Database Concepts
Genealogy of Relational DBMSs
Overview

1. What are Databases – Basic Concepts
2. Relational Databases – Data as Tables
3. Database Design Using the ER Model
4. Relational DB Design and Design Theory
5. The Database Language SQL
6. Query Basics: Algebra & Calculus
7. Transactions, Integrity and Triggers
8. Views and Access Control
9. Application Programming
Further English Literature

R. Ramakrishnan and J. Gehrke.  
*Database Management Systems.*  

R. Elmasri, S.B. Navathe.  
*Fundamentals of Database Systems.*  

H. Garcia-Molina, J. D. Ullman, J. Widom  
Part I

What are Databases?
What are Databases?

1. Overview & Motivation
2. Architectures
3. Areas of Application
4. History
Educational Objective for Today . . .

- Motivation for using database systems
- Knowledge of basic architectures
What are Databases?

▶ Data = logically grouped pieces of information
▶ Base¹
  : the bottom or lowest part of something : the part on which something rests or is supported
  : something (such as a group of people or things) that provides support for a place, business, etc.

¹http://www.merriam-webster.com/dictionary/base
Areas of Application
How is Data Managed?

Without databases

▶ Each application manages its own data
▶ Data is stored multiple times $\rightarrow$ redundancy
▶ Problems
  ▶ Waste of storage space
  ▶ “Forgetting” of changes
  ▶ No centralized, “standardized” data management
Problems of Data Redundancy

- Other software systems cannot process large amounts of data efficiently
- Many users or applications cannot access the same data in parallel without interfering with each other
- Application developers / users cannot develop / use applications without knowing
  - internal representation of data
  - storage media or computers
  (no data independence)
- No data security; potential loss of data
Idea: Data Integration Using Database Systems

DBMS = Database Management System = Software for Managing Databases

DBS = Database System

structured data, which is managed by DBMS
Motivation

► Database systems are center piece of modern IT systems

► ... ubiquitous

► Database specialists are in high demand
Questions

1. How to organize (model and use) data?
2. How to store data safely and persistently?
3. How to process huge amounts of data (\(\geq\) terabytes) efficiently?
4. How can many users (\(\geq\) 10,000) access data concurrently?
Principles: Codd’s Nine Rules

1. **Integration**: uniform, non-redundant data management
2. **Operations**: insert, query, update, delete
3. **Catalog**: access to the database description in a data dictionary
4. **User views**: different users/applications must be able to have a different perception of the data
5. **Integrity**: ensure conformity of database contents with real world
6. **Security**: prevention of unauthorized access
7. **Transactions**: multiple DB operations handled as an atomic unit
8. **Synchronisation**: coordination of concurrent transactions
9. **Recovery**: data backup and recovery after system errors
Data Independence and Schemata

- Based on coarse DBMS architecture
- Decouple user and implementation view
- Goals include:
  - Separate modeling view from internal storage
  - Portability
  - Simplify tuning
  - Standardized interfaces
Schema Architecture /2

► Connection between
  ► Conceptual schema (result of data definition)
  ► Internal schema (definition of file structure and access paths)
  ► External schemata (result of view definition)
  ► Application programs (result of application programming)
Schema Architecture /3

- Separation schema — instance
  - Schema (metadata, data description)
  - Instance (user data, database state or shape)

- Database schema consists of
  - Internal, conceptual, external schemata and application programs

- Conceptual schema contains, e.g.:
  - Structure descriptions
  - Integrity descriptions
  - Authorization rules (DB accesses that a user may perform)
Data Independence /2

- Stability of user interface with respect to changes
- **Physical**: Changes to file structure or access paths do not influence the conceptual schema
- **Logical**: Changes to conceptual schema and certain external schemata do not influence other external schemata or application programs
Data Independence /3

- Potential impact of changes to the conceptual schema:
  - external schemata may be affected (changing attributes)
  - application programs may be affected (recompilation of application programs, adaptations may be necessary)

- But: necessary changes are recognized and monitored by the DBMS
Application Example: Music Store

Musician: Neil Young

Title: Living With War

Year: 2006

Tracks: 10

Price: 9.99 €

Critique(s): Album Kaufen

In a move that deliberately echoes the rush release of "Ohio" in the wake of the Kent State shootings, Neil Young bashed out his 2006 protest record Living With War in a matter of days, sometimes recording songs the day they were written, and then seized the opportunities of the
Layer Architecture by Example

▷ Conceptual view: presentation in tables (relations)

<table>
<thead>
<tr>
<th>Artist</th>
<th>MNr</th>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td></td>
<td>Apocalyptica</td>
<td>Finland</td>
</tr>
<tr>
<td>104</td>
<td></td>
<td>Subway To Sally</td>
<td>Germany</td>
</tr>
<tr>
<td>105</td>
<td></td>
<td>Rammstein</td>
<td>Germany</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Album</th>
<th>ANr</th>
<th>Title</th>
<th>Year</th>
<th>Genre</th>
<th>MNr → Artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1014</td>
<td></td>
<td>Amplified</td>
<td>2006</td>
<td>Rock</td>
<td>103</td>
</tr>
<tr>
<td>1015</td>
<td></td>
<td>Nord Nord Ost</td>
<td>2005</td>
<td>Rock</td>
<td>104</td>
</tr>
<tr>
<td>1016</td>
<td></td>
<td>Rosenrot</td>
<td>2005</td>
<td>Rock</td>
<td>105</td>
</tr>
<tr>
<td>1021</td>
<td></td>
<td>Engelskrieger</td>
<td>2003</td>
<td>Rock</td>
<td>104</td>
</tr>
<tr>
<td>1025</td>
<td></td>
<td>Reflections</td>
<td>2006</td>
<td>Rock</td>
<td>103</td>
</tr>
</tbody>
</table>
Layer Architecture by Example /2

► External view: data in a flat relation

<table>
<thead>
<tr>
<th>ANr</th>
<th>Title</th>
<th>Year</th>
<th>Genre</th>
<th>Artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1014</td>
<td>Amplified</td>
<td>2006</td>
<td>Rock</td>
<td>Apocalyptica</td>
</tr>
<tr>
<td>1015</td>
<td>Nord Nord Ost</td>
<td>2005</td>
<td>Rock</td>
<td>Subway To Sally</td>
</tr>
<tr>
<td>1016</td>
<td>Rosenrot</td>
<td>2005</td>
<td>Rock</td>
<td>Rammstein</td>
</tr>
<tr>
<td>1021</td>
<td>Engelskrieger</td>
<td>2003</td>
<td>Rock</td>
<td>Subway To Sally</td>
</tr>
<tr>
<td>1025</td>
<td>Reflections</td>
<td>2006</td>
<td>Rock</td>
<td>Apocalyptica</td>
</tr>
</tbody>
</table>
Layer Architecture by Example /3

- External view: data in a hierarchically structured relation

<table>
<thead>
<tr>
<th>Artist</th>
<th>Album</th>
<th>Year</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apocalyptica</td>
<td>Amplified</td>
<td>2006</td>
<td>Rock</td>
</tr>
<tr>
<td></td>
<td>Reflections</td>
<td>2003</td>
<td>Rock</td>
</tr>
<tr>
<td>Subway To Sally</td>
<td>Nord Nord Ost</td>
<td>2005</td>
<td>Metal</td>
</tr>
<tr>
<td></td>
<td>Engelskrieger</td>
<td>2003</td>
<td>Rock</td>
</tr>
<tr>
<td>Rammstein</td>
<td>Rosenrot</td>
<td>2005</td>
<td>Rock</td>
</tr>
</tbody>
</table>
Layer Architecture by Example /4

- Internal presentation

Tree-like Access on Album Number

Partial Storage of Data Sets in a Tree

Space for Data Sets Overflow

<table>
<thead>
<tr>
<th>Album Number</th>
<th>Year</th>
<th>Title</th>
<th>Genre</th>
<th>Artist</th>
<th>Year</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1014</td>
<td>Amplified</td>
<td>2006</td>
<td>Rock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1015</td>
<td>Nord Nord Ost</td>
<td>2005</td>
<td>Rock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.99</td>
<td>Rock</td>
<td>103</td>
<td>....</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.99</td>
<td>Rock</td>
<td>104</td>
<td>....</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System Architectures

- Description of components of a database system
- Standardized interfaces between components
- Architecture proposals
  - ANSI-SPARC architecture
    ↦ three layer architecture
  - Five layer architecture
    ↦ describes transformation components in detail

Lecture “Database Implementation Techniques”
ANSI-SPARC Architecture

- ANSI: American National Standards Institute
- SPARC: Standards Planning and Requirement Committee
- Proposal from 1978
- Basically refines coarse architecture
  - Internal layer / operation system refined
  - Multiple interactive and programming components
  - Interfaces named and standardized
ANSI-SPARC Architecture /2

- External Layer
  - Query
  - Updates
- Conceptual Layer
  - Optimizer
  - Mapping
  - Disk Access
- Internal Layer
  - Data Dictionary
  - View Definition
  - Data Definition
  - File Organisation
  - DB-Operations
  - Embedding
  - GUIs

P1

...
Classification of Components

- Definition components: data definition, file organization, view definition
- Programming components: DB programming with embedded DB operations
- User components: interactive application programs, query and update
- Transformation components: optimizer, analysis, disk access
- Data dictionary: contains data from definition components, provides other components with this information
Five Layer Architecture

Refinement of transformation steps

SOI  Set-Oriented Interface

Data System

Translation, Access Path Selection, Access Control, Integrity Control

ROI  Record-Oriented Interface

Access System

Data Dictionary, Currency Pointer, Sorting, Concurrency control

IRI  Internal-Record Interface

Storage System

Record Manager, Access Path Management, Lock Management, Logging, Recovery

SBI  System Buffer Interface

Buffer Manager

System Buffer Management, Page Replacement

FI  File Interface

Operating System

External Storage Management

DI  Device Interface

External Storage
Application Architectures

- Architecture of database applications typically based on client-server model: server ≡ database system

Client
(Customer)

Server
(Service Provider)

1. Request

2. Processing

3. Answer
Application Architectures /2

- Separation of functionality of an application
  - Presentation and user interaction
  - Application logic (“business logic”)
  - Data management functionality (store, query, ...).

Two Layer Architecture

Three Layer Architecture
Some concrete systems

- (Object-)relational DBMS
  - Oracle11g, IBM DB2 V.10, Microsoft SQL Server 2012, SAP HANA
  - MySQL (www.mysql.org), PostgreSQL (www.postgresql.org)
- Pseudo DBMS
  - MS Access
- NoSQL systems
  - Graph database systems (InfiniteGraph, neo4j), document databases (MongoDB), key-value stores, ....
Areas of Application

▶ Classical areas of application:
▶ Many objects (15,000 books, 300 users, 100 books borrowed per week, . . .)
▶ Few object types (BOOK, USER, BORROWING)
▶ For instance, book keeping systems, order tracking systems, library systems, . . .

▶ Current applications:
▶ E-Commerce, decision supporting systems (data warehouses, OLAP), NASA’s Earth Observation System (petabyte databases), data mining
Database Sizes

**eBay Data Warehouse** 10 PB (≈ 10 \cdot 10^{15} \text{ bytes})
Teradata DBMS, 72 nodes, 10,000 users, millions of queries/day

**WalMart Data Warehouse** 2.5 PB
Teradata DBMS, NCR MPP hardware; product information (sales etc.) of 2,900 stores; 50,000 queries/week

**Facebook** 400 TB
x.000 MySQL server
Hadoop/Hive, 610 nodes, 15 TB/day

**US Library of Congress** 10-20 TB
not digitized

▶ PB for Petabyte is in the order of 10^{15}
Historical Developments: 60s

- Start of 60s: elementary files, application specific file structure (device-dependent, redundant, inconsistent)
- End of 60s: file management systems (SAM, ISAM) with service programs (sorting) (device-independent, but redundant and inconsistent)
- DBS based on hierarchical model, network model
  - Pointer structures between data
  - Weak separation of internal / conceptual layer
  - Navigational DML
  - Separation DML / programming language
Historical Developments: 70s and 80s

- 70s: database systems (device and data independence, redundancy free, consistent)
- Relational database systems
  - Data in table structures
  - 3 layer concept
  - Declarative DML
  - Separation DML / programming language
History of RDBMS

▶ 1970: Ted Codd (IBM) → relational model as conceptual basis of relational DBS
▶ 1974: System R (IBM) → first prototype of an RDBMS
   ▶ Two modules: RDS, RSS; ca. 80,000 LOC (PL/1, PL/S, assembler), ca. 1.2 MB of code
   ▶ Query language SEQUEL
   ▶ First deployment 1977
▶ 1975: University of California at Berkeley (UCB) → Ingres
   ▶ Query language QUEL
   ▶ Predecessor of Postgres, Sybase, ... 
▶ 1979: Oracle Version 2
Historical Developments: (80s and) 90s

- **Knowledge base systems**
  - Data in table structures
  - Strongly declarative DML, integrated database programming language

- **Object-oriented database systems**
  - Data in more complex object structures (separation of object and its data)
  - Declarative or navigational DML
  - Often integrated database programming language
  - Often incomplete separation of layers
Historical Developments: Today

► New hardware architectures
  ► Multicore processors, terabytes of main memory: in-memory database systems (e.g., SAP HANA)
► Support for specific applications
  ► Cloud databases: Hosting of databases, scalable data management solutions (Amazon RDS, Microsoft Azure)
  ► Data stream processing: online processing of live data, e.g., stock exchange data, sensor data, RFID data, ... (StreamBase, MS StreamInsight, IBM Infosphere Streams)
  ► Big Data: Handle petabytes of data through highly scalable, parallel processing, data analysis (Hadoop, Hive, Google Spanner & F1, ...)
  ► NoSQL databases ("Not only SQL"): non-relational databases, flexible schema (document-centered), "light-weight" because SQL functionality like transactions is omitted, powerful declarative query languages with joins, etc. (CouchDB, MongoDB, Cassandra, ... )
Trends

- User-generated content, e.g., Google:
  - Daily processing of 20 PB
  - 15 hours of video uploaded to YouTube every minute
  - Reading 20 PB would take 12 years with a 50 MB/s hard disk drive

- Linked data and data web
  - Provision, exchange and linking of structured data on the Web
  - Enables querying (with query languages like SPARQL) and further processing
  - Examples: DBpedia, GeoNames
Summary

▶ Motivation for using database systems
▶ Codd’s rules
▶ 3 layer schema architecture & data independence
▶ Areas of application
Control Questions

▶ What is the advantage of using database systems compared to application-specific data management?
▶ What does data independence mean and how is it achieved?
▶ Which are areas of application of database systems?
Part II

Relational Databases – Data as Tables
Relational Databases – Data as Tables

1. Relations for Tabular Data
2. SQL Data Definition
3. Basic Operations: The Relational Algebra
4. SQL as a Query Language
5. Manipulation Operations in SQL
Educational Objective for Today . . .

- Basic understanding of the structure of relational databases
- Knowledge of base operations of relational query languages
- Elementary ability to use SQL
Relational Model

- Conceptually, a database is a set of tables

<table>
<thead>
<tr>
<th>WINES</th>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1042</td>
<td>La Rose Grand Cru</td>
<td>Red</td>
<td>1998</td>
<td>Château La Rose</td>
</tr>
<tr>
<td></td>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Red</td>
<td>2003</td>
<td>Creek</td>
</tr>
<tr>
<td></td>
<td>3456</td>
<td>Zinfandel</td>
<td>Red</td>
<td>2004</td>
<td>Helena</td>
</tr>
<tr>
<td></td>
<td>2171</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>2001</td>
<td>Creek</td>
</tr>
<tr>
<td></td>
<td>3478</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>1999</td>
<td>Helena</td>
</tr>
<tr>
<td></td>
<td>4711</td>
<td>Riesling Reserve</td>
<td>White</td>
<td>1999</td>
<td>Müller</td>
</tr>
<tr>
<td></td>
<td>4961</td>
<td>Chardonnay</td>
<td>White</td>
<td>2002</td>
<td>Bighorn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORIGIN</th>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td></td>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td></td>
<td>Château La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td></td>
<td>Château La Pointe</td>
<td>Pomerol</td>
<td>Bordeaux</td>
</tr>
<tr>
<td></td>
<td>Müller</td>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
<tr>
<td></td>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
</tbody>
</table>

- Table = “Relation”
Presentation of Relations; Terminology

- **Bold fields:** relation schema
- Further entries in the table: relation
- A table row: tuple
- A column heading: attribute
- An entry: attribute value

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>A₁</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Aₙ</td>
</tr>
</tbody>
</table>

Relation Schema

Relation

Tuple
Integrity Constraints: Keys

▶ Attributes of a column unambiguously identify stored tuples: key property

▶ E.g., Vineyard for table ORIGIN

<table>
<thead>
<tr>
<th>ORIGIN</th>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
<td></td>
</tr>
<tr>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
<td></td>
</tr>
<tr>
<td>Château La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
<td></td>
</tr>
<tr>
<td>Château La Pointe</td>
<td>Pomerol</td>
<td>Bordeaux</td>
<td></td>
</tr>
<tr>
<td>Müller</td>
<td>Rheingau</td>
<td>Hessen</td>
<td></td>
</tr>
<tr>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
<td></td>
</tr>
</tbody>
</table>

▶ Combinations of attributes can also be keys!

▶ Keys can be marked by underlining them
Integrity Constraints: Foreign Keys

- Keys in one table can be used as unambiguous references in another table (or even in the same table!): **Foreign key**, referential integrity
- E.g., **Vineyard** as a reference to **ORIGIN**
- A foreign key is a **key in a “foreign” table**
## Foreign Keys /2

### WINES

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard → ORIGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1042</td>
<td>La Rose Grand Cru</td>
<td>Red</td>
<td>1998</td>
<td>Château La Rose</td>
</tr>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Red</td>
<td>2003</td>
<td>Creek</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>Red</td>
<td>2004</td>
<td>Helena</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>2001</td>
<td>Creek</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>1999</td>
<td>Helena</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Reserve</td>
<td>White</td>
<td>1999</td>
<td>Müller</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>White</td>
<td>2002</td>
<td>Bighorn</td>
</tr>
</tbody>
</table>

### ORIGIN

<table>
<thead>
<tr>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td>Château La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>Château La Pointe</td>
<td>Pomerol</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>Müller</td>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
<tr>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
</tbody>
</table>
The `create table` Statement

```sql
create table base_relation_name (  
    column_name\textsubscript{1} domain\textsubscript{1} [not null],  
    ...  
    column_name\textsubscript{k} domain\textsubscript{k} [not null])
```

▶ Effect of this command is both

▶ to store the relation schema in the data dictionary, and
▶ to prepare an “empty base relation” in the database
Possible Domains in SQL

- integer (also: integer4, int),
- smallint (also: integer2),
- float($p$) (also, for short, float),
- decimal($p$, $q$) and numeric($p$, $q$) with $q$ decimal places,
- character($n$) (also, for short, char($n$), with $n = 1$ just char) for character strings of fixed length $n$,
- character varying($n$) (also, for short, varchar($n$) for variable-length character strings up to the maximum length $n$,
- bit($n$) or bit varying($n$) like varchar but for bit strings, and
- date, time, timestamp for specifying dates, times and the combination of date and time.
Example for `create table`

```sql
create table WINES (  
    WineID int not null,  
    Name varchar(20) not null,  
    Color varchar(10),  
    Vintage int,  
    Vineyard varchar(20),  
    primary key(WineID))
```

- `primary key` marks column as `key attribute`
create table WINES (  
WineID int,  
Name varchar(20) not null,  
Color varchar(10),  
Vintage int,  
Vineyard varchar(20),  
primary key(WineID),  
foreign key(Vineyard)  
    references ORIGIN(Vineyard))

▶ foreign key marks column as a foreign key
Null Values

- **not null** precludes null values as attribute values for certain columns.
- SQL uses **null** to refer to null values; we use ⊥.
- **null** has the semantics of “unknown value”, “value does not apply” oder “value does not exist”; however, **null** itself does not belong to any domain.
- **null** can occur in any column, except for key attributes or columns marked **not null**.
Additional Notes on Data Definition in SQL

- Apart from primary and foreign keys, SQL allows specifying the following:
  - Default values for attributes using the `default` clause,
  - `create domain` statement to define custom domains (data types), and
  - `check` clause to specify further local integrity constraints within the domains, attributes and relation schemata being defined
Query Operations on Tables

- **Basic operations** on tables that allow computing new result tables from saved database tables
- Operations are combined to form the so-called **relational algebra**
- Mathematics: algebra is defined by a domain and operations defined on that domain
  → for database queries, the contents of the database are the values (of the domain), operations are **functions to compute query results**
- Query operations can be **freely combined** and form an algebra to perform “calculations on tables” – the so-called relational algebra
Relational Algebra: Overview

- Selection
- Projection

Join
Selection Projection

<table>
<thead>
<tr>
<th>a1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>b2</td>
</tr>
<tr>
<td>a2</td>
<td>b3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3</td>
<td>c4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>b3</td>
</tr>
<tr>
<td>a2</td>
<td>b4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>c4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a1</th>
<th>b2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>b2</td>
<td>c3</td>
</tr>
<tr>
<td>a2</td>
<td>b3</td>
<td>c4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a2</th>
<th>b3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>b3</td>
<td>c4</td>
</tr>
</tbody>
</table>
Selection $\sigma$

- **Selection**: Choose rows in a table based on a selection predicate

$$\sigma_{\text{Vintage} > 2000}(\text{WINES})$$

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Red</td>
<td>2003</td>
<td>Creek</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>Red</td>
<td>2004</td>
<td>Helena</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>2001</td>
<td>Creek</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>White</td>
<td>2002</td>
<td>Bighorn</td>
</tr>
</tbody>
</table>
Projection $\pi$

- **Projection**: Choose columns by specifying a list of attributes

$$\pi_{\text{Region}}(\text{ORIGIN})$$

<table>
<thead>
<tr>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>South Australia</td>
</tr>
<tr>
<td>California</td>
</tr>
<tr>
<td>Bordeaux</td>
</tr>
<tr>
<td>Hessen</td>
</tr>
</tbody>
</table>

- **Projection removes duplicate tuples.**
Natural Join

- Join: connects tables via same-named columns, combining two tuples if they have equal values in those columns

**WINES ⋊ ORIGIN**

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>...</th>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1042</td>
<td>La Rose Grand Cru</td>
<td>...</td>
<td>Ch. La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>...</td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>...</td>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>...</td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td>...</td>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Reserve</td>
<td>...</td>
<td>Müller</td>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>...</td>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
</tbody>
</table>

- The vineyard “Château La Pointe” is missing from the result ~_tuples that do not find a partner (dangling tuples), are eliminated
Combining Operations

\[ \pi \text{Name,Color,Vineyard} \left( \sigma \text{Vintage} > 2000 \left( \text{WINES} \right) \right) \times \\ \sigma \text{Region='California'} \left( \text{ORIGIN} \right) \]

yields

<table>
<thead>
<tr>
<th>Name</th>
<th>Color</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zinfandel</td>
<td>Red</td>
<td>Helena</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>White</td>
<td>Bighorn</td>
</tr>
</tbody>
</table>
### Renaming $\beta$

**Renaming to adapt attribute names:**

<table>
<thead>
<tr>
<th>WINELIST</th>
<th>RECOMMENDATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
<td>Wine</td>
</tr>
<tr>
<td>Creek Shiraz</td>
<td></td>
</tr>
<tr>
<td>Zinfandel</td>
<td></td>
</tr>
<tr>
<td>Pinot Noir</td>
<td></td>
</tr>
<tr>
<td>Riesling Reserve</td>
<td></td>
</tr>
</tbody>
</table>

**RECOMMENDATION**

| | |
| La Rose Grand Cru | |
| Riesling Reserve | |
| Merlot Selection | |
| Sauvignon Blanc | |

**Adapt with:**

$\beta_{\text{Name} \leftarrow \text{Wine}}$ (RECOMMENDATION)
Set Operations

- **Union** $r_1 \cup r_2$ of two relations $r_1$ and $r_2$: collects the tuple sets of two relations in a common schema
- Both relations must have the same set of attributes

\[ \text{WINELIST} \cup \beta_{\text{Name} \leftarrow \text{Wine}}(\text{RECOMMENDATION}) \]

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Zinfandel</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Riesling Reserve</td>
</tr>
<tr>
<td>Merlot Selection</td>
</tr>
<tr>
<td>Sauvignon Blanc</td>
</tr>
</tbody>
</table>
Set Operations /2

- **Difference** \( r_1 - r_2 \) removes from the first relation all tuples that are present in the second relation.

\[
\text{WINELIST} - \beta_{\text{Name} \leftarrow \text{Wine}}(\text{RECOMMENDATION})
\]

yields:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Zinfandel</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
</tbody>
</table>
Set Operations /3

- **Intersection** $r_1 \cap r_2$: yields all tuples that are present in both relations

\[
\text{WINELIST} \cap \beta_{\text{Name} \leftarrow \text{Wine}}(\text{RECOMMENDATION})
\]
yields:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Riesling Reserve</td>
</tr>
</tbody>
</table>
SQL Query as a Standard Language

- Query a single table

```sql
select Name, Color
from WINES
where Vintage = 2002
```

- SQL has **multi-set semantics** — SQL does not automatically suppress duplicate table entries!

- Set semantics by using **distinct**

```sql
select distinct Name
from WINES
```
Joining Tables

- Cross join as basic join

```sql
select *
from WINES, ORIGIN
```

- Join with operator `natural join`

```sql
select *
from WINES natural join ORIGIN
```

- Alternatively, join by specifying a `join condition`

```sql
select *
from WINES, ORIGIN
where WINES.Vineyard = ORIGIN.Vineyard
```
Combining Conditions

- Expression in relational algebra

\[ \pi \text{Name, Color, Vineyard} (\sigma \text{Vintage} > 2000 (\text{WINES}) \bowtie \sigma \text{Region} = \text{'California'} (\text{ORIGIN})) \]

- Query in SQL

```sql
select Name, Color, WINES.Vineyard
from WINES, ORIGIN
where Vintage > 2000 and
    Region = 'California' and
    WINES.Vineyard = ORIGIN.Vineyard
```
Set Operations in SQL

- In SQL, union is realized by an extra operator, `union`
- Differences by using nested queries

```sql
SELECT * 
FROM WINEMAKER 
WHERE Name NOT IN ( 
    SELECT Surname 
    FROM CRITIC )
```
Manipulation Operations in SQL

- **insert**: Insert one or more tuples into a base relation or view
- **update**: Change one or more tuples in a base relation or view
- **delete**: Delete one or more tuples from a base relation or view
- Local and global integrity constraints must be checked automatically by the system when executing manipulation operations.
The update Statement

Syntax:

\[
\text{update} \ \text{base\_relation} \\
\text{set} \ \text{attribute}_1 = \text{expression}_1 \\
\quad \ldots \\
\quad \text{attribute}_n = \text{expression}_n \\
\quad [\ \text{where} \ \text{condition} \ ]
\]
Example for update

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Red</td>
<td>2003</td>
<td>Creek</td>
<td>7.99</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>Red</td>
<td>2004</td>
<td>Helena</td>
<td>5.99</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>2001</td>
<td>Creek</td>
<td>10.99</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>1999</td>
<td>Helena</td>
<td>19.99</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Reserve</td>
<td>White</td>
<td>1999</td>
<td>Müller</td>
<td>14.99</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>White</td>
<td>2002</td>
<td>Bighorn</td>
<td>9.90</td>
</tr>
</tbody>
</table>

**update WINES**

**set** Price = Price * 1.10

**where** Vintage < 2000
### Example for update: New Values

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Red</td>
<td>2003</td>
<td>Creek</td>
<td>7.99</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>Red</td>
<td>2004</td>
<td>Helena</td>
<td>5.99</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>2001</td>
<td>Creek</td>
<td>10.99</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>1999</td>
<td>Helena</td>
<td>21.99</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Reserve</td>
<td>White</td>
<td>1999</td>
<td>Müller</td>
<td>16.49</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>White</td>
<td>2002</td>
<td>Bighorn</td>
<td>9.90</td>
</tr>
</tbody>
</table>
Additional Notes on `update`

- Operations on single tuples can be achieved by using the primary key:

```sql
update WINES
set Price = 7.99
where WineID = 3456
```

- Update the whole relation:

```sql
update WINES
set Price = 11
```
The delete Statement

Syntax:

```sql
delete
from base_relation
[ where condition ]
```

Delete a tuple from the WINES relation:

```sql
delete from WINES
where WineID = 4711
```
Additional Notes on delete

▶ Deletion of multiple tuples is the common case:

```
delete from WINES
where Color = 'White'
```

▶ Delete the whole relation:

```
delete from WINES
```
Additional Notes on delete /2

▶ Deletions can lead to violation of integrity constraints!
▶ Example: Violation of the foreign key property if there are still wines from this origin:

```sql
delete from ORIGIN
where District = 'Hessen'
```
The insert Statement

► Syntax:

```
insert
into base_relation
    [ (attribute_1, ..., attribute_n) ]
values (constant_1, ..., constant_n)
```

► Optional list of attributes allows for insertion of incomplete tuples
**insert Examples**

```sql
insert into ORIGIN (Vineyard, Region)
values ('Wairau Hills', 'Marlborough')
```

- Not all attributes given $\rightarrow$ Value of missing attribute District will be `null`

```sql
insert into ORIGIN
values ('Château Lafite', 'Medoc', 'Bordeaux')
```
Inserting Computed Data

▶ Syntax:

\[
\text{insert} \\
\text{into } \text{base\_relation} \\
\hspace{1cm} \left[ (\text{attribute}_1, \ldots, \text{attribute}_n) \right] \\
\text{SQL-query}
\]

▶ Example:

\[
\text{insert into WINES (} \\
\hspace{1cm} \text{select ProdID, ProdName, 'Red', ProdYear,} \\
\hspace{1cm} \text{ 'Château Lafite'} \\
\hspace{1cm} \text{from SUPPLIER} \\
\hspace{1cm} \text{where SName = 'Aspri Spirits')}
\]
Summary

- Relational model: database as a set of tables
- Integrity constraints in the relational model
- Table definition in SQL
- Relational algebra: query operators
- Basic concepts of SQL queries and manipulations
Control Questions

▶ What is a relation?
▶ What are the defining properties of the relational algebra?
▶ How are objects from the real world represented in a relational database?
▶ How can tables in SQL be defined and manipulated?
▶ What are integrity constraints?
Part III

Entity-Relationship Model
Entity-Relationship Model

1 Database Models

2 ER Model

3 Further ER Model Concepts
Educational Objective for Today . . .

- Knowing the concepts of the entity-relationship model
- Ability to conceptually model an application domain
Basics of Database Models

A database model is a system of concepts to describe databases. It defines the syntax and semantics of database descriptions for a database system.

- Database descriptions = database schemata
A Database Model Defines . . .

1. **Static properties**
   1.1 Objects
   1.2 Relationships
   
   including the primitive data types, which can describe data about the relations and objects,

2. **Dynamic properties** such as
   
   2.1 Operations
   2.2 Relationships between operations,

3. **Integrity constraints** of
   
   3.1 Objects
   3.2 Operations
Database Models

- Classical database models are especially suited for
  - Large amounts of data with a relatively static structure and
  - Describing static properties and integrity constraints
- Design models: (E)ER model, UML, ...
- Realization models: relational model, object-oriented models, ...
## Databases versus Programming Languages

<table>
<thead>
<tr>
<th>Database concept</th>
<th>Type system of a programming language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Database Model</strong></td>
<td><strong>Type system</strong></td>
</tr>
<tr>
<td>Relation, Attribute …</td>
<td>int, struct …</td>
</tr>
<tr>
<td><strong>Database schema</strong></td>
<td>Declaration of variable</td>
</tr>
<tr>
<td>relation WINE = (...)</td>
<td>var x: int, y: struct Wine</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>Values</td>
</tr>
<tr>
<td>WINE(4961, 'Chardonnay', 'White', ...)</td>
<td>42, 'Cabernet Sauvignon'</td>
</tr>
</tbody>
</table>
# Levels of Abstraction

<table>
<thead>
<tr>
<th>Models</th>
<th>Data</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>entity-relationship model</td>
<td>structograms</td>
</tr>
<tr>
<td>concrete</td>
<td>hierarchical model</td>
<td>Pascal</td>
</tr>
<tr>
<td></td>
<td>network model</td>
<td>C, C++</td>
</tr>
<tr>
<td></td>
<td>relational model</td>
<td>Java, C#</td>
</tr>
</tbody>
</table>

---

Prof. Thomas Leich  
Harz University of Applied Sciences  
Copyright: Gunter Saake, University of Magdeburg
Overview of Database Models

- HM
- NWM
- RM
- SQL
- NF²
- eNF²
- ER
- SDM
- OEM
- UML
- ODMG
- ORDM
- SQL:1999
- SQL:2003
- SQL:2016
- XML
- NoSQL
- NewSQL
- SQL:2016

Mid 1960
1970
1980
1990
2000
2010
2020

close to implementation
abstract

Copyright: Gunter Saake, University of Magdeburg

Page 94/523
Overview of Database Models /2

▶ HM: hierarchical model, NWM: network model, RM: relational model
▶ NF²: model of nested (non-first-normal form = NF²) relations, eNF²: extended NF² model
▶ ER: entity-relationship model, SDM: semantic data models
▶ OODM / C++: object-oriented data models based on object-oriented programming languages, such as C++, OEM: object-oriented design models (e.g., UML), ORDM: object-relational data models
The ER Model

Entity: object of the real or a virtual world, about which information is to be stored, e.g., **Products** (wine, catalog), winemaker or critic; but also information about events, e.g., **Orders**

Relationship: describes a relationship between entities, e.g., a customer **orders** a wine or wine is being **offered** by a winemaker

Attribute: represents a property of an entity or a relationship, e.g., **Name** of customer, **Color** of a wine or **Date** of an order
ER Example

- **Grape**
  - **Name**
  - **Color**
  - **Proportion**
  - **Located in**
  - **Produced by Producer**
  - **Made of**
  - **Res. Sugar**
  - **Year**

- **Wine**
  - **Name**
  - **Color**
  - **Made of**
  - **Proportion**
  - **Region**
  - **Location in**
  - **Producer**
  - **Vineyard**
  - **Address**

- **Dish**
  - **Name**
  - **Side dish**
  - **Proportion**
  - **Recommended by**

- **Critic**
  - **Name**
  - **Organization**

- **License**
  - **LicenseNo.**
  - **Amount**
  - **Has**

- **Organization**
  - **Name**

- **Country**

- **Region**

- **Organization**
  - **Name**

- **Producer**
  - **Vineyard**
  - **Address**

- **License**
  - **LicenseNo.**
  - **Amount**

- **Recommended by**
  - **Critic**
  - **Organization**
Values

- **Values**: primitive elements of data, which can be represented directly
- Value domains are described by **datatypes**, which, apart from the set of possible values, also characterize the basic operations on those values
- ER model: pre-defined primitive datatypes, such as the integers `int`, the character sequences `string`, dates `date` etc.
- Every datatype represents a domain, including operations and predicates on values of this domain
Entities

- **Entities** are the pieces of information to be represented in a database.
- In contrast to values, entities cannot be represented directly. They can only be observed through their properties.
- Entities are grouped according to their entity types, such as $E_1, E_2$...

Set of current entities:

$$\sigma(E_1) = \{e_1, e_2, \ldots, e_n\}$$
Attribute

- **Attribute** models properties of entities or relationships
- All entities of an entity type have the same kinds of properties; attributes are therefore declared for the entity type

![Diagram showing entity Wine with attributes Name, Color, and Year]

- Textual notation $E(A_1 : D_1, \ldots, A_m : D_m)$
Key-based Identification

Key attributes: Subset of all attributes of an entity type $E(A_1, \ldots, A_m)$

$\{S_1, \ldots, S_k\} \subseteq \{A_1, \ldots, A_m\}$

In every state of the database, current values of the key attributes uniquely identify instances of the entity type $E$

If multiple keys would be possible: Choice of a primary key

Notation: Highlight by underlining:

$E(\ldots, \underline{S_1}, \ldots, S_i, \ldots)$
Relationship Types

- Relationships between entities are grouped into relationship types.
- In general: arbitrary number $n \geq 2$ of entity types can participate in a relationship type.
- Every $n$-ary relationship type $R$ refers to $n$ entity types $E_1, \ldots, E_n$.
- Instances of a relationship type:

$$\sigma(R) \subseteq \sigma(E_1) \times \sigma(E_2) \times \cdots \times \sigma(E_n)$$
Relationship Types /2

- **Notation**

  ![Diagram](image)

  \[ R(E_1, E_2, \ldots, E_n) \]

- **Textual notation:** If an entity type participates in a relationship type multiple times: roles can be assigned

  married(Wife: Person, Husband: Person)
Relationship Attributes

- Relationships can also have attributes.
- Attribute are declared at the relationship type; this also holds for the set of possible values \( \sim \text{relationship attributes} \).

Textual notation: \( R(E_1, \ldots, E_n; A_1, \ldots, A_k) \)
Characteristics of Relationships

▶ **Degree:**
  ▶ Number of participating entity types
  ▶ Often: binary
  ▶ Example: *Supplier supplies Product*

▶ **Cardinality Constraints:**
  ▶ Number of incoming instances of an entity type
  ▶ Typical forms: 1:1, 1:n, m:n
  ▶ Represent integrity constraints
  ▶ Example: *maximum of 5 Products per Order*
Binary vs. N-ary Relationships
Instances in the Example
Reconstruction of Instances

\[\begin{align*}
  d_1 &\rightarrow c_1 &\rightarrow w_1 \\
  d_1 &\rightarrow c_2 &\rightarrow w_2 \\
  d_2 &\rightarrow c_2 &\rightarrow w_1 \\
  d_1 &\rightarrow c_2 &\rightarrow w_1
\end{align*}\]

\[\begin{align*}
  \text{But also: } d_1 &- c_2 - w_1
\end{align*}\]
1:1-Relationships

▶ Every $e_1$ of entity type $E_1$ is assigned to at most one entity $e_2$ out of $E_2$ and vice versa

▶ Examples: *Brochure describes Product*, *Husband is married to Wife*
1:N Relationships

- Every entity $e_1$ of entity type $E_1$ is assigned to an arbitrary number of entities $E_2$, but for every entity $e_2$, there is at most one $e_1$ in $E_1$
- Examples: Supplier supplies Product, Mother has Children
N:1 Relationship

- Inverse of 1:N, also **functional** relationship
- Binary relationships that define a **function**: Every entity of entity type $E_1$ is assigned to at most one entity of entity type $E_2$.

$$R : E_1 \rightarrow E_2$$
1:1 Relationship

Licence

Has

Producer
M:N Relationships

- No restrictions
- Example: *Order consists of Products*

![Diagram showing M:N relationship between E₁ and E₂]
[\text{min, max}] \text{ Notation}

\[
\begin{align*}
\textbf{E}_1 & \quad \text{[min}_1, \text{ max}_1] \\
\diamond \quad \text{R} & \quad \text{[min}_n, \text{ max}_n] \\
\textbf{E}_2 & \quad \text{[min}_2, \text{ max}_2] \\
\vdots
\end{align*}
\]

- Restricts the possible \textbf{number of times} an instance of an entity type can participate in a relationship by giving a minimum and a maximum value
- Notation for expressing cardinalities in a relationship type:

\[
R(\textbf{E}_1, \ldots, \textbf{E}_i[\text{min}_i, \text{max}_i], \ldots, \textbf{E}_n)
\]

- Cardinality constraints:

\[
\text{min}_i \leq |\{r \mid r \in R \land r.E_i = \epsilon_i\}| \leq \text{max}_i
\]

- Special notation for $\text{max}_i$ is $\ast$
Expressing Cardinalities

- \([0, \ast]\) means “no restrictions” (default)
- \(R(E_1[0, 1], E_2)\) corresponds to a (partial) functional relationship \(R : E_1 \rightarrow E_2\), because every instance out of \(E_1\) is assigned to at most one instance out of \(E_2\)
- Total functional relationships are modelled by \(R(E_1[1, 1], E_2)\)
Expressing Cardinalities: Examples

► Partial functional relationship

\[ \text{stored\_on(P\_{0,1},S\_{0,3})} \]

“Every product in the warehouse is stored on one shelf. However, products that are currently out of stock are not assigned to a shelf. At most three products can share the same shelf.”

► Total functional relationship

\[ \text{supplies(S\_{0,*},P\_{1,1})} \]

“Every product is supplied by exactly one supplier. However, a supplier can very well supply more than one product.”
Alternative Ways to Express Cardinalities

Product \( [1,1] \) \( \xrightarrow{\text{Delivered By}} \) \( [0,*] \) Supplier

Product \( N \) \( \xrightarrow{\text{Delivered By}} \) Supplier \( 1 \)
Dependent Entity Types

- *Dependent Entity Type*: Identification through functional relationship

![ER diagram](image)

- Dependent entities in the ER model: Functional relationship used as key
Dependent Entity Types /2

Possible instantiations for dependent entities

- Name: Pinot Noir  
  Color: Red  
  Year: 2004  
  Res. Sugar: 1,2

- Name: Zinfandel  
  Color: Red  
  Year: 2003  
  Res. Sugar: 1,4

- Name: Riesling Reserve  
  Color: Weiß  
  Year: 1999  
  Res. Sugar: 6,7
Dependent Entity Types /3

Alternative notation

- Vintage Year
- Belongs To
- Wine

- Year
- Res.
- Sugar
- Name
- Color
The IS-A Relationship

- Specialization/generalization relationship or IS-A relationship
- Textual notation: $E_1 \text{ IS-A } E_2$
- IS-A relationship semantically corresponds to an injective functional relationship

![Diagram showing IS-A relationship between Sparkling Wine and Wine with attributes Production, Name, and Color]
Properties of the IS-A Relationship

- Every sparkling wine instance is assigned to exactly one wine instance
  \[ \leadsto \] sparkling wine instances are identified by their functional IS-A relationship

- Not every wine is a sparkling wine

- Attributes of the entity type \texttt{Wine} also apply to sparkling wines: “inherited” attributes
  \[
  \text{Sparkling\_wine(Name,Color,Production) of \texttt{Wine}}
  \]

- Not only attribute declarations are inherited, but also the current values of each instance
Instantiations of IS-A Relationship

Sparkling Wine

Wine

w1
w2
w3
w4
w5
w6

w1
w2
w4

w1
w2
w4

Prof. Thomas Leich
Harz University of Applied Sciences
Copyright: Gunter Saake, University of Magdeburg
Alternative Notation for IS-A Relationship

Production

Sparkling Wine

Name

Wine

Color

Wine

Sparkling Wine
Expressing Cardinalities: IS-A

- It holds for every relationship $E_1 \text{ IS-A } E_2$ that: $\text{IS-A}(E_1[1, 1], E_2[0, 1])$
- Every instance of $E_1$ participates exactly once in the IS-A relationship, whereas instances of the supertype $E_2$ do not have to participate
- This does not affect aspects like attribute inheritance
Optionality of Attributes

![Database schema diagram]

- Vineyard
- Address
- Name
- Region
- Country

Producers are associated with areas, which may or may not have an address. Areas can be located in various countries and regions.
## Overview of Concepts

<table>
<thead>
<tr>
<th>Term</th>
<th>Informal Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>The piece of information to be represented</td>
</tr>
<tr>
<td>Entity type</td>
<td>Grouping of entities with the same properties</td>
</tr>
<tr>
<td>Relationship type</td>
<td>Grouping of relationships between entities</td>
</tr>
<tr>
<td>Attribute</td>
<td>Property value of an entity or a relationship</td>
</tr>
<tr>
<td>Key</td>
<td>Identifying property of an entity</td>
</tr>
<tr>
<td>Cardinalities</td>
<td>Restrict relationship types with regards to the number of times an entity can participate in a relationship</td>
</tr>
<tr>
<td>Degree</td>
<td>Number of entity types that participate in a relationship type</td>
</tr>
<tr>
<td>Functional relationship</td>
<td>Relationship Type with functional property</td>
</tr>
<tr>
<td>Dependent entities</td>
<td>Entities that cannot exist independently from other entities</td>
</tr>
<tr>
<td>IS-A relationship</td>
<td>Specialization of entity types</td>
</tr>
<tr>
<td>Optionality</td>
<td>Attribute or functional relationships as partial functions</td>
</tr>
</tbody>
</table>
Summary

- Database model, database schema, database (instance)
- Entity-relationship model
- Further concepts of the ER model
- Based on: chapter 3 in Datenbanken - Konzepte und Sprachen von Gunter Saake, Kai-Uwe Sattler und Andreas Heuer and chapter 7 in Fundamentals of Database Systems by Ramez Elmasri and Shamkant B. Navathe
Control Questions

▶ What defines a database model? What is the distinction between model and schema?
▶ Which concepts does the ER model define?
▶ Which properties characterize relationship types?
▶ How are dependent entity types different from regular entity types?
Part IV

Database Design
Database Design

1. Phases of Database Design
2. Further Steps During Design
3. Capacity-preserving Transformations
4. ER-to-RM Transformation
Educational Objective for Today . . .

- Goals and steps of the database design process
- Rules to transform ER schemata into relational schemata
Goal of Design

▶ Data management for multiple application systems, for multiple years
▶ Therefore: special importance
▶ Design requirements
  ▶ For every application, it should be possible to derive application data from data in the database — efficiently
  ▶ Only store “sensible” (actually needed) data
  ▶ Avoid redundancies
Phase Model

- Requirement Analysis
- Conceptional Design
- Distribution Design
- Logical Design
- Data Definition
- Physical Design
- Implementation & Maintenance
Requirements Analysis

▶ **Approach:** Collecting information needs from all specialist divisions

▶ **Result:**
  ▶ Informal description (text, tabular lists, forms, etc.) of the problem domain
  ▶ Separation of the information about data (data analysis) from the information about functions (functional analysis)

▶ **“Classical” DB design:**
  ▶ Only data analysis and following steps

▶ **Functional design:**
  ▶ See methods of software engineering
Conceptual Design

- First formal description of the problem domain
- Language means: semantical data model
- Process:
  - Modeling of views, e.g., for different specialist divisions
  - Analysis of existing views with respect to conflicts
  - Integration of views into a full schema
- Result: full conceptual schema, e.g., ER diagram
Phases of Conceptual Design

Conceptional Design

View Design

View Analysis

View Integration
Further Steps During Design

▶ ER modeling of different views of the complete information, e.g., for different specialist divisions of a company claimer conceptual design
  ▶ Analysis and integration of views
  ▶ Result: full conceptual schema
▶ Distribution design when using distributed storage
▶ Transformation to concrete implementation model (e.g., relational model) claimer logical design
▶ Data definition, implementation and maintenance claimer physical design
View Integration

- Analysis of existing view with respect to conflicts
- Integration of views into a full schema
Integration Conflicts

- **Naming conflicts**: Homonyms / synonyms
  - Homonyms: bank (money / river); order (command / request for goods)
  - Synonyms: car, vehicle, automobile

- **Typing conflicts**: different structures for the same element

- **Domain mismatch**: different domains for an element

- **Identifier conflicts**: e.g., different keys for the same element

- **Structural conflicts**: same fact expressed in different ways
Distribution Design

- If data should be distributed to several machines, a way of distributed storage must be determined

- E.g., for a relation `CUSTOMER (CNo, Name, Address, Zipcode, Account)`

  - **Horizontal** distribution:
    - `CUSTOMER_1 (CNo, Name, Address, Zipcode, Account)`
    - `where` Zipcode < 50 000
    - `CUSTOMER_2 (CNo, Name, Address, Zipcode, Account)`
    - `where` Zipcode >= 50 000

  - **Vertical** distribution (connection via attribute CNo):
    - `CUSTOMER_Adr (CNo, Name, Address, Zipcode)`
    - `CUSTOMER_Account (CNo, Account)`
Logical Design

- **Language means:** Data model of the chosen "implementation" DBMS, e.g., relational model

- **Process:**
  1. (Automatical) transformation of the conceptual schema, e.g.,
     ER → relational model
  2. Improvement of the relational schema based on quality criteria
     (normalization, see Chapter 6):
     Design goals: avoid redundancies, . . .

- **Result:** logical schema, e.g., collection of relation schemata
Data Definition

- Translation of logical schema into a concrete schema
- **Language means:** DDL and DML of DBMS (e.g., Oracle, DB2, SQL Server)
  - Database declaration in the DDL of the DBMS
  - Realization of integrity constraints
  - Definition of views
Physical Design

▶ Supplement physical design with support for efficient access, e.g., by defining indexes
▶ Index
  ▶ Access path: data structure for additional, key-based access to tuples (⟨key attribute value, tuple address⟩)
  ▶ Usually implemented as a B*-tree
▶ Language means: storage structure (definition) language SSL
Indexes in SQL

```
create [ unique ] index indexname
    on relname ( attrname [ asc | desc ], attrname [ asc | desc ], ...
         )
```

▶ Example

```
create index WineIdx on WINES (Name)
```
Necessity of Access Paths

- Example: Table with 100 GB of data, hard disk transfer rate of ca. 50 MB/s
- Operation: Search for a tuple (selection)
- Implementation: Sequential search
- Cost: $\frac{102.400}{50} = 2.048$ sec. $\approx 34$ min.
Implementation and Maintenance

▶ Phases of . . .
  ▶ Maintenance,
  ▶ Further optimization of the physical layer,
  ▶ Adaptation to new requirements or operating system platforms,
  ▶ Porting to new database management systems
  ▶ etc.
Transformation of the Conceptual Schema

- Translation to logical schema
  - Example: ER $\rightarrow$ RM
  - Correct?
  - Quality of transformation?

- Preservation of *information capacity*
  - Is it possible, after the transformation, to store exactly the same data as before?
  - ... or more?
  - ... or less?
Capacity-increasing Transformation

- Transformation into

\[ R = \{\text{LicenseNo}, \text{Vineyard}\} \]

with exactly one key

\[ K = \{\{\text{LicenseNo}\}\} \]

- Possible invalid relation:

<table>
<thead>
<tr>
<th>Has</th>
<th>LicenseNo</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>007</td>
<td>Helena</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>Helena</td>
</tr>
</tbody>
</table>
Capacity-preserving Transformation

- Correct instantiation

\[
\text{Has} \quad \begin{array}{|c|c|}
\hline
\text{LicenseNo} & \text{Vineyard} \\
\hline
007 & Helena \\
42 & Müller \\
\hline
\end{array}
\]

- Correct set of keys

\[
K = \{\{\text{LicenseNo}\}, \{\text{Vineyard}\}\}
\]
Capacity-decreasing Transformation

Relation schema with one key \{WName\}

Instantiation that is no longer valid:

<table>
<thead>
<tr>
<th>ConsistsOf</th>
<th>WName</th>
<th>GrapeName</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Zinfandel Red Blossom</td>
<td>Zinfandel</td>
</tr>
<tr>
<td></td>
<td>Bordeaux Blanc</td>
<td>Cabernet Sauvignon</td>
</tr>
<tr>
<td></td>
<td>Bordeaux Blanc</td>
<td>Muscadelle</td>
</tr>
</tbody>
</table>

Capacity-preserving when using the keys of both entity types as the new key in the relation schema

\[ K = \{\{WName, GrapeName\}\} \]
Example Transformation ER-RM: Input

```
Grape
  Consists Of
  Amount
  Color
  Grape Name

Producer
  Vineyard
  Address

Wine
  WName
  Color
  Year
  Res. Sugar

Produce
```
Example Transformation ER-RM: Result

1. GRAPE = \{\text{Color}, \text{GrapeName}\} with \ K_{\text{GRAPE}} = \{\{\text{GrapeName}\}\}

2. ConsistsOf = \{\text{GrapeName}, \text{WName}, \text{Amount}\} with
   \ K_{\text{ConsistsOf}} = \{\{\text{GrapeName}, \text{WName}\}\}

3. WINE = \{\text{Color}, \text{WName}, \text{Vintage}, \text{Res.Sugar}\} with
   \ K_{\text{WINE}} = \{\{\text{WName}\}\}

4. PRODUCE = \{\text{WName}, \text{Vineyard}\} with \ K_{\text{PRODUCE}} = \{\{\text{WName}\}\}

5. PRODUCER = \{\text{Vineyard}, \text{Address}\} with
   \ K_{\text{PRODUCER}} = \{\{\text{Vineyard}\}\}
ER Transformation into Relations

- **Entity types and relationship types**: both transformed into relation schemata
- **Attributes**: attributes of the relation schema, keys are adopted
- **Cardinalities** of the relationships: expressed in respective relation schemata by choice of keys
- In some cases: **merge** of the relation schemata of entity and relationship types
- Introduce foreign key constraints between the remaining relation schemata
Transformation of Relationship Types

- New relation schema with all attributes of the relationship type; additionally, adopt all primary keys of the participating entity types

- **Determining keys:**
  - **m:n relationship:** both primary keys together form the key in the new relation schema
  - **1:n relationship:** primary keys of the n-side (in the functional notation, this is the side without the arrowhead) form key in the new relation schema
  - **1:1 relationship:** both primary keys become a key in the new relation schema; the primary key is then chosen from these keys
**n:m Relationships**

- **WName**, **Color**, **Amount**
- **Grape Name**, **Color**
- **Wine Consists Of**
  - **Grape Name**, **WName**, **Amount**
  - **Grape Name**
  - **WName**
  - **Color**

**Transformation**

1. \(\text{GRAPE} = \{\text{Color}, \text{GrapeName}\}\) with \(K_{\text{GRAPE}} = \{\{\text{GrapeName}\}\}\)
2. \(\text{ConsistsOf} = \{\text{GrapeName}, \text{WName}, \text{Amount}\}\) with \(K_{\text{ConsistsOf}} = \{\{\text{GrapeName}, \text{WName}\}\}\)
3. \(\text{WINE} = \{\text{Color}, \text{WName}, \text{Vintage}, \text{Res. Sugar}\}\) with \(K_{\text{WINE}} = \{\{\text{WName}\}\}\)

**Attributes** **GrapeName** and **WName** together are key
1:n Relationships

- (Preliminary) transformation
  - PRODUCER with the attributes Vineyard and Address,
  - AREA with the attributes Name and Region, and
  - LocatedIn with the attributes Vineyard and Name and the primary key of the $n$-side Vineyard as primary key of this schema.
Possible Merges

- **Optional relationships** ([0,1] or [0,n]) are not merged
- With cardinalities [1,1] or [1,n] (**mandatory relationships**), merge is possible:
  - **1:n relationship**: the entity-relation schema of the n-side can be integrated into the relation schema of the relationship
  - **1:1 relationship**: both entity-relation schemata can be integrated into the relation schema of the relationship
1:1 Relationships

▶ (Preliminary) transformation
  ▶ PRODUCER with the attributes Vineyard and Address
  ▶ LICENSE with the two attributes LicenseNo and Hectoliters
  ▶ Has with the primary keys of both participating entity types each as key of this schema, that is LicenseNo and Vineyard
1:1 Relationships: Merge

- Transformation with merge
  - Merged relation:

<table>
<thead>
<tr>
<th>PRODUCER</th>
<th>Vineyard</th>
<th>Address</th>
<th>LicenseNo</th>
<th>Hectoliters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vineyard</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotkäppchen</td>
<td>Freiberg</td>
<td>42-007</td>
<td>10 000</td>
<td></td>
</tr>
<tr>
<td>Vineyard Müller</td>
<td>Dagstuhl</td>
<td>42-009</td>
<td>250</td>
<td></td>
</tr>
</tbody>
</table>

- Producers without license require null values:

<table>
<thead>
<tr>
<th>PRODUCER</th>
<th>Vineyard</th>
<th>Address</th>
<th>LicenseNo</th>
<th>Hectoliters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vineyard</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotkäppchen</td>
<td>Freiberg</td>
<td>42-007</td>
<td>10 000</td>
<td></td>
</tr>
<tr>
<td>Vineyard Müller</td>
<td>Dagstuhl</td>
<td>⊥</td>
<td>⊥</td>
<td></td>
</tr>
</tbody>
</table>

- Free Licenses lead to additional null values:

<table>
<thead>
<tr>
<th>PRODUCER</th>
<th>Vineyard</th>
<th>Address</th>
<th>LicenseNo</th>
<th>Hectoliters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vineyard</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotkäppchen</td>
<td>Freiberg</td>
<td>42-007</td>
<td>10 000</td>
<td></td>
</tr>
<tr>
<td>Vineyard Müller</td>
<td>Dagstuhl</td>
<td>⊥</td>
<td>⊥</td>
<td>42-003</td>
</tr>
</tbody>
</table>
Dependent Entity Types

Transformation

1. \( \text{WINEVINTAGE} = \{\text{WName}, \text{Res.Sugar}, \text{Year}\} \) with \( K_{\text{WINEVINTAGE}} = \{\{\text{WName}, \text{Year}\}\} \)
2. \( \text{WINE} = \{\text{Color}, \text{WName}\} \) with \( K_{\text{WINE}} = \{\{\text{WName}\}\} \)

- Attribute \text{WName} in \text{WINEVINTAGE} is foreign key to relation \text{WINE}
IS-A Relationship

Transformation

1. WINE = \{Color, WName\} with \(K_{WINE} = \{|\{WName\}|\}
2. SPARKLING_WINE = \{WName, Production\} with \(K_{SPARKLING_WINE} = \{|\{WName\}|\}

- WName in SPARKLING_WINE is foreign key with respect to relation WINE
Recursive Relationships

Transformation

1. $\text{AREA} = \{\text{Name, Region}\}$ with $K_{\text{AREA}} = \{\{\text{Name}\}\}$
2. $\text{ADJOINS} = \{\text{to, from}\}$ with $K_{\text{ADJOINS}} = \{\{\text{to, from}\}\}$
Recursive Functional Relationships

▶ Transformation

1. \( \text{CRITIC} = \{\text{Name}, \text{Organization}, \text{Mentorname}\} \) with \( K_{\text{CRITIC}} = \{\{\text{Name}\}\} \)

▶ Mentorname is foreign key to attribute Name of relation CRITIC.
N-ary Relationships

Every participating entity type is treated according to the rules stated above.

For relationship `Recommends`, the primary keys of the three participating entity types are included in the resulting relation schema.

Relationship has a generic type (k:m:n relationship): all primary keys together form the key.
N-ary Relationships: Result

1. \( \text{RECOMMENDS} = \{\text{WName}, \text{DName}, \text{Name}\} \) with 
   \( K_{\text{RECOMMENDS}} = \{\{\text{WName}, \text{DName}, \text{Name}\}\} \)
2. \( \text{DISH} = \{\text{DName}, \text{Side\_Dish}\} \) with 
   \( K_{\text{DISH}} = \{\{\text{DName}\}\} \)
3. \( \text{WINE} = \{\text{Color}, \text{WName}, \text{Vintage}, \text{Res\_Sugar}\} \) with 
   \( K_{\text{WINE}} = \{\{\text{WName}\}\} \)
4. \( \text{CRITIC} = \{\text{Name}, \text{Organization}\} \) with 
   \( K_{\text{CRITIC}} = \{\{\text{Name}\}\} \)

▶ The three key attributes of \( \text{RECOMMENDS} \) are foreign keys to 
the respective source relations (\( \text{CRITIC, WINE, DISH} \)).
Overview of Transformations

<table>
<thead>
<tr>
<th>ER Concept</th>
<th>Is Translated into Relational Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity type $E_i$</td>
<td>Relation schema $R_i$</td>
</tr>
<tr>
<td>Attributes of $E_i$</td>
<td>Attributes of $R_i$</td>
</tr>
<tr>
<td>Primary key $P_i$</td>
<td>Primary key $P_i$</td>
</tr>
<tr>
<td>Relationship type</td>
<td>Relation schema $R_i$</td>
</tr>
<tr>
<td>Its attributes</td>
<td>Attributes: $P_1$, $P_2$</td>
</tr>
<tr>
<td>$1 : n$</td>
<td>Further attributes</td>
</tr>
<tr>
<td>$1 : 1$</td>
<td>$P_2$ becomes primary key of the relationship</td>
</tr>
<tr>
<td>$m : n$</td>
<td>$P_1$ and $P_2$ become key of the relationship</td>
</tr>
<tr>
<td>$1 : n$</td>
<td>$P_1 \cup P_2$ becomes primary key of the relationship</td>
</tr>
<tr>
<td>IS-A relationship</td>
<td>$R_1$ gets an additional key $P_2$</td>
</tr>
</tbody>
</table>

$E_1$, $E_2$: Entity types participating in a relationship,
$P_1$, $P_2$: Their primary keys,
$1 : n$ relationship: $E_2$ is $n$-side,
IS-A relationship: $E_1$ is a special entity type
Summary

- Phases of database design
- Capacity-preserving transformations
- Transformation ER → relational
Control Questions

▶ Which steps does the database design process comprise?
▶ Which requirements do the transformations between each design step have to fulfill? Why?
▶ How are concepts of the ER model translated into concepts of the relational model?
▶ How are the different cardinalities of relationship types accounted for during transformation?
Part V

Relational Database Design Theory
Relational Database Design Theory

1. Target Model of the Logical Design
2. Relational DB Design
3. Normal Forms
4. Transformation Properties
5. Design Methods
Educational Objective for Today . . .

- Know how to refine the relational design
- Understanding of normal forms
- Methodology and techniques for normalization
## Relation Model

### WINES

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1042</td>
<td>La Rose Grand Cru</td>
<td>Rot</td>
<td>1998</td>
<td>Château La Rose</td>
</tr>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Rot</td>
<td>2003</td>
<td>Creek</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>Rot</td>
<td>2004</td>
<td>Helena</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>Rot</td>
<td>2001</td>
<td>Creek</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td>Rot</td>
<td>1999</td>
<td>Helena</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Reserve</td>
<td>Weiś</td>
<td>1999</td>
<td>Müller</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>Weiś</td>
<td>2002</td>
<td>Bighorn</td>
</tr>
</tbody>
</table>

### PRODUCER

<table>
<thead>
<tr>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td>Château La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>Château La Pointe</td>
<td>Pomerol</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>Müller</td>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
<tr>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
</tbody>
</table>
## Terms of the Relational Model

<table>
<thead>
<tr>
<th>Term</th>
<th>Informal Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Column of a table</td>
</tr>
<tr>
<td>Value domain</td>
<td>Possible values of an attribute</td>
</tr>
<tr>
<td>Attribute value</td>
<td>Element of a value domain</td>
</tr>
<tr>
<td>Relation schema</td>
<td>Set of attributes</td>
</tr>
<tr>
<td>Relation</td>
<td>Set of rows in a table</td>
</tr>
<tr>
<td>Tuple</td>
<td>Row in a table</td>
</tr>
<tr>
<td>Database schema</td>
<td>Set of relation schemas</td>
</tr>
<tr>
<td>Database</td>
<td>Set of relations (base relations)</td>
</tr>
</tbody>
</table>
## Terms of the Relational Model /2

<table>
<thead>
<tr>
<th>Term</th>
<th>Informal Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Minimal set of attributes, whose values uniquely identify a tuple in a table</td>
</tr>
<tr>
<td>Primary key</td>
<td>A key designated during database design</td>
</tr>
<tr>
<td>Foreign key</td>
<td>Set of attributes that are key in another relation</td>
</tr>
<tr>
<td>Foreign key constraint</td>
<td>All attribute values of the foreign key show up as keys in the other relation</td>
</tr>
</tbody>
</table>
Integrity Constraints

- Identifying set of attributes \( K := \{B_1, \ldots, B_k\} \subseteq R: \)
  \[
  \forall t_1, t_2 \in r \ [t_1 \neq t_2 \implies \exists B \in K : t_1(B) \neq t_2(B)]
  \]

- **Key**: is minimal identifying set of attributes
  - \( \{\text{Name}, \text{Vintage}, \text{Vineyard}\} \) and
  - \( \{\text{WineID}\} \) for WINES

- **Prime attribute**: element of a key
- **Primary key**: designated key
- **Superkey**: every superset of a key (= identifying set of attributes)
- **Foreign key**: \( X(R_1) \rightarrow Y(R_2) \)
  \[
  \{t(X) | t \in r_1\} \subseteq \{t(Y) | t \in r_2\}
  \]
Relation with Redundancies

WINES

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>...</th>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1042</td>
<td>La Rose Gr. Cru</td>
<td></td>
<td>Ch. La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td></td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td></td>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td></td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td></td>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Res.</td>
<td></td>
<td>MÃ¼ller</td>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td></td>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
</tr>
</tbody>
</table>
Update Anomalies

- Insertion into the redundancy-containing relation WINES:

```sql
insert into WINES (WineID, Name, Color, Vintage, Vineyard, District, Region)
values (4711, 'Chardonnay', 'Weiś', 2004, 'Helena', 'Rheingau', 'California')
```

- WineID 4711 already assigned to another wine: violates FD WineID → Name

- Up to now, vineyard Helena was located in Napa Valley: violates FD Vineyard → District

- Rheingau is not located in California: violates FD District → Region

- Also: update- and delete anomalies
Functional Dependencies

- **Functional dependency** between two sets of attribute $X$ and $Y$ of a relation holds iff for each tuple of the relation, the attribute values of the $X$ components determine the attribute values of the $Y$ components.

- If two tuples have the same values for the $X$ attributes, they also have the same values for all $Y$ attributes.

- Notation for functional dependency (FD): $X \rightarrow Y$

- Example:
  
  WineID → Name, Vineyard
  District → Region

- But not: Vineyard → Name
Keys as a Special Case

▶ For example on Slide 5-177
  WineID → Name, Color, Vintage, Vineyard, District, Region

▶ Always: WineID → WineID,
  then whole schema on the right side

▶ If left side minimal: Key

▶ Formally: \( X \) is key if \( FD \ X \rightarrow R \) holds for relation schema \( R \)
  and \( X \) is minimal

Goal of database design: Transform all existing functional
dependencies into “key dependencies”, without losing
semantic information
Deriving FDs

\[ r \begin{array}{|c|c|c|} \hline
A & B & C \\
\hline
a_1 & b_1 & c_1 \\
a_2 & b_1 & c_1 \\
a_3 & b_2 & c_1 \\
a_4 & b_1 & c_1 \\
\hline
\end{array} \]

- Satisfies \( A \to B \) and \( B \to C \)
- Then \( A \to C \) also holds
- Not derivable: \( C \to A \) or \( C \to B \)
Deriving FDs /2

- If for \( f \) over \( R \), it holds that \( \text{SAT}_R(F) \subseteq \text{SAT}_R(f) \), then \( F \) implies the FD \( f \) (short: \( F \models f \))

- Previous example:

\[
F = \{ A \rightarrow B, B \rightarrow C \} \models A \rightarrow C
\]

- Computing the closure: Determine all functional dependencies that can be derived from a given set of FDs

- Closure \( F_R^+ := \{ f \mid (f \text{ FD over } R) \land F \models f \} \)

- Example:

\[
\{ A \rightarrow B, B \rightarrow C \}^+ = \{ A \rightarrow B, B \rightarrow C, A \rightarrow C, AB \rightarrow C, A \rightarrow BC, \ldots, AB \rightarrow AB, \ldots \}
\]
## Derivation Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Reflexivity</td>
<td>$X \supseteq Y \implies X \rightarrow Y$</td>
</tr>
<tr>
<td>F2</td>
<td>Augmentation</td>
<td>${X \rightarrow Y} \implies XZ \rightarrow YZ$ and $XZ \rightarrow Y$</td>
</tr>
<tr>
<td>F3</td>
<td>Transitivity</td>
<td>${X \rightarrow Y, Y \rightarrow Z} \implies X \rightarrow Z$</td>
</tr>
<tr>
<td>F4</td>
<td>Decomposition</td>
<td>${X \rightarrow YZ} \implies X \rightarrow Y$</td>
</tr>
<tr>
<td>F5</td>
<td>Union</td>
<td>${X \rightarrow Y, X \rightarrow Z} \implies X \rightarrow YZ$</td>
</tr>
<tr>
<td>F6</td>
<td>Pseudo-transitivity</td>
<td>${X \rightarrow Y, WY \rightarrow Z} \implies WX \rightarrow Z$</td>
</tr>
</tbody>
</table>

F1-F3 known as **Armstrong axioms** (sound, complete)

- **Sound**: Rules do not derive FDs that are not logically implied
- **Complete**: All implied FDs are derived
- **Independent** (i.e., minimal w.r.t. $\subseteq$): No rule can be omitted

---

1 w.r.t. = with respect to
Alternative Set of Rules

- B-Axioms or RAP-rules
  - Reflexivity \( \{\} \implies X \rightarrow X \)
  - Accumulation \( \{X \rightarrow YZ, Z \rightarrow AW\} \implies X \rightarrow YZA \)
  - Projectivity \( \{X \rightarrow YZ\} \implies X \rightarrow Y \)

- Rule set is complete because it allows to derive the Armstrong axioms
Membership Problem

Can a certain FD $X \rightarrow Y$ be derived from a given set $F$, i.e., is it implied by $F$?

Membership problem: “$X \rightarrow Y \in F^+$?”

- Closure over a set of attributes $X$ w.r.t. $F$ is
  $X_F^+ := \{ A \mid X \rightarrow A \in F^+ \}$

- Membership problem can be solved in linear time by solving the modified problem

  Membership problem (2): “$Y \subseteq X_F^+$?”
Algorithm CLOSURE

- Compute $X_F^+$, the closure of $X$ w.r.t. $F$

CLOSURE$(F, X)$:

$X^+ := X$

repeat

$\overline{X}^+ := X^+ /* R\text{-}rule */$

forall FDs $Y \rightarrow Z \in F$

if $Y \subseteq X^+$ then $X^+ := X^+ \cup Z /* A\text{-}rule */$

until $X^+ = \overline{X}^+$

return $X^+$

MEMBER$(F, X \rightarrow Y)$: /* Test if $X \rightarrow Y \in F^+$ */

return $Y \subseteq \text{CLOSURE}(F, X) /* P\text{-}rule */$

- Example: $A \rightarrow C \in \{A \rightarrow B, B \rightarrow C\}^+$?
Minimal Cover

... to minimize a set of FDs

forall FD $X \rightarrow Y \in F$ /* Left reduction */
forall $A \in X$ /* A superflous? */
if $Y \subseteq \text{Closure}(F, X \setminus \{A\})$
then replace $X \rightarrow Y$ with $(X - A) \rightarrow Y$ in $F$

forall remaining FD $X \rightarrow Y \in F$ /* Right reduction */
forall $B \in Y$ /* B superflous? */
if $B \subseteq \text{Closure}(F - \{X \rightarrow Y\} \cup \{X \rightarrow (Y - B)\}, X)$
then replace $X \rightarrow Y$ with $X \rightarrow (Y - B)$

Eliminate FDs of the form $X \rightarrow \emptyset$
Combine FDs of the form $X \rightarrow Y_1, X \rightarrow Y_2, \ldots$ into $X \rightarrow Y_1 Y_2 \ldots$
Normal Forms . . .

► . . . determine properties of relation schemata
► . . . forbid certain combinations of functional dependencies in relations
► . . . should prevent redundancies and anomalies
First Normal Form

- Allows only *atomic* attributes in relation schemas, i.e., only elements of standard datatypes, such as `integer` or `string`, are allowed as attribute values, but not `array` or `set`.

- Not in 1NF:

<table>
<thead>
<tr>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
<th>WName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ch. La Rose Creek</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Helena</td>
<td>Barossa Valley</td>
<td>South Australia</td>
<td>Creek Shiraz, Pinot Noir</td>
</tr>
<tr>
<td>MÃ¼ller</td>
<td>Rheingau</td>
<td>Hessen</td>
<td>Zinfandel, Pinot Noir</td>
</tr>
<tr>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>California</td>
<td>Riesling Reserve</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Chardonnay</td>
</tr>
</tbody>
</table>
In first normal form:

<table>
<thead>
<tr>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
<th>WName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ch. La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Creek Shiraz</td>
<td></td>
<td></td>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Helena</td>
<td>Napa Valley</td>
<td>California</td>
<td>Zinfandel</td>
</tr>
<tr>
<td>Helena Pinot Noir</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MÃ¼ller Riesling Reserve</td>
<td>Rheingau</td>
<td>Hessen</td>
<td></td>
</tr>
<tr>
<td>Bighorn Chardonnay</td>
<td>Napa Valley</td>
<td>California</td>
<td></td>
</tr>
</tbody>
</table>
Second Normal Form

- **Partial dependency**: An attribute functionally depends on only part of the key

<table>
<thead>
<tr>
<th>Name</th>
<th>Vineyard</th>
<th>Color</th>
<th>District</th>
<th>Region</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
<td>Ch. La Rose</td>
<td>Rot</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
<td>39.00</td>
</tr>
<tr>
<td>Creek Shiraz</td>
<td>Creek</td>
<td>Rot</td>
<td>Barossa Valley</td>
<td>South Australia</td>
<td>7.99</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>Creek</td>
<td>Rot</td>
<td>Barossa Valley</td>
<td>South Australia</td>
<td>10.99</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>Helena</td>
<td>Rot</td>
<td>Napa Valley</td>
<td>California</td>
<td>5.99</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>Helena</td>
<td>Rot</td>
<td>Napa Valley</td>
<td>California</td>
<td>19.99</td>
</tr>
<tr>
<td>Riesling Reserve</td>
<td>Müller</td>
<td>Weiß</td>
<td>Rheingau</td>
<td>Hessen</td>
<td>14.99</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>Bighorn</td>
<td>Weiß</td>
<td>Napa Valley</td>
<td>California</td>
<td>9.90</td>
</tr>
</tbody>
</table>

\[ f_1: \ Name, \ Vineyard \rightarrow \ Price \]
\[ f_2: \ Name \rightarrow \ Color \]
\[ f_3: \ Vineyard \rightarrow \ District, \ Region \]
\[ f_4: \ District \rightarrow \ Region \]

- Second normal form eliminates such partial dependencies for non-key attributes
Elimination of Partial Dependencies

Key K

Part of Key X

dependent Attribute A
Second Normal Form /2

- Example relation in 2NF

  R1(Name, Vineyard, Price)
  R2(Name, Color)
  R3(Vineyard, District, Region)
Second Normal Form /3

▲ Note: Partially dependent attribute is only problematic if it is not a prime attribute
▲ 2NF formally: Extended relation schema \( \mathcal{R} = (R, \mathcal{K}) \), FD set \( F \) over \( R \)

▲ \( Y \) partially depends on \( X \) w.r.t. \( F \) if the FD \( X \rightarrow Y \) is not left-reduced
▲ \( Y \) fully depends on \( X \) if the FD \( X \rightarrow Y \) is left-reduced
▲ \( \mathcal{R} \) is in 2NF if \( \mathcal{R} \) is in 1NF and every non-prime attribute of \( R \) fully depends on every key of \( \mathcal{R} \)
Third Normal Form

- Eliminates transitive dependencies (in addition to the other kinds of dependencies)
- For instance, Vineyard $\rightarrow$ District and District $\rightarrow$ Region in relation on Slide 5-191
- Note: 3NF only considers non-key attributes as endpoints of transitive dependencies
Elimination of Transitive Dependencies

Key K

Set of Attributes X

dependent Attribute A

Prof. Thomas Leich
Harz University of Applied Sciences
Copyright: Gunter Saake, University of Magdeburg
Third Normal Form /2

- Transitive dependency in $R_3$, i.e., $R_3$ violates 3NF
- Example relation in 3NF
  $R_{3\_1}$(Vineyard, District)
  $R_{3\_2}$(District, Region)
Third Normal Form: Formally

- Relation schema $R$, $X \subseteq R$ and $F$ is an FD set over $R$

- $A \in R$ is called transitive dependent on $X$ w.r.t. $F$ if and only if there is a $Y \subseteq R$ for which it holds that $X \rightarrow Y, Y \not\rightarrow X, Y \rightarrow A, A \notin XY$

- Extended relation schema $\mathcal{R} = (R, \mathcal{K})$ is in 3NF w.r.t. $F$ if and only if

\[
\not\exists A \in R : \ A \text{ is non-prime attribute in } R \\land A \text{ transitive dependent on a } K \in \mathcal{K} \text{ w.r.t. } F.
\]
Boyce-Codd Normal Form

▶ Stronger version of 3NF: Elimination of transitive dependencies also between prime attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Vineyard</th>
<th>Dealer</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
<td>Château La Rose</td>
<td>Weinkontor</td>
<td>39.90</td>
</tr>
<tr>
<td>Creek Shiraz</td>
<td>Creek</td>
<td>Wein.de</td>
<td>7.99</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>Creek</td>
<td>Wein.de</td>
<td>10.99</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>Helena</td>
<td>GreatWines.com</td>
<td>5.99</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>Helena</td>
<td>GreatWines.com</td>
<td>19.99</td>
</tr>
<tr>
<td>Riesling Reserve</td>
<td>MÃ¼ller</td>
<td>Weinkeller</td>
<td>19.99</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>Bighorn</td>
<td>Wein-Dealer</td>
<td>9.90</td>
</tr>
</tbody>
</table>

▶ FDs:

- Name, Vineyard → Price
- Vineyard → Dealer
- Dealer → Vineyard

▶ Candidate keys: \{ Name, Vineyard \} and \{ Name, Dealer \}

▶ Example relation meets 3NF but not BCNF
Boyce-Codd-Normalform /2

- Extended relation schema $\mathcal{R} = (R, \mathcal{K})$, FD set $F$
- BCNF formally:

$$\nexists A \in R : A \text{ transitively depends on a } K \in \mathcal{K} \text{ w.r.t. } F.$$

- Schema in BCNF:
  - WINES(Name, Vineyard, Price)
  - WINE_TRADE(Vineyard, Dealer)
- However, BCNF may violate dependency preservation, therefore often stop at 3NF
Minimality

- Avoid global redundancies
- Meet other criteria (such as normal forms) with as few schemas as possible
- Example: Set of attributes $ABC$, set of FDs $\{A \rightarrow B, B \rightarrow C\}$
- Database schema in third normal form:

$$S = \{(AB, \{A\}), (BC, \{B\})\}$$

$$S' = \{(AB, \{A\}), (BC, \{B\}), (AC, \{A\})\}$$

Redundancies in $S'$
# Schema Properties

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Schema Property</th>
<th>Key Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1NF</td>
<td>Only atomic attributes</td>
<td></td>
</tr>
<tr>
<td>2NF</td>
<td>No non-prime attribute that partially depends on a key</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>3NF</td>
<td>No non-prime attribute that transitively depends on a key</td>
</tr>
<tr>
<td></td>
<td>BCNF</td>
<td>No attribute that transitively depends on a key</td>
</tr>
<tr>
<td>S2</td>
<td>Minimality</td>
<td>Minimal number of relation schemas that satisfies the other properties</td>
</tr>
</tbody>
</table>
Transformation Properties

- When decomposing a relation in multiple relations, care must be taken that...

  1. . . . only semantically sensible and consistent application data is presented (dependency preservation), and
  2. . . . all application data can be derived from the base relations (lossless-join decomposition)
Dependency Preservation

- **Dependency preservation**: A set of dependencies can be transformed into an equivalent second set of dependencies.
- More specifically: into the set of key dependencies because these can be validated efficiently by the database system.
  - The set of dependencies shall be equivalent to the set of key constraints in the resulting database schema.
  - Equivalence ensures that, on a semantic level, the key dependencies express the exact same integrity constraints as the functional and other dependencies did before.
Dependency Preservation: Example

- Decomposition of the relation schema WINES (Slide 5-191) into 3NF:

  \begin{align*}
  &R_1(\text{Name}, \text{Vineyard}, \text{Price}) \\
  &R_2(\text{Name}, \text{Color}) \\
  &R_{3\_1}(\text{Vineyard}, \text{District}) \\
  &R_{3\_2}(\text{District}, \text{Region})
  \end{align*}

  with key dependencies

  \begin{align*}
  &\text{Name, Vineyard} \rightarrow \text{Price} \\
  &\text{Name} \rightarrow \text{Color} \\
  &\text{Vineyard} \rightarrow \text{District} \\
  &\text{District} \rightarrow \text{Region}
  \end{align*}

- Equivalent to FDs $f_1 \ldots f_4$ (Slide 5-191) $\leadsto$ dependency-preserving
Dependency Preservation: Example /2

- Zip code (a.k.a. postal code) structure of the Deutsche Post

  ADDRESS(ZIP (Z), City (C), Street (S), Street Number (N))

  and functional dependencies \( F \)

  \( CSN \rightarrow Z, Z \rightarrow C \)

- Candidate keys: \( CSN \) and \( ZSN \) \( \sim \) 3NF

- Does not meet BCNF (because \( ZSN \rightarrow Z \rightarrow C \)): therefore decomposition of \( ADDRESS \)

- But: every decomposition would destroy \( CSN \rightarrow Z \)

- Set of resulting FDs is not equivalent to \( F \), the decomposition is therefore not dependency-preserving
Dependency Preservation: Formally

- Locally extended database schema
  \[ S = \{(R_1, \mathcal{K}_1), \ldots , (R_p, \mathcal{K}_p)\}; \]
  a set \( F \) of local dependencies

\( S \) fully characterizes \( F \) (or: is dependency-preserving w.r.t. \( F \)) if and only if

\[ F \equiv \{ K \rightarrow R \mid (R, \mathcal{K}) \in S, K \in \mathcal{K} \} \]
Lossless-Join Decomposition

- In order to satisfy the criteria of the normal forms, relation schemas sometimes have to be decomposed into smaller relation schemas
- In order to restrict to “sensible” decomposition, require that the original relation can be recreated from the decomposed relations using a natural join
  \[\text{lossless-join decomposition}\]
Lossless-Join Decomposition: Examples

- Decompose the relation schema $R = ABC$ into

  $R_1 = AB$ and $R_2 = BC$

- Decomposition is not join-lossless given the dependencies

  $F = \{A \rightarrow B, C \rightarrow B\}$

- In contrast, the decomposition is join-lossless given the dependencies

  $F' = \{A \rightarrow B, B \rightarrow C\}$
Lossless-Join Decomposition

- **Original relation:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Decomposition:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Join (join-lossless):**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Non-Join-Lossless Decomposition

- Original relation:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

- Decomposition:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

- Join (not join-lossless):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Lossless-Join Decomposition: Formally

The decomposition of a set of attributes $X$ in $X_1, \ldots, X_p$ with $X = \bigcup_{i=1}^{p} X_i$ is called a lossless-join decomposition under a set of dependencies $F$ over $X$ if and only if

\[ \forall r \in \text{SAT}_X(F) : \pi_{X_1}(r) \bowtie \cdots \bowtie \pi_{X_p}(r) = r \]

holds.

- Simple criterion for a join-lossless decomposition into two relation schemas: Decomposition of $X$ into $X_1$ and $X_2$ is join-lossless under $F$, if $X_1 \cap X_2 \rightarrow X_1 \in F^+$ or $X_1 \cap X_2 \rightarrow X_2 \in F^+$
## Transformation Properties

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Transformation Property</th>
<th>Key Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Dependency Preservation</td>
<td>All given dependencies are represented by keys</td>
</tr>
<tr>
<td>T2</td>
<td>Lossless-Join Decomposition</td>
<td>Original relations can be recreated by joining base relations</td>
</tr>
</tbody>
</table>
Design Methods: Goals

- Given: Universe $\mathcal{U}$ and set of FDs $F$
- Locally extended database schema
  
  $$S = \{(R_1, K_1), \ldots, (R_p, K_p)\}$$
  
  compute with

  - **T1**: Dependency Preservation ($S$ fully characterizes $F$)
  - **S1**: $S$ is in 3NF under $F$
  - **T2**: Lossless-Join Decomposition
  - **S2**: Minimality, i.e.,
    
    $\forall S' : S'$ satisfies $T1$, $S1$, $T2$ and $|S'| < |S|$
Design Methods: Example

- Database schemas badly designed if only one of these four criteria is not fulfilled
- Example: $S = \{(AB, \{A\}), (BC, \{B\}), (AC, \{A\})\}$ fulfills $T_1$, $S_1$ and $T_2$ under $F = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$ in third relation $AC$ tuple redundant or inconsistent
- Correct: $S' = \{(AB, \{A\}), (BC, \{B\})\}$
Decomposition

Given: Initial universal relation schema \( \mathcal{R} = (\mathcal{U}, \mathcal{K}(F)) \) with all attributes and a set of implied keys implied by FDs \( F \) over \( R \)

- Set of attributes \( \mathcal{U} \) and set of FDs \( F \)
- Find all \( K \rightarrow \mathcal{U} \) with \( K \) minimal, for which \( K \rightarrow \mathcal{U} \in F^+ (\mathcal{K}(F)) \)

Wanted: Decomposition into \( D = \{ \mathcal{R}_1, \mathcal{R}_2, \ldots \} \) of 3NF-relation schemas
Decomposition: Algorithm

\textbf{DECOMPOSE}(\mathcal{R})

Set \( D := \{ \mathcal{R} \} \)

while \( \mathcal{R}' \in D \), does not meet 3NF

\begin{tabular}{l}
/* Find attribute \( A \) that is transitively dependent on \( K \) */
\end{tabular}

\begin{tabular}{l}
if Key \( K \) with \( K \rightarrow Y, Y \not\rightarrow K, Y \rightarrow A, A \not\in KY \) then
\end{tabular}

\begin{tabular}{l}
/* Decompose relation schema \( R \) w.r.t. \( A \) */
\end{tabular}

\begin{align*}
R_1 &:= R - A, \\
R_2 &:= YA \\
\mathcal{R}_1 &:= (R_1, K), \\
\mathcal{R}_2 &:= (R_2, K_2 = \{Y\}) \\
D &:= (D - \mathcal{R}') \cup \{\mathcal{R}_1\} \cup \{\mathcal{R}_2\}
\end{align*}

\begin{tabular}{l}
end if
\end{tabular}

end while

return \( D \)
Decomposition: Example

▶ Initial relation schema $R = ABC$
▶ Functional dependencies $F = \{A \rightarrow B, B \rightarrow C\}$
▶ Keys $K = A$
Decomposition: Example /2

- Initial relation schema \( R \) with Name, Vineyard, Price, Color, District, Region

- Functional dependencies

\[
\begin{align*}
  f_1 & : \text{Name, Vineyard} \rightarrow \text{Price} \\
  f_2 & : \text{Name, Vineyard} \rightarrow \text{Vineyard} \\
  f_3 & : \text{Name, Vineyard} \rightarrow \text{Name} \\
  f_4 & : \text{Name} \rightarrow \text{Color} \\
  f_5 & : \text{Vineyard} \rightarrow \text{District, Region} \\
  f_6 & : \text{District} \rightarrow \text{Region}
\end{align*}
\]
Decomposition: Assessment

- Advantages: 3NF, lossless-join decomposition
- Disadvantages: other criteria not fulfilled, depends on order, NP-hard (search for keys)
Synthesis Method

- Principle: Synthesis transforms original set of FDs $F$ into a resulting set of key dependencies $G$ such that $F \equiv G$
- “Dependency Preservation” built into the method
- 3NF and minimality also achieved, independent of order
- Computational complexity: quadratic
Comparison Decomposition — Synthesis

\[ \mathcal{U}, \text{FDs } F \]

\[ R, \mathcal{K} \]

\[ R_1, K_1 \ldots R_n, K_n \]

\[ R'_1, K'_1 \ldots R'_m, K'_m \]

\[ \text{FDs } F' \]

\[ \text{FDs } F'' \]

Decomposition Synthesis
Synthesis Method: Algorithm

Given: Relation schema $R$ mit FDs $F$
Wanted: Join-lossless and dependency-preserving decomposition into $R_1, \ldots, R_n$ where all $R_i$ are in 3NF
Algorithm:

SYNTHESIZE($F$):

$\hat{F} := \text{MINIMALCOVER}(F)$ /* Determine minimal cover */

Compute equivalence classes $C_i$ of FDs from $\hat{F}$ with equal or equivalent left sides, i.e., $C_i = \{X_i \rightarrow A_{i1}, X_i \rightarrow A_{i2}, \ldots\}$

For each equivalence class $C_i$ create a schema of the form $R_{C_i} = \{X_i \cup \{A_{i1}\} \cup \{A_{i2}\} \cup \ldots\}$

if none of the schemas $R_{C_i}$ contains a key from $R$

then create additional relation schema $R_K$ with attributes from $R$, which form the key

return $\{R_K, R_{C_1}, R_{C_2}, \ldots\}$
Equivalence Classes

- Class of FDs whose left sides are equal or equivalent
- Left sides are equivalent if they determine each other functionally.
- Relation schema $R$ with $X_i, Y \subset R$, set of FDs $X_i \rightarrow X_j$ and $X_i \rightarrow Y$ with $1 \leq i, j \leq n$ can be expressed as

\[(X_1, X_2, \ldots, X_n) \rightarrow Y\]
Equivalence Classes: Example

- **Set of FDs**

\[ F = \{ A \rightarrow B, AB \rightarrow C, A \rightarrow C, B \rightarrow A, C \rightarrow E \} \]

- **Minimal cover**

\[ \hat{F} = \{ A \rightarrow B, B \rightarrow C, B \rightarrow A, C \rightarrow E \} \]

- **Aggregation into equivalence classes**

\[ C_1 = \{ A \rightarrow B, B \rightarrow C, B \rightarrow A \} \]
\[ C_2 = \{ C \rightarrow E \} \]

- **Result of synthesis**

\((ABC, \{\{A\}, \{B\}\}), (CE, \{C\})\)
Achieving a Lossless-Join Decomposition

- Achieve a lossless-join decomposition by a simple “trick”:
  - Extend the original set of FDs $F$ with $\mathcal{U} \rightarrow \delta$, where $\delta$ is a dummy attribute
  - $\delta$ is removed after synthesis
- Example: $\{A \rightarrow B, C \rightarrow E\}$
  - Result of synthesis $(AB, \{A\}), (CE, \{C\})$ is not lossless, because the universal key is not part of any schema
  - Dummy-FD $ABCE \rightarrow \delta$; reduced to $AC \rightarrow \delta$
  - Yields third relation schema $(AC, \{AC\})$
Synthesis: Example

- Relation schema and set of FDs from Slide 5-219
- Steps
  1. Minimal cover: removal of $f_2, f_3$ as well as Region in $f_5$
  2. Equivalence classes:

$$C_1 = \{ \text{Name, Vineyard} \rightarrow \text{Price} \}$$
$$C_2 = \{ \text{Name} \rightarrow \text{Color} \}$$
$$C_3 = \{ \text{Vineyard} \rightarrow \text{District} \}$$
$$C_4 = \{ \text{District} \rightarrow \text{Region} \}$$

3. Derivation of relation schemas
Summary

▶ Functional dependencies
▶ Normal forms (1NF – 3NF, BCNF)
▶ Dependency preservation and lossless-join decomposition
▶ Design methods
Control Questions

▸ What is the goal of normalizing relational schemas?

▸ Which properties of relational schemas do the normal forms take into account?

▸ What is the difference between 3NF and BCNF?

▸ What does it mean for a decomposition to be dependency-preserving?
What is a lossless-join decomposition?
Part VI

The Relational Query Language

SQL
The Relational Query Language SQL

1. The SFW Block in Detail
2. Extensions of the SFW Block
3. Recursion
Educational Objective for Today . . .

- Advanced knowledge of the relational SQL
- Knowledge of extensions of the SFW block
- Understanding the formulation and evaluation of recursive queries
Structure of an SQL Query

```
-- query
select projection-list
from relations-list
[ where condition ]
```

**select**
- Projection list
- Arithmetic operations and aggregation functions

**from**
- Relations to use, optionally aliases (renamings)

**where**
- Selection and join conditions
- Nested queries (another SFW block)
Selection of Tables: The `from` Clause

- Most simple form:
  - Each relation name may be followed by an optional tuple variable

```sql
select *
from relations_list
```

- Example query:

```sql
select *
from WINES
```
Cartesian Product

➤ With more than one relation, the Cartesian product (a.k.a. cross product) is computed:

```sql
select *
from WINES, PRODUCER
```

➤ All combinations are returned!
Tuple Variables for Repeated Access

- Using tuple variables, a relation can be accessed several times:

```sql
select *
from WINES w1, WINES w2
```

- Columns are then called:

```
w2.WineID, w2.Name, w2.Color, w2.Vintage, w2.Vineyard
```
Natural Join in SQL92

- Early versions of SQL
  - Standard that is usually implemented in current systems
  - Only know cross product, no explicit join operator
  - Join achieved with predicate after `where`

- Example for natural join:

```sql
select *
from WINES, PRODUCER
where WINES.Vineyard = PRODUCER.Vineyard
```
Joins as Explicit Operators: natural join

- Newer SQL versions
  - Know several explicit join operators
  - Can be seen as an abbreviation of the detailed query with cross product

```
select *
from WINES natural join PRODUCER
```
Joins as Explicit Operators: join

- Join with arbitrary predicate:

```sql
select *
from WINES join PRODUCER
    on WINES.Vineyard = PRODUCER.Vineyard
```

- Equi-joins on columns using the same name with `using`:

```sql
select *
from WINES join PRODUCER
    using (Vineyard)
```
Joins as Explicit Operators: cross join

- Cross product (a.k.a. Cartesian product)

```sql
select *
from WINES, PRODUCER
```

- As cross join

```sql
select *
from WINES cross join PRODUCER
```
Tuple Variable for Intermediate Results

- “Intermediate relations” from SQL operations or an SFW block can be named using tuple variables

```
select Result.Vineyard
from (WINES natural join PRODUCER) as Result
```

- For `from`, tuple variables are mandatory
- `as` is optional
The select Clause

- Determines projection attributes

\[
\text{select} \ [\text{distinct}] \ projection\text{-}list
\]

\[
\text{from} \ \ldots
\]

- \textit{mit}

\[
projection\text{-}list \ := \ \{\text{attribute} \mid
\text{arithmetic}\text{-}expression \mid
\text{aggregation}\text{-}function\} \ [, \ \ldots]\]

- Attributes of the relation after the \texttt{from}, optionally with a prefix that specifies names of relations or names of tuple variables
- Arithmetic expressions over attributes of these relations, as well as constants
- Aggregation functions over attributes of these relations
The select Clause /2

- Special case of the projection list: *
  - Yields all attributes of the relation(s) from the `from` part

```sql
select *
from WINES
```
distinct Eliminates Duplicates

\[ \text{select Name from WINES} \]

- Yields the result relation as a multi-set:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Zinfandel</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Riesling Reserve</td>
</tr>
<tr>
<td>Chardonnay</td>
</tr>
</tbody>
</table>
distinct Eliminates Duplicates /2

```sql
select distinct Name from WINES
```

► Yields projection from the relational algebra:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Zinfandel</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Riesling Reserve</td>
</tr>
<tr>
<td>Chardonnay</td>
</tr>
</tbody>
</table>
Tuple Variables and Relation Names

- Query

```
select Name from WINES
```

- is equivalent to

```
select WINES.Name from WINES
```

- and

```
select W.Name from WINES W
```
 Prefixes for Unambiguousness

```
select Name, Vintage, Vineyard  -- (wrong!)
from WINES natural join PRODUCER
```

▶ Attribute Vineyard exists in both tables, WINES and PRODUCER!

▶ Correct with prefix:

```
select Name, Vintage, PRODUCER.Vineyard
from WINES natural join PRODUCER
```
Prefixes for Unambiguousness /2

When using tuple variables, the name of a tuple variable can be used to qualify an attribute:

```sql
select w1.Name, w2.Vineyard
from WINES w1, WINES w2
```
The **where Clause**

```sql
select ... from ...  
where condition
```

- **Forms of the condition:**
  - Comparing an attribute with a constant:
    ```sql
    attribute \( \theta \) constant
    ```
    possible comparison symbols \( \theta \) depend on the domain;
    e.g., \( =, <>, >, <, >= \) or \( <= \).
  - Comparison between two attributes with compatible domains:
    ```sql
    attribute1 \( \theta \) attribute2
    ```
  - Logical **connectors** **or**, **and** and **not**
Join Condition

- Join condition has the form:

\[ \text{relation1.attribute} = \text{relation2.attribute} \]

- Example:

```sql
select Name, Vintage, PRODUCER.Vineyard
from WINES, PRODUCER
where WINES.Vineyard = PRODUCER.Vineyard
```
Range Selection

- Range selection

attribute between constant$_1$ and constant$_2$

is an abbreviation for

attribute $\geq$ constant$_1$ and attribute $\leq$ constant$_2$

- Restricts attribute values to the closed interval $[\text{constant}_1, \text{constant}_2]$

- Example:

```sql
select * from WINES
where Vintage between 2000 and 2005
```
Imprecise Selection

- Notation

```
attribute like special-constant
```

- Pattern matching in strings (search for multiple substrings)
- Special constant can contain the wildcard characters ‘%’ and ‘_’
  - ‘%’ stands for no character or an arbitrary string of characters
  - ‘_’ stands for exactly one character
Imprecise Selection /2

Example

```sql
select * from WINES
where Name like 'La Rose%'
```

is shorthand for

```sql
select * from WINES
where Name = 'La Rose'
  or Name = 'La RoseA'
  or Name = 'La RoseAA'
  ...
  or Name = 'La RoseB'
  or Name = 'La RoseBB'
  ...
  or Name = 'La Rose Grand Cru'
  ...
  or Name = 'La Rose Grand Cru Classe'
  ...
  or Name = 'La RoseZZZZZZZZZZZZ
```
Set Operations

- Set operation require compatible domains for pairs of corresponding attributes:
  - Both domains are equal, or
  - both domains are based on character (irrespective of the length of the strings), or
  - both domains are numeric (irrespective of the exact types), such as integer or float.

- Result schema := schema of the “left” relation

```sql
select A, B, C from R1
union
select A, C, D from R2
```
Set Operations in SQL

- \textit{Union, intersection and difference as union, intersect and except}

- Can be used orthogonally:

```sql
select *
from (select Vineyard from PRODUCER
       except select Vineyard from WINES)
```

equivalent to

```sql
select *
from PRODUCER except corresponding WINES
```
Set Operations in SQL /2

- Via corresponding by clause: specification of the list of attributes over which to perform the set operation

```
select *
from PRODUCER except corresponding by (Vineyard) WINES
```

- When using union: Default case is duplicate removal (union distinct); without duplicate removal when using union all
### Set Operations in SQL

#### R union S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

#### R union all S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

#### R union corresponding S

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

#### R union corresponding by (A) S

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

---

© Hochschule Harz
Hochschule für angewandte Wissenschaften

Prof. Thomas Leich
Harz University of Applied Sciences
Copyright: Gunter Saake, University of Magdeburg

Page 257/523
Nesting Queries

► Necessary for comparing sets of values:
  ► Standard comparisons in combination with the quantifiers
    all (∀) or any (∃)
  ► Special predicates for working with sets, in and exists
in Predicate and Nested Queries

► Notation:

\[ \text{attribute in ( SFW-block )} \]

► Example:

```sql
select Name
from WINES
where Vineyard in ( select Vineyard from PRODUCER
where Region='Bordeaux')
```
Evaluation of Nested Queries

1. Evaluation of the inner query regarding the vineyards from Bordeaux

2. Insertion of the results as a set of constants in the outer query after `in`

3. Evaluation of the modified query

```sql
select Name
from WINES
where Vineyard in (
    'Château La Rose', 'Château La Pointe')
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
</tbody>
</table>
Evaluation of Nested Queries /2

Internal evaluation: transformation into a join

```
select Name
from WINES natural join PRODUCER
where Region = 'Bordeaux'
```
Negation of the \textit{in} Predicate

▶ Simulation of the difference operator

\[ \pi_{\text{Vineyard}}(\text{PRODUCER}) - \pi_{\text{Vineyard}}(\text{WINES}) \]

using the SQL query

\begin{verbatim}
select Vineyard from PRODUCER
where Vineyard not in (select Vineyard from WINES )
\end{verbatim}
## Expressiveness of the SQL Kernel

<table>
<thead>
<tr>
<th>Relational Algebra</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projection</td>
<td><code>select</code> distinct</td>
</tr>
<tr>
<td>Selection</td>
<td><code>where</code> without nesting</td>
</tr>
<tr>
<td>Join</td>
<td><code>from, where</code> <code>from with join</code> or <code>natural join</code></td>
</tr>
<tr>
<td>Renaming</td>
<td><code>from with tuple variable; as</code></td>
</tr>
<tr>
<td>Difference</td>
<td><code>where</code> with nesting <code>except corresponding</code></td>
</tr>
<tr>
<td>Intersection</td>
<td><code>where</code> with nesting <code>intersect corresponding</code></td>
</tr>
<tr>
<td>Union</td>
<td><code>union corresponding</code></td>
</tr>
</tbody>
</table>
Additional Notes on SQL

- Extensions of the SFW block
  - Further join operations inside the `from` clause (outer join),
  - Other kinds of conditions and conditions using quantifiers inside the `where` clause,
  - Application of scalar operations and aggregation functions inside the `select` clause,
  - Additional clauses `group by` and `having`

- Recursive queries
Scalar Expressions

- Renaming of columns: *expression as new-name*
- Scalar operations on
  - Numeric domains: for instance +, −, ∗ and /,
  - Strings: Operations such as `char_length` (current length of a string), concatenation `||` and the `substring` operation (extract a substring starting at a certain position in the string),
  - Dates and time intervals: operations such as `current_date` (current date), `current_time` (current time), +, − and ∗
- Conditional expressions
- Type conversion
- Notes:
  - Scalar expressions can comprise multiple attributes
  - Application is performed tuple-wise: one output tuple is created for each input tuple
Scalar Expressions /2

Return the names of all Grand-Cru wines

```sql
select substring(Name from 1 for
    (char_length(Name) - position('Grand Cru' in Name)))
from WINES where Name like '%Grand Cru'
```

Assumption: additional attribute ProdDate in WINES

```sql
alter table WINES add column ProdDate date
update WINES set ProdDate = date '2004-08-13'
where Name = 'Zinfandel'
```

Query:

```sql
select Name, year(current_date - ProdDate) as Age
from WINES
```
Conditional Expressions

- **case** expression: return a value depending on the Evaluation of predicate

```plaintext
case
  when \( \text{predicate}_1 \) then \( \text{expression}_1 \)
  ...
  when \( \text{predicate}_{n-1} \) then \( \text{expression}_{n-1} \)
  [ else \( \text{expression}_n \) ]
end
```

- Use in **select**- and **where** clause

```plaintext
select case
  when Color = 'Red' then 'Red wine'
  when Color = 'White' then 'White wine'
  else 'Other'
end as WineType, Name from WINES
```
**Type Conversion**

- Explicit conversion of the types of expressions

  \[ \text{cast(expression as typname)} \]

- Example: int values as strings for the concatenation operator

  \[
  \text{select cast(Vintage as varchar)} \ || \ ' ' \ || \ \text{Name as Description from WINES}
  \]
Quantifiers and Set Comparisons

► Quantifiers: all, any, some and exists

► Notation

\[
\text{attribute } \theta \{ \text{all} \mid \text{any} \mid \text{some} \} (\text{select attribute from ... where ...})
\]

► all: where condition is fulfilled if for all tuples of the inner SFW block, the \(\theta\)-comparison with attribute evaluates to true

► any and some: where condition is fulfilled if the \(\theta\)-comparison evaluates to true for at least one tuple of the inner SFW block
Conditions with Quantifiers: Examples

► Determine the oldest wine

```sql
select *
from WINES
where Vintage <= all (  
sel ect Vintage from WINES)
```

► All vineyards that produce red wines

```sql
select *
from PRODUCER
where Vineyard = any (  
sel ect Vineyard from WINES  
where Color = 'Red')
```
Comparison of Sets of Values

- Test for equality of two sets impossible with quantifiers alone
- Example: “Return all producers that produce both, red and white wines.”
- Wrong query

```sql
select Vineyard
from WINES
where Color = 'Red' and Color = 'White'
```

- Correct query

```sql
select w1.Vineyard
from WINES w1, WINES w2
where w1.Vineyard = w2.Vineyard
    and w1.Color = 'Red' and w2.Color = 'White'
```
The \textbf{exists/not exists} Predicate

- Simple form of nesting

\texttt{exists ( SFW-block )}

- Yields \texttt{true} if the result of the inner query is \texttt{not} empty
- Especially useful for \texttt{correlated subqueries} (a.k.a. synchronized subqueries)
  - In the inner query, the relation names and tuple variable names from the \texttt{from} part of the outer query are used
Synchronized Subqueries

- Vineyards with 1999 red wine

```sql
select * from PRODUCER
where 1999 in (select Vintage from WINES
where Color=’Red’ and WINES.Vineyard = PRODUCER.Vineyard)
```

- Conceptual evaluation
  1. Examination of the first PRODUCER tuple the outer query (Creek) and insertion into the inner query
  2. Evaluation of the inner query

```sql
select Vintage from WINES
where Color=’Red’ and WINES.Vineyard = ’Creek’
```

3. Continue at step 1. with second tuple ...

- Alternative: reformulation as join
Example for exists

- Vineyards from Bordeaux without known wines

```
select * from PRODUCER e
where Region = 'Bordeaux' and not exists (  
  select * from WINES  
  where Vineyard = e.Vineyard)
```
Aggregation Functions and Grouping

- Aggregation functions calculate new values for the whole column, such as the sum or the average of the values of a column.
- Example: Determination of the average price of articles or the total sales of all sold products.
- With additional grouping: calculation of functions per group, e.g., the average price per Product group or the total sales per customer.
Aggregation Functions

- **Aggregation functions in Standard-SQL:**
  - **count**: calculates the number of values in a column or alternatively (in a special case count(∗)) the number of tuples of a relation
  - **sum**: calculates the sum of all values in a column (only for numeric values)
  - **avg**: calculates the arithmetic mean of the values of a column (only for numeric domains)
  - **max** resp. **min**: calculate the biggest or smallest value of a column
Arguments of a aggregation function:

- an attribute of the `from` clause specified relation,
- a valid scalar expression or,
- in the clause of the `count` function also the symbol `*`
Aggregation Functions /3

► Before the argument (except of the case \texttt{count(*)}) optional also the keywords \texttt{distinct} or \texttt{all}

  ▶ \texttt{distinct}: before application of aggregation functions, duplicate values are removed from the set of values on which the function is applied

  ▶ \texttt{all}: duplicates are used in calculations (default setting)

  ▶ null values are always eliminated before the function is applied (except of the case of \texttt{count(*)})
Aggregation Functions – Examples

► Number of wines

```sql
select count(*) as Number
from WINES
```

results in

<table>
<thead>
<tr>
<th>Number</th>
<th>7</th>
</tr>
</thead>
</table>

```
Aggregation Functions – Examples /2

▶ Number of distinct wine regions:

```sql
SELECT count(distinct Region)
FROM PRODUCER
```

▶ Wines that are older than the average:

```sql
SELECT Name, Vintage
FROM WINES
WHERE Vintage < (SELECT avg(Vintage) FROM WINES)
```

▶ All producers that deliver exactly one wine:

```sql
SELECT * FROM PRODUCER e
WHERE 1 = (SELECT count(*) FROM WINES w
WHERE w.Vineyard = e.Vineyard)
```
Aggregation Functions /2

► Nesting of aggregation functions is not allowed

```sql
select f_1(f_2(A)) as Result
from R ...
```

-- (Wrong!)

► Possible formalization:

```sql
select f_1(Temp) as Result
from ( select f_2(A) as Temp from R ... )
```
Aggregation Functions in `where` Clause

- Aggregation functions give only one value \(\leadsto\) Application in Constants-

"Selections of the `where`-"Clause possible

- All producers that deliver exactly one wine:

\[
\begin{align*}
\text{select} & \quad \text{* from PRODUCER e} \\
\text{where} & \quad 1 = ( \\
& \quad \text{select count(*) from WINES w} \\
& \quad \text{where w.Vineyard = e.Vineyard})
\end{align*}
\]
group by and having

▶ Notation

```
select ...  
from ...    
[where ...] 
[group by attribute-list ] 
[having condition ]
```
Grouping: Scheme

Relation REL:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Query:

```
select A, sum(D) from REL where ...
group by A, B
having A<4 and sum(D)<10 and max(C)=4
```
Grouping: Step 1

▶ from and where

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

...
Grouping: Step 2

▶ group by A, B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
Grouping: Step 3

- **select A, sum(D)**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>sum(D)</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>
Grouping: Step 4

- having $A < 4$ and $\text{sum}(D) < 10$ and $\text{max}(C) = 4$

<table>
<thead>
<tr>
<th>A</th>
<th>sum(D)</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

$\text{max}(C) = 4$

$A$: 

- $1$ with sum(D): $9$

$\text{sum}(D)$

<table>
<thead>
<tr>
<th>A</th>
<th>sum(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Copyright: Gunter Saake, University of Magdeburg
Grouping - Example

- Number of red and white wines:

```sql
select Color, count(*) as Number
from WINES
group by Color
```

- Result relation:

<table>
<thead>
<tr>
<th>Color</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>5</td>
</tr>
<tr>
<td>white</td>
<td>2</td>
</tr>
</tbody>
</table>
having - Example

- Region with more than one wine

```sql
select Region, count(*) as Number
from PRODUCER natural join WINES
group by Region
having count(*) > 1
```
Attributes for Aggregation resp. having

- Valid attributes after `select` at grouping on relation with scheme $R$
  - Grouping attributes $G$
  - Aggregations on non-grouping attributes $R - G$
- Valid attributes for `having`
  - dito
Outer Joins

- Additionally to classic join (inner join): in SQL-92 also outer join \rightarrow Adoption of “dangling tuples” into the result and completion with null values
- **outer join** takes all tuples of both operands (long version: full outer join)
- **left outer join** resp. **right outer join** takes all tuples of the left resp. right operand
- Outer natural join each with keyword **natural**, e.g. **natural left outer join**
## Outer Joins /2

<table>
<thead>
<tr>
<th>LEFT</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RIGHT</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NATURAL JOIN</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTER</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEFT</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RIGHT</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Outer Join: Example

```
select Region, count(WineID) as Number
from PRODUCER natural left outer join WINES
group by Region
```

<table>
<thead>
<tr>
<th>Region</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barossa Valley</td>
<td>2</td>
</tr>
<tr>
<td>Napa Valley</td>
<td>3</td>
</tr>
<tr>
<td>Saint-Emilion</td>
<td>1</td>
</tr>
<tr>
<td>Pomerol</td>
<td>0</td>
</tr>
<tr>
<td>Rheingau</td>
<td>1</td>
</tr>
</tbody>
</table>
Simulation of the Outer Join

▶ Left outer join

```sql
select *
from PRODUCER natural join WINES
union all
select PRODUCER.*, cast(null as int),
    cast(null as varchar(20)),
    cast(null as varchar(10)), cast(null as int),
    cast(null as varchar(20))
from PRODUCER e
where not exists (select *
    from WINES
where WINES.Vineyard = e.Vineyard)
```
Sorting with order by

- Notation
  
  \texttt{order by attribute-list}

- Example:
  
  \texttt{select * from WINES order by Vintage}

- Sorting ascending (\texttt{asc}) or descending (\texttt{desc})

- Sorting as last operation of a query \implies \texttt{Sort attribute must be contained in the select clause}
Sorting /2

- Sorting also with calculated attributes (aggregates) as sort criterion

```sql
select Vineyard, count(*) as Number
from PRODUCER natural join WINES
group by Vineyard
order by Number desc
```
Sorting: Top-k-Queries

Query, that gives the best $k$ elements for a ranking function

```sql
select w1.Name, count(*) as Rank
from WINES w1, WINES w2
where w1.Vintage <= w2.Vintage  -- Step 1
group by w1.Name, w1.WineID  -- Step 2
having count(*) <= 4  -- Step 3
order by Rank  -- Step 4
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zinfandel</td>
<td>1</td>
</tr>
<tr>
<td>Creek Shiraz</td>
<td>2</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>3</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>4</td>
</tr>
</tbody>
</table>
Sorting: Top-k-Queries

- Determination of the $k = 4$ youngest wines
- Explanation
  - Step 1: assignment of all wines that are older
  - Step 2: grouping by names, determination of the rank
  - Step 3: restriction to ranks $\leq 4$
  - Step 4: sorting by rank
Handling of Null Values

- Scalar Expressions: Result null, when null value is used in calculation
- In all aggregation functions (except of \texttt{count(*))} null values are removed before the function is applied
- Almost all comparisons with null values result in \texttt{unknown} (instead of \texttt{true} or \texttt{false})
- Exception: \texttt{is null} gives \texttt{true} and \texttt{is not null} gives \texttt{false}
- Boolean expressions are then based on three-valued logic
# Handling of Null Values /2

<table>
<thead>
<tr>
<th>and</th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>or</th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection of Null Values

- **Null-Selection** selects tuples that contain null values for a certain attribute

- **Notation**
  
  \[
  \text{attribute is null}
  \]

- **Example**

  ```
  select * from PRODUCER
  where Region is null
  ```
Named Queries

- Query expression that can be referenced multiple times in a query
- Notation

```sql
with query-name [(column-list)] as
( query-expression )
```

- Query without with

```sql
select *
from WINES
where Vintage - 2 >= ( 
    select avg (Vintage) from WINES
) 
and Vintage + 2 <= ( 
    select avg (Vintage) from WINES
)
```
Named Queries /2

Query with with

```sql
with AGE(Average) as (  
    select avg(Vintage) from WINES)  
select *  
from WINES, AGE  
where Vintage - 2 >= Average  
and Vintage + 2 <= Average
```
Recursive Queries

▶ Application: *Bill of Material*-Queries, Calculation of the transitive closure (flight connection etc.)

▶ Example:

<table>
<thead>
<tr>
<th>BUSLINE</th>
<th>Departure</th>
<th>Arrival</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Penrice</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Tanunda</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Tanunda</td>
<td>Seppeltsfield</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Tanunda</td>
<td>Bethany</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Bethany</td>
<td>Lyndoch</td>
<td>14</td>
</tr>
</tbody>
</table>
Recursive Queries /2

- Bus trips with max. two transfers

```sql
select Departure, Arrival
from BUSLINE
where Departure = 'Nuriootpa'
union
from BUSLINE B1, BUSLINE B2
union
select B1.Departure, B3.Arrival
from BUSLINIE B1, BUSLINIE B2, BUSLINIE B3
and B2.Arrival = B3.Departure
```
Recursion in SQL:2003

- Formulation via extended with recursive-query
- Notation

```sql
with recursive recursive-table as (  
    query-expression  -- recursive part
)
[traversal-clause] [cycle-clause]
query-expression  -- non-recursive part
```

- Non-recursive part: query of recursion table
Recursion in SQL:2003 /2

▶ Recursive part:

```
  -- Initialization
  select ...
  from table where ...

  -- Recursion step
  union all
  select ...
  from table, recursion table
  where recursion condition
```
Recursion in SQL:2003: Example

```sql
with recursive TOUR(Departure, Arrival) as ( 
  select Departure, Arrival 
  from BUSLINE 
  where Departure = 'Nuriootpa' 
  union all 
  select T.Departure, B.Arrival 
  from TOUR T, BUSLINE B 
  where T.Arrival = B.Departure)
select distinct * from TOUR
```
## Step-Wise Composition of the Recursion

### Table TOUR

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Departure</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Penrice</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Tanunda</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Penrice</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Tanunda</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Seppeltsfield</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Bethany</td>
</tr>
</tbody>
</table>

### Step 2

<table>
<thead>
<tr>
<th>Departure</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuriootpa</td>
<td>Penrice</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Tanunda</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Seppeltsfield</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Bethany</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Lyndoch</td>
</tr>
</tbody>
</table>
Recursion: Example /2

- Arithmetic operations in the recursion step

```sql
WITH recursive TOUR(Departure, Arrival, Route) AS (
    SELECT Departure, Arrival, Distance AS Route
    FROM BUSLINE
    WHERE Departure = 'Nuriootpa'
    UNION ALL
    SELECT T_Departure, B_Arrival, Route + Distance AS Route
    FROM TOUR T, BUSLINE B
    WHERE T_Arrival = B_Departure
)
SELECT DISTINCT * FROM TOUR
```
Safety of Recursive Queries

- Safety (= finiteness of the calculation) is the most important requirement on a query language
- Problem: cycles in the recursion

```
insert into BUSLINE (Departure, Arrival, Distance)
values ('Lyndoch', 'Tanunda', 12)
```

- Handling in SQL
  - Limitation of the recursion depth
  - Cycle detection
Safety of Recursive Queries /2

▶ Restriction on the recursion depth

```sql
with recursive TOUR(Departure, Arrival, Transitions) as (
  select Departure, Arrival, 0
  from BUSLINE
  where Departure = 'Nuriootpa'
  union all
  select T.Departure, B.Arrival, Transitions + 1
  from TOUR T, BUSLINE B
  where T.Arrival = B.Departure and Transitions < 2)
select distinct * from TOUR
```

Safety through Cycle Detection

- Cycle Clause
  - at detection of duplicates in the calculation path \textit{attrib}:
    \begin{verbatim}
    Cycle = '*'
    \end{verbatim}
    (Pseudo column of type \texttt{char(1)})
  - Guarantee the finiteness of the result “by hand”

\begin{verbatim}
cycle attrib set marke to '*' default '-'
\end{verbatim}
Safety through Cycle Detection

```sql
with recursive TOUR(Departure, Arrival, Way) as (  
    select Departure, Arrival, Departure || ' - ' || Arrival as Way  
    from BUSLINIE where Departure = 'Nuriootpa'  
    union all  
    select T.Departure, B.Arrival, Way || ' - ' || B. Arrival as Way  
    from TOUR T, BUSLINIE B where T.Arrival = B.Departure)  
cycle Arrival set Cycle to '*' default '-'  
select Way, Cycle from TOUR
```

<table>
<thead>
<tr>
<th>Way</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuriootpa-Penrice</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Seppeltsfield</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Bethany</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Bethany-Lyndoch</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Bethany-Lyndoch-Tanunda</td>
<td>*</td>
</tr>
</tbody>
</table>
SQL-Versions

► History
  ► SEQUEL (1974, IBM Research Labs San Jose)
  ► SEQUEL2 (1976, IBM Research Labs San Jose)
  ► SQL (1982, IBM)
  ► ANSI-SQL (SQL-86; 1986)
  ► ISO-SQL (SQL-89; 1989; three Languages Level 1, Level 2, + IEF)
  ► (ANSI / ISO) SQL2 (as SQL-92 adopted)
  ► (ANSI / ISO) SQL3 (as SQL:1999 adopted)

► Despite of standardization: partly incompatible among systems of certain producers
Summary

- SQL as standard language
- SQL-Core with reference to relational algebra
- Extensions: Grouping, Recursion etc.
Control Questions

► What are the options to formalize joins?
► What do aggregations and grouping calculate?
► Which operations can be used for the handling of null values?
► What is the purpose of recursive queries in SQL?
Part VII

Query Basics: Algebra & Calculus
Query Basics: Algebra & Calculus

1. Criteria for Queries
2. Query Algebras
3. Relational Algebra: Extensions
4. Query Calculus
5. Examples for the Domain Calculus
Learning goals for today . . .

- Understanding of formal basics of relational query languages
- Knowledge to formalize queries with relational algebra
- Knowledge to formalize calculus queries
Introduction

▶ so far:
  ▶ Relation schemata with basic relations that are saved in databases

▶ now:
  ▶ "derived" relation schemata with virtual relations that are calculated from basis relations (basis relations keep unchanged)
Terms

▶ **Query:** Sequence of operations that calculate a result relation from basis relations
  - Present result relations interactively on a monitor or
  - further processing via program ("Embedding")

▶ **View:** Sequence of operations that is long-term saved under a view name and that can be called again with this name; results in a view relation

▶ **Snapshot:** Result relation of a query that is stored under a Snapshot-Name, but that is never calculated twice (with changed basis relations) (e.g., annual balance sheets)
Criteria for Query Languages

- **Ad-Hoc-Formalization**: User should be able to formalize a query without a need to write a complete program.

- **Descriptiveness**: User should formalize "What do I want?" instead of "How do I get what I want?"

- **Collection-based**: Each operation should operate on a collection of data at the same time, not navigating on single elements ("one-tuple-at-a-time")

- **Closure**: Result is again a relation that can be used as input for the next query.
Criteria for Query Languages /2

- **Adequacy**: All constructs of the underlying data model are supported
- **Orthogonality**: Language constructs are in similar situations also similar applicable
- **Optimizability**: Language consists of few operations for which optimization rules exist
- **Efficiency**: Any operation is efficient executable (in the relation model each operation has a complexity \( \leq O(n^2) \), \( n \) number of tuples of the relation).
Criteria for Query Languages

- **Safety**: No query, that is syntactically correct, may result in an endless loop or give an infinite result.
- **Limitation**: (comes from Safety, Optimizability, Efficiency) Query language should not be a complete programming language.
- **Completeness**: The language must at least being able to express the queries of a standard language (such as the to be introduced relation algebra of this chapter or the safe relation calculus).
Query Algebra

- Math: Algebra defined with range of values and on this defined operators
- For database queries: Contents of the database are values, and operators define functions for the calculation of query results
  - Relational Algebra
  - Algebra Extensions
Relational Algebra

▶ **Hide Columns**: Projection $\pi$
▶ **Search Rows**: Selection $\sigma$
▶ **Joining Tables**: Join $\Join$
▶ **Union of Tables**: Union $\cup$
▶ **Subtract Tables from each other**: Difference $-$
▶ **Rename Columns**: Renaming $\beta$

(important for $\Join$ and $\cup$, $-$)
Relational Algebra: Overview

Selection

Projection

Join

Selection

Projection

Join

Selection

Projection

Join
Projection

► Syntax

\[ \pi_{\text{AttributeSet}}(\text{Relation}) \]

► Semantics

\[ \pi_X(r) := \{ t(X) \mid t \in r \} \]

For \( r(R) \) and \( X \subseteq R \) attribute set in \( R \)

► Property for \( Y \subseteq X \subseteq R \)

\[ \pi_Y(\pi_X(r)) = \pi_Y(r) \]

► Attention: \( \pi \) removes duplicates (Set Semantics)
Projection: Example

\[\pi_{\text{Region}}(\text{PRODUCER})\]

<table>
<thead>
<tr>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>South Australia</td>
</tr>
<tr>
<td>Kalifornien</td>
</tr>
<tr>
<td>Bordeaux</td>
</tr>
<tr>
<td>Hessen</td>
</tr>
</tbody>
</table>
**Projection: Example 2**

\[
\pi_{\text{District}, \text{Region}}(\text{PRODUCER})
\]

<table>
<thead>
<tr>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>Napa Valley</td>
<td>Kalifornien</td>
</tr>
<tr>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>Pomerol</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
</tbody>
</table>
Selection

- Syntax

\[ \sigma_{\text{Condition}}(\text{Relation}) \]

- Semantics (for \( A \in R \))

\[ \sigma_{A=a}(r) := \{ t \in r \mid t(A) = a \} \]
Selection Conditions

- **Constant Selection**
  
  \[
  \text{Attribute } \theta \text{ Constant}
  \]
  
  boolean predicate \( \theta \) is \( = \) or \( \neq \), for linear ordered range of values also \( \leq \), \( < \), \( \geq \) or \( > \)

- **Attribute Selection**
  
  \[
  \text{Attribute1 } \theta \text{ Attribute2}
  \]

- logic connection of multiple Constant- or Attribute-Selections with \( \land \), \( \lor \) or \( \neg \)
Selection: Properties

▶ Commutativity

\[ \sigma_{A=a}(\sigma_{B=b}(r)) = \sigma_{B=b}(\sigma_{A=a}(r)) \]

▶ when \( A \in X, \ X \subseteq R \)

\[ \pi_X(\sigma_{A=a}(r)) = \sigma_{A=a}(\pi_X(r)) \]

▶ Distributivity respect. \( \cup, \ \cap, \ \ominus \)

\[ \sigma_{A=a}(r \cup s) = \sigma_{A=a}(r) \cup \sigma_{A=a}(s) \]
Selection: Example

\[ \sigma_{\text{Vintage} > 2000}(\text{WINES}) \]

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>Red</td>
<td>2003</td>
<td>Creek</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>Red</td>
<td>2004</td>
<td>Helena</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>Red</td>
<td>2001</td>
<td>Creek</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>White</td>
<td>2002</td>
<td>Bighorn</td>
</tr>
</tbody>
</table>
Join

- Syntax of the natural join
  \[\text{Relation}_1 \bowtie \text{Relation}_2\]

- Semantics
  \[r_1 \bowtie r_2 := \left\{ t \mid t(R_1 \cup R_2) \land \left[ \forall i \in \{1, 2\} \exists t_i \in r_i : t_i = t(R_i) \right] \right\}\]

- Join links tables over equally named columns at equal attribute values
Join: Properties

- Schema for \( r(R) \bowtie r(S) \) is union of the attribute sets
  \[ RS = R \cup S \]
- from \( R_1 \cap R_2 = \{\} \) follows \( r_1 \bowtie r_2 = r_1 \times r_2 \)
- Commutativity: \( r_1 \bowtie r_2 = r_2 \bowtie r_1 \)
- Associativity: \( (r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3) \)
- thus allowed:
  \[ \bowtie_{i=1}^{p} r_i \]
## Join: Example

**WINES \(\bowtie\) PRODUCER**

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>...</th>
<th>Vineyard</th>
<th>District</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1042</td>
<td>La Rose Grand Cru</td>
<td>...</td>
<td>Ch. La Rose</td>
<td>Saint-Emilion</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>2168</td>
<td>Creek Shiraz</td>
<td>...</td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>3456</td>
<td>Zinfandel</td>
<td>...</td>
<td>Helena</td>
<td>Napa Valley</td>
<td>Kalifornien</td>
</tr>
<tr>
<td>2171</td>
<td>Pinot Noir</td>
<td>...</td>
<td>Creek</td>
<td>Barossa Valley</td>
<td>South Australia</td>
</tr>
<tr>
<td>3478</td>
<td>Pinot Noir</td>
<td>...</td>
<td>Helena</td>
<td>Napa Valley</td>
<td>Kalifornien</td>
</tr>
<tr>
<td>4711</td>
<td>Riesling Reserve</td>
<td>...</td>
<td>Müller</td>
<td>Rheingau</td>
<td>Hessen</td>
</tr>
<tr>
<td>4961</td>
<td>Chardonnay</td>
<td>...</td>
<td>Bighorn</td>
<td>Napa Valley</td>
<td>Kalifornien</td>
</tr>
</tbody>
</table>
Renaming

▶ Syntax

\[ \beta_{\text{new} \leftarrow \text{old}}(\text{Relation}) \]

▶ Semantic

\[ \beta_{B \leftarrow A}(\mathcal{r}) := \{ t' \mid \exists t \in r : t'(R - A) = t(R - A) \land t'(B) = t(A) \} \]

▶ changes attribute names from old to new

\[ \beta_{\text{Name} \leftarrow \text{LastName}} (\text{CRITIC}) \]

▶ with renaming now possible

▶ Join, where Cartesian products were applied (different attributes get equal naming),
▶ Cartesian products, where Joins were applied (equal attributes get different naming),
▶ Set operations
Calculation of the Cross Product

- Natural Join degenerates to a cross product, when no shared attributes exist
- Enforce by renaming
  - Example: $R_1(A, B, C)$ and $R_2(C, D)$
    
    $R_1 \times R_2 \equiv R_1 \bowtie \beta_{E \leftarrow C}(R_2)$
  
- Cross product + selection simulates natural join
  
  $R_1 \bowtie R_2 \equiv \sigma_{R_1.C = R_2.C}(R_1 \times R_2)$
Set Operations: Semantics

- formal for $r_1(R)$ and $r_2(R)$
  - Union $r_1 \cup r_2 := \{t \mid t \in r_1 \lor t \in r_2\}$
  - Intersection $r_1 \cap r_2 := \{t \mid t \in r_1 \land t \in r_2\}$
  - Difference $r_1 - r_2 := \{t \mid t \in r_1 \land t \notin r_2\}$
- Intersection $\cap$ is superfluous as $r_1 \cap r_2 = r_1 - (r_1 - r_2)$
Independence and Completeness

▶ Minimal relational algebra:

\[ \Omega = \pi, \sigma, \bowtie, \beta, \cup \text{ and } \neg \]

▶ independence: no operator can be left off without losing completeness

▶ other independent set: \( \bowtie \) and \( \beta \) replaced by \( \times \)

▶ relational Completeness: every other set of operators with same expressive power as \( \Omega \)

▶ strict relational completeness: for any expression with operators out of \( \Omega \) there is an equivalent expression also with the other set of operations
Relational Algebra: Extensions

- Further Join operations
- Division
- Grouping and nested grouping relations
- ...
Join Variants

- for \( L(AB), R(BC), S(DE) \)

- **Equi-Join**: Equality condition over explicit specified and possibly different attributes

  \[ r(R) \bowtie_{C=D} r(S) \]

- **Theta-Join** (\( \theta \)-join): arbitrary join condition

  \[ r(R) \bowtie_{C>D} r(S) \]

- **Semi-Join**: only attributes of one operand appear in the result

  \[ r(L) \bowtie r(R) = \pi_L(r(L) \bowtie r(R)) \]

- **Outer Join**
Outer Join

- Adoption of "dangling tuples" into the result and fill up with null values
- **Full Outer Join** takes all tuples of both operands

\[ r \bowtie s \]

- **Left Outer Join** takes all tuples of the left operand

\[ r \leftarrow s \]

- **Right Outer Join** takes all tuples of the right operand

\[ r \rightleftarrows s \]
### Outer Join /2

**LEFT**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**RIGHT**

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**natural join**

**left outer join**

**right outer join**

**full outer join**
Problem: Quantifiers

- Express universal quantification (allquantor) in relational algebra, even though quantification in the selection condition is not allowed
  - Division (can be derived from $\Omega$)
  - $r_1(R_1)$ and $r_2(R_2)$ given with $R_2 \subseteq R_1, R' = R_1 - R_2$. Then is
    \[
    r'(R') = \{ t \mid \forall t_2 \in r_2 \exists t_1 \in r_1 : t_1(R') = t \land t_1(R_2) = t_2 \}
    \]
    \[
    = r_1 \div r_2
    \]
  - Division of $r_1$ by $r_2$
    \[
    r_1 \div r_2 = \pi_{R'}(r_1) - \pi_{R'}((\pi_{R'}(r_1) \Join r_2) - r_1)
    \]
## Division Example I

**WINE\_RECOMMENDATION**

<table>
<thead>
<tr>
<th>Wine</th>
<th>Critic</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
<td>Parker</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>Parker</td>
</tr>
<tr>
<td>Riesling Reserve</td>
<td>Parker</td>
</tr>
<tr>
<td>La Rose Grand Cru</td>
<td>Clarke</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>Clarke</td>
</tr>
<tr>
<td>Riesling Reserve</td>
<td>Gault-Millau</td>
</tr>
</tbody>
</table>

**GUIDES1**

<table>
<thead>
<tr>
<th>Critic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parker</td>
</tr>
<tr>
<td>Clarke</td>
</tr>
</tbody>
</table>

**GUIDES2**

<table>
<thead>
<tr>
<th>Critic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parker</td>
</tr>
<tr>
<td>Gault-Millau</td>
</tr>
</tbody>
</table>
Division Example II

▶ Division with first table

\[ \text{WINE\_RECOMMENDATION} \div \text{GUIDES1} \]

gives

<table>
<thead>
<tr>
<th>Wine</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
</tbody>
</table>

▶ Division with second critics table

\[ \text{WINE\_RECOMMENDATION} \div \text{GUIDES2} \]

gives

<table>
<thead>
<tr>
<th>Wine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Riesling Reserve</td>
</tr>
</tbody>
</table>
Term Division

Analogy to the arithmetic operation of integer division

The integer division is in this sense the inverse to the multiplication by giving the result of the biggest number for which the multiplication with the divisor is smaller than the dividend.

Analogously holds: $r = r_1 \div r_2$ is the biggest relation for which $r \circ r_2 \subseteq r_1$ is.
Division in SQL

- Simulation of the Allquantor (Division)
- with double Negation:

```sql
select distinct Wine
from WINE_RECOMMENDATION w1
where not exists (
    select * from GUIDES2 g
    where not exists (
        select * from WINE_RECOMMENDATION w2
        where g.Critic = w2.Critic and w1.Wine = w2.Wine))
```

- "Gives all wines, thus no wine exists that is not recommended by all critics in the relation GUIDES2".
- (We use the relation GUIDES2, as the in the textbook stated result relation refers to this comparison relation.)
Grouping

- Grouping Operator $\gamma$:

\[
\gamma(f_1(x_1), f_2(x_2), \ldots, f_n(x_n); A(r(R)))
\]

- Extends the attribute scheme of $r(R)$ by new attributes that corresponds with the function application $f_1(x_1), f_2(x_2), \ldots, f_n(x_n)$

- Application of the function $f_i(x_i)$ on the subset of the tuples of $r(R)$ which have the same attribute values for the attributes $A$

```sql
select f_1(x_1), f_2(x_2), \ldots, f_n(x_n); A
from R
group by A
```
Semantics of the Grouping Operator

- empty attribute set $A = \emptyset$:

$$\gamma_{F(X) ; \emptyset}(r(R)) = r(R) \times r(R)^{F(X)}$$

with $r(R)^{F(X)}$ is relation with attribute $F(X)$ and a tuple as value of $F(X)$ on $r(R)$

- without function:

$$\gamma_{\emptyset ; \emptyset}(r(R)) = r(R)$$

- general case:

$$\gamma_{F(X) ; A}(r(R)) = \bigcup_{t \in R} \gamma_{F(X) ; \emptyset}(\sigma_{A=t.A}(r(R)))$$
Query Calculus

- Calculus: A formal logic language to formalize statements
- Goal: The use of such a calculus to formalize database queries
- Logic-based approach:
  - Database contents match an assignment of a predicate logic
  - Query: derived predicates
A General Calculus

- Motivation: Mathematical Notation

\[ \{x^2 \mid x \in \mathbb{N} \land x^3 > 0 \land x^3 < 1000\} \]

- Query has the form

\[ \{f(\bar{x}) \mid p(\bar{x})\} \]

- \( \bar{x} \) denotes set of free variables

\[ \bar{x} = \{x_1 : D_1, \ldots, x_n : D_n\} \]
A General Calculus /2

- Function $f$ denotes result function over $\bar{x}$
  - Important special cases: Declaration of a variable itself ($f$ is here the identity function) and tuple construction (result of type **tuple of**)

- $p$ selection predicate over free variables $\bar{x}$
  - Terms of variables, constants, and function applications
  - Predicates of data types, such as $\leq$, $<$, $>$, $\geq$, ...
    → atomic formulas over terms
  - Relation to current database → database predicates, e.g., relation name in the RM
  - predicate logical operators $\land$, $\lor$, $\neg$, $\forall$, $\exists$
    → formulas
Result Declaration of a Query

\[ \bar{x} = \{x_1 : D_1, \ldots, x_n : D_n\} \]

1. Declare all assignments of free variables in \( \bar{x} \), for which the predicate \( p \) gets true.
2. Apply function \( f \) on the values given by this assignment.

Under which circumstances do calculus queries give infinite results?

→ Safety of queries
Relational Calculus

▶ **Domain Calculus**: Variables take values of elementary data type (*domains*).

▶ **Tuple calculus**: Variables vary over tuple values (according to the lines of a relation).
Tuple Calculus

- Basics of SFW-Queries in SQL
- Variables are tuple valued
- Example:

\[ \{w \mid w \in \text{WINES} \land w.\text{Color} = \text{’Red’}\} \]
Tuple Calculus: Example

▶ constructed tuple

\[ \{ \langle w.\text{Name}, w.\text{Vineyard} \rangle \mid w \in \text{WINES} \land w.\text{Color} = 'Red' \} \]

▶ Join

\[ \{ \langle e.\text{Vineyard} \rangle \mid e \in \text{PRODUCER} \land w \in \text{WINES} \land e.\text{Vineyard} = w.\text{Vineyard} \} \]

▶ Nesting

\[ \{ \langle w.\text{Name}, w.\text{Vineyard} \rangle \mid w \in \text{WINES} \land \exists e \in \text{PRODUCER} (w.\text{Vineyard} = e.\text{Vineyard} \land e.\text{Region} = 'Bordeaux') \} \]
Motivation: The Language QBE

- "Query by Example"
- Queries in QBE: Entries in table frameworks
- Intuition: Example entries in tables
- Precursor of several table-based query interfaces of commercial systems
- Based on logical calculus with domain variables
Queries in QBE: Selection and Projection

Query: "All rock-albums of the years before 2006"

{\( t \mid \text{Album}(\_ , t , j , 'Rock' , \_ , \_ ) \land j < 2006 \)}
Queries in QBE: Join

Query: "All Rock albums of German musicians"

\[
\{ t \mid \text{Album}(__, t, __, 'Rock', __, m) \land \text{Musician}(m, __, 'Germany') \}
\]
Queries in QBE: Self-Join

- Query: "Countries with two or more musicians"

<table>
<thead>
<tr>
<th>Musician</th>
<th>MNr</th>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>_one</td>
<td>-_one</td>
<td>P. _country</td>
<td>_country</td>
</tr>
</tbody>
</table>

\[
\{l \mid \text{Musician}(x, _, l) \land \text{Musician}(y, _, l) \land x \neq y\}
\]
QBE in MS-Access

- MS-Access: Database program for Windows
  - Basis relations with keys
  - Foreign keys with graphical statement of relations
  - Graphical definition of queries (SQL-similar)
  - Interactive definition of forms and reports
- Support of QBE
Access: Projection and Selection

![Diagram showing a database query for selecting wines with specific criteria.](image)

- **Feld**: Name, Jahrgang, Farbe, Weingut
- **Tabelle**: WEINE
- **Sortierung**: WEINE
- **Anzeigen**: Checkboxes for Name, Jahrgang, Farbe
- **Kriterien**: Age > 2005, Color: "Rot"
Access: Join

<table>
<thead>
<tr>
<th>Feld</th>
<th>Tabelle</th>
<th>Sortierung</th>
<th>Anzeigen</th>
<th>Kriterien</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>WEINE</td>
<td>WEINE</td>
<td>❑</td>
<td>&quot;Napa Valley&quot;</td>
</tr>
<tr>
<td>Jahrgang</td>
<td>WEINE</td>
<td></td>
<td>❑</td>
<td></td>
</tr>
<tr>
<td>Farbe</td>
<td>ERZEUGER</td>
<td></td>
<td>❑</td>
<td></td>
</tr>
<tr>
<td>Weingut</td>
<td></td>
<td></td>
<td>❑</td>
<td></td>
</tr>
<tr>
<td>Anbaugebiet</td>
<td>ERZEUGER</td>
<td></td>
<td>❑</td>
<td></td>
</tr>
<tr>
<td>Region</td>
<td></td>
<td></td>
<td>❑</td>
<td></td>
</tr>
</tbody>
</table>
Domain Calculus

- **Terms:**
  - Constants, such as 42 or ’MZ-4’
  - Variables for data types, such as $x$
    Data type declaration usually takes place implicitly and is not declared explicitly!
  - Function application $f(t_1, \ldots, t_n)$: Function $f$, terms $t_i$, such as $\text{plus}(12, x)$ resp. in infix notation $12 + x$

- **Atomic formulas:**
  - Predicate application $\Theta(t_1, \ldots, t_n)$,
    $\Theta \in \{<, >, \leq, \geq, \neq, =, \ldots\}$ data-type predicate, terms $t_i$
    Binary predicates are usually in infix-notation.
    Example: $x = y$, $42 > x$ or $3 + 7 = 11$. 
Atomic formulas (advanced):

Predicate application for database predicates noted as $R(t_1, \ldots, t_n)$ for a relation name $R$
Providing: $n$ must be the arity of the relation $R$ and all $t_i$
must be of the same type
Example: $\text{PRODUCER}(x, 'Hessen', z)$

Formula as usual with $\land$, $\lor$, $\neg$, $\forall$ and $\exists$
Domain Calculus /3

- Query: \{x_1, \ldots, x_n \mid \phi(x_1, \ldots, x_n)\}
  - \phi is formula over the variables contained in the result list \(x_1\) to \(x_n\)
  - Result is set of tuples
  - Tuple construction comes implicitly from the values of the variables in the result list

- Example

\[
\{x \mid \text{PRODUCER}(x, y, z) \land z = 'Hessen'\}
\]
Basic Domain Calculus

- Restriction of the domain calculus:
  - **Range of values**: Integer
  - **Data type predicates** are restricted to equivalence and elementary comparison operations as in relational algebra
  - **Function applications** are not allowed; besides domain variables only constants can be used as terms

- used for theoretical investigations, e.g., proves of properties
Safety

- **Safe Queries** (also *semantic safe queries*):

  Queries that give a finite result for each database state $\sigma(R)$

  - Example for unsafe queries:

    $$\{x, y \mid \neg R(x, y)\}$$

  - Example for safe queries:

    $$\{x, y \mid R(x, y)\}$$
Safe Queries /2

▶ Further example for safe queries:

\[ \{ x, y \mid y = 10 \land x > 0 \land x < 10 \} \]

Safety comes directly from the rules of arithmetics.

Semantic safety is in general not decidable!
Syntactically Safe Queries

- **Syntactically Safe Queries**: Queries that are subject to syntactic restrictions to enforce semantic safety.

- **Basic idea**: Any free variable $x_i$ must be bound to a finite domain anywhere in $\phi(x_1, \ldots)$ by a positive occurrence $x_i = t$ or $R(\ldots, x_i, \ldots)$.

- Binding to finite domains must hold for the whole condition, in particular for all branches of an disjunction.
Safe Queries Overview

queries

safe queries

syntactic safe queries
Examples Domain Calculus

▶ Query: "All vineyards of producers in Hessen."

\[ \{ x \mid \text{PRODUCER}(x, y, z) \land z = 'Hessen' \} \]

▶ Simplified notation: Each unbound variable (here \( y \) and \( z \)) in the condition part existentially bound with \( \exists \)

▶ Complete version:

\[ \{ x \mid \exists y \exists z \text{PRODUCER}(x, y, z) \land z = 'Hessen' \} \]

▶ Saving of domain variables, by using constants as parameters of the predicate:

\[ \{ x \mid \text{PRODUCER}(x, y, 'Hessen') \} \]
Examples Domain Calculus /2

► Abbreviation for arbitrary, different existentially bound variables is the _ symbol:

\{x \mid \text{PRODUCER}(x, _, z) \land z = 'Hessen'\}

► Different appearances of the symbol _ stand for pairwise different variables
Query: "Regions with more than two vineyards."

\[
\{ z \mid \text{PRODUCER}(x, y, z) \land \text{PRODUCER}(x', y', z) \land x \neq x' \}
\]

- Query shows a join binding on the third attribute of the PRODUCER-relation
- Join binding can easily come from the use of the same domain variables as parameter in different relation predicates
Examples Domain Calculus /4

► Query: "From which region are which wines with the vintage before 1970 in the supply?"

\{y, r \mid \text{WINES}(x, y, z, j, w) \land \text{PRODUCER}(w, a, r) \land j < 1970\}

► Join over two relations
Example Domain Calculus /5

Query: "From which regions are red wines?"

\[
\{ z \mid \text{PRODUCER}(x, y, z) \land \exists a \exists b \exists c \exists d (\text{WINE}(a, b, c, d, x) \land c = \text{’Red’}) \}
\]

Use of a existentially bound subquery

Such subqueries could be dissolved due to the rules of predicate logic as follows:

\[
\{ z \mid \text{PRODUCER}(x, y, z) \land (\text{WINE}(a, b, c, d, x) \land c = \text{’Red’}) \}
\]
Examples Domain Calculus /6

► Query: "Which vineyard has only wines of the vintage after 1995 in its supply?"

\{ x \mid \text{PRODUCER}(x, y, z) \land \forall a \forall b \forall c \forall d (\text{WINE}(a, b, c, d, x) \Rightarrow d > 1995) \}

► Universally bound subformulas cannot be dissolved
Expressiveness of the Domain Calculus

Domain calculus is strict relational complete, i.e. to any term $\tau$ of the relational algebra there is an equivalent (safe) term $\eta$ of the domain calculus.
Implementation of Relational Operations

Given: Relation schema $R(A_1, \ldots, A_n)$ and $S(B_1, \ldots, B_m)$

- Union (for $n = m$)

  $$R \cup S \triangleq \{x_1 \ldots x_n \mid R(x_1, \ldots, x_n) \lor S(x_1, \ldots, x_n)\}$$

- Difference (for $n = m$)

  $$R - S \triangleq \{x_1 \ldots x_n \mid R(x_1, \ldots, x_n) \land \neg S(x_1, \ldots, x_n)\}$$

- Natural Join

  $$R \bowtie S \triangleq \{x_1 \ldots x_n x_{n+1} \ldots x_{n+m-i} \mid R(x_1, \ldots, x_n) \land S(x_1, \ldots, x_i, x_{n+1}, \ldots, x_{n+m-i})\}$$

Assumption: the first $i$ attributes of $R$ and $S$ are the join attributes, thus $A_j = B_j$ for $j = 1 \ldots i$
Implementation of Relational Operations /2

► Projection

\[ \pi_A(R) \doteq \{ y_1 \ldots y_k \mid \exists x_1 \ldots \exists x_n (R(x_1, \ldots, x_n) \land y_1 = x_{i_1} \land \cdots \land y_k = x_{i_k}) \} \]

Attribute list of the projection: \( \bar{A} = (A_{i_1}, \ldots, A_{i_k}) \)

► Selection

\[ \sigma_{\phi}(R) \doteq \{ x_1 \ldots x_n \mid R(x_1, \ldots, x_n) \land \phi' \} \]

\( \phi' \) comes from \( \phi \), by insertion of the variable \( x_i \) at the place of the attribute names \( A_i \)
Summary

▶ Formal models for queries in database systems
▶ Relational algebra
  ▶ Operational approach
  ▶ Queries as nesting of operations on relations
▶ Query calculus
  ▶ Logic-based approach
  ▶ Queries as derived predicates
  ▶ see Book: sections 4.2.3, 4.2.4 and 9.3
Control Questions

▶ Which meaning do equivalence, independence and completeness have in the relational algebra?
▶ How can the semantic of the extended SQL-operations be expressed in relational algebra?
▶ What is the difference between relational algebra and relational query calculus?
▶ What is the role of safety in queries?
Part VIII

Transactions, Integrity and Triggers
Transactions, Integrity and Triggers

1. Basic Terms
2. Term Transaction
3. Transactions in SQL
4. Integrity Constraints in SQL
5. Trigger
Learning goals for today . . .

▶ Understanding of fundamentals of integrity control in databases
▶ Knowledge to formalize and implement integrity constraints
▶ Knowledge of the transaction concept in databases
Integrity

- Integrity constraint (*also: assertion*): Condition for the "permissibility" or "correctness"
- with respect to databases:
  - (single) database states,
  - state transitions from an old to a new database state,
  - long term database evolution
## Classification of Integrity

<table>
<thead>
<tr>
<th>Constraint Class</th>
<th>Temporal Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>database state</td>
</tr>
<tr>
<td>dynamic</td>
<td>transitional state</td>
</tr>
<tr>
<td></td>
<td>temporal state sequence</td>
</tr>
</tbody>
</table>

- **Static**: database state
- **Dynamic**: transitional state sequence
Inherent Integrity Constraints in the RM

1. *Type Integrity:*
   - SQL allows domain definitions for a range of values for attributes
   - Permission or forbidding of null values

2. *Key Integrity:*
   - Specification of a key for a relation

3. *Referential Integrity:*
   - Specification of foreign keys
Example Scenarios

▶ Seat reservation for flights simultaneously from multiple travel agencies
  → Seat could be sold multiple times when multiple travel agencies identify the seat as available

▶ Overlapping account operations of a bank

▶ Statistics database operations
  → results are corrupted when data is changed during the calculation
Transaction

A transaction is a sequence of operations (actions) that transforms the database from a consistent state into a consistent, possibly changed, state, while the ACID-principle must be hold.

▶ Aspects:

▶ Semantic Integrity: Correct (consistent) DB-state after a transaction has finished
▶ Operational Integrity: Prevent fault caused by "simultaneous" access of multiple users on the same data
ACID-Properties

- **Atomicity:**
  Transaction is executed completely or not at all

- **Consistency:**
  Database is before the start and after the end of a transaction in a consistent state

- **Isolation:**
  User, who is working on a database, should have the impression that she works alone on the database

- **Durability (Persistence):**
  The result of transaction has to be saved "permanently" in a database after the transaction competed successfully
Commands of a Transaction Language

▶ Begin of a transaction: Begin-of-Transaction-Command \textbf{BOT} (implicit in SQL!)

▶ \textbf{commit}: the transaction should try to finish successfully
  ▶ success is not guaranteed!

▶ \textbf{abort}: the transaction has to be aborted
  ▶ abort is guaranteed!
Transaction: Integrity Violation

▶ Example:
  ▶ Transfer of an amount $A$ from a household post $K1$ to another post $K2$
  ▶ Condition: Sum of the account balances stays constant

▶ Simplified notation
  \[ \text{Transfer} = < K1:=K1-A; K2:=K2+A >; \]

▶ Realization in SQL: as sequence of two elementary changes
  \[ \rightsquigarrow \text{Condition is not necessarily fulfilled between single changing steps!} \]
Transaction: Behavior at System Crash

![Diagram showing transactions and time]

- $T_1$
- $T_2$
- $T_3$
- $T_4$
- $T_5$

Time $t_f$

Crash
Transaction: Behavior at System Crash /2

▶ Consequences:
  ▶ Contents of the volatile memory at the time $t_f$ is unusable
    $\rightarrow$ transactions in different ways affected by this

▶ Transaction states:
  ▶ Still active transactions at the time of the failure ($T_2$ and $T_4$)
  ▶ Already finished transactions before the time of the failure
    ($T_1$, $T_3$ and $T_5$)
Simplified Model for Transactions

▶ Representation of database changes of a transaction
  ▶ $\text{read}(A, x)$: assign the value of the DB-object $A$ to the variable $x$
  ▶ $\text{write}(x, A)$: save the value of the variable $x$ in the DB-object $A$

▶ Example of a transaction $T$:

\[
\text{read}(A, x); \quad x := x - 200; \; \text{write}(x, A);
\]
\[
\text{read}(B, y); \quad y := y + 100; \; \text{write}(y, B);
\]

▶ Execution variants for two transactions $T_1, T_2$:
  ▶ serially, e.g. $T_1$ before $T_2$
  ▶ "mixed", e.g. alternating steps of $T_1$ and $T_2"
Problems with Multi-User Operation

- Nonrepeatable Read
- Dependencies on not released data: Dirty Read
- The Phantom-Problem
- Lost Update
Nonrepeatable Read

Example:

- Assurance $x = A + B + C$ at the end of transaction $T_1$
- $x, y, z$ are local variables
- $T_i$ is the transaction $i$
- Integrity constraint $A + B + C = 0$
### Example for Nonrepeatable Read

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{read}(A, x);$</td>
<td>$\text{read}(A, y);$</td>
</tr>
<tr>
<td></td>
<td>$\text{write}(y, A);$</td>
</tr>
<tr>
<td>$\text{read}(B, y);$</td>
<td>$\text{read}(C, z);$</td>
</tr>
<tr>
<td>$x := x + y;$</td>
<td>$z := z + y;$</td>
</tr>
<tr>
<td>$\text{read}(C, z);$</td>
<td>$\text{write}(z, C);$</td>
</tr>
<tr>
<td>$x := x + z;$</td>
<td>$\text{commit};$</td>
</tr>
<tr>
<td>$\text{commit};$</td>
<td>$\text{commit};$</td>
</tr>
</tbody>
</table>
## Dirty Read

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A, x$);</td>
<td>read($A, x$);</td>
</tr>
<tr>
<td>$x := x + 100$;</td>
<td>read($B, y$);</td>
</tr>
<tr>
<td>write($x, A$);</td>
<td>$y := y + x$;</td>
</tr>
<tr>
<td>abort;</td>
<td>write($y, B$);</td>
</tr>
<tr>
<td></td>
<td>commit;</td>
</tr>
</tbody>
</table>
The Phantom-Problem

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>select count (*) into $X$ from Customer;</td>
<td>insert into Customer values ('Meier', 0, ...);</td>
</tr>
<tr>
<td>update Customer set Bonus = Bonus + 10000/$X$; commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>
Lost Update

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
$T_1$ & $T_2$ & $A$
\hline
\texttt{read}(A, x); & \texttt{read}(A, x); & 10
\hline
\texttt{x} := x + 1; & \texttt{x} := x + 1; & 10
\hline
\texttt{write}(x, A); & \texttt{write}(x, A); & 11
\hline
\end{tabular}
\end{center}
Serializability

An interleaved execution of multiple transactions is called **serializable**, if its effect is identical to the effect of a (arbitrarily chosen) serial execution of these transactions.

- **Problem for checking serializability:**
  - there are $n!$ different serial execution orders for $n$ transactions...

- **Schedule:** Plan of execution for transactions (ordered list of transaction operations)
Transactions in SQL-DBS

Weakening of ACID in SQL: Isolation levels

```
set transaction
[ { read only | read write }, ]
[isolation level
 { read uncommitted | read committed |
  repeatable read |
  serializable }, ]
[ diagnostics size ...]
```

Default settings:

```
set transaction read write,
   isolation level serializable
```
Meaning of Isolation Levels

▶ read uncommitted
▶ weakest level: access to not committed data, only for read only transactions
▶ statistic and similar transactions (approximate overview, incorrect values possible)
▶ no locks → efficient executable, other transactions are not hindered

▶ read committed
▶ only read finally written values, but nonrepeatable read possible

▶ repeatable read
▶ no nonrepeatable read, but phantom-problem can occur

▶ serializable
▶ guarantees serializability
## Isolation Levels: read committed

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>set transaction isolation level read committed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><strong>select</strong> Name from WINES where WineID = 1014 → <em>Riesling</em></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td><strong>update</strong> WINES set Name = 'Riesling Superiore' where WineID = 1014</td>
</tr>
<tr>
<td>3</td>
<td><strong>select</strong> Name from WINES where WineID = 1014 → <em>Riesling</em></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><strong>commit</strong></td>
</tr>
<tr>
<td>5</td>
<td><strong>select</strong> Name from WINES where WineID = 1014 → <em>Riesling Superiore</em></td>
<td></td>
</tr>
</tbody>
</table>
### Isolation Levels: read committed /2

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set transaction</td>
<td>update WINES</td>
</tr>
<tr>
<td></td>
<td>isolation level</td>
<td>set Name = 'Riesling Super-</td>
</tr>
<tr>
<td></td>
<td>read committed</td>
<td>ore'</td>
</tr>
<tr>
<td>2</td>
<td>select Name from WINES</td>
<td>where WineID = 1014</td>
</tr>
<tr>
<td></td>
<td>where WineID = 1014</td>
<td>update WINES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set Name = 'Riesling Superiore'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where WineID = 1014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→ <strong>blocked</strong></td>
</tr>
<tr>
<td>3</td>
<td>update WINES</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td>set Name = 'Superiore Ries-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lining'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>where WineID = 1014</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>commit</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Isolation Levels: serializable

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set transaction</td>
<td>update WINES</td>
</tr>
<tr>
<td></td>
<td>isolation level</td>
<td>set Name = 'Riesling Superior'</td>
</tr>
<tr>
<td></td>
<td>serializable</td>
<td>where WineID = 1014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abort</td>
</tr>
<tr>
<td>2</td>
<td>select Name into N from WINES where WineID = 1014</td>
<td>update WINES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set Name = 'Riesling Superior'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where WineID = 1014</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>5</td>
<td>update WINES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set Name = 'Superior'</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>where WineID = 1014</td>
<td></td>
</tr>
</tbody>
</table>

- $T_1$: Transaction $T_1$ sets the isolation level to serializable.
- $T_2$: Transaction $T_2$ queries the database and updates the wine name to 'Riesling Superior'.
- The update is committed.
- Transaction $T_1$ aborts.
Integrity Constraints in SQL-DDL

- **not null**: Null values prohibited
- **default**: Specification of default values
- **check** (search-condition): Attribute specific constraint (usually One-Tuple-Integrity-Condition)
- **primary key**: Specification of a primary key
- **foreign key** (Attribute(e))
  - references **Table** (Attribute(e)): Specification of the referential integrity
Integrity Constraints: Range of Values

▶ create domain: Establishing of a user defined range of values
▶ Example

```sql
create domain WineColor varchar(5)
    default 'Red'
    check (value in ('Red', 'White', 'Rose'))
```

▶ Application

```sql
create table WINES (  
    WineID int primary key,
    Name varchar(20) not null,
    Color WineColor,
    ...
)
```
Integrity Constraints: check-Clause

▷ **check**: Establishing of further local integrity constraints within the defined range of values, attributes and relational scheme

▷ Example: Restriction of permitted values

▷ Example

```sql
create table WINES (  
  WineID int primary key,  
  Name varchar(20) not null,  
  Year int check(Year between 1980 and 2010),  
  ...  
)```
Preservation of Referential Integrity

- Checking of foreign keys after database changes

- for $\pi_A(r_1) \subseteq \pi_K(r_2)$,
  - e.g. $\pi_{\text{Vineyard}}(\text{WINES}) \subseteq \pi_{\text{Vineyard}}(\text{PRODUCER})$
    - Tuple $t$ is inserted into $r_1$ ⇒ check, whether $t' \in r_2$ exists with: $t'(K) = t(A)$, d.h. $t(A) \in \pi_K(r_2)$
      - if not ⇒ reject
    - Tuple $t'$ is removed from $r_2$ ⇒ check, whether $\sigma_{A=t'(K)}(r_1) = \{\}$, i.e. no tuple from $r_1$ references $t'$
      - if not empty ⇒ reject or remove tuple from $r_1$, that reference $t'$ (at cascading deletion)
Checking Modes of Constraints

▶ on update | delete
Specification of a triggering event that starts the checking of the condition

▶ cascade | set null | set default | no action
Cascading: Handling of some integrity violations propagates over multiple levels, e.g. deletion as reaction on a violation of the referential integrity

▶ deferred | immediate sets the checking time for a condition
  ▶ deferred: put back to the end of the transaction
  ▶ immediate: immediate verification at any relevant database change
Checking Modes: Example

- Cascading deletion

```sql
create table WINES (
    WineID int primary key,
    Name varchar(50) not null,
    Price float not null,
    Jahr int not null,
    Vineyard varchar(30),
    foreign key (Vineyard) references PRODUCER (Vineyard)
    on delete cascade)
```
The assertion-Clause

- Assertion: Predicate expressed by a condition that always has to be fulfilled by a database
- Syntax (SQL:2003)

```sql
create assertion name check ( predicate )
```

- Example:

```sql
create assertion Prices check
    ( ( select sum (Price) 
       from WINES) < 10000 )
```

```sql
create assertion Prices2 check
    ( not exists ( 
       select * from WINES where Price > 200 ) )
```
Trigger

- Trigger: Statement/Procedure that is executed automatically by the DBMS at the occurrence of a specific event

- Application:
  - Enforcement of integrity conditions ("implementation" of integrity rules)
  - Auditing of DB-actions
  - Propagation of DB-changes

- Definition:

  ```
  create trigger ...
  after <Operation>
  <Procedure>
  ```
Example for Triggers

- Realization of a calculated attribute with two triggers:
  - Introduction of new tasks
    
    ```
    create trigger TaskCounter+
    on insertion of Task A:
    update Customer
    set NrTasks = NrTasks + 1
    where CName = new A.CName
    ```

  - Analogously for deletion of tasks:
    
    ```
    create trigger TaskCounter-
    on deletion ...
    update ....- 1 ...
    ```
Trigger: Design and Implementation

- Specification of
  - Event and condition for activation of the trigger
  - Action(s) for the execution
- Syntax in SQL:2003 defined
- Available in most commercial systems (but with different syntax)
SQL:2003-Trigger

- Syntax:

```sql
create trigger <Name:>
after | before <Event>
on <Relation>
[ when <Condition> ]
begin atomic < SQL-statements > end
```

- Event:

  - insert
  - update [ of <list of attributes> ]
  - delete
Further Specifications for Triggers

▶ for each row resp. for each statement: Activation of the trigger for each single change of a set-valued change or just once for the whole change

▶ before resp. after: Activation before or after the change

▶ referencing new as resp. referencing old as: Binding of a tuple variable on the new introduced resp. just removed ("old") tuple of a relation

⇝ tuple of the difference relation
Example for Triggers

▶ No customer account can fall below 0:

```
create trigger bad_account
after update of Acc on CUSTOMER
referencing new as INSERTED
when (exists
    (select * from INSERTED where Acc < 0)
)
begin atomic
    rollback;
end
```

⇝ similar trigger for insert
Example for triggers /2

Producers must be removed, if they do not offer any wine:

```sql
create trigger useless_Vineyard
  after delete on WINES
  referencing old as o
  for each row
  when (not exists
       (select * from WINES W
        where W.Vineyard = o.Vineyard))
  begin atomic
    delete from PRODUCER where Vineyard = o.Vineyard;
  end
```
Integrity Enforcement with Triggers

1. Specify object $o_i$, for which the condition $\phi$ should be monitored
   - Usually monitor multiple $o_i$ when condition is across relations
   - Candidates for $o_i$ are tuples of the relation names that occur in $\phi$

2. Specify the elemental database changes $u_{ij}$ on objects $o_i$ that can violate $\phi$
   - Rules: e.g., check existence requirements on deletion and updates, but not on insertion etc.
3. Specify, depending on the application, the reaction $r_i$ on the integrity violation
   ▶ Reset the transaction (rollback)
   ▶ Correcting database changes

4. Formulate following triggers:

   ```sql
   create trigger t-phi-ij after u_{ij} on o_i
   when \neg \phi
   begin r_i end
   ```

5. If possible, simplify the created trigger
Trigger in Oracle

- Implementation in PL/SQL
- Notation

```sql
CREATE [ OR REPLACE ] TRIGGER trigger-name
    before | after
    insert or update [ of columns ]
    or delete on table
    [ for each row
    [ when ( predicate ) ] ]
PL/SQL-Block
```
Trigger in Oracle: Types

- Statement level trigger: Trigger is triggered before resp. after the DML-statement
- Row level trigger: Trigger is triggered before resp. after each single modification (*one tuple at a time*)

Trigger on row level:

- Predicate for restriction (**when**)
- Access on old (**old.col**) resp. new (**new.col**) tuple
  - for **delete**: only (**old.col**)
  - for **insert**: only (**new.col**)
  - in **when**-clause only (**new.col**) resp. (**old.col**)
Trigger in Oracle /2

- Transaction abortion with `raise_application_error(code, message)`
- Distinction of the type of the DML-statement

```sql
if deleting then ... end if;
if updating then ... end if;
if inserting then ... end if;
```
Trigger in Oracle: Example

- No customer account can fall below 0:

```sql
create or replace trigger bad_account
after insert or update of Acc on Customer
for each row
when (:new.Acc < 0)
begin
    raise_application_error(-20221, 'Not below 0');
end;
```
Summary

▶ Enforcement of correctness resp. integrity of the data
▶ Inherent integrity constraints of the relational model
▶ Additional SQL-integrity constraints: check-clause, assertion-statement
▶ Trigger for "implementation" of integrity constraints resp. rules
Control Questions

▶ What is the purpose of integrity enforcement? Which types of integrity constraints are there?

▶ How can integrity constraints and rules be formulated in SQL systems?

▶ What requirements result from the ACID-principle? How are these achieved in database systems?
Part IX

Views and Access Control
Views and Access Control

1 View Concept
2 Updates via Views
3 Assignment of Rights
4 Privacy-Aspects
Learning goals for today . . .

▶ Understanding of the view concept of databases
▶ Knowledge to formalize and to use views in SQL
▶ Knowledge of the problems with updates via views
▶ Knowledge to data protection aspects in context with aggregated / statistical data
Views

Views: virtual relations (resp. virtual database objects in other data models)

- Views are external DB-schemata that follow the 3-level-schema architecture
- View definition
  - Relation schema (implicit or explicit)
  - Calculation rule for virtual relations, such as SQL-query
Views /2

▶ Advantages
  ▶ Simplification of queries for the user of the database, e.g. by realization of often required sub-queries
  ▶ Possibility of structuring of the database description, specific to user classes
  ▶ Logic data independence enables robustness of the interface for applications against changes to the database structure (accordingly vice versa)
  ▶ Description of access rights on the database in context with the access control

▶ Problems
  ▶ Automatic query transformation
  ▶ Execution of updates on views
Three-Level Schema Architecture

- External Schema 1
- External Schema N
- Conceptual Schema
- Internal Schema

Query Processing

Data Representation
Definition of Views in SQL

```
create view ViewName [ SchemaDeclaration ]
as SQLQuery
[ with check option ]
```

- Schema declaration is optional (could be derived from SQL query)
Views - Example

all red wines from Bordeaux:

```sql
create view RedWines as
    select Name, Vintage, WINES.Vineyard
    from WINES natural join PRODUCER
    where Color = 'Red'
    and Region = 'Bordeaux'
```
Problem Areas of Views

- Execution of updates via views
- Automatic query transformation
Criteria for Updates via Views

- **Effect Conformity**
  User sees effect as if the update was done directly on the view relation.

- **Minimality**
  Basis database should only be changed minimal to preserve the mentioned effect.

- **Consistency Preservation**
  Updates of a view must not lead to integrity violations of the basis database.

- **Respecting the Database Protection**
  If a view is implemented for data protection purposes, then the consciously faded out part of the basis database must not be effected by changes of the view.
Projection View

\[ \text{WNW} := \pi_{\text{WineID}, \text{Name}, \text{Vineyard}}(\text{WINES}) \]

- In SQL with `create view`-statement:

```sql
create view WNW as
    select WineID, Name, Vineyard from WINES
```

- Update statement for the view `WNW`:

```sql
insert into WNW values (3333, 'Dornfelder', 'Müller')
```

- Corresponding statement on the basis relation `WINES`:

```sql
insert into WINES
    values (3333, 'Dornfelder', null, null, 'Müller')
```

→ Problem of Consistence preservation if Color or Vintage declared as `not null`!
Selection Views

\[ WJ := \sigma_{\text{Vintage}>2000}(\pi_{\text{WineID},\text{Vintage}}(\text{WINES})) \]

create view WJ as
select WineID, Vintage
from WINES
where Vintage > 2000

Tuple migration: Tuple
WINES(3456, 'Zinfandel', 'Red', 2004, 'Helena'), gets "moved out" of the view:

update WINES
set Vintage = 1998
where WineID = 3456
Control of Tuple Migration

create view WJ as
select WineID, Vintage
from WINES
where Vintage > 2000
with check option
Join Views

\[ WE := \text{WINES} \bowtie \text{PRODUCER} \]

**In SQL:**

```sql
create view WE as
select WineID, Name, Color, Vintage, WINES.Vinyard, Area, Region
from WINES, PRODUCER
where WEINE.Vineyard = ERZEUGER.Vineyard
```

**Update operations usually not clearly translatable:**

```sql
insert into WE
values (3333, 'Dornfelder', 'Red', 2002, 'Helena', 'Barossa Valley', 'South Australia')
```
Join Views /2

- Update is transformed to

```sql
insert into WINES
values (3333, 'Dornfelder', 'Red', 2002, 'Helena')
```

- Plus
  1. Insert statement on ERZEUGER:

```sql
insert into PRODUCER
values ('Helena', 'Barossa Valley', 'South Australia')
```

  2. Or alternative:

```sql
update PRODUCER
set Area = 'Barossa Valley', Region = 'South Australia'
where Vineyard = 'Helena'
```

better regarding minimality requirement, but contradicts effect conformity!
Aggregation Views

```sql
create view FM (Color, MinVintage) as
select Color, min(Vintage)
from WINES
group by Color
```

▶ Following update is not clearly realizable:

```sql
update FM
set MinVintage = 1993
where Color = 'Red'
```
Classification of Problem Areas

1. Violation of the schema definition (e.g. introduction of null values at projection view)
2. Data protection: Avoid side effects on invisible part of the database (tuple migration, selection views)
3. Not always clear transformation: choice problem
4. Aggregation views (among others): no useful transformation possible at all
5. Elemental view updates should exactly comply with an atomic change on basis relation: 1:1-Relation between view tuples and tuples of the basis relation (no projection of keys)
Handling of Views in SQL

- SQL-92-Standard
  - Integrity-violating view changes are prohibited
  - Data-protection-violating view updates: user control (with check option)
  - View with unclear transformation: view not updateable (SQL-92 more restrictive than necessary)
Restrictions for View Updates

▶ Only selection and projection views updateable (join and set operations prohibited)
▶ 1:1-Relation of view tuples to basis tuples: no distinct in projection view
▶ Arithmetic and aggregation functions in the select-part are prohibited
▶ Exactly one reference on one relation name in the from-part permitted (also no self join)
▶ No sub-queries with "self reference" in the where-part permitted (use relation name in the top SFW-block not in the from-parts of sub-queries)
▶ group by and having prohibited
Evaluation of Queries on Views

- Simple syntactical transformation:
  - **select**: View attributes, probably renamed resp. replaced by calculation term
  - **from**: Names of the original relations
  - Conjunctive linking of the **where**-clauses of the view definition and queries (probably renaming)
Problems with Aggregation Views

create view FM (Color, MinVintage) as
select Color, min(Vintage)
from WINES
group by Color

▶ Query: Wine colors with old vintages

select Color
from FM
where MinVintage < 1995
Problems with Aggregation Views /2

- After simple syntactic transformation:

```sql
select Color
from WINES
where min(Vintage) < 1995
group by Color
```

- No syntactic correct SQL-query – correct would be:

```sql
select Color
from WINES
group by Color
having min(Vintage) < 1995
```
Problems with Aggregation Views /3

Query

```sql
select max (MinVintage) 
from FM
```

Should be transformed as follows:

```sql
select max(min (Vintage)) 
from WINES 
group by Color
```

But: Nested aggregation functions are prohibited in SQL!
Assignment of Access Rights in Databases

- **Access rights**
  
  \[(\text{AuthorizationID}, \text{DB-Excerpt}, \text{Operation})\]

- AuthorizationID is internal identification of a "database user"

- Database excerpts: relations and views

- DB-Operations: read, insert, update, remove
Assignment of Rights in SQL

```
grant <Rights>
on <Table>
to <UserList>
[with grant option]
```
Assignment of Rights in SQL /2

► Explanations:
  ► In <Rights>-List: all resp. long form all privileges or list of select, insert, update, delete
  ► After on: relation and view name
  ► After to: Authorization identifications (also public, group)
  ► Special right: right on passing of rights (with grant option)
Authorization for public

```sql
create view MyJobs as
select *
from JOB
where KName = user;

grant select, insert
on MyJobs
to public;

"Every user can see her jobs and can insert new jobs
(but not remove!)."
```
Taking Back of Rights

```
revoke <Rights>
on <Table>
from <UserList>
[restrict | cascade ]
```

- **restrict**: If rights already passed to thirds: abort of revoke
- **cascade**: Propagate revocation of the rights with `revoke` to all users that received them from this user with `grant`
Privacy: Term and Areas of Application

**Privacy**: The right of each individual on a save and private room, that can only be violated by others in exceptional cases.

- Electronic highway toll system: Monitoring of vehicles
- Credit card activities and diverse payback resp. discount cards: buying behavior of customers
- Mobile communication systems: movement profiles of users
- RFID-technology: e.g. in retail trade the customer behavior, flow of goods, etc.
Statistic Databases

- Databases in which single entries are subject to data protection, but statistic information about all users is accessible
- Statistic information = aggregated data (average income etc.)
- Problem: Extraction of single information with indirect queries
Statistic Databases: Example

Example: User X can query data about the account holder as well as statistic data, but no single account balances

1. Simplification of search criterion (only one customer gets selected)

```sql
select count (*) from ACCOUNT
where Place = 'Manebach' and Age = 24 and ...
```

2. Name of the account holder

```sql
select Name from ACCOUNT
where Place = 'Manebach' and Age = 24 and ...
```

3. Statistic query, that actually gives a single entry

```sql
select sum(Balance) from ACCOUNT
where Place = 'Manebach' and Age = 24 and ...
```

Remedy: no query that select less than \( n \) tuples
Statistic Database: Example /2

- X wants to find out balance of Y
- X knows, that Y does not live in Ilmenau
- X has queried, that more than \( n \) account holders live in Ilmenau

1. Sum of the balances of customers from Ilmenau

\[
\text{select sum(Balance) from Account where Place = 'Ilmenau'}
\]

2. Sum of the balances of customers from Ilmenau + Customer Y

\[
\text{select sum(Balance) from Account where Name = :Y or Place = 'Ilmenau'}
\]

3. Difference of the results gives balance of Y

- Remedy: prohibition of statistic queries that affect pairwise an average of more than \( m \) given tuples
Statistic Databases: Conclusion

- Critical parameters
  - Result size $n$
  - Size of the overlapping of the result sets $m$

If only results of aggregate functions are permitted, then a person needs $1 + (n - 2)/m$ queries to determine a single attribute value.
## k-Anonymity

For many purposes (clinical studies etc.) detail data (micro data) is required.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>ZIP</th>
<th>Gender</th>
<th>Marital State</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>*****</td>
<td>38</td>
<td>98693</td>
<td>male</td>
<td>married</td>
<td>cold</td>
</tr>
<tr>
<td>*****</td>
<td>29</td>
<td>39114</td>
<td>female</td>
<td>single</td>
<td>fever</td>
</tr>
<tr>
<td>*****</td>
<td>29</td>
<td>39114</td>
<td>female</td>
<td>single</td>
<td>anemia</td>
</tr>
<tr>
<td>*****</td>
<td>34</td>
<td>98693</td>
<td>male</td>
<td>married</td>
<td>cough</td>
</tr>
<tr>
<td>*****</td>
<td>34</td>
<td>98693</td>
<td>male</td>
<td>married</td>
<td>broken bone</td>
</tr>
<tr>
<td>*****</td>
<td>27</td>
<td>18055</td>
<td>male</td>
<td>single</td>
<td>fever</td>
</tr>
<tr>
<td>*****</td>
<td>27</td>
<td>18055</td>
<td>female</td>
<td>single</td>
<td>cold</td>
</tr>
</tbody>
</table>
k-Anonymity: Problem

- Is for a person of this relation known that he is:
  - male
  - 38 years old
  - married
  - living in 98693 Ilmenau
- cold
- Further relation (Name etc.), e.g. by join with other data
- Solution: Data Swapping (??)
**k-Anonymity**

**k-Anonymity:** a certain fact cannot be differentiated among a given amount of $k$ tuples

- A query for an arbitrary combination of age, gender, marital state and ZIP code gives either an empty relation or at least $k$ tuples
**k-Anonymity: Approaches**

- **Generalization**: Replace attribute values by more general values that are gathered from a generalization hierarchy
  - Generalization of the age of the person to age classes: \{35, 39\} $\rightsquigarrow$ 30-40
  - Leave off digits of the ZIP code: \{39106, 39114\} $\rightsquigarrow$ 39**

- **Suppression of tuples**: Removing of tuples that violate the $k$-anonymity and thus are identifiable
Control Questions

▸ What is a database view? How are views defined?

▸ Are views updateable? Under which conditions?

▸ How can data protection be achieved in databases?
Summary

▶ Views to structure databases
▶ Problems with updates via views
▶ Access right system in SQL-DBS
▶ Privacy aspects
Part X

Application Programming
Application Programming

1. Programming Language Connection
2. JDBC
3. SQLJ
4. LINQ
5. Object-Relational Mapping
6. Procedural SQL-Extensions: SQL/PSM
Learning goals for today . . .

- Knowledge about concepts and interfaces for access on SQL-databases out of programming languages
- Understanding of procedural interfaces on the example of JDBC
- Knowledge on embedded SQL and procedural SQL-extensions
- Basic knowledge on object-relational mapping
Programming Language Connection

- Coupling types:
  - Procedural or CALL-interfaces (call level interface)
    - Examples: SQL/CLI, ODBC, JDBC, …
  - Embedding of a DB-language into programming languages
    - Static embedding: Precompiler-principle
      - SQL-Statements defined at compile time
    - Examples: Embedded SQL, SQLJ
    - Dynamic embedding:
      - construction of SQL-statements at runtime

- Language extensions and new language developments
  - Examples: SQL/PSM, PL/SQL, Transact-SQL, PL/pgSQL
Cursor-Concept

Cursor: iterator over list of tuples (query result)
JDBC: Overview

- Database access interface for Java
- Abstract, database neutral interface
- Comparable with ODBC
- Low-Level-API: direct usage of SQL
- Java-Package `java.sql`
  - `DriverManager`: Entrance point, loading of drivers
  - `Connection`: Database connection
  - `Statement`: Execution of statement with a connection
  - `ResultSet`: Manages results of a query, access on single columns
JDBC: Structure

DriverManager

getConnection

Connection

createStatement

Statement

executeQuery

ResultSet

Statement

ResultSet
JDBC: Driver Concept

Java Application

JDBC Driver Manager

Native Protokoll Driver

JDBC Net Driver

JDBC ODBC Bridge

Native API Driver

DB Middleware

ODBC

Client Library

Client Library
JDBC: Sequence of Events

1. Establishing of a connection to the database
   ▶ Specification of connection information
   ▶ Selection and loading of the driver

2. Sending of a SQL-query
   ▶ Definition of the statement
   ▶ Assignment of parameters

3. Processing of the query results
   ▶ Navigation over result relation
   ▶ Access on columns
JDBC: Connection Establishment

1. Loading drivers

   ```java
   Class.forName("com.company.DBDriver");
   ```

2. Establish connection

   ```java
   String url = "jdbc:subprotocol:datasource";
   Connection con = DriverManager.getConnection(url, "scott", "tiger");
   ```

JDBC-URL specifies

- Data source / Database
- Connection mechanism (Protocol, Server and Port)
JDBC: Query Execution

1. Create statement

   Statement stmt = con.createStatement();

2. Execute statement

   String query = "select Name, Vintage from WINES";
   ResultSet rSet = stmt.executeQuery(query);

Class java.sql.Statement

- Execution of queries (SELECT) with executeQuery
- Execution of changing statements (DELETE, INSERT, UPDATE) with executeUpdate
JDBC: Result Processing

1. Navigation over result set (Cursor-Principle)

```java
while (rSet.next()) {
    // Processing of single tuples
    ...
}
```

2. Access of column values with `getType`-methods
   - with column index
     ```java
     String wName = rSet.getString(1);
     ```
   - with column name
     ```java
     String wName = rSet.getString("Name");
     ```
JDBC: Exception Handling

- Exception handling with **try-catch**-mechanism
- SQLException for all SQL- and DBMS-exceptions

```java
try {
    // call of JDBC-methods
    ...
} catch (SQLException exc) {
    System.out.println("SQLException: " + exc.getMessage());
}
```
JDBC: Update Operations

- DDL- and DML-statements with `executeUpdate`
- Gives number of affected rows (for DML-statements)

```java
Statement stmt = con.createStatement();
int rows = stmt.executeUpdate(
    "update WINES set Price = Price * 1.1 " +
    "where Vintage < 2000");
```
JDBC: Transaction Management

▶ Methods of Connection
  ▶ commit()
  ▶ rollback()

Auto-Commit-Mode
  ▶ Implicit commit after each statement
  ▶ Transaction consists just out of one single statement
  ▶ Switch mode with setAutoCommit(boolean)
SQLJ: Embedded SQL for Java

- Embedding of SQL-statements in Java source code
- Precompilation of the extended source codes onto real Java code with the translator **sqlj**
- Checking of the SQL-statements
  - Correct syntax
  - Accordance of the statements with the DB-scheme
  - Type compatibility of the for data transfer used variables
- Usage of JDBC-drivers
SQLJ: Principle
SQLJ-Statements

- Identification with #sql declaration
- Class definition for iterators
- SQL-statements: Queries, DML- and DDL-statements

```sql
#sql { SQL-statement };
```

- Example:

```sql
#sql { insert into PRODUCER (Vineyard, Region) values
    ( 'Wairau Hills', 'Marlborough' ) };
```
Host-Variables

- Variables of a host-language (here Java) that can occur in SQL-statements
- Usage: Exchange of data between the host-language and SQL
- Identification with ":variable"
- Example:

```java
String name;
int wineID = 4711;
#sql { select Name into :name
    from WINES where WineID = :wineID }
System.out.println("Wine = " + name);
```
Iterators

1. Declaration of the iterator

```sql
#sql public iterator WineIter(String Name, String Vineyard, int Vintage);
```

2. Definition of the iterator object

```java
WineIter iter;
```

3. Execution of the statement

```sql
#sql iter = { select Name, Vineyard, Vintage from WINES };
```

4. Navigation

```java
while (iter.next()) {
    System.out.println(iter.Name() + " " + iter.Vineyard() + " " + iter.Vintage());
}
```
Dynamic SQL

- SQL-Statements as during runtime constructed Strings

```sql
exec sql begin declare section;
  queryString char(256) varying;
exec sql end declare section;
exec sql declare QueryObjekt statement;
queryString =
  'delete from WINES where WineID = 4711';
...
exec sql prepare QueryObjekt from :queryString;
exec sql execute QueryObjekt;
```
Language Integrated Query (LINQ)

- Embedding of a DB-language (SQL) into a programming language (C#)
- Specialized class methods

```csharp
IEnumerable<string> res = wines
    .Where(w => w.Color = "Red")
    .Select(w => new { w.Name });
```

- Own language constructs (since C# 3.0)

```csharp
IEnumerable<op> res = from w in wines
    where w.Color = "Red"
    select new { w.Name };
```
Object-Relational Mapping

- Use of
  - Relational back ends (SQL-DBMS)
  - Object-relational applications, applications servers, middleware, . . .
- Implementation of "'business logic'" in form of objects (customer, order, process, . . .)
  - e.g., as Java Bean, CORBA-object
- Requires: Mapping class ↔ relation
- Aspects:
  - Conceptual mapping
  - Runtime support
- Technologies/Products: JDO, Hibernate, ADO.NET Entity Framework...
Object-Relational Mapping: Principle

Application

Application Objects

Runtime System

Object Model

Mapping

Database Schema

Objects

Object Model

Database Schema

Application

Application

Objects

Object Model

Database Schema

Prof. Thomas Leich
Harz University of Applied Sciences
Copyright: Gunter Saake, University of Magdeburg
Classes and Tables

▶ OO: Class defines properties of objects (intention) + covers set of all objects (extension)
▶ RM: Relation covers all tuples, relational scheme describes structure
▶ Obvious: class = table
▶ But: normalization decomposes relations!
  ▶ 1 class = 1 table
  ▶ 1 class = $n$ tables
  ▶ $n$ classes = 1 table
Classes and Tables: Example

Wine

<table>
<thead>
<tr>
<th>WineID</th>
<th>Name</th>
<th>Color</th>
<th>Vintage</th>
<th>Vineyard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Relations

- **Embedded foreign key** in the relation of the class, i.e. the identifier of the associated object is saved as foreign key in additional columns
- **Foreign key tables**: the relation instance is represented as tuple with the keys of the involved objects
- **Mapping of the relating classes on a single table**: violation of the normal form
- **Concrete**
  - 1:1-Relation: embedded foreign keys
  - 1:n-Relation: embedded foreign keys of foreign key tables
  - Relations with attributes: Foreign key tables
  - m:n-Relations: Foreign key tables
  - Three- and more valued relations: Foreign key tables
Relations /2

- **Producer**
  - Name: string
  - District: string
  - Region: string
  - Winemaker: list of string

- **Vineyard**

- **District**

- **Region**

- **Winemaker**

- **Vineyard Name**

- **Name**
Hibernate

- Java-framework for object-relational mapping
- Idea: Mapping of Java-objects to tuples of a relational database
- Principle: Java-class + mapping rule $\Rightarrow$ SQL-table
- No explicit SQL-statements required!
- Support of the navigation over relations (automatic loading of the referenced objects)
- Queries on some languages (HQL resp. QBC/QBE)
public class Wine {
    private int id;
    private String name;
    private String color;
    private int vintage;
    private String vineyard;

    public void setName(String n) { name = n; }
    public String getName() { return name; }
    public void setColor(String c) { color = c; }
    public String getColor() { return color; }
    public void setVintage(int v) { vintage = v; }
    public int getVintage() { return vintage; }
    ...
}
Hibernate: Example /2

- Declaration of the mapping in a XML-Mapping-File
- Mapping rule is interpreted during runtime

```xml
<hibernate-mapping>
  <class name="Wine" table="WINES">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
    <property name="color" />
    <property name="vintage" column="vintage" />
    <property name="vineyard" />
  </class>
</hibernate-mapping>
```
Transaction tx = null;

Wine wine = new Wine();
wine.setName("Pinot Noir");
wine.setColor("Red");
wine.setVintage(1999);
wine.setVineyard("Helena");

try {
    tx = session.beginTransaction();
    session.save(wine);
    tx.commit();
} catch (HibernateException exc) {
    if (tx != null) tx.rollback();
}
Hibernate: Queries

- Queries with Hibernate’s query language HQL
- Formulation on the *conceptual* scheme (Java-classes)
- Select-clause not required (results are always objects)
- Example

```java
Query query =
    session.createQuery("from Wine where Color = 'Red'");
Iterator iter = query.iterate();
while (iter.hasNext()) {
    Wine wine = (Wine) iter.next();
    ...
}
```
SQL/PSM: The Standard

- SQL-Standard for procedural extensions
- PSM: Persistent Stored Modules
  - Stored modules of procedures and functions
  - Single routines
  - Integration of external routines (implemented in C, Java, ...)
- Syntactic constructs for loops, conditions etc.
- Basis for method implementation for object-relational concepts
Advantages of Stored Procedures

► Proved structuring tool for larger applications
► Specification of functions and procedures done in the database language; thus only depending on DBMS
► Optimization by DBMS possible
► Execution of the procedures completely under control of the DBMS
► Central control of the procedures allows a redundancy free representation of relevant aspects of the application functionality
► Concepts and mechanisms of the right assignment of the DBMS can be extended on procedures
► Procedures can be used for integrity protection (e.g., as action part of triggers)
SQL/PSM: Variable Declaration

- Declare variables before consumption
- Specification of identifier and data type
- Optional with initial value

```
declare Price float;
declare Name varchar(50);
declare Set int default 0;
```
SQL/PSM: Flow Control

► Assignment

```sql
set var = 42;
```

► Conditional branching

```sql
if <Condition> then <Statement>
[  else <Statement>  ] end if;
```
Loops

```sql
loop <Statement> end loop;
while <Condition> do
    <Statement> end while;
repeat <Statement>
    until <Condition> end repeat;
```
Loops with cursor

```sql
for LoopVariable as CursorName cursor for CursorDeclaration
  do
    Statement
  end for;
```

SQL/PSM: Flow Control /3
declare wlist varchar(500) default '';
declare pos integer default 0;

for w as WineCurs cursor for
    select Name from WINES where Vintage = 'Helena'
do
    if pos > 0 then
        set wlist = wlist || ',' || w.Name;
    else
        set wlist = w.Name;
    end if;
    set pos = pos + 1;
end for;
SQL/PSM: Exception Handling

▶ Triggering of an exception (Condition)

```sql
signal <ConditionName>;
```

▶ Declaration of exceptions

```sql
declare missing_vineyard condition;
declare invalid_region condition for sqlstate value '40123';
```
begin
    declare exit handler for ConditionName
    begin
        -- statements for exception handling
    end
    -- statements that can trigger exceptions
end
SQL/PSM: Functions

Function definition

```sql
create function taste (rz int)
returns varchar(20)
begin
    return case
        when rz <= 9 then 'Dry'
        when rz > 9 and rz <= 18 then 'Medium-Dry'
        when rz > 18 and rz <= 45 then 'Smooth'
        else 'Sweet'
    end
end
```
SQL/PSM: Functions /2

- Call inside of a query

```sql
select Name, Vineyard, taste(residualSugar)
from WINES
where Color = 'Red' and taste(residualSugar) = 'Dry'
```

- usage outside of queries

```sql
set wine_taste = taste(12);
```
**Procedure definition**

```sql
create procedure winelist (in prod varchar(30),
                        out wlist varchar(500))
begin
    declare pos integer default 0;

    for w as WineCurs cursor for
        select Name from WINES where Vintage = prod
    do
        -- see example of slide 10-513
    end for;
end; end;
```
SQL/PSM: Procedures /2

- Usage via call-statement

```sql
declare wlist varchar(500);
call winelist ('Helena', wlist);
```
SQL/PSM: Access Characteristics

- Properties of procedures that affect query execution and optimization
  - **deterministic**: Routine gives same results for same parameters
  - **no sql**: Routine contains no SQL-statements
  - **contains sql**: Routine contains SQL-statements (standard for SQL-routines)
  - **reads sql data**: Routine executes SQL-queries (select-statements)
  - **modifies sql data**: Routine that contains DML-statements (insert, update, delete)
Control Questions

▶ What concepts exist that can access SQL-databases?
▶ What are advantages and disadvantages of call-level-interfaces such as JDBC in comparison with embedding of SQL?
▶ How can application objects be mapped to SQL-tables? What tasks are therefore required?
Summary

- Connection between SQL and imperative languages
- Call-level-interfaces vs. embedded SQL
- Object relational mapping
- SQL/PSM: imperative extension of SQL → implementation of functions and procedures