitive.
- Suppose that G_1 results from G_2 by replacing a subformula F_1 by F_2 and let $F_1 \equiv F_2$.
  Then $G_1 \equiv G_2$.
- If $F \vdash G$, then $F \wedge G \equiv F$.

Some Equivalences (1)

- Commutativity (for and, or, iff):
  - $F \land G \equiv G \land F$
  - $F \lor G \equiv G \lor F$
  - $F \leftrightarrow G \equiv G \leftrightarrow F$
- Associativity (for and, or, iff):
  - $F_1 \land (F_2 \land F_3) \equiv (F_1 \land F_2) \land F_3$
  - $F_1 \lor (F_2 \lor F_3) \equiv (F_1 \lor F_2) \lor F_3$
  - $F_1 \leftrightarrow (F_2 \leftrightarrow F_3) \equiv (F_1 \leftrightarrow F_2) \leftrightarrow F_3$

Some Equivalences (2)

- Distribution Law:
  - $F \land (G_1 \lor G_2) \equiv (F \land G_1) \lor (F \land G_2)$
  - $F \lor (G_1 \land G_2) \equiv (F \lor G_1) \land (F \lor G_2)$
- Double Negation:
  - $\neg(\neg F) \equiv F$
- De Morgan's Law:
  - $\neg(F \land G) \equiv (\neg F) \lor (\neg G)$.
  - $\neg(F \lor G) \equiv (\neg F) \land (\neg G)$.

Some Equivalences (3)

- Replacements of Implication Operators:
  - $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (F \leftarrow G)$
  - $F \leftarrow G \equiv G \rightarrow F$
  - $F \rightarrow G \equiv \neg F \vee G$
  - $F \leftarrow G \equiv F \vee \neg G$
- Together with De Morgan's Law this means that e.g. $\{\neg, \vee\}$ are sufficient, all other logical junctors $\{\wedge, \leftarrow, \rightarrow, \leftrightarrow\}$ can be expressed with them.

Some Equivalences (4)

- Removing Negation:
  - $\neg(t_1 < t_2) \equiv t_1 \geq t_2$
  - $\neg(t_1 \leq t_2) \equiv t_1 > t_2$
  - $\neg(t_1 = t_2) \equiv t_1 \neq t_2$
  - $\neg(t_1 \neq t_2) \equiv t_1 = t_2$
  - $\neg(t_1 \geq t_2) \equiv t_1 < t_2$
  - $\neg(t_1 > t_2) \equiv t_1 \leq t_2$

Some Equivalences (5)

- Law of the excluded middle:
  - $F \vee \neg F \quad \equiv \quad \top \quad$ (always true)
  - $F \wedge \neg F \quad \equiv \quad \bot \quad$ (always false)
- Simplifications of formulas with logical constants $\top$ (true) and $\bot$ (false):
  - $F \wedge \top \equiv F \quad F \wedge \bot \equiv \bot$
  - $F \vee \top \equiv \top \quad F \vee \bot \equiv F$
  - $\neg \top \equiv \bot \quad \neg \bot \equiv \top$

Some Equivalences (6)

- Replacements for quantifiers:
  - $\forall s\, X\colon\ F \equiv \neg(\exists s\, X\colon\ (\neg F))$
  - $\exists s\, X\colon\ F \equiv \neg(\forall s\, X\colon\ (\neg F))$
- Moving logical junctors over quantifiers:
  - $\neg(\forall s\, X\colon\ F) \equiv \exists s\, X\colon\ (\neg F)$
  - $\neg(\exists s\, X\colon\ F) \equiv \forall s\, X\colon\ (\neg F)$
  - $\forall s\, X\colon\ (F \wedge G) \equiv (\forall s\, X\colon\ F) \wedge (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \vee G) \equiv (\exists s\, X\colon\ F) \vee (\exists s\, X\colon\ G)$

Some Equivalences (7)

- Moving quantifiers: If $X \notin \mathit{free}(F)$:
  - $\forall s\, X\colon\ (F \vee G) \equiv F \vee (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \wedge G) \equiv F \wedge (\exists s\, X\colon\ G)$

  If in addition $\mathcal{I}[s]$ cannot be empty:
  - $\forall s\, X\colon\ (F \wedge G) \equiv F \wedge (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \vee G) \equiv F \vee (\exists s\, X\colon\ G)$
- Removing unnecessary quantifiers: If $X \notin \mathit{free}(F)$ and $\mathcal{I}[s]$ cannot be empty:
  - $\forall s\, X\colon\ F \equiv F$
  - $\exists s\, X\colon\ F \equiv F$

Some Equivalences (8)

- Exchanging quantifiers: If $X \neq Y$:
  - $\forall s_1 X\colon (\forall s_2 Y\colon F) \equiv \forall s_2 Y\colon (\forall s_1 X\colon F)$
  - $\exists s_1 X\colon (\exists s_2 Y\colon F) \equiv \exists s_2 Y\colon (\exists s_1 X\colon F)$
- Renaming bound variables: If $Y \notin \mathit{free}(F)$ and F' results from F by replacing every free occurrence of X in F by Y:
  - $\forall s X\colon F \equiv \forall s Y\colon F'$
  - $\exists s X\colon F \equiv \exists s Y\colon F'$

Some Equivalences (9)

- Equality is an equivalence relation:
  - $t = t \equiv \top$ (reflexivity)
  - $t_1 = t_2 \equiv t_2 = t_1$ (symmetry)
  - $t_1 = t_2 \wedge t_2 = t_3 \equiv t_1 = t_2 \wedge t_2 = t_3 \wedge t_1 = t_3$ (transitivity)
- Compatibility to function and predicate symbols:
  - $f(t_1, \dots, t_n) = t \wedge t_i = t_i' \equiv$
    $f(t_1, \dots, t_{i-1}, t_i', t_{i+1}, \dots, t_n) = t \wedge t_i = t_i'$
  - $p(t_1, \dots, t_n) \wedge t_i = t_i' \equiv$
    $p(t_1, \dots, t_{i-1}, t_i', t_{i+1}, \dots, t_n) \wedge t_i = t_i'$

Normal Forms (1)

Definition

- A formula F is in Prenex Normal Form iff it is closed and has the form

$$\Theta_1\, s_1\, X_1 \;\ldots\; \Theta_n\, s_n\, X_n : \quad G$$

  where $\Theta_1, \ldots, \Theta_n \in \{\forall, \exists\}$ and G is quantifier-free.

- A formula F is in Disjunctive Normal Form iff it is in Prenex Normal Form, and G has the form

$$(G_{1,1} \wedge \cdots \wedge G_{1,k_1}) \;\vee \cdots \vee\; (G_{n,1} \wedge \cdots \wedge G_{n,k_n}),$$

  where each $G_{i,j}$ is an atomic formula or a negated atomic formula.

Normal Forms (2)

- Conjunctive Normal Form is like disjunctive normal form, but G must have the form

$$(G_{1,1} \vee \cdots \vee G_{1,k_1}) \wedge \cdots \wedge (G_{n,1} \vee \cdots \vee G_{n,k_n}).$$

- Under the assumption of non-empty domains, every formula can be equivalently translated into prenex normal form, disjunctive normal form, and conjunctive normal form.

Outline

Motivation

- Functions are often only partially defined e.g.
  - division by 0,
  - square root of a negative number,
  - integer overflow.
- Often, table columns, i.e., attributes of objects are missing values e.g. not every customer
  - has a fax machine or
  - discloses his birthday.
- Hence, partial function are relevant in real-life.

Interpretation

- Formally, a function symbol $f(s_1, \dots, s_n)\colon s$ is interpreted as function

  $$\mathcal{I}[f]\colon\ \mathcal{I}[s_1] \times \cdots \times \mathcal{I}[s_n] \ \rightarrow\ \mathcal{I}[s] \cup \{\mathit{null}\},$$

  where *null* is a designated value (different from all elements in $\mathcal{I}[s]$).
- For term evaluation, *null*-values are propagated in a "bottom-up"-fashion: if a function has argument "*null*", it returns "*null*".

Example (1)

- Suppose we also record the semester of students which might not always be known:

Student			
SID	FirstName	LastName	Semester
101	Lisa	Weiss	3
102	Michael	Schmidt	5
103	Daniel	Sommer	
104	Iris	Meier	3

- Consider the following query:

$$\{\texttt{S.first\_name}, \texttt{S.last\_name} \quad [\texttt{student S}] \mid \texttt{S.semester} \leq 3\}$$

Example (2)

- With SQL semantics, this query would not return Daniel Sommer.
- This is also the case, when querying students in later semesters:

$$\{\texttt{S.first\_name}, \texttt{S.last\_name} \quad [\texttt{student S}] \mid \texttt{S.semester} > 3\}$$

- This is (also in SQL) equivalent to query:

$$\{\texttt{S.first\_name}, \texttt{S.last\_name} \quad [\texttt{student S}] \mid \neg(\texttt{S.semester} \leq 3)\}$$

Example (3)

- Daniel Sommer would also be omitted by the answer of this query:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}]\ | \\ \texttt{S.semester} \leq 3\ \vee\ \neg(\texttt{S.semester} \leq 3)\}$$

- This violates the law of the excluded middle.
- A two-valued logic with truth-values "true" and "false" does not suffice in this situation.
- A third truth-value, "undefined" (or "null"), is required.

Truth of a Formula (1)

- If F is an atomic formula of form $p(t_1, \dots, t_n)$ or $t_1 = t_2$, and one of its argument terms t_i evaluates to *null*, then F evaluates to the third truth value u.
- If F is of form $\neg G$, then its truth value is depends on the truth value of G as follows:

G	$\neg G$
f	t
u	u
t	f

Truth of a Formula (2)

- Logical binary connectives are evaluated as follows:

G_1	G_2	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
f	f	f	f	t	t	t
f	u	f	u	u	t	u
f	t	f	t	f	t	f
u	f	f	u	t	u	u
u	u	u	u	u	u	u
u	t	u	t	u	t	u
t	f	f	t	t	f	f
t	u	u	t	t	u	u
t	t	t	t	t	t	t

Truth of a Formula (3)

- The principle is simple: the truth value u is passed on, if the value of the formula is not already determined by the other input value.
- E.g. is $\mathsf{u} \wedge \mathsf{f} = \mathsf{f}$, because it does not matter whether the left input value is t or f.
- In other words: A partial condition, which evaluates to u, should not affect the overall truth value as much as possible.

Truth of a Formula (3)

- An existential statement $\exists\, s\, X\colon G$ is true under $\langle \mathcal{I}, \mathcal{A} \rangle$, iff there exists a value in $d \in \mathcal{I}[s]$ such that

$$\langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle [G] = \mathrm{t}.$$

- Otherwise, false in SQL semantics.
- *null* must not be substituted for X: $\mathit{null} \notin \mathcal{I}[s]$.

Truth of a Formula (4)

- Accordingly, a universal statement $\forall\, s\, X\colon G$ is true under $\langle \mathcal{I}, \mathcal{A} \rangle$ iff for each $d \in \mathcal{I}[s]$:
  - $\langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathrm{t}$ or
  - $\langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathrm{u}$.
- Such a statement is only false if there exists an variable assignment $\mathcal{A}'$ such that $\langle \mathcal{I}, \mathcal{A}' \rangle[G] = \mathrm{f}$, where $\mathcal{A}'$ only differs from $\mathcal{A}$ by the value assigned to X.
- Hence, it holds: $\forall\, s\, X\colon G \quad \equiv \quad \neg\exists\, s\, X\, \neg G$.

Equivalences

- Note that some equivalences from two-valued logic do not apply:
  - $t = t$ is not a tautology: if $t = \mathit{null}$, the equation evaluates to u.
  - $F \vee \neg F$ (law of excluded middle).
  - Equivalences with quantifiers that require $\mathcal{I}[s]$ to be not empty.

Check for Null

- To check whether a term evaluates to *null*, one needs another form of atomic formulas.
- t `is null` is true (t) under $\langle \mathcal{I}, \mathcal{A} \rangle$ iff $\langle \mathcal{I}, \mathcal{A} \rangle[t] = \mathit{null}$ and false (f), otherwise.
- For better readability, we may also write t `is not null` instead of $\neg(t$ `is null`$)$.

Total Functions

- Since all functions are partial in this context, one has to explicitly enforce by integrity constraints that a distinct function is total.
- e.g. the last name of students is mandatory

  ∀ `student S: S.last_name is not null`.

- Since this is very common, data models usually provide a shorthand, e.g. in SQL we can declare a table row as "`NOT NULL`".

Outline

Summary

- Signature and formulas
- Interpretations and models
- Database signature
  - Datatype signature
  - Domain calculus
  - Tuple calculus
- Integrity constraints (and keys)
- Queries
- Equivalence
- Incomplete information

Relational model: Overview

Outline

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.
    - `ENO`: Exercise number (within category).
    - `TOPIC`: Topic of the exercise.
    - `MAXPT`: Max. no. of points (How many points is it worth?).
- `RESULTS`: one row for each submitted solution to an exercise.
    - `SID`: Student who wrote the solution.
    - `CAT`, `ENO`: Identification of the exercise.
    - `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).
- 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.
- Extensible DBMS allow the user to define new data types (e.g. multimedia and geometric data types).
- This extensibility is one important feature of modern object-relational systems.

Data Values (2)

- As explained in Part 2 (Logic), 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
  - for each $D \in \mathcal{S}_{\mathcal{D}}$ a set $\mathit{val}(D)$ of possible values of that type ($\mathit{val}(D) := \mathcal{I}_{\mathcal{D}}[D]$).
- 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:

$$\begin{tikzcd}
\begin{array}{c}\text{Column}\\ \text{``EXERCISES.ENO''}\end{array} \arrow[dr] & & \\
& \begin{array}{c}\text{Domain}\\ \text{``EX\_NUMBER''}\end{array} \arrow[r] & \begin{array}{c}\text{Data Type}\\ \text{``NUMERIC(2)''}\end{array} \\
\begin{array}{c}\text{Column}\\ \text{``RESULTS.ENO''}\end{array} \arrow[ur] & &
\end{tikzcd}$$

- 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.

Atomic Attribute Values

- 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.

- In contrast, the NF^2 ("Non First Normal Form") data model allows table entries to be complete tables in themselves.
- Support for "complex values" (sets, lists, records, nested tables) is another typical feature of object-relational systems.

Relational DB Schemas (1)

- A relation schema ρ (schema of a single relation) defines
  - a (finite) sequence $A_1 \dots A_n$ of attribute names,
  - for each attribute A_i a data type (or domain) D_i.
- A relation schema can be written as

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

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).
- That is, $\mathcal{R} = \big(\{R_1, \ldots, R_m\},\ sch,\ \mathcal{C}\big)$.

- Compared to the definitions in Part 2 (Logic), the attribute names are new.

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).
- For every column (identified by the combination of table name and column name) there is a unique data type.
- The columns within a table are ordered, i.e. there is a first, second, etc. column.
- 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.

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:

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

Schemas: Notation (2)

- 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), ...)
```

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

EXERCISES			
CAT	ENO	TOPIC	MAXPT
⋮	⋮	⋮	⋮

Tuples (1)

- An *n*-tuple is a sequence of *n* values.
- E.g. XY-coordinates are pairs (X, Y) of real numbers.
  Pairs are tuples of length 2 ("2-tuples").
- 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.
- E.g. one row in the example table "`EXERCISES`" is the tuple ('H', 1, 'Rel. Algeb.', 10).

Database States (1)

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

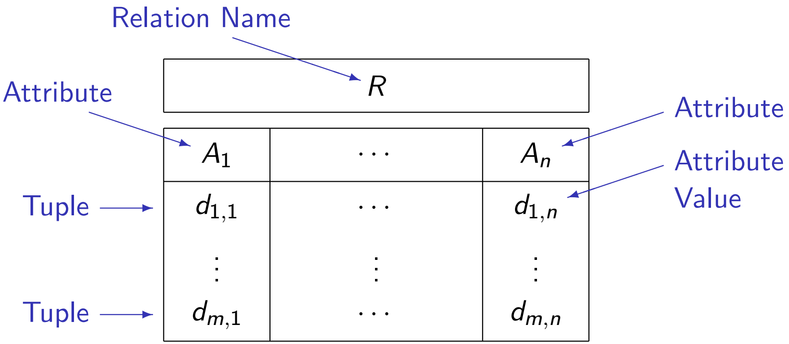
$$\mathcal{I}[R_i]\ \subseteq\ val(D_{i,1})\ \times\ \cdots\ \times\ val(D_{i,n_i}).$$

- I.e. a DB state interprets the symbols in the DB schema: It maps relation names to relations.

Database States (2)

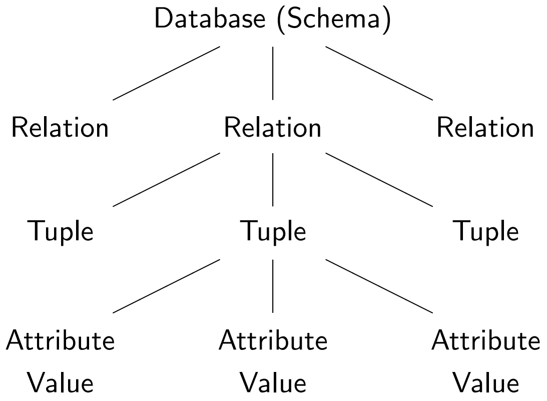
- In mathematics, the term "relation" is defined as "a subset of a cartesian product".
- E.g. an order relation such as "$<$" on the natural numbers is formally: $\{(X, Y) \in \mathbb{N} \times \mathbb{N} \mid X < Y\}$.
- Relations are sets of tuples. Therefore:
  - The sequence of the tuples is undefined.
    - The tabular representation is a bit misleading, there is no first, second, etc. row.
    - 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).

Summary (1)



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

Summary (2)



(∼ Classes)

(∼ Objects)

(Data)

Update Operations (1)

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

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

  - Deletion (of a tuple from a relation):

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

  - Modification / Update (of a tuple):

    $$\begin{aligned}\mathcal{I}_{\mathrm{new}}[R] := \big(\mathcal{I}_{\mathrm{old}}[R] &\setminus \{(d_1, \ldots, d_i, \ldots, d_n)\}\big)\\ &\cup \{(d_1, \ldots, d_i', \ldots, d_n)\}\end{aligned}$$

Update Operations (2)

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

Outline

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 $(A_1\colon D_1, \dots, A_n\colon D_n)$, then

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

- "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.
  - No value exists.
  - The attribute is not applicable to this tuple.
  - 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 (3)

Advantages

- 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.
- If null values are not allowed, users will invent fake values to fill the missing columns.

Null Values (4)
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.
- Most programming languages do not have null values. This complicates application programs.

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.
- 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:

  $$\texttt{STUDENTS(SID, FIRST, LAST, EMAIL}^{o}\texttt{)}$$

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

Outline

Constraints

Overview

- (Integrity) Constraints are conditions that every (valid) database state must satisfy
- 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.

Keys: Unique Identification (1)

- A key of a relation R is an attribute/column A that uniquely identifies the tuples/rows in R.
- 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".
- E.g. this relation satisfies the key `FIRST`, `LAST`:

STUDENTS			
SID	FIRST	LAST	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`.
- 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.
- However, the key "LAST" is not satisfied by the example state on Slide 221.
- If the database designer wants to permit this state, the key constraint "LAST" is too strong.

- Once the key "LAST" is excluded, the composed key "FIRST, LAST" becomes again interesting.
- The database designer must now find out whether there could ever be two students in the course with the same first and last name.

Keys: Minimality (3)

- Natural keys nearly always could possibly have exceptions, yet, if exceptions are extremely rare, 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.
  - Advantage Students can be identified in application programs by first name and last name.

Keys: Minimality (4)

- 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.

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".
- Other keys are called "alternate/secondary keys".

Keys: Notation (1)

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

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

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))
```

Keys and Null Values

- The primary key cannot be null, other keys should not be null.
- 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).
- But SQL does not enforce this constraint.

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

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.

Outline

Foreign Keys (1)

- The relational model has no explicit relationships, links or pointers.
- Values for the key attributes identify a tuple.
- To refer to tuples of R in a relation S, include the primary key of R among the attributes of S.
- 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

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 Part 4 Relational Algebra), the projection $\pi_{\mathtt{SID}}$ returns the values of the column SID. Then the foreign key condition is:

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

Foreign Keys (4)

- The foreign key constraint ensures that for every tuple t in RESULTS there is a tuple u in STUDENTS such that $t.\mathtt{SID} = u.\mathtt{SID}$.
- The key constraint for STUDENTS ensures that there is at most one such tuple u.
- Enforcing foreign key constraints ensures the "referential integrity" of the database.
- 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 (5)

- 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

<u>CAT</u>	<u>ENO</u>	···	MAXPT
H	1	···	10
H	2	···	10
M	1	···	14

Foreign Keys (6)

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

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:

$$\begin{array}{l}\texttt{RESULTS(}\underline{\texttt{SID}} \rightarrow \texttt{STUDENTS,}\\ \qquad\qquad\texttt{(}\underline{\texttt{CAT}}\texttt{, }\underline{\texttt{ENO}}\texttt{)} \rightarrow \texttt{EXERCISES, POINTS)}\\ \texttt{STUDENTS(}\underline{\texttt{SID}}\texttt{, FIRST, LAST, EMAIL)}\\ \texttt{EXERCISES(}\underline{\texttt{CAT}}\texttt{, }\underline{\texttt{ENO}}\texttt{, TOPIC, MAXPT)}\end{array}$$

- 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.

  $\texttt{COURSE\_CATALOG(}\underline{\texttt{NO}}\texttt{, TITLE, DESCRIPTION)}$

  $\texttt{COURSE\_OFFER(}\underline{\texttt{CRN}}\texttt{, CRSNO} \rightarrow \texttt{COURSE\_CATALOG, TERM,}$

  $\qquad\qquad\texttt{(INST\_FIRST, INST\_LAST)} \rightarrow \texttt{FACULTY)}$

  $\texttt{FACULTY(}\underline{\texttt{FIRST}}\texttt{, }\underline{\texttt{LAST}}\texttt{, OFFICE, PHONE)}$

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)
```

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.

More about Foreign Keys (2)

Mutual References

- It is possible that parent and child are the same table, e.g.

  EMP($\underline{\texttt{EMPNO}}$, ENAME, JOB, MGR$^{o}\rightarrow$EMP, DEPTNO$\rightarrow$DEPT)
  PERSON($\underline{\texttt{NAME}}$, MOTHER$^{o}\rightarrow$PERSON, FATHER$^{o}\rightarrow$PERSON)

- Two relations can reference each other, e.g.

  EMPLOYEES($\underline{\texttt{EMPNO}}$, ..., DEPT$\rightarrow$DEPARTMENTS)
  DEPARTMENTS($\underline{\texttt{DNO}}$, ..., LEADER$^{o}\rightarrow$EMPLOYEES).

Please Remember

- Foreign keys are not themselves keys!
- 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`.
- Update of the key `SID` of the parent table `STUDENTS` when the old value is still referenced.

Foreign Keys and Updates (2)

- Deletions from `RESULTS` (child table) and insertions into `STUDENTS` (parent table) can never lead to a violation of the foreign key constraint.
- The insertion of dangling references 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.
- The foreign key is set to a declared default value.

Foreign Keys and Updates (4)

Reactions on Updates of Referenced Key Values

- The update is rejected. The DB state remains unchanged.
- The update cascades.
- 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.
- Other alternatives exist only in few systems at the moment.

Outline

Summary

- Relational database schemas
- Relational database states
- Update operations
- Null values
- Keys and foreign keys

Relational algebra: Overview

Outline

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" (primary key).
  - `FIRST`, `LAST`: First and last name.
  - `EMAIL`: Email address (can be null).
- `EXERCISES`: one row for each graded exercise.
  - `CAT`: Exercise category (key together with `ENO`).
  - `ENO`: Exercise number (within category).
  - `TOPIC`: Topic of the exercise.
  - `MAXPT`: Max. no. of points (How many points is it worth?).
- `RESULTS`: one row for each submitted solution to an exercise.
  - `SID`: Student who wrote the solution.
  - `CAT`, `ENO`: Identification of the exercise.
  - `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.

Relational Algebra (1)

- Relational algebra (RA) is a theoretical query language for the relational model.
- Relational algebra is not used in any commerical system on the user interface level.
- However, variants of it are used to represent queries internally (for query optimization and execution).
- Knowledge of relational algebra will help in understanding SQL and relational database systems.

Relational Algebra (2)

- An algebra is a set together with operations on this set.
- For instance, the set of integers together with the operations $+$ and $*$ forms an algebra.
- In the case of relational algebra, the set is the set of all finite relations.
- One operation of relational algebra is $\cup$ (union).
  This is natural since relations are sets.

Relational Algebra (3)

- Another operation of relational algebra is selection.
- E.g. $\sigma_{\texttt{SID=101}}$ selects all tuples in the input relation that have the value "101" in column "SID":

$\sigma_{\texttt{SID=101}}$ (

RESULTS

<u>SID</u>	<u>CAT</u>	<u>ENO</u>	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

) =

<u>SID</u>	<u>CAT</u>	<u>ENO</u>	POINTS
101	H	1	10
101	H	2	8
101	M	1	12

Relational Algebra (4)

- Since the output of a relational algebra operation is again a relation, it can be input for another relational algebra operation.
- A query is then a term/expression in this algebra.
- Arithmetic expressions like $(\mathtt{x} + 2) * \mathtt{y}$ are familiar.
- In relational algebra, relations are connected:

$$\pi_{\mathsf{FIRST},\mathsf{LAST}}(\mathsf{STUDENTS} \bowtie \sigma_{\mathsf{CAT}='\mathsf{M}'}(\mathsf{RESULTS})).$$

Relational Algebra (5)

- Null values are usually excluded in the definition of relational algebra, except when operations like the outer join are defined.
- Relational algebra treats relations as sets, i.e. any duplicate tuples are automatically eliminated.
- Relational algebra is much simpler than SQL, it has only five basic operations and can be completely defined on one page.
- Relational algebra is also a yardstick for measuring the expressiveness of query languages.
- E.g., every query that can be formulated in relational algebra can also be formulated in SQL.

Selection (1)

- The operation σ_φ selects a subset of the tuples of a relation, namely those which satisfy the condition φ.
  Selection acts like a filter on the input set.
- Example:

$$\sigma_{A=1}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 1 & 4 \\ 2 & 5 \\ \hline\end{array}\right) = \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 1 & 4 \\ \hline\end{array}$$

- The selection condition has the following form:

  ⟨Term⟩ ⟨Comparison-Operator⟩ ⟨Term⟩

- The selection condition returns a Boolean value (true or false) for a given input tuple.

Selection (2)

- ⟨Term⟩ (or "expression") is something that can be evaluated to a data type element for a given tuple:
  - an attribute name,
  - a data type constant, or
  - an expression composed from attributes and constants by data type operations like $+$, $-$, $*$, $/$.
- ⟨Comparison-Operator⟩ is
  - $=$ (equals), $\neq$ (not equals),
  - $<$ (less than), $>$ (greater than), $\leq$,
  - or other data type predicates (e.g. `LIKE`).
- Examples for Conditions:
  - `LAST = 'Smith'`
  - `POINTS >= 8`
  - `POINTS = MAXPT`

Selection (3)

- Of course, the attributes used in the selection condition must appear in the input table:

$$\sigma_{C=1}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) \quad = \quad \textbf{Error}$$

- The following is legal, but the selection is superfluous, because the condition is always true:

$$\sigma_{A=A}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) \quad = \quad \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}$$

Selection (4)

- It is no error if the result of a relational algebra expression happens to be empty in a specific state:

$$\sigma_{A=3}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) \quad = \quad \emptyset$$

- It is legal, but most probably an error, to use a condition that is always false (inconsistent):

$$\sigma_{1=2}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) \quad = \quad \emptyset$$

Selection (5)

- $\sigma_{\varphi}(\mathtt{R})$ corresponds to the following SQL query:

```
SELECT *
FROM   R
WHERE  φ
```

- I.e. selection corresponds to the `WHERE`-clause.
- A different relational algebra operation called “projection” corresponds to the `SELECT`-clause in SQL.
  This can be slightly confusing.

Extended Selection (1)

- In the basic selection operation, only simple conditions consisting of a single comparison ("atomic formula") are possible.
- However, one can extend the possible conditions by permitting to combine the single conditions by the logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not):

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$	$\varphi_1 \vee \varphi_2$	$\neg\varphi_1$
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Extended Selection (2)

- $\varphi_1 \land \varphi_2$ is called the “conjunction of φ_1 and φ_2”
- $\varphi_1 \lor \varphi_2$ is called the “disjunction of φ_1 and φ_2”
- $\neg\varphi_1$ is called the “negation of φ_1”.
- One can write “`and`”, “`or`” and “`not`” instead of the symbols “$\land$”, “$\lor$”, “$\neg$” used in logic.

Extended Selection (3)

- The selection condition must permit evaluation for each input tuple in isolation.
- This extended form of selection is not necessary, since it can always be expressed with the basic operations of relational algebra.
  But it is convenient.
- E.g. $\sigma_{\varphi_1 \wedge \varphi_2}(\mathtt{R})$ is equivalent to $\sigma_{\varphi_1}\big(\sigma_{\varphi_2}(\mathtt{R})\big)$.

Projection (1)

- The projection π eliminates attributes (columns) from the input relation.
- Example:

$$\pi_{A,C}\left(\begin{array}{|c|c|c|}\hline A & B & C \\ \hline 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \\ \hline\end{array}\right) = \begin{array}{|c|c|}\hline A & C \\ \hline 1 & 7 \\ 2 & 8 \\ 3 & 9 \\ \hline\end{array}$$

Projection (2)

- In general, the projection $\pi_{\mathtt{A}_{\mathtt{i}_1},\dots,\mathtt{A}_{\mathtt{i}_k}}(\mathtt{R})$ produces for each input tuple $(\mathtt{A}_1 : \mathtt{d}_1, \dots, \mathtt{A}_\mathtt{n} : \mathtt{d}_\mathtt{n})$ an output tuple $(\mathtt{A}_{\mathtt{i}_1} : \mathtt{d}_{\mathtt{i}_1}, \dots, \mathtt{A}_{\mathtt{i}_k} : \mathtt{d}_{\mathtt{i}_k})$.
- I.e. the attribute values are not changed, but only the explicitly mentioned attributes are retained.
  All other attributes are "projected away".

Projection (3)

- Normally, there is one output tuple for every input tuple. However, if two input tuples lead to the same output tuple, the duplicate will be eliminated.
- Example:

$$\pi_B\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 4 \\ 2 & 5 \\ 3 & 4 \\ \hline \end{array}\right) = \begin{array}{|c|}\hline B \\ \hline 4 \\ 5 \\ \hline \end{array}$$

Projection (4)

- Attributes can be renamed: $\pi_{\mathtt{B_1 \leftarrow A_{i_1}, \ldots, B_k \leftarrow A_{i_k}}}(\mathtt{R})$ transforms the input tuple $(\mathtt{A_1} : \mathtt{d_1}, \ldots, \mathtt{A_n} : \mathtt{d_n})$ into the output tuple $(\mathtt{B_1} : \mathtt{d_{i_1}}, \ldots, \mathtt{B_k} : \mathtt{d_{i_k}})$.
- Return values can be computed by datatype operations such as $+$ or `||` (string concatenation):

$$\pi_{\mathsf{SID},\ \mathsf{NAME} \leftarrow \mathsf{FIRST}\ ||\ '\ '\ ||\ \mathsf{LAST}}(\mathsf{STUDENTS}).$$

- Columns can be created with constant values:

$$\pi_{\mathsf{SID}, \mathsf{FIRST}, \mathsf{LAST}, \mathsf{GRADE} \leftarrow '\mathsf{A}'}(\mathsf{STUDENTS}).$$

Projection (5)

- The projection is a mapping, which is applied to every input tuple.
- Each input tuple is mapped locally to an output tuple.
- Only functions which are defined based on single input tuples are allowed.

Projection (6)

- $\pi_{A_1,\ldots,A_n}(R)$ corresponds to the SQL query:

```
SELECT A_1, ..., A_n
FROM   R
```

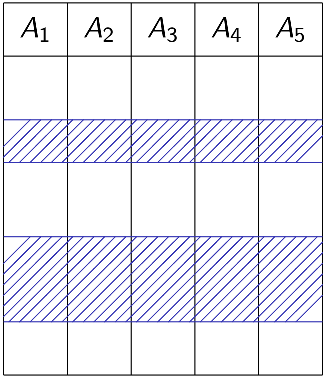
- $\pi_{B_1 \leftarrow A_1,\ldots,B_n \leftarrow A_n}(R)$ is written in SQL as follows:

```
SELECT A_1 AS B_1, ..., A_n AS B_n
FROM   R
```

- The keyword `AS` can be left out ("syntactic sugar").

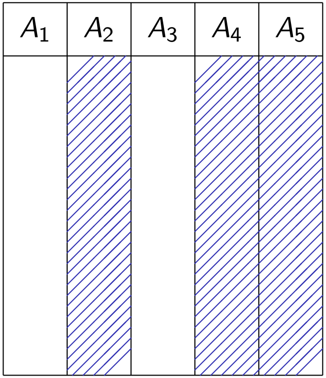
Summary

Selection σ



(Filters some rows)

Projection π



(Maps each row)

Combining Operations (1)

- Since the result of a relational algebra operation is also a relation, it can act as input to another algebra operation.
- For instance, to compute the exercises solved by student 102:

$$\pi_{\mathsf{CAT},\,\mathsf{ENO}}(\sigma_{\mathsf{SID}=102}(\mathsf{RESULTS}))$$

- An intermediate result can be stored in a temporary relation (can also be seen as macro definition):

$$\mathtt{S102} \ := \ \sigma_{\mathtt{SID}=102}(\mathtt{RESULTS});$$
$$\pi_{\mathtt{CAT},\,\mathtt{ENO}}(\mathtt{S102})$$

Combining Operations (2)

- Expressions of relational algebra may become clearer if depicted as operator tree:

$$\begin{tikzcd}
\pi_{\mathtt{CAT},\mathtt{ENO}} \\
\sigma_{\mathtt{SID}=102} \arrow[u] \\
\mathtt{RESULTS} \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & * & \\
 & + \arrow[ur] & & y \arrow[ul] \\
x \arrow[ur] & & 2 \arrow[ul] &
\end{tikzcd}$$

- For comparison, an operator tree for the arithmetic expression $(x + 2) * y$ is shown on the right.

Combining Operations (3)

- In SQL, σ and π (and $\times$, see below) can be combined in a single SELECT-expression:

```
SELECT CAT, ENO
FROM   RESULTS
WHERE  SID = 102
```

- Complex queries can be constructed step by step:

```
CREATE VIEW S102
AS SELECT *
   FROM   RESULTS
   WHERE  SID = 102
```

- Then S102 can be used like a stored table.

Basic Operands

- The leaves of the operator tree are
  - the names of database relations
  - constant relations (explicitly enumerated tuples).
- A relation name R is a legal expression of relational algebra.
  Its value is the entire relation stored under that name.
  It corresponds to the SQL query:

```
SELECT *
FROM   R
```

- It is not necessary to write a projection on all attributes.

Exercises

- Which of the following relational algebra expressions are syntactically correct? What do they mean?
  - ☐ $\mathtt{STUDENTS}$.
  - ☐ $\sigma_{\mathtt{MAXPT} \neq 10}(\mathtt{EXERCISES})$.
  - ☐ $\pi_{\mathtt{FIRST}}(\pi_{\mathtt{LAST}}(\mathtt{STUDENTS}))$.
  - ☐ $\sigma_{\mathtt{POINTS} \leq 5}(\sigma_{\mathtt{POINTS} \geq 1}(\mathtt{RESULTS}))$.
  - ☐ $\sigma_{\mathtt{POINTS}}(\pi_{\mathtt{POINTS} = 10}(\mathtt{RESULTS}))$.

Outline

Cartesian Product (1)

- Often, answer tuples must be computed that are derived from two (or more) input tuples.
- This is done by the cartesian product $\times$.
- $R \times S$ concatenates ("glues together") each tuple from R with each tuple from S.

Cartesian Product (2)

- Example:

A	B
1	2
3	4

$\times$

C	D
6	7
8	9

$=$

A	B	C	D
1	2	6	7
1	2	8	9
3	4	6	7
3	4	8	9

- Since attribute names must be unique within a tuple, the cartesian product may only be applied when R and S have no attribute in common.
- This is no real restriction, since we may rename the attributes first (with π) and then apply $\times$.

Cartesian Product (3)

- We may also define $\times$ such that it automatically renames double attributes:
  - E.g. for relations $R(A, B)$ and $S(B, C)$ the product $R \times S$ has attributes $(R.A, R.B, S.B, S.C)$.
  - As in SQL, one can also use the names A and C, because they uniquely identify the attributes.

Cartesian Product (4)

- If the relation R contains n tuples, and the relation S contains m tuples, then $R \times S$ contains $n * m$ tuples.
- The cartesian product is in itself seldom useful, because it leads to a "blowup" in relation size.
- The problem is that $R \times S$ combines each tuple from R with each tuple from S. Usually, the goal is to combine only selected pairs of tuples.
- Thus, the cartesian product is useful only as input for a following selection.

Cartesian Product (5)

- $R \times S$ is written in SQL as

```
SELECT *
FROM   R, S
```

- In SQL it is no error if the two relations have common attribute names, since one can reference attributes also in the form "$R.A$" or "$S.A$".

Renaming

- An operator $\rho_R(S)$ that prepends "`R.`" to all attribute names is sometimes useful:

$$\rho_R\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 2 \\ 3 & 4 \\ \hline\end{array}\right) \quad = \quad \begin{array}{|r|r|}\hline R.A & R.B \\ \hline 1 & 2 \\ 3 & 4 \\ \hline\end{array}$$

- This is only an abbreviation for an application of the projection: $\pi_{\mathtt{R.A}\leftarrow\mathtt{A},\,\mathtt{R.B}\leftarrow\mathtt{B}}(\mathtt{S})$.
- Otherwise, attribute names in relational algebra do not automatically contain the relation name.

Join (1)

- Since this combination of cartesian product and selection is so common, a special symbol has been introduced for it:

$$R \underset{A=B}{\bowtie} S \quad \text{is an abbreviation for} \quad \sigma_{\mathtt{A=B}}(\mathtt{R} \times \mathtt{S}).$$

- This operation is called "join": It is used to join two tables (i.e. combine their tuples).
- The join is one of the most important and useful operations of the relational algebra.

Join (2)

STUDENTS ⋈ RESULTS						
SID	FIRST	LAST	EMAIL	CAT	ENO	POINTS
101	Ann	Smith	⋯	H	1	10
101	Ann	Smith	⋯	H	2	8
101	Ann	Smith	⋯	M	1	12
102	Michael	Jones	(null)	H	1	9
102	Michael	Jones	(null)	H	2	9
102	Michael	Jones	(null)	M	1	10
103	Richard	Turner	⋯	H	1	5
103	Richard	Turner	⋯	M	1	7

- Student Maria Brown does not appear, because she has not submitted any homework and did not participate in the exam.
- What is shown above, is the natural join of the two tables. However, in the following first the standard join is explained.

Join (3)

- The join condition does not have to take the form $A = B$ (although this is most common). It can be an arbitrary condition, for instance also $A < B$.
- A typical application of a join is to combine tuples based on a foreign key, e.g.

$$\mathtt{RESULTS} \underset{\mathtt{SID}=\mathtt{SID}'}{\bowtie} \pi_{\mathtt{SID}'\leftarrow\mathtt{SID},\mathtt{FIRST},\mathtt{LAST},\mathtt{EMAIL}}(\mathtt{STUDENTS})$$

Join (4)

- The join not only combines tuples, but also acts as a filter: It eliminates tuples without join partner. (Note: Foreign key ensures that join partner exists.)

A	B
1	2
3	4

$\underset{B=C}{\bowtie}$

C	D
4	5
6	7

$=$

A	B	C	D
3	4	4	5

- A “semijoin” ($\ltimes$, $\rtimes$) works only as a filter.
- An “outer join” does not work as a filter: It preserves all input tuples.

Natural Join (1)

- Another useful abbreviation is the "natural join" $\bowtie$.
- It combines tuples which have equal values in attributes with the same name.

A	B
1	2
3	4

$\bowtie$

B	C
4	5
4	8
6	7

$=$

A	B	C
3	4	5
3	4	8

Natural Join (2)

- The natural join of two relations
  - $\mathtt{R}(\mathtt{A}_1,\dots,\mathtt{A}_n,\mathtt{B}_1,\dots,\mathtt{B}_k)$ and
  - $\mathtt{S}(\mathtt{B}_1,\dots,\mathtt{B}_k,\mathtt{C}_1,\dots,\mathtt{C}_m)$

  produces in database state $\mathcal{I}$ all tuples of the form

  $$(\mathtt{a}_1,\dots,\mathtt{a}_n,\mathtt{b}_1,\dots,\mathtt{b}_k,\mathtt{c}_1,\dots,\mathtt{c}_m)$$

  such that
  - $(\mathtt{a}_1,\dots,\mathtt{a}_n,\mathtt{b}_1,\dots,\mathtt{b}_k)\in\mathcal{I}(\mathtt{R})$ and
  - $(\mathtt{b}_1,\dots,\mathtt{b}_k,\mathtt{c}_1,\dots,\mathtt{c}_m)\in\mathcal{I}(\mathtt{S})$.

Natural Join (3)

- The natural join not only corresponds to a cartesian product followed by a selection, but also
  - automatically renames one copy of each common attribute before the cartesian product, and
  - uses a projection to eliminate these double attributes at the end.
- E.g., given $\mathtt{R}(\mathtt{A},\mathtt{B})$, and $\mathtt{S}(\mathtt{B},\mathtt{C})$, then $\mathtt{R} \bowtie \mathtt{S}$ is an abbreviation for

$$\pi_{A,B,C}(\sigma_{B=B'}(R \times \pi_{B'\leftarrow B,\,C}(S))).$$

Natural Join (4)
Relational Database Design

- In order to support the natural join, it is beneficial to give attributes from different relations, which are typically joined together, the same name.
- Even if the utilized query language does not have a natural join, this provides good documentation.
- Try to avoid giving the same name to attributes which will probably not be joined.

Joins in SQL

- $\mathtt{R} \underset{\mathtt{A}=\mathtt{B}}{\bowtie} \mathtt{S}$ is normally written in SQL like $\sigma_{\mathtt{A}=\mathtt{B}}(\mathtt{R} \times \mathtt{S})$:

```
SELECT *
FROM   R, S
WHERE  A = B
```

- Attributes can also referenced with explicit relation name (required if the attribute name appears in both relations):

```
SELECT *
FROM   R, S
WHERE  R.A = S.B
```

Algebraic Laws (1)

- The join satisfies the associativity condition:

$$(R \bowtie S) \bowtie T \;=\; R \bowtie (S \bowtie T).$$

- Therefore, the parentheses are not needed:

$$R \;\bowtie\; S \;\bowtie\; T.$$

- The join is not quite commutative: The sequence of columns (from left to right) will be different.
- However, if a projection follows later, this does not matter (one can also introduce π for this purpose):

$$\pi_{\dots}(R \bowtie S) \;=\; \pi_{\dots}(S \bowtie R).$$

Algebraic Laws (2)

- Further algebraic laws hold, which are utilized in the query optimizer of a relational DBMS.
- E.g., if the condition φ refers only to S, then

$$\sigma_\varphi(R \bowtie S) \;=\; R \bowtie \sigma_\varphi(S).$$

The right hand side can often be evaluated more efficiently (depending on relation sizes, indexes).

A Common Query Pattern (1)

- The following query structure is very common:

$$\pi_{A_1,\dots,A_k}(\sigma_\varphi(R_1 \bowtie \cdots \bowtie R_n)).$$

  - First join all tables which are needed to answer the query.
  - Second, select the relevant tuples.
  - Third, project on the attributes which should appear in the output.

A Common Query Pattern (2)

- Patterns are often useful conventions of thought.
- But relational algebra operations can be combined in any way.
  It is not necessary to adhere to this pattern.
- In contrast, in SQL the keywords `SELECT` and `FROM` are required, and the sequence must always be

```
SELECT ... FROM ... WHERE ...
```

A Common Query Pattern (3)

- $\pi_{\mathtt{A_1,\ldots,A_k}}(\sigma_\varphi(\mathtt{R_1} \bowtie \cdots \bowtie \mathtt{R_n}))$ is written in SQL as:

  SELECT A_1, ..., A_k
  FROM R_1, ..., R_n
  WHERE φ AND $\langle$Join Conditions$\rangle$

- It is a common mistake to forget a join condition.
- Usually, every two relations are linked (directly or indirectly) by equations, e.g. $\mathtt{R_1.B_1} = \mathtt{R_2.B_2}$.

A Common Query Pattern (4)

- To formulate a query, think first about the relations needed:
    - Usually, the natural language version of the query names certain attributes.
    - Each such attribute requires at least one relation which contains this attribute.
    - Finally, sometimes intermediate relations are required in order to make the join meaningful.
    - E.g., suppose that relations $\mathtt{R}(\mathtt{A},\mathtt{B})$, $\mathtt{S}(\mathtt{B},\mathtt{C})$, $\mathtt{T}(\mathtt{C},\mathtt{D})$ are given and attributes `A` and `D` are needed. Then $\mathtt{R} \bowtie \mathtt{T}$ would not be correct.
    - Instead, the join must be $\mathtt{R} \bowtie \mathtt{S} \bowtie \mathtt{T}$.
    - It often helps to have a graphical representation of the foreign key links between the tables (which correspond to typical joins).

Self Joins (1)

- Sometimes, it is necessary to refer to more than one tuple from one relation at the same time.
- E.g. who got more points than student 101 for any exercise?
- In this case, two tuples of the relation `RESULTS` are needed in order to compute one result tuple:
    - One tuple for the student 101.
    - One tuple for the same exercise, in which `POINTS` is greater than in the first tuple.

Self Joins (2)

- This requires a generalization of the above query pattern, where two copies of a relation are joined (at least one must be renamed first).

$$S := \rho_{\mathtt{X}}(\mathtt{RESULTS}) \underset{\substack{\mathtt{X.CAT} = \mathtt{Y.CAT} \\ \land\, \mathtt{X.ENO} = \mathtt{Y.ENO}}}{\bowtie} \rho_{\mathtt{Y}}(\mathtt{RESULTS});$$

$$\pi_{\mathtt{X.SID}}(\sigma_{\mathtt{X.POINTS} > \mathtt{Y.POINTS} \land \mathtt{Y.SID} = \mathtt{101}}(S))$$

- Such joins of a table with itself are sometimes called "self joins".

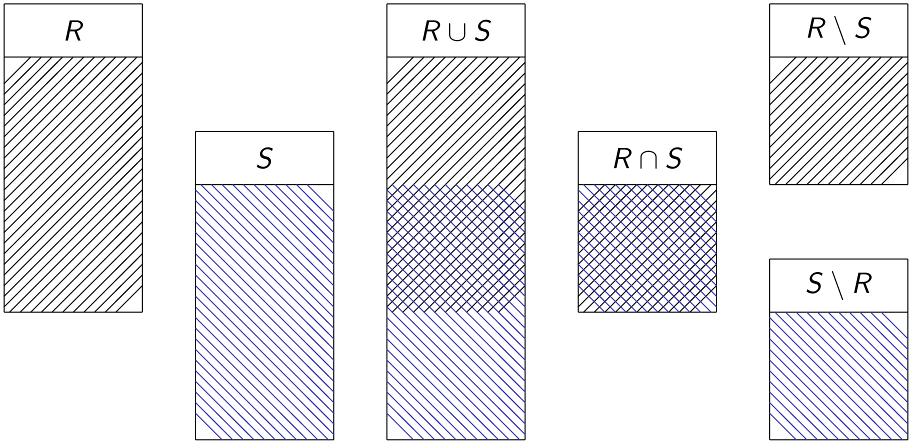
Outline

Set Operations (1)

- Since relations are sets (of tuples), the usual set operations $\cup$, $\cap$, $\setminus$ can also be applied to relations.
- However, both input relations must have the same schema.
- $\mathtt{R} \cup \mathtt{S}$ contains all tuples which are contained in R, in S, or in both relations (Union).
- $\mathtt{R} \setminus \mathtt{S}$ contains all tuples which are contained in R, but not in S (Set Difference).
- $\mathtt{R} \cap \mathtt{S}$ contains all tuples which are contained in both, R and S (Intersection).
- Intersection is (like the join) a derived operation: It can be expressed in terms of $\setminus$:

$$\mathtt{R} \cap \mathtt{S} = \mathtt{R} \setminus (\mathtt{R} \setminus \mathtt{S}).$$

Set Operations (2)



Union (1)

- Without $\cup$, every result column can contain only values from a single column of the stored tables.
- E.g. suppose that besides the registered students, who submit homeworks and write exams, there are also guests that attend the course:

$$\texttt{GUESTS}(\underline{\texttt{FIRST}}, \underline{\texttt{LAST}}, \texttt{EMAIL}^{o}).$$

- The task is to produce a list of email addresses of registered students and guests in one query.

Union (2)

- With $\cup$, this is simple:

$$\pi_{\mathtt{EMAIL}}(\mathtt{STUDENTS}) \cup \pi_{\mathtt{EMAIL}}(\mathtt{GUESTS}).$$

- This query cannot be formulated without $\cup$.
- Another typical application of $\cup$ is a case analysis:

$$\mathtt{MPOINTS} := \pi_{\mathtt{SID},\mathtt{POINTS}}\big(\sigma_{\mathtt{CAT}=\mathtt{'M'}\,\wedge\,\mathtt{ENO}=1}(\mathtt{RESULTS})\big);$$

$$\begin{array}{l}
\quad\pi_{\mathtt{SID},\,\mathtt{GRADE}\leftarrow\mathtt{'A'}}\big(\sigma_{\mathtt{POINTS}\,\geq\,12}(\mathtt{MPOINTS})\big)\\
\cup\,\pi_{\mathtt{SID},\,\mathtt{GRADE}\leftarrow\mathtt{'B'}}\big(\sigma_{\mathtt{POINTS}\,\geq\,10\,\wedge\,\mathtt{POINTS}\,<\,12}(\mathtt{MPOINTS})\big)\\
\cup\,\pi_{\mathtt{SID},\,\mathtt{GRADE}\leftarrow\mathtt{'C'}}\big(\sigma_{\mathtt{POINTS}\,\geq\,7\;\;\wedge\,\mathtt{POINTS}\,<\,10}(\mathtt{MPOINTS})\big)\\
\cup\,\pi_{\mathtt{SID},\,\mathtt{GRADE}\leftarrow\mathtt{'F'}}\big(\sigma_{\mathtt{POINTS}\,<\,7}\;(\mathtt{MPOINTS})\big)
\end{array}$$

Union (3)

- In SQL, `UNION` can be written between two `SELECT`-expressions:

```
SELECT SID, 'A' AS GRADE
FROM   RESULTS
WHERE  CAT = 'M' AND ENO = 1 AND POINTS >= 12
UNION
SELECT SID, 'B' AS GRADE
FROM   RESULTS
WHERE  CAT = 'M' AND ENO = 1
AND    POINTS >= 10 AND POINTS < 12
UNION
...
```

Set Difference (1)

- The operators σ, π, $\times$, $\bowtie$, $\cup$ have a monotonic behaviour, e.g.

$$R \subseteq S \implies \sigma_\varphi(R) \subseteq \sigma_\varphi(S)$$

- Then it follows that also every query Q that uses only the above operators behaves monotonically:
  - Let $\mathcal{I}_1$ be a database state, and let $\mathcal{I}_2$ result from $\mathcal{I}_1$ by the insertion of one or more tuples.
  - Then every tuple t contained in the answer to Q in $\mathcal{I}_1$ is also contained in the answer to Q in $\mathcal{I}_2$.

Set Difference (2)

- If the query must behave nonmonotonically, it is clear that the previous operations are not sufficient, and one must use set difference "$\setminus$". E.g.
  - Which student has not solved any exercise?
  - Who got the most points in Homework 1?
  - Who has solved all exercises in the database?

Set Difference (3)

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

Set Difference (4)

- E.g. which student has not solved any exercise?

$$\mathtt{NO\_SOL} := \pi_{\mathtt{SID}}(\mathtt{STUDENTS}) \setminus \pi_{\mathtt{SID}}(\mathtt{RESULTS});$$

$$\pi_{\mathtt{FIRST},\,\mathtt{LAST}}(\mathtt{STUDENTS} \bowtie \mathtt{NO\_SOL})$$

Set Difference (5)

- When using $\setminus$, a typical pattern is the anti-join.
- E.g. given $\mathtt{R}(\mathtt{A}, \mathtt{B})$ and $\mathtt{S}(\mathtt{B}, \mathtt{C})$, the tuples from R that do not have a join partner in S can be computed as follows:

$$R \bowtie \big(\pi_B(R) \setminus \pi_B(S)\big).$$

- The following is equivalent: $R \setminus \pi_{A,B}(R \bowtie S)$.

Set Difference (6)

- Note that in order for the set difference $R \setminus S$ to be applicable, it is not required that $S \subseteq R$.
- E.g. this query computes the SIDs of students that have solved Homework 2, but not Homework 1:

$$\begin{array}{ll} & \pi_{\texttt{SID}}\big(\sigma_{\texttt{CAT}=\texttt{'H'} \land \texttt{ENO}=2}(\texttt{RESULTS})\big) \\ \setminus & \pi_{\texttt{SID}}\big(\sigma_{\texttt{CAT}=\texttt{'H'} \land \texttt{ENO}=1}(\texttt{RESULTS})\big) \end{array}$$

- It is no problem that there might also be students that have solved Homework 1, but not Homework 2.

Set Difference (7)

- Suppose that R and S are represented in SQL as
  - SELECT A_1, ..., A_n FROM R_1, ..., R_m WHERE φ_1
  - SELECT B_1, ..., B_n FROM S_1, ..., S_k WHERE φ_2
- Then $R \setminus S$ can be represented as

```
SELECT $A_1$, ..., $A_n$
FROM   $R_1$, ..., $R_m$
WHERE  $\varphi_1$ AND NOT EXISTS
       (SELECT * FROM $S_1$, ..., $S_k$
        WHERE  $\varphi_2$
        AND    $B_1 = A_1$ AND ... AND $B_n = A_n$)
```

Outline

Outer Join (1)

- The usual join eliminates tuples without partner:

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2

- The left outer join guarantees that tuples from the left table will appear in the result:

A	B
a_1	b_1
a_2	b_2

$⟕$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	
a_2	b_2	c_2

Outer Join (2)

- The right outer join preserves tuples from the right table:

A	B
a_1	b_1
a_2	b_2

⟖

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2
	b_3	c_3

- The full outer join does not eliminate any tuples:

A	B
a_1	b_1
a_2	b_2

⟗

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	
a_2	b_2	c_2
	b_3	c_3

Outer Join (3)

- E.g. students with their homework results, students without homework result are listed with null values:

$$\mathtt{STUDENTS} ⟕ \pi_{\mathtt{SID,ENO,POINTS}}(\sigma_{\mathtt{CAT='H'}}(\mathtt{RESULTS}))$$

SID	FIRST	LAST	EMAIL	ENO	POINTS
101	Ann	Smith	...	1	10
101	Ann	Smith	...	2	8
102	Michael	Jones	(null)	1	9
102	Michael	Jones	(null)	2	9
103	Richard	Turner	...	1	5
104	Maria	Brown	...	(null)	(null)

Outer Join (4)

- The outer join is a derived operation (like $\bowtie$, $\cap$), i.e. it can be simulated with the five basic relational algebra operations.
- E.g. consider relations $R(A, B)$ and $S(B, C)$.
- The left outer join $R \,⟕\, S$ is an abbreviation for

$$R \bowtie S \quad \cup \quad (R \setminus \pi_{A,B}(R \bowtie S)) \times \{(C\colon \mathit{null})\}$$

(where $\bowtie$ can be further replaced by $\times$, σ, π).

Outline

Syntax (1)

- A set $\mathcal{S}_\mathcal{D}$ of data type names, and for each $D \in \mathcal{S}_\mathcal{D}$ a set $\mathit{val}(D)$ of values.
- A set $\mathcal{A}$ of possible attribute names (identifiers).
- A relational database schema $\mathcal{S}$ that consists of
  - a finite set of relation names $\mathcal{R}$, and
  - for every $R \in \mathcal{R}$, a relation schema $\mathit{sch}(R)$.
- Constraints are not important here.

Syntax (2)

Base Cases

- One recursively defines the set of relational algebra (RA) expressions (queries) together with the relation schema of each RA expression.
  - $\boxed{R}$: For every $R \in \mathcal{R}$, the relation name R is an RA expression with schema $sch(R)$.
  - $\boxed{\{(A_1 \colon d_1, \dots, A_n \colon d_n)\}}$ ("relation constant") is an RA expression if $A_1, \dots, A_n \in \mathcal{A}$, and $d_i \in val(D_i)$ for $1 \leq i \leq n$ with $D_1, \dots, D_n \in \mathcal{S}_\mathcal{D}$. The schema of this RA expression is $(A_1 \colon D_1, \dots, A_n \colon D_n)$.

Syntax (3)

Recursive cases I

- Let Q be an RA expression with schema $\rho = (A_1 \colon D_1, \ldots, A_n \colon D_n)$.
  Then also the following are RA expressions:
  - $\boxed{\sigma_{A_i = A_j}(Q)}$ for $i, j \in \{1, \ldots, n\}$.
    It has schema ρ.
  - $\boxed{\sigma_{A_i = d}\,(Q)}$ for $i \in \{1, \ldots, n\}$ and $d \in val(D_i)$.
    It has schema ρ.
  - $\boxed{\pi_{B_1 \leftarrow A_{i_1}, \ldots, B_m \leftarrow A_{i_m}}(Q)}$ for $i_1, \ldots, i_m \in \{1, \ldots, n\}$ and $B_1, \ldots, B_m \in \mathcal{A}$
    such that $B_j \neq B_m$ for $j \neq m$.
    It has schema $(B_1 \colon D_{i_1}, \ldots, B_m \colon D_{i_m})$.

Syntax (4)

Recursive cases II

- Let Q_1 and Q_2 be RA expressions with the same schema ρ. Then also the following are RA expressions with schema ρ:
  - $\boxed{(Q_1) \cup (Q_2)}$
  - $\boxed{(Q_1) \setminus (Q_2)}$
- Let Q_1 and Q_2 be RA expressions with the schemas $(A_1 \colon D_1, \ldots, A_n \colon D_n)$ and $(B_1 \colon E_1, \ldots, B_n \colon E_m)$, respectively. If $\{A_1, \ldots, A_n\} \cap \{B_1, \ldots, B_m\} = \emptyset$, then also the following is an RA expression:
  - $\boxed{(Q_1) \times (Q_2)}$
    It has schema: $(A_1 : D_1, \ldots, A_n : D_n, B_1 : E_1, \ldots, B_m : E_m)$.
- Nothing else is a relational algebra expression.

Abbreviations

- Parentheses can be left out if the structure is clear (or the possible structures are equivalent).
- As explained above, additional algebra operations (like the join) can be introduced as abbreviations.

Semantics (1)

- A database state $\mathcal{I}$ defines a finite relation $\mathcal{I}(R)$ for every relation name R in the database schema.
- The result of a query Q, i.e. an RA expression, in a database state $\mathcal{I}$ is a relation.
  The query result is written $\mathcal{I}[Q]$ and defined recursively corresponding to the structure of Q:
  - If Q is a relation name R, then $\mathcal{I}[Q] := \mathcal{I}(R)$.
  - If Q is a constant relation $\{(A_1 : d_1, \dots, A_n : d_n)\}$, then $\mathcal{I}[Q] := \{(d_1, \dots, d_n)\}$.

Semantics (2)

- Definition of the result $\mathcal{I}[Q]$ of an RA expression Q in state $\mathcal{I}$, continued:
  - If Q has the form $\sigma_{A_i=A_j}(Q_1)$, then
    $$\mathcal{I}[Q] := \{(d_1, \ldots, d_n) \in \mathcal{I}[Q_1] \mid d_i = d_j\}.$$
  - If Q has the form $\sigma_{A_i=d}(Q_1)$, then
    $$\mathcal{I}[Q] := \{(d_1, \ldots, d_n) \in \mathcal{I}[Q_1] \mid d_i = d\}.$$
  - If Q has the form $\pi_{B_1 \leftarrow A_{i_1}, \ldots, B_m \leftarrow A_{i_m}}(Q_1)$, then
    $$\mathcal{I}[Q] := \{(d_{i_1}, \ldots, d_{i_m}) \mid (d_1, \ldots, d_n) \in \mathcal{I}[Q_1]\}.$$

Semantics (3)

- Definition of $\mathcal{I}[Q]$, continued:
  - If Q has the form $(Q_1) \cup (Q_2)$ then
    $$\mathcal{I}[Q] := \mathcal{I}[Q_1] \cup \mathcal{I}[Q_2].$$
  - If Q has the form $(Q_1) \setminus (Q_2)$ then
    $$\mathcal{I}[Q] := \mathcal{I}[Q_1] \setminus \mathcal{I}[Q_2].$$
  - If Q has the form $(Q_1) \times (Q_2)$, then
    $$\begin{aligned}\mathcal{I}[Q] := \{(d_1, \ldots, d_n, e_1, \ldots, e_m) \mid \\ (d_1, \ldots, d_n) \in \mathcal{I}[Q_1], \\ (e_1, \ldots, e_m) \in \mathcal{I}[Q_2]\}.\end{aligned}$$

Monotonicity

- Definition A database state $\mathcal{I}_1$ is smaller than (or equal to) a database state $\mathcal{I}_2$, written $\mathcal{I}_1 \subseteq \mathcal{I}_2$, if and only if $\mathcal{I}_1(R) \subseteq \mathcal{I}_2(R)$ for all relation names R in the schema.
- Theorem If an RA expression Q does not contain the $\setminus$ (set difference) operator, then the following holds for all database states $\mathcal{I}_1, \mathcal{I}_2$:

$$\mathcal{I}_1 \subseteq \mathcal{I}_2 \implies \mathcal{I}_1[Q] \subseteq \mathcal{I}_2[Q].$$

Equivalence (1)

- Definition Two RA expressions Q_1 and Q_2 are equivalent if and only if they have the same schema and for all database states $\mathcal{I}$ the following holds:

  $$\mathcal{I}[Q_1] = \mathcal{I}[Q_2].$$

- There are two notions of equivalence, depending on whether one considers all structurally possible states or only states that satisfy the constraints.

Equivalence (2)

- Examples for equivalences
  - $\sigma_{\varphi_1}(\sigma_{\varphi_2}(Q))$ is equivalent to $\sigma_{\varphi_2}(\sigma_{\varphi_1}(Q))$.
  - $(Q_1 \times Q_2) \times Q_3$ is equivalent to $Q_1 \times (Q_2 \times Q_3)$.
  - If A is an attribute in the schema of Q_1: $\sigma_{A=d}(Q_1 \times Q_2)$ is equivalent to $(\sigma_{A=d}(Q_1)) \times Q_2$
- Theorem The equivalence of relational algebra expressions is undecidable.

Limitations (1)

- Let R be a relation name with schema $(A\colon D, B\colon D)$ and let $val(D)$ be infinite.
- The transitive closure of $\mathcal{I}(R)$ is the set of all $(d, e) \in val(D) \times val(D)$ such that there are $n \in \mathbb{N}$ $(n \geq 1)$ and $d_0, \ldots, d_n \in val(D)$ with $d = d_0$, $e = d_n$ and $(d_{i-1}, d_i) \in \mathcal{I}(R)$ for $i = 1, \ldots, n$.
- E.g. R could be the relation “`PARENT`”, then the transitive closure are all ancestor-relationships (parents, grandparents, great-grandparents, $\ldots$).

Limitations (2)

- Theorem There is no RA expression Q such that $\mathcal{I}[Q]$ is the transitive closure of $\mathcal{I}(R)$ for all database states $\mathcal{I}$.
- E.g. in the ancestor example, one would need an additional join for every additional generation.
- Therefore, if one does not know, how many generations the database contains, one cannot write a query that works for all possible database states.

Limitations (3)

- This of course implies that relational algebra is not computationally complete:
  - Not every function from database states to relations for which a C program exists can be formulated in relational algebra.
  - However, this can also not be expected, since one wants to be sure that query evaluation always terminates. This is guaranteed for RA.

Limitations (4)

- All RA queries can be computed in time that is polynomically in the size of the database.
- Thus, also very complex functions cannot be formulated in relational algebra.
- As the transitive closure shows, not all problems of polynomial complexity can be formulated in RA.

Expressive Power (1)

- A query language $\mathcal{L}$ for the relational model is called strong relationally complete if for every database schema $\mathcal{S}$ and for every RA expression Q_1 with respect to $\mathcal{S}$ there is a query $Q_2 \in \mathcal{L}_{\mathcal{S}}$ such that for all database states $\mathcal{I}$ with respect to $\mathcal{S}$ the two queries produce the same result: $\mathcal{I}[Q_1] = \mathcal{I}[Q_2]$.
- I.e. the requirement is that every relational algebra expression can be translated into an equivalent query in that language.

Expressive Power (2)

- E.g. SQL is strong relationally complete.
- If the translation of queries is possible in both directions, the two query languages have the same expressive power.
- “Relationally complete” (without “strong”) permits to use a sequence of queries and to store intermediate results in temporary relations.

Expressive Power (3)

- The following languages have the same expressive power (queries can be translated between them):
  - Relational algebra
  - SQL without aggregations and with mandatory duplicate elimination.
  - Tuple relational calculus (first order logic with variables for tuples)
  - Domain relational calculus
  - Datalog (a Prolog variant) without recursion
- Thus, the set of functions that can be expressed in RA is at least not arbitrary.

Outline

Summary

- The five basic operations of relational algebra are:
  - σ_{φ}: Selection
  - $\pi_{\mathtt{A}_1,\dots,\mathtt{A}_k}$: Projection
  - $\times$: Cartesian Product
  - $\cup$: Union
  - $\setminus$: Set Difference
- Derived operations: The general join $\underset{\varphi}{\bowtie}$, the natural join $\bowtie$, the renaming operator ρ, the intersection $\cap$.

Relational normal form: Overview

Outline

Introduction (1)

- Relational database design theory is based mainly on a class of constraints called “Functional Dependencies” (FDs). FDs are a generalization of keys.
- This theory defines when a relation is in a certain normal form (e.g. Third Normal Form, 3NF) for a given set of FDs.
- It is usually bad if a schema contains relations that violate the conditions of a normal form.

Introduction (2)

- If a normal form is violated, data is stored redundantly, and information about different concepts is intermixed. E.g. consider the following table:

COURSES			
CRN	TITLE	INAME	PHONE
22268	DB	Brass	9404
42232	DS	Brass	9404
31822	IS	Spring	9429

- The phone number of “Brass” is stored two times. In general, the phone number of an instructor will be stored once for every course he/she teaches.

Introduction (3)

- Of course, it is no problem if a column contains the same value two times (e.g. consider a `Y/N` column).
- But in this case, the following holds: If two rows have the same value in the column `INAME`, they must have the same value in the column `PHONE`.
- This is an example of a functional dependency: `INAME` $\rightarrow$ `PHONE`.
- Because of this rule, one of the two `PHONE` entries for `Brass` is redundant.

Introduction (4)

- Table entries are redundant if they can be reconstructed from other table entries and additional information (like the FD in this case).
- Redundant information in database schemas is bad:
  - Storage space is wasted.
  - If the information is updated, all redundant copies must be updated. If one is not careful, the copies become inconsistent (Update Anomaly).

Introduction (5)

- In the example, the information about the two concepts “Course” and “Instructor” are intermixed in one table.
  This is bad:
  - The phone number of a new faculty member can be stored in the table only together with a course (Insertion Anomaly).
  - When the last course of a faculty member is deleted, his/her phone number is lost (Deletion Anomaly).

Introduction (6)

- Third Normal Form (3NF) is considered part of a decent database design.
- Boyce-Codd Normal Form (BCNF) is slightly stronger, easier to define, and better matches intuition.
  Intuitively, BCNF means that all FDs are already enforced by keys.
- Only BCNF is defined here.
- If a table is in BCNF, it is automatically in 3NF.

Outline

Functional Dependencies (1)

- Functional dependencies (FDs) are generalizations of keys.
- A functional dependency specifies that an attribute (or attribute combination) uniquely determines another attribute (or other attributes).
- Functional dependencies are written in the form

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m.$$

- This means that whenever two rows have the same values in the attributes $A_1, \ldots, A_n$, then they must also agree in the attributes $B_1, \ldots, B_m$.

Functional Dependencies (2)

- As noted above, the FD "INAME $\rightarrow$ PHONE" is satisfied in the following example:

COURSES			
CRN	TITLE	INAME	PHONE
22268	DB	Brass	9404
42232	DS	Brass	9404
31822	IS	Spring	9429

- If two rows agree in the instructor name, they must have the same phone number.

Functional Dependencies (3)

- A key uniquely determines every attribute, i.e. the FDs "`CRN`$\rightarrow$`TITLE`", "`CRN`$\rightarrow$`INAME`", "`CRN`$\rightarrow$`PHONE`" are trivially satisfied:
  - There are no two distinct rows that have the same value for a key (`CRN` in this case).
  - Therefore, whenever rows t and u agree in the key (`CRN`), they must actually be the same row, and therefore agree in all other attributes, too.
- Instead of the three FDs above, one can also write the single FD "`CRN` $\rightarrow$ `TITLE, INAME, PHONE`".

Functional Dependencies (4)

- In the example, the FD "`INAME` $\rightarrow$ `TITLE`" is not satisfied: There are two rows with the same `INAME`, but different values for `TITLE`.
- In the example, the FD "`TITLE` $\rightarrow$ `CRN`" is satisfied.
- However, like keys, FDs are constraints: They must hold in all possible database states, not only in a single example state.

Functional Dependencies (5)

- Therefore, it is a database design task to determine which FDs should hold. This cannot be decided automatically, and the FDs are needed as input for the normalization check.
- In the example, the DB designer must find out whether it can ever happen that two courses are offered with the same title (e.g. two sessions of a course that is overbooked).
- If this can happen, the FD "`TITLE` $\rightarrow$ `CRN`" does not hold in general.

Functional Dependencies (6)

- Sequence and multiplicity of attributes in an FD are unimportant, since both sides are formally sets of attributes:

$$\{A_1, \dots, A_n\} \to \{B_1, \dots, B_m\}.$$

- In discussing FDs, the focus is on a single relation R.
  All attributes A_i, B_i are from this relation.
- The FD $A_1, \dots, A_n \to B_1, \dots, B_m$ is equivalent to the m FDs:

$$\begin{array}{ccc} A_1, \dots, A_n & \to & B_1 \\ \vdots & \vdots & \vdots \\ A_1, \dots, A_n & \to & B_m. \end{array}$$

FDs vs. Keys

- FDs are a generalization of keys: $A_1, \dots, A_n$ is a key of

  $$R(A_1, \dots, A_n, B_1, \dots, B_m)$$

  if and only if the FD "$A_1, \dots, A_n \rightarrow B_1, \dots, B_m$" holds.
- Given the FDs for a relation, one can compute a key by finding a set of attributes $A_1, \dots, A_n$ that functionally determines the other attributes.

Trivial FDs

- A functional dependency $\alpha \rightarrow \beta$ such that $\beta \subseteq \alpha$ is called trivial.
- Examples are:
  - `TITLE` $\rightarrow$ `TITLE`
  - `INAME, PHONE` $\rightarrow$ `PHONE`
- Trivial functional dependencies are always satisfied (in every database state, no matter whether it satisfies other constraints or not).
- Trivial FDs are not interesting.

Implication of FDs (1)

- `CRN`$\rightarrow$`PHONE` is nothing new when one knows already `CRN`$\rightarrow$`INAME` and `INAME`$\rightarrow$`PHONE`.
- A set of FDs $\{\alpha_{\mathtt{1}} \rightarrow \beta_{\mathtt{1}}, \ldots, \alpha_{\mathtt{n}} \rightarrow \beta_{\mathtt{n}}\}$ implies an FD $\alpha \rightarrow \beta$ if and only if every DB state which satisfies the $\alpha_{\mathtt{i}} \rightarrow \beta_{\mathtt{i}}$ for $i = 1, \ldots, n$ also satisfies $\alpha \rightarrow \beta$.

Implication of FDs (2)

- One is normally not interested in all FDs which hold, but only in a representative set that implies all other FDs.
- Implied dependencies can be computed by applying the Armstrong Axioms:
  - If $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ trivially holds (Reflexivity).
  - If $\alpha \rightarrow \beta$, then $\alpha \cup \gamma \ \rightarrow \ \beta \cup \gamma$ (Augmentation).
  - If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (Transitivity).

Implication of FDs (3)

- A simpler way to check whether $\alpha \to \beta$ is implied by given FDs is to compute first the attribute cover α^+ of α and then to check whether $\beta \subseteq \alpha^+$.
- The attribute cover $\alpha^+_{\mathcal{F}}$ of a set of attributes α is the set of all attributes B that are uniquely determined by α (with respect to given FDs $\mathcal{F}$).

$$\alpha^+_{\mathcal{F}} := \{B \mid \text{The FDs } \mathcal{F} \text{ imply } \alpha \to B\}.$$

- A set of FDs $\mathcal{F}$ implies $\alpha \to \beta$ if and only if $\beta \subseteq \alpha^+_{\mathcal{F}}$.

Implication of FDs (4)

- The cover is computed as follows:

Input:	α (Set of attributes)
	$\alpha_1 \rightarrow \beta_1, \ldots, \alpha_n \rightarrow \beta_n$ (Set of FDs)
Output:	α^+ (Set of attributes, Cover of α)

```
Method: $x := \alpha$;
        **while** $x$ did change **do**
            **for each** given FD $\alpha_i \rightarrow \beta_i$ **do**
                **if** $\alpha_i \subseteq x$ **then**
                    $x := x \cup \beta_i$;
        **output** $x$;
```

Implication of FDs (5)

- Consider the following FDs:

$$\begin{aligned}&\texttt{ISBN} \rightarrow \texttt{TITLE, PUBLISHER}\\&\texttt{ISBN, NO} \rightarrow \texttt{AUTHOR}\\&\texttt{PUBLISHER} \rightarrow \texttt{PUB\_URL}\end{aligned}$$

- Suppose we want to compute $\{\texttt{ISBN}\}^+$.
- We start with $\texttt{x} = \{\texttt{ISBN}\}$.

Implication of FDs (6)

- The first of the given FDs, namely

  $$\texttt{ISBN} \rightarrow \texttt{TITLE, PUBLISHER}$$

  has a left hand side (ISBN) that is contained in the current set x (actually, $x = \{\texttt{ISBN}\}$).
- Therefore, we can extend x by the attributes on the right hand side of this FD, i.e. TITLE, and PUBLISHER:

  $$x = \{\texttt{ISBN}, \texttt{TITLE}, \texttt{PUBLISHER}\}.$$

Implication of FDs (7)

- Now the third of the FDs, namely

  $$\texttt{PUBLISHER} \rightarrow \texttt{PUB\_URL}$$

  is applicable:
  Its left hand side is contained in x.
- Therefore, we can add the right hand side of this FD to x and get

  $$x = \{\texttt{ISBN}, \texttt{TITLE}, \texttt{PUBLISHER}, \texttt{PUB\_URL}\}.$$

- The last FD, namely

  $$\texttt{ISBN, NO} \rightarrow \texttt{AUTHOR}$$

  is still not applicable, because `NO` is missing in x.

Implication of FDs (8)

- After checking again that there is no way to extend the set x any further with the given FDs, the algorithm terminates and prints

$$\{\texttt{ISBN}\}^+ = \{\texttt{ISBN}, \texttt{TITLE}, \texttt{PUBLISHER}, \texttt{PUB\_URL}\}.$$

- From this, we can conclude that the given FDs imply e.g.

$$\texttt{ISBN} \rightarrow \texttt{PUB\_URL}.$$

- In the same way, one can compute e.g. the cover of $\{\texttt{ISBN}, \texttt{NO}\}$. It is the entire set of attributes.

How to Determine Keys (1)

- Given a set of FDs (and the set of all attributes $\mathcal{A}$ of a relation), one can determine all possible keys for that relation.
- $\alpha \subseteq \mathcal{A}$ is a key if and only if $\alpha^+ = \mathcal{A}$.
- Normally, one is only interested in minimal keys.

How to Determine Keys (2)

- One can construct a key also in a less formal way.
- So one starts with the set of required attributes (that do not appear on any right side).
- If the required attributes do not already form a key, one adds attributes: The left hand side of an FD or directly one of the missing attributes.

Outline

Motivation (1)

- Consider again the example:

COURSES			
<u>CRN</u>	TITLE	INAME	PHONE
22268	DB	Brass	9404
42232	DS	Brass	9404
31822	IS	Spring	9429

- As noted above, the FD INAME→PHONE leads to problems, one of which is the redundant storage of certain facts (e.g. the phone number of “Brass”).
- The FD INAME→PHONE leads exactly then to redundancies, if there are several lines with the same value for INAME (left side of the FD).
- Then these lines must also have the same value for PHONE (right side of the FD). Of these, all but one copy are redundant.

Motivation (2)

- Actually, any FD $A_1, \dots, A_n \rightarrow B_1, \dots, B_m$ will cause redundant storage unless $A_1, \dots, A_n$ is a key, so that each combination of attribute values for $A_1, \dots, A_n$ can occur only once.
- Avoid (proper) FDs by transforming them into key constraints. This is what normalization does.

Motivation (3)

- The problem in the example is also caused by the fact that information about different concepts is stored together (faculty members and courses).
- Formally, this follows also from "`INAME`$\rightarrow$`PHONE`":
  - `INAME` is like a key for only part of the attributes.
  - It identifies faculty members, and `PHONE` depends only on the faculty member, not on the course.
- Again: The left hand side of an FD should be a key.

Boyce-Codd Normal Form (BCNF)

- A Relation R is in BCNF if and only if all its FDs are already implied by key constraints.
- I.e. for every FD "$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$" one of the following conditions must hold:
  - The FD is trivial, i.e. $\{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\}$.
  - The FD follows from a key, because $\{A_1, \ldots, A_n\}$ or some subset of it is already a key.

Boyce-Codd Normal Form

Check

- From the given FDs $\mathcal{F}$ one can first determine the keys of the relation, and then apply the definition directly:
  For each FD $\alpha \rightarrow \beta$ from $\mathcal{F}$: If $\beta \not\subseteq \alpha$, then α contains one of the keys (plus possibly other attributes).
- However, one can also check for each FD $\alpha \rightarrow \beta$ with $\beta \not\subseteq \alpha$ whether the attribute cover α^+ is already the set of all the attributes of the relation.
  Then α or a subset of it is just a key.

Examples (1)

- `COURSES(`<u>`CRN`</u>`, TITLE, INAME, PHONE)` with the FDs
  - `CRN` $\rightarrow$ `TITLE, INAME, PHONE`
  - `INAME` $\rightarrow$ `PHONE`

  is not in BCNF because the FD "`INAME` $\rightarrow$ `PHONE`" is not implied by a key:
  - "`INAME`" is not a key of the entire relation.
  - The FD is not trivial.
- However, without the attribute `PHONE` (and its FD), the relation is in BCNF:
  - `CRN` $\rightarrow$ `TITLE, INAME` corresponds to the key.

Examples (2)

- Suppose that each course meets only once per week and that there are no cross-listed courses. Then

  `CLASS(CRN, TITLE, DAY, TIME, ROOM)`

  satisfies the following FDs (plus implied ones):
  - `CRN` $\rightarrow$ `TITLE, DAY, TIME, ROOM`
  - `DAY, TIME, ROOM` $\rightarrow$ `CRN`
- The keys are `CRN` and `DAY, TIME, ROOM`.
- Both FDs have a key on the left hand side, so the relation is in BCNF.

Examples (3)

- Suppose that `PRODUCT(NO, NAME, PRICE)` has these FDs:

  (1) $\texttt{NO} \rightarrow \texttt{NAME}$ (3) $\texttt{PRICE, NAME} \rightarrow \texttt{NAME}$

  (2) $\texttt{NO} \rightarrow \texttt{PRICE}$ (4) $\texttt{NO, PRICE} \rightarrow \texttt{NAME}$

- This relation is in BCNF:
  - The first two FDs show that `NO` is a key. Since their left hand side is a key, they are no problem.
  - The third FD is trivial and can be ignored.
  - The fourth FD has a superset of the key on the left hand side, which is also no problem.

Splitting Relations (1)

- A table which is not in BCNF can be split into two tables ("decomposition"), e.g. split `COURSES` into

  $$\texttt{COURSES\_NEW}(\underline{\texttt{CRN}}, \texttt{ TITLE}, \texttt{ INAME}\rightarrow\texttt{INSTRUCTORS})$$
  $$\texttt{INSTRUCTORS}(\underline{\texttt{INAME}}, \texttt{ PHONE})$$

- General case: If $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ violates BCNF, create a relation $S(\underline{A_1}, \ldots, \underline{A_n}, B_1, \ldots, B_m)$ and remove $B_1, \ldots, B_m$ from the original relation.

Splitting Relations (2)

- When splitting relations, it is of course important that the transformation is “lossless”, i.e. that the original relation can be reconstructed by means of a join:

$$\texttt{COURSES} = \texttt{COURSES\_NEW} \bowtie \texttt{INSTRUCTORS}.$$

- I.e. the original relation can be defined as a view:

```
CREATE VIEW COURSES(CRN, TITLE, INAME, PHONE)
AS
SELECT C.CRN, C.TITLE, C.INAME, I.PHONE
FROM   COURSES_NEW C, INSTRUCTORS I
WHERE  C.INAME = I.INAME
```

Splitting Relations (3)

- The split of the relations is guaranteed to be lossless if the intersection of the attributes of the new tables is a key of at least one of them ("decomposition theorem"):

$$\{\texttt{CRN},\ \texttt{TITLE},\ \texttt{INAME}\} \cap \{\texttt{INAME},\ \texttt{PHONE}\} = \{\texttt{INAME}\}.$$

- The above method for transforming relations into BCNF does only splits that satisfy this condition.
- It is always possible to transform a relation into BCNF by lossless splitting (if necessary repeated).

Splitting Relations (4)

- Not every lossless split is reasonable:

STUDENTS		
SSN	FIRST_NAME	LAST_NAME
111-22-3333	John	Smith
123-45-6789	Maria	Brown

- Splitting this into STUD_FIRST(SSN,FIRST_NAME) and STUD_LAST(SSN,LAST_NAME) is lossless, but
  - is not necessary to enforce a normal form and
  - only requires costly joins in later queries.

Splitting Relations (5)

- Losslessness means that the resulting schema can represent all states which were possible before.
- However, the new schema allows states which do not correspond to a state in the old schema: Now instructors without courses can be stored.
- Thus, the two schemas are not equivalent: The new one is more general.

Splitting Relations (6)

- If instructors without courses are possible in the real world, the decomposition removes a fault in the old schema (insertion and deletion anomaly).
- If they are not,
  - a new constraint is needed that is not necessarily easier to enforce than the FD, but at least
  - the redundancy is avoided (update anomaly).

Outline

Summary

- Data base design
- Redundancy causes anomalies on
  - Deletion
  - Insertion
  - Update
- Functional dependencies
  - Implication
  - Computation of attribute cover
- BCNF eliminates all redundancies based on functional dependencies
  - Check
  - Split

Bibliography

- The following list of references is compiled from the open source bibliography available at

  `https://github.com/krr-up/bibliography`

- Feel free to submit corrections via pull requests!

[1] S. Abiteboul, R. Hull, and V. Vianu.
Foundations of Databases.
Addison-Wesley, 1995.

[2] C. Aggarwal, editor.
volume 31 of *Advances in Database Systems*.
Springer-Verlag, 2007.

[3] C. Aggarwal, editor.
Data Streams — Models and Algorithms, volume 31 of *Advances in Database Systems*.
Springer-Verlag, 2007.

[4] K. Apt, H. Blair, and A. Walker.
Towards a theory of declarative knowledge.
In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann Publishers, 1987.

[5] M. Arenas, L. Bertossi, and J. Chomicki.

Consistent query answers in inconsistent databases.
In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'99)*, pages 68–79. ACM Press, 1999.

[6] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors.
The Description Logic Handbook: Theory, Implementation, and Applications.
Cambridge University Press, 2003.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom.
Models and issues in data stream systems.
In L. Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 1–16. ACM Press, 2002.

[8] C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.

Cambridge University Press, 2003.

[9] S. Ceri, G. Gottlob, and L. Tanca.
Logic Programming and Databases.
Springer-Verlag, 1990.

[10] R. Elmasri and S. Navathe.
Fundamentals of database systems.
Addison-Wesley, 1994.

[11] R. Fagin, J. Ullman, and M. Vardi.
On the semantics of updates in databases. preliminary report.
In *Proceedings of the Second ACM Conference SIGACT-SIGMOD*, pages 352–365, 1983.

[12] H. Gallaire, J. Minker, and J. Nicolas.
Logic and databases: A deductive approach.
Computing Surveys, 16(2):153–185, 1984.

[13] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele.

Potassco User Guide.
University of Potsdam, 2 edition, 2015.

[14] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Answer Set Solving in Practice.
Synthesis Lectures on Artificial Intelligence and Machine Learning.
Morgan and Claypool Publishers, 2012.

[15] M. Gelfond and Y. Kahl.
Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.
Cambridge University Press, 2014.

[16] M. Gelfond and V. Lifschitz.
Classical negation in logic programs and disjunctive databases.
New Generation Computing, 9:365–385, 1991.

[17] H. Katsuno and A. Mendelzon.
On the difference between updating a knowledge database and revising it.

In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 387–394. Morgan Kaufmann Publishers, 1991.

[18] V. Lifschitz.
Closed-world databases and circumscription.
Artificial Intelligence, 27:229–235, 1985.

[19] V. Lifschitz.
Nonmonotonic databases and epistemic queries.
In J. Myopoulos and R. Reiter, editors, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 381–386. Morgan Kaufmann Publishers, 1991.

[20] V. Lifschitz.
Introduction to answer set programming.
Unpublished draft, 2004.

[21] V. Lifschitz, F. van Harmelen, and B. Porter, editors.

Handbook of Knowledge Representation.
Elsevier Science, 2008.

[22] L. Liu and M. Özsu, editors.
Encyclopedia of Database Systems.
Springer-Verlag, 2009.

[23] V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.
Artifical Intelligence. Springer-Verlag, 1993.

[24] J. Minker, editor.
Foundations of Deductive Databases and Logic Programming.
Morgan Kaufmann Publishers, 1988.

[25] R. Reiter.
On closed world data bases.
In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, New York, 1978.

[26] R. Reiter.

Towards a logical reconstruction of relational database theory.
In M. Brodie, J. Myopoulos, and J. Schmidt, editors, *On conceptual modeling: Perspectives from Artificial Intelligence, Datbases and Programming Languages*, pages 191–233. Springer-Verlag, 1984.

[27] R. Reiter.
On asking what a database knows.
In J. Lloyd, editor, *Computational Logic*, pages 96–113. Springer-Verlag, 1990.

[28] J. Ullman.
Principles of Database Systems.
Computer Science Press, Rockville MD, 1982.

[29] J. Ullman.
Principles of Database and Knowledge-Base Systems.
Computer Science Press, 1988.