Database Systems
Spring 2017

Introduction
SL01

- Organization of the course
- The database field, basic definitions
- DB applications, functionality, users and languages
- Data models, schemas, instances, and redundancy
- Main characteristics of the database approach

Organization of the Course

- Database curricula at IfI
- Literature
- Lectures
- Exercises
- Content

About me

- I have been a database system person since more than 20 years.
- My previous affiliations (and the first example of a database table):

Affiliations

Start	End	Institution	Country
1990	1994	ETH Zürich	CH
1994	1995	University of Arizona	USA
1995	2003	Aalborg University	DK
2003	2009	Free University of Bozen-Bolzano	IT
2009	now	University of Zürich	CH

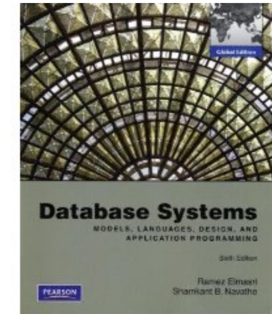
About the Database Systems Course

- Slides can be accessed through the course web page: `http://www.ifi.uzh.ch/dbtg/teaching/courses/DBS.html`
- The slides are designed as a **working script**.
- The textbook is Database Systems by Elmasri and Navathe.
- Use the slides, and optionally the textbook, for preparation throughout the semester.
- During the lecture we will solve illustrative examples on the board. Interaction during class is welcome.
- What is important
  - Being able to **apply** your knowledge to relevant **examples**.
  - Being able to be **precise** about the key concepts of database systems.

About Database Sytems @IfI

- **Database Systems** (DBS), Spring, 4th semester
- **Praktikum Datenbanksysteme** (PDBS), Fall, 5th semester
- **Distributed Databases** (DDBS), Fall even years, 5th semester
- **Seminar Database Systems** (SDBS), Spring, 6th or 8th semester
- **XML und Datenbanken**, (XMLDB), Spring, 6th or 8th semester
- **Data Warehousing**, Spring (DW), Spring even years, 8th semester
- **Temporal and Spatial Data Management** (TSDM), Fall odd years, 9th semester

Literature and Acknowledgments



Reading List for SL01:

- Database Systems, Chapters 1 and 2, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Database Systems, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

DBS Course/1

- Lectures:
  - Tuesday 10:15-12:00 in BIN-0.K.02
  - Wednesday 12:15-13:45 in BIN-0.K.02
- The final exam is written and takes place Tuesday, June 20, 10:15 - 12:00 (check official web pages for details).
- All written material (slides, exercises, exam) is in English.

  The assessment consists of the completion of 9 out of 12 exercises and the participation at the final exam. Both parts have to be passed independently.

- Office hours after appointment with TAs (after exercise hour or by email).
- There is no re-exam.

DBS Course/2

- The weekly exercises are a crucial part of the course.
- The exercises take place Tuesday 12:15-13:45. Start is February 28. During the first week there are no exercises.
- TAs: Oksana Dolmatova (English), Yvonne Mülle (German), Kevin Wellenzohn (English).
- Please sign up for the exercise groups by the end of this week by filling the Doodle (cf. course web page). We will balance the load across groups.
- Hand in of the exercises is Tuesday 12:15 in the exercise room or before to TA directly.
- Exercises are only valid for the current year.

DBS Exercises

- **Exercises**
  - **28.2** Relational algebra
  - **07.3** Domain relational calculus
  - **14.3** SQL (metadata, DDL, simple DML), PostgreSQL
  - **21.3** Transformations: RA - DRC - SQL
  - **28.3** Advanced SQL
  - **04.4** Functions and triggers
  - **11.4** Relational database design
  - **25.4** Functional dependencies, multivalued dependencies
  - **02.5** Entity relationship (ER) model
  - **09.5** Physical design and indexing
  - **16.5** Query trees and plans, cost computation
  - **23.5** Transaction processing
  - **30.5** Wrap up

DBS Syllabus/1

1. **Database systems**, chapter 1 and 2
   - The field, terminology, database system, schema, instance, functionality, architecture
2. **Relational model, algebra, and calculus**, chapter 3 and 6
   - The relational model, relational algebra, relational calculus
3. **SQL**, chapter 4
   - Data definition language, data manipulation language
4. **Constraints, triggers, views, DB access**, chapter 5 and 12
   - column constraints, table constraints, assertions, referential integrity, triggers, stored procedures
5. **Relational database design**, chapter 14 and 15
   - design goals, keys, functional dependencies, normal forms, lossless join decompositions, higher normal forms

DBS Syllabus/2

6. **Conceptual database design**, chapter 7 and 8
   - The design process, the entity-relationship model, entity-relationship to relational model mapping
7. **Physical database design**, chapter 16 and 17
   - Physical Storage media, file and buffer manager, indices, B-trees, hashing
8. **Query processing and optimization**, chapter 18 and 19
   - Measures of query cost, selection and join operation, transformation of relational expressions, evaluation plans
9. **Transactions, concurrency, recovery**, chapter 20, 21, 22
   - ACID properties, SQL transactions, concurrency protocols, log-based recovery

Notational Conventions/1

Relational Algebra (RA):

- **constant**: 'abc', 14, 3.14, ...
- **attribute**: *Name*, *X*, ... (upper case)
- **relation name**: *Employee*, *R*, ... (upper case)
- **tuple**: t, t_1, ... (lower case)
- **relation**: *emp*, r, s, ... (lower case)
- **schema**: $sch(emp) = Employee(Name, Addr)$, ...
- **database**: *D*, *DB*, ... (upper case)

Domain Relational Calculus (DRC), First Order Predicate Logic (FOPL):

- **constant**: 'abc', 14, ...
- **variable**: *X*, *Y*, ... (upper case)
- **predicate**: *p*, *q*, ... (lower case)

Notational Conventions/2

SQL:

- **constant**: 'abc', 14, ...
- **SQL keyword**: **SELECT**, **FROM**, ... (all cap, blue, bold)
- **attribute**: *Name*, *Salary*, ... (upper case)
- **table**: *r*, *dept*, ... (lower case)

Entity Relationship Model (ER Model):

- **constant**: 'abc', 14, ...
- **attribute**: *Name*, *Gender*, ... (green, upper case)
- **relationship**: *workFor*, ... (red, lower case)
- **entity**: *Company*, *Emp*, ... (blue, upper case)

The Database Field

- Professional Resources
- Products
- Activities of Database People
- Basic Terminology and Definitions

The Field/1

- Conference Publications
  - SIGMOD/PODS
  - VLDB
  - ICDE
  - EDBT/ICDT
- Journal Publications
  - ACM Transaction on Database System (TODS)
  - The VLDB Journal (VLDBJ)
  - Information Systems (IS)
  - IEEE Transactions on Knowledge and Data Engineering (TKDE)
- DBLP Bibliography (Michael Ley, Uni Trier, Germany)
  - http://dblp.uni-trier.de/db/
- DBWorld mailing list
  - http://www.cs.wisc.edu/dbworld/

The Field/2

home | browse | search | about

The DBLP Computer Science Bibliography

maintained by the DBLP Team - Welcome - FAQ

DBLP is available from three hosts: Trier I - Trier II - Dagstuhl

Search

- Author
- Faceted search (L3S Research Center, Leibniz Universität Hannover)
- Free search (isearch)
- CompleteSearch (Hannah Bast, University of Freiburg)

Bibliographies

- **Conferences:** SIGMOD, VLDB, PODS, ER, EDBT, ICDE, POPL, ...
- **Journals:** CACM, TODS, TOIS, TOPLAS, DKE, VLDB J., Inf. Systems, TPLP, TCS, ...
- **Series:** LNCS/LNAI, IFIP
- **Books:** Reference - Collections

Full Text: ACM SIGMOD Anthology

Links

- Computer Science Organizations: ACM (DL / SIGs), IEEE (Xplore), IEEE Computer Society (DL), IFIP, GI, ...
- Related Services: Google Scholar, MS Academic Search, CiteSeer/ CiteSeerX, CS BibTeX (DBLP), io-port.net, CoRR, HAL, NZ-DL, Zentralblatt MATH, MathSciNet, Erdös Number Proj., Math Genealogy Proj., BibSonomy, CiteULike, ScientificCommons, Libra, Arnetminer, RePEc, ...

Products

- Commercial Products
  - Oracle
  - DB2 (IBM)
  - SQL Server (Microsoft)
  - Teradata
  - SAP HANA
  - ...
- Open Source Products
  - PostgreSQL
  - MySQL (Oracle), MariaDB
  - MongoDB (NoSQL)
  - CouchDB (NoSQL, JSON, MapReduce)
  - Cassandra
  - MonetDB
  - ...

We will use PostgreSQL for this course.

Oracle's Solution Stack

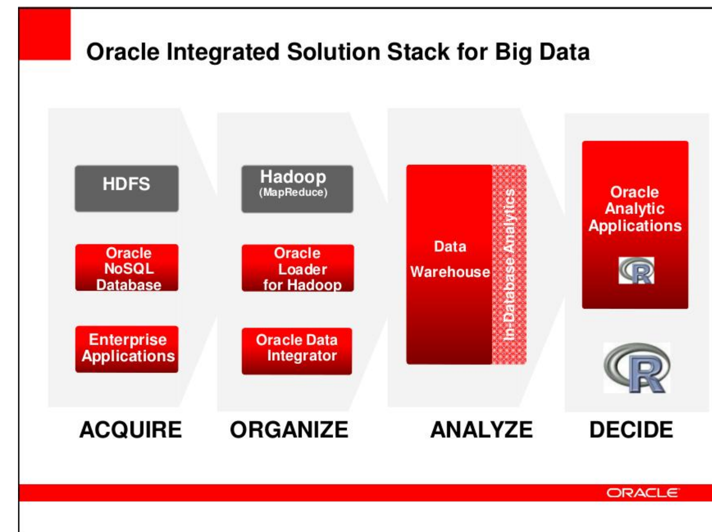


Image: Roger Wullschleger, Oracle @ DBTA Workshop on Big Data, Bern, 2012

Basic Definitions/1

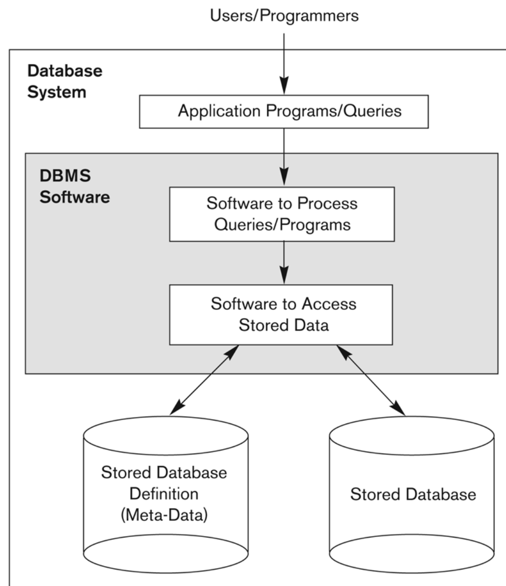
About, data, information, and knowledge:

- **Data** are facts that can be recorded:
  - book('Lord of the Rings', 3, 10)
- **Information** = data + meaning
  - book:
  - title = 'Lord of the rings',
  - volume nr = 3,
  - price in USD = 10
- **Knowledge** = information + application

Basic Definitions/2

- **Mini-world**: The part of the real world we are interested in
- **Data**: Known facts about the mini-world that can be recorded
- **Database (DB)**: A collection of related data
- **Database Management System (DBMS)**: A software package to facilitate the creation/maintenance/querying of databases
- **Database System (DBS)**: DB + DBMS
- **Meta Data**: Information about the structure of the DB.
  - which tables? which columns? which users? which access rights?
  - Meta data is organized as a DB itself.

Basic Definitions/3



DBMS Languages/1

- A DBMS offers two types of languages:
  - data definition language (DDL) to create and drop tables, etc
  - data manipulation language (DML) to select, insert, delete, and update data
- The standard language for database systems is SQL
  - SQL stands for Structured Query Language
  - Example SQL query: `SELECT * FROM r`
  - the original name was SEQUEL
  - "Intergalactic data speak" [Michael Stonebraker].
- SQL offers a DDL and a DML.

DBMS Languages/2

- We distinguish between
  - High level or declarative (non-procedural) languages
  - Low level or procedural languages
- High level or declarative language:
  - For example, the SQL language
  - Set-oriented (retrieve multiple results)
  - Specify **what** data to retrieve and not how to retrieve it
- Low level or procedural language:
  - Retrieve data one record at a time
  - Specify **how** to retrieve data
  - Constructs such as looping are needed to retrieve multiple records, along with positioning pointers.

Review 1.1

1. Give examples of declarative and procedural approaches from the real world.

Applications, Functionality, Users and Interfaces

- Application Areas of Database Systems
- Functionality of Database Systems
- Users of Database Systems
- DBMS Interfaces

Applications of Database Systems

- Traditional Applications
  - Numeric and Textual Databases
- More Recent Applications:
  - Multimedia Databases
  - Geographic Information Systems (GIS)
  - Data Warehouses
  - Real-time and Active Databases
  - Many other applications
- Examples:
  - Bank (accounts)
  - Insurances
  - Stores (inventory, sales)
  - Reservation systems
  - University (students, courses, rooms)
  - online sales (amazon.com)
  - online newspapers (nzz.ch)

Typical Activities/Jobs of Database People

- Data modeling (e.g., UZH)
- Handling large volumes of complex data (scientific data, astrophysics, genome data, etc)
- Distributed databases
- Design of migration strategies
- User interface design
- Development of algorithms
- Design of languages
- New data models and systems
  - XML/semi-structured databases
  - Stream data processing
  - Temporal and spatial databases
  - GIS systems
- etc.

Functionality of Database Systems/1

Typical DBMS functionality:

- **Define** a particular database in terms of its data types, structures, and constraints
- **Construct** or **load** the initial database contents on a persistent storage medium
- **Manipulating** the database:
  - Retrieval: Querying, generating reports
  - Modification: Insertions, deletions and updates to its content
  - Accessing the database through Web applications
- **Sharing** by a set of concurrent users and application programs while, at the same time, keeping all data valid and consistent

Functionality of Database Systems/2

Additional DBMS functionality:

- Other features of DBMSs:
  - **Protection** or **security** measures to prevent unauthorized access
  - **Active** processing to take internal actions on data
  - **Presentation** and visualization of data
  - **Maintaining** the database and associated programs over the lifetime of the database application (called database, software, and system maintenance)

Users of Database Systems/1

Database users have very different tasks. There are those who use and control the database content, and those who design, develop and maintain database applications.

- **Database administrators**:
  - Responsible for authorizing access to the database, for coordinating and monitoring its use, acquiring software and hardware resources, controlling its use and monitoring efficiency of operations.
- **Database Designers**:
  - Responsible to define the content, the structure, the constraints, and functions or transactions against the database. They must communicate with the end-users and understand their needs.

Users of Database Systems/2

- **End-users**: They use the data for queries, reports and some of them update the database content. End-users can be categorized into:
  - **Casual**: access database occasionally when needed
  - **Naïve**: they make up a large section of the end-user population.
    - They use previously well-defined functions in the form of "canned transactions" against the database.
    - Examples are bank-tellers or reservation clerks.
  - **Sophisticated**:
    - These include business analysts, scientists, engineers, others thoroughly familiar with the system capabilities.
    - Many use tools in the form of software packages that work closely with the stored database.
  - **Stand-alone**:
    - Mostly maintain personal databases using ready-to-use packaged applications.
    - An example is a tax program user that creates its own internal database or a user that maintains an address book

DBMS Interfaces/1

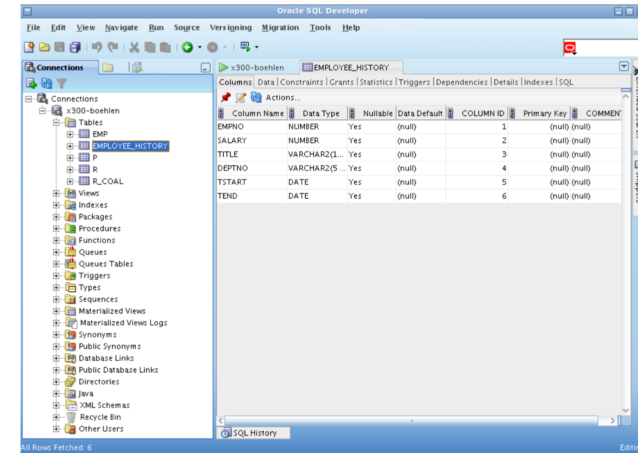
- **User-friendly** interfaces
  - Menu-based, forms-based, graphics-based, etc.
- **Stand-alone** query language interfaces
  - Example: Entering SQL queries at the DBMS interactive SQL interface (e.g. psql in PostgreSQL, sqlplus in Oracle)
- **Program interfaces** for embedding DML in programming languages
- **Web Browser** as an interface
- **Speech** as Input and Output
- **Parametric interfaces**, e.g., bank tellers using function keys.
- **Interfaces for the DBA**:
  - Creating user accounts, granting authorizations
  - Setting system parameters
  - Changing schemas or access paths

DBMS Interfaces/2

- Programmer interfaces for embedding DML in programming languages:
  - Embedded Approach:
    embedded SQL (for C, C++, etc.)
    SQLJ (for Java)
  - Procedure Call Approach:
    JDBC for Java
    ODBC for other programming languages
  - Database Programming Language Approach:
    e.g., ORACLE has PL/SQL, a programming language based on SQL;
    language incorporates SQL and its data types as integral components

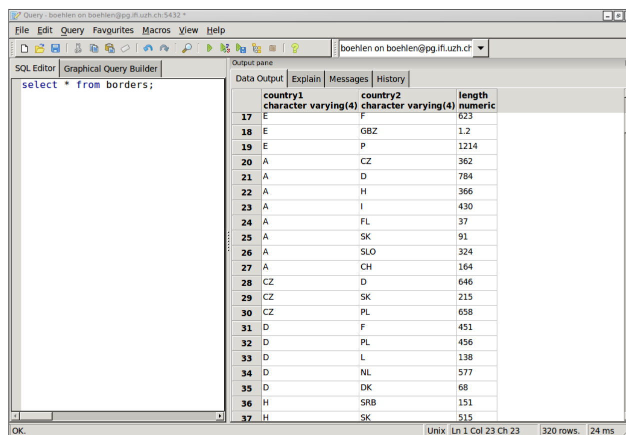
DBMS Interfaces/3

- **Oracle SQL Developer** is a graphical tool for DB development.
- With SQL Developer you can browse database objects, run SQL statements and SQL scripts, and edit and debug PL/SQL statements.



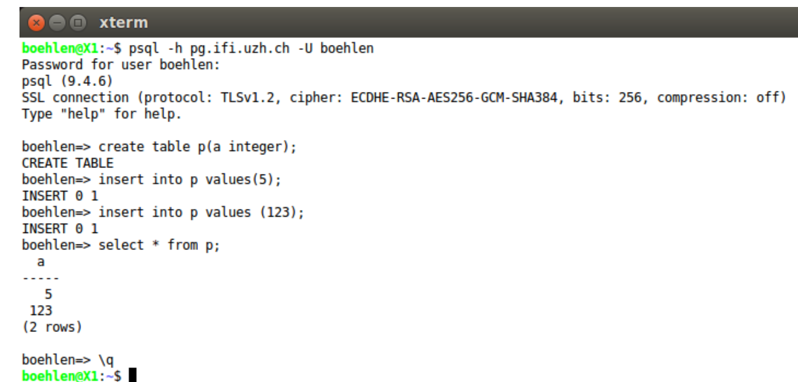
DBMS Interfaces/4

- **pgadmin** is the administration and development platform for PostgreSQL.
- The graphical interface supports all PostgreSQL features, from writing simple SQL queries to developing complex databases.



DBMS Interfaces/5

- Command line tool `psql`:



DBMS Interfaces/6

- There are various database system utilities to perform certain functions such as:
  - **Loading data** stored in files into a database. Includes data conversion tools.
  - **Backing up** the database periodically on tape.
  - **Reorganizing** database file structures.
  - **Report generation** utilities.
  - **Performance monitoring** utilities.
  - Other functions, such as sorting, user monitoring, data compression, etc.

Models, Schemas, Instances and Redundancy

- Data Models
- Database Schema
- Database Instance
- Redundancy

Data Models

- Data Model:
  - A set of concepts to describe the **structure** of a database, the **operations** for manipulating these structures, and certain **constraints** that the database should obey.
- Structure and Constraints:
  - Different constructs are used to define the database structure
  - Constructs typically include **elements** (and their **data types**) as well as **groups** of elements (e.g. **record**, **table**), and **relationships** among such groups
  - **Constraints** specify some restrictions on valid data; these constraints must be enforced at all times
- Operations
  - **Operations** are used for specifying database retrievals and updates by referring to the constructs of the data model.
  - Operations on the data model may include basic model operations (e.g. generic insert, delete, update) and user-defined operations (e.g. compute_student_gpa, update_inventory)

Categories of Data Models

- Conceptual (high-level, semantic) data models:
  - Provide concepts that are close to the way many users perceive data. (Also called entity-based or object-based data models.)
- Physical (low-level, internal) data models:
  - Provide concepts that describe details of how data is stored in the computer. These are usually specified in an ad-hoc manner through DBMS design and administration manuals
- Implementation (representational) data models:
  - Provide concepts that fall between the above two, used by many commercial DBMS implementations (e.g. the relational data model is used in many commercial systems).

Database Schema

- **Database Schema**:
  - The description of a database.
  - Includes descriptions of the database structure, data types, and the constraints on the database.
- **Schema Diagram**:
  - An illustrative display of (most aspects of) a database schema.
- **Schema Construct**:
  - A component of the schema or an object within the schema, e.g., Student, Course.
- The database schema changes very infrequently.
- Schema is also called **intension**.

Database Instance

- **Database Instance**:
  - The actual data stored in a database at a particular moment in time. This includes the collection of all the data in the database.
  - Also called database state (or occurrence or snapshot).
  - The term instance is also applied to individual database components, e.g., record instance, table instance, entity instance
- **Initial Database Instance**: Refers to the database instance that is initially loaded into the system.
- **Valid Database Instance**: An instance that satisfies the structure and constraints of the database.
- The database instance changes every time the database is updated.
- Instance is also called **extension**.

Example of a Database Description

- Mini-world for the example:
  - Part of a UNIVERSITY environment.
- Some mini-world *entities* (an entity is a specific thing in the mini-world):
  - STUDENTs
  - COURSEs
  - SECTIONs (of COURSEs)
  - DEPARTMENTs
  - INSTRUCTORs
- Some mini-world *relationships* (a relationship relates things of the mini-world):
  - SECTIONs are of specific COURSEs
  - STUDENTs take SECTIONs
  - COURSEs have prerequisite
  - COURSE INSTRUCTORs teach SECTIONs
  - COURSEs are offered by DEPARTMENTs
  - STUDENTs major in DEPARTMENTs

Example of a Database Schema

Student

Name	StudNr	Class	Major

Course

CourseName	CourseNr	CreditHours	Department

Prerequisite

CourseNr	PrerequisiteNr

Section

SectionID	CourseNr	Semester	Year	Instructor

GradeReport

StudNr	SectionId	Grade

Example of a Database Instance

course(Course)

CourseName	CourseNr	CreditHours	Department
'Intro to Computer Science'	'CS1310'	4	'CS'
'Data Structures'	'CS3320'	4	CS
'Discrete Mathematics'	'MATH2410'	3	'MATH'
'Databases'	'CS3360'	3	'CS'

section(Section)

SectionID	CourseNr	Semester	Year	Instructor
85	'MATH2410'	'Fall'	04	'King'
92	'CS1310'	'Fall'	04	'Anderson'
102	'CS3320'	'Spring'	05	'Knuth'
112	'MATH2410'	'Fall'	05	'Chang'
119	'CS1310'	'Fall'	05	'Anderson'
135	'CS3380'	'Fall'	05	'Stone'

gradeReport(GradeReport)

StudNr	SectionId	Grade
17	112	'B'
17	119	'C'
8	85	'A'
8	92	'A'
8	102	'B'
8	135	'A'

prerequisite(Prerequisite)

CourseNr	PrerequisiteNr
'CS3380'	'CS3320'
'CS3380'	'MATH2410'
'CS3320'	'CS1310'

Redundancy

- During the design of a database the number of tables and their schemas must be determined.
- A key goal of database design is to avoid redundancy.

- **Redundancy** is present if information is stored multiple times.
- Example of redundancy: storing the same address multiple times
- Redundancy leads to update anomalies and inconsistent data (e.g., a person has multiple and partially invalid addresses)
- The goal of database design, and specifically of database normalization, is to eliminate redundancy.
- The term **controlled redundancy** is used if duplication of information is allowed and if the duplication is controlled by the DBMS.

Review 1.2/1

Consider the university database instance shown above.

1. Explain why this schema contains redundancy.
2. Give an example of a change that leads to update anomalies.
3. Propose a modified schema that eliminates the redundancy.

Review 1.2/2

Main Characteristics of Database Systems

- Three Schema Architecture
- Data Independence
- Main Characteristics
- Advantages and Disadvantages of Database Systems
- History

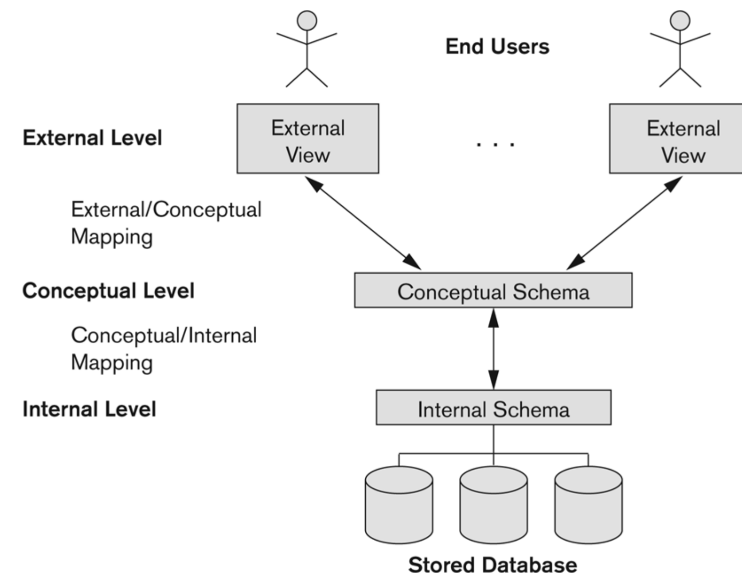
The ANSI/SPARC Three Schema Architecture/1

- Proposed to support DBMS characteristics of:
  - Data independence
  - Multiple views of the data
- Not explicitly used in commercial DBMS products, but has been useful in explaining database system organization.
- Defines DBMS schemas at three levels:
  - **Internal schema** at the internal level to describe physical storage structures and access paths (e.g indexes).
    - Typically uses a physical data model.
  - **Conceptual schema** at the conceptual level to describe the structure and constraints for the whole database for a community of users.
    - Uses a conceptual or an implementation data model.
  - **External schemas** at the external level to describe the various user views.
    - Usually uses the same data model as the conceptual schema.

The ANSI/SPARC Three Schema Architecture/2

- Mappings among schema levels are needed to transform requests and data.
  - Programs refer to an external schema, and are mapped by the DBMS to the internal schema for execution.
  - Data extracted from the internal DBMS level is reformatted to match the user's external view (e.g., formatting the results of an SQL query for display in a Web page)

The ANSI/SPARC Three Schema Architecture/3



Data Independence

- **Logical Data Independence**:
  - The capacity to change the conceptual schema without having to change the external schemas and their associated application programs.
- **Physical Data Independence**:
  - The capacity to change the internal schema without having to change the conceptual schema.
  - For example, the internal schema may be changed when certain file structures are reorganized or new indexes are created to improve database performance
- When a schema at a lower level is changed, only the mappings between this schema and higher-level schemas need to be changed in a DBMS that fully supports data independence.
- The higher-level schemas themselves are unchanged.
  - Hence, the application programs need not be changed since they refer to the external schemas.

Review 1.3

1. Give real world examples of data independence.

Main Characteristics of Database Approach/1

- Insulation between programs and data:
  - Called **data independence**.
  - Allows changing data structures and storage organization without having to change the DBMS access programs.
- Control of **redundancy**:
  - Database systems control (and minimize) redundancy
  - The control allows to avoid inconsistent data (happens if only one copy is updated)
- **Data abstraction**:
  - A data model is used to hide storage details and present the users with a conceptual view of the database.
  - Programs refer to the data model constructs rather than data storage details
- Support of **multiple views** of the data:
  - Each user may see a different view of the database, which describes only the data of interest to that user.

Main Characteristics of Database Approach/2

- **Sharing** of data and **multi-user** transaction processing:
  - Allowing a set of concurrent users to retrieve from and to update the database.
  - Concurrency control within the DBMS guarantees that each transaction is correctly executed or aborted
  - Recovery subsystem ensures each completed transaction has its effect permanently recorded in the database
  - OLTP (Online Transaction Processing) is a major part of database applications. This allows hundreds of concurrent transactions to execute per second.
- **Self-describing** nature of a database system:
  - A DBMS catalog stores the description of a particular database (e.g. data types, data structures, and constraints)
  - The description is called **metadata**.
  - This allows the DBMS software to work with different database applications.

Main Characteristics of Database Approach/3

Example of a DBMS catalog (just the idea; oversimplified):

relations

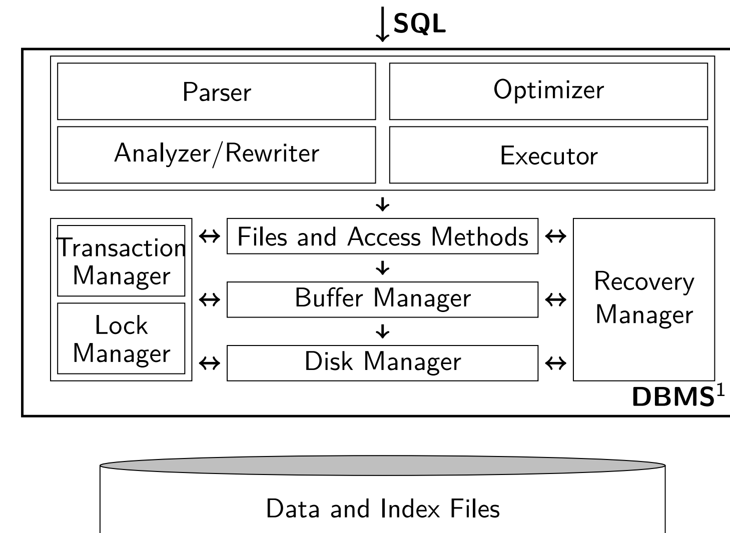
RelationName	NrOfColumns
'Student'	4
'Course'	4
'Section'	5
'GradeReport'	3
'Prerquisite'	2

columns

ColumnName	DataType	BelongsToRelation
'Name'	'CHARACTER(30)'	'Student'
'StudentNr'	'CHARACTER(4)'	'Student'
'Class'	'INTEGER(1)'	'Student'
...	...	...

- PostgreSQL 8.3.9: 74 objects in the system catalog
- Oracle 10.2: 1821 objects in the system catalog

DBMS Architecture



[1] Image: Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill 2003

Advantages of Using a DBMS/1

- Controlling redundancy in data storage.
- Restricting unauthorized access to data.
- Providing persistent storage for program objects.
- Providing storage structures (e.g., indexes) for efficient query processing.
- Providing backup and recovery services.
- Providing multiple interfaces to different classes of users.
- Representing complex relationships among data.
- Enforcing integrity constraints on the database (= **good data quality**).
- Drawing inferences and actions from the stored data using deductive and active rules.

Advantages of Using a DBMS/2

- Potential for enforcing standards:
  - This is very crucial for the success of database applications in large organizations. Standards refer to data item names, display formats, screens, report structures, meta-data (description of data), Web page layouts, etc.
- Reduced application development time:
  - Incremental time to add each new application is reduced.
- Flexibility to change data structures:
  - Database structure may evolve as new requirements are defined.
- Availability of current information:
  - Extremely important for on-line transaction systems such as airline, hotel, car reservations.
- Economies of scale:
  - Wasteful overlap of resources and personnel can be avoided by consolidating data and applications across departments.

When to not use a DBMS

- Main inhibitors of using a DBMS:
  - High initial investment and possible need for additional hardware.
  - Overhead for providing generality, security, concurrency control, recovery, and integrity functions.
- When a DBMS may be unnecessary:
  - If the database and applications are simple, well defined, and not expected to change.
  - If there are stringent real-time requirements that may not be met because of DBMS overhead.
  - If access to data by multiple users is not required.
- When no DBMS may suffice:
  - If the database system is not able to handle the complexity of data because of modeling limitations
  - If the database users need special operations not supported by the DBMS.

History of Database Technology/1

- **Network Model**:
  - The first network DBMS was implemented by Honeywell in 1964-65 (IDS System).
  - Adopted heavily due to the support by CODASYL (Conference on Data Systems Languages) (CODASYL - DBTG report of 1971).
- Advantages:
  - The network model is able to model complex relationships.
  - Can handle most situations for modeling using record types and relationship types.
  - Language is navigational; uses constructs like FIND, FIND member, FIND owner, FIND NEXT within set, GET, etc.
  - Programmers can do optimal navigation through the database.
- Disadvantages:
  - Navigational and procedural nature of processing
  - Database contains a complex array of pointers that thread through a set of records.
  - Little scope for automated query optimization

History of Database Technology/2

- **Hierarchical Data Model**:
  - Initially implemented in a joint effort by IBM and North American Rockwell around 1965. Resulted in the IMS family of systems.
  - IBM's IMS product had (and still has) a very large customer base worldwide
  - Hierarchical model was formalized based on the IMS system
  - Other systems based on this model: System 2k (SAS inc.)
- Advantages:
  - Simple to construct and operate
  - Corresponds to a number of natural hierarchically organized domains, e.g., organization chart
  - Language is simple; Uses constructs like GET, GET UNIQUE, GET NEXT, GET NEXT WITHIN PARENT, etc.
- Disadvantages:
  - Navigational and procedural nature of processing
  - Database is visualized as a linear arrangement of records
  - Little scope for "query optimization"

History of Database Technology/3

- **Relational Model**:
  - Proposed in 1970 by E.F. Codd (IBM)
  - Heavily researched and experimented within IBM Research and universities
  - First commercial system in 1981-82.
  - Today in most commercial products (e.g. DB2, ORACLE, MS SQL Server, SYBASE, INFORMIX).
  - Several free open source implementations, e.g. MySQL, PostgreSQL
  - Currently most dominant for developing database applications.
  - SQL relational standards: SQL-89 (SQL1), SQL-92 (SQL2), SQL-99, SQL3, ...
- Advantages:
  - High level of abstraction (conceptual and physical level are separated)
  - Elegant mathematical model
  - High level (declarative) query languages
- Disadvantages:
  - Performance (was slow at the beginning because there is no navigational access to data)

History of Database Technology/4

- **Object-oriented models**:
  - Object-oriented database management systems (OODBMSs) were introduced in late 1980s and early 1990s to cater to the need of complex data processing in CAD and other applications.
  - OBJECTSTORE, VERSANT, GEMSTONE, O2, ORION, IRIS.
  - Object Database Standard: ODMG-93, ODMG-version 2.0, ODMG-version 3.0.
  - Pure OODBMSs have disappeared. Many relational DBMSs have incorporated object database concepts, leading to a new category called object-relational DBMSs (ORDBMSs).
- **Data on the web and E-commerce applications**:
  - Web contains data in HTML with links among pages.
  - This has given rise to a new set of applications and E-commerce is using standards like XML.
  - Script programming languages such as PHP and JavaScript allow generation of dynamic Web pages that are partially generated from a database.

History of Database Technology/5

- **New functionality** is being added to DBMSs in the following areas:
  - Scientific Applications
  - XML (eXtensible Markup Language)
  - Image Storage and Management
  - Audio and Video Data Management
  - Data Warehousing and Data Mining
  - Spatial Data Management
  - Time Series and Historical Data Management
  - Key-value stores (NoSQL)
- The above gives rise to new research and development in incorporating new data types, complex data structures, new operations and storage and indexing schemes in database systems.

Summary/1

- Data models, schemas, instances
  - **data model** = structures + constraints + operations
  - **schema** = intension; schema consists of structures and constraints; schema changes infrequently
  - **relation instance** = relation = extension; relation instance is the actual data that is compatible with the schema; changes often
- Key characteristics of database systems
  - **controlled redundancy**: database systems is aware of redundancy and provides support for updates that could violate the consistency of the data
  - **data independence**: separation of program and data; makes it possible to, e.g., reorganize internal schema without changing conceptual schema
  - **data abstraction**: high level query language that is independent of storage structure
  - **data dictionary** (metadata) that stores information about the database itself (self-describing)

Summary/2

- Three-Schema Architecture
  - multiple views of the data
  - ANSI/SPARC three schema arcitecture
  - external, conceptual, and internal schema
- DBMS Languages and Interfaces
  - stand-alone command line interfaces: psql, sqlplus, ...
  - programming interfaces: ODBC, JDBC
  - database development tools: pgadmin, SQL developer
- Architectures and History
  - network
  - hierarchical
  - relational
  - object-oriented, object-relational

Database Systems
Spring 2017

The Relational Model
SL02

- The Relational Model
- Basic Relational Algebra Operators
- Additional Relational Algebra Operators
- Extended Relational Algebra Operators
- Modification of the Database
- Relational Calculus

Literature and Acknowledgments

Reading List for SL02:

- Database Systems, Chapters 3 and 6, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Fundamentals of Database Systems, Fourth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

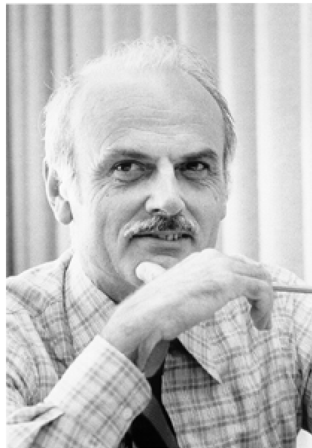
The Relational Model

- schema, attribute, domain, tuple, relation, database
- superkey, candidate key, primary key
- entity constraints, referential integrity

The Relational Model/1

- The **relational model** is based on the concept of a relation.
- A relation is a mathematical concept based on the ideas of sets.
- The relational model was proposed by Codd from IBM Research in the paper:
  - *A Relational Model for Large Shared Data Banks*, Communications of the ACM, June 1970
- The above paper caused a major revolution in the field of database management and earned Codd the coveted ACM Turing Award.
- The strength of the relational approach comes from the formal foundation provided by the theory of relations.
- In practice, there is a standard model based on SQL. There are several important differences between the formal model and the practical model, as we shall see.

The Relational Model/2



- Edgar Codd, a mathematician and IBM Fellow, is best known for creating the relational model for representing data that led to today's 12 billion database industry.
- Codd's basic idea was that relationships between data items should be based on the item's values, and not on separately specified linking or nesting.
- The idea of relying only on value-based relationships was quite a radical concept at that time, and many people were skeptical. They didn't believe that machine-made relational queries would be able to perform as well as hand-tuned programs written by expert human navigators.

http://www.research.ibm.com/resources/news/20030423_edgarpassaway.shtml

Relation Schema

- $R(A_1, A_2, \ldots, A_n)$ is a **relation schema**
- R is the **name** of the relation.
- $A_1, A_2, \ldots, A_n$ are **attributes**
- Example of a relation schema:
  $Customer(CustName, CustStreet, CustCity)$
- $attr(R)$ denotes the set of attributes of relation schema with name R:
  $attr(Customer) = \{CustName, CustStreet, CustCity\}$

Attribute

- Each attribute of a relation has a **name**
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are required to be **atomic**; that is, indivisible
  - The value of an attribute can be an account number, but cannot be a set of account numbers
- The attribute name designates the role played by a domain in a relation:
  - Used to interpret the meaning of the data elements corresponding to that attribute
  - Example: The domain Date may be used to define two attributes named "Invoice-date" and "Payment-date" with different meanings

Domain

- A domain has a logical definition:
  - Example: USA_phone_numbers are the set of 10 digit phone numbers valid in the U.S.
- A domain also has a data-type or a format defined for it.
  - The USA_phone_numbers may have a format: (ddd)ddd-dddd where each d is a decimal digit.
  - Dates have various formats such as year, month, date formatted as yyyy-mm-dd, or as dd mm,yyyy etc.
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
  - We ignore the effect of null values in our main presentation and consider their effect later

Tuple

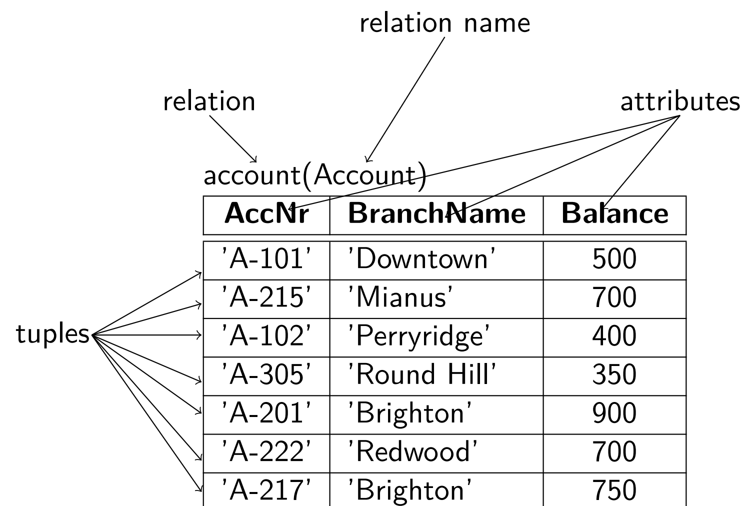
- A tuple is an *ordered set* (= list) of values
- Angle brackets ⟨...⟩ are used as notation; sometimes regular parentheses (...) are used as well
- Each value is derived from an appropriate domain
- A customer tuple is a 3-tuple and would consist of three values, for example:
  - ('Adams', 'Spring', 'Pittsfield')

Relational Instance

- $r(R)$ denotes a *relation* (or relation instance) r on *relation schema* with name R
- Example: $customer(Customer)$
- A relation instance is a subset of the Cartesian product of the domains of its attributes. Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$
- Formally, given sets $D_1, D_2, \ldots, D_n$ a **relation** r is a subset of $D_1 \times D_2 \times \ldots \times D_n$
- Example:

$$\begin{aligned}
D_1 &= CustName = \{\text{'Jones'}, \text{'Smith'}, \text{'Curry'}, \text{'Lindsay'}, \ldots\}\\
D_2 &= CustStreet = \{\text{'Main'}, \text{'North'}, \text{'Park'}, \ldots\}\\
D_3 &= CustCity = \{\text{'Harrison'}, \text{'Rye'}, \text{'Pittsfield'}, \ldots\}\\
r &= \{\ (\text{'Jones'}, \text{'Main'}, \text{'Harrison'}), (\text{'Smith'}, \text{'North'}, \text{'Rye'}),\\
&\qquad (\text{'Curry'}, \text{'North'}, \text{'Rye'}), (\text{'Lindsay'}, \text{'Park'}, \text{'Pittsfield'})\ \}\\
&\subseteq CustName \times CustStreet \times CustCity
\end{aligned}$$

Example of a Relation



AccNr	BranchName	Balance
'A-101'	'Downtown'	500
'A-215'	'Mianus'	700
'A-102'	'Perryridge'	400
'A-305'	'Round Hill'	350
'A-201'	'Brighton'	900
'A-222'	'Redwood'	700
'A-217'	'Brighton'	750

The Customer Relation

customer

CustName	CustStreet	CustCity
'Adams'	'Spring'	'Pittsfield'
'Brooks'	'Senator'	'Brooklyn'
'Curry'	'North'	'Rye'
'Glenn'	'Sad Hill'	'Woodside'
'Green'	'Walnut'	'Stamford'
'Hayes'	'Main'	'Harrison'
'Johnson'	'Alma'	'Palo Alto'
'Jones'	'Main'	'Harrison'
'Lindsay'	'Park'	'Pittsfield'
'Smith'	'North'	'Rye'
'Turner'	'Putnam'	'Stamford'
'Williams'	'Nassau'	'Princeton'

Characteristics of Relations

- Relations are unordered, i.e., the order of tuples is irrelevant (tuples may be stored and retrieved in an arbitrary order)
- The attributes in $R(A_1, ..., A_n)$ and the values in $t = \langle v_1, ..., v_n \rangle$ are ordered.

depositor(Depositor)

CustName	AccNr
'Hayes'	'A-102'
'Johnson'	'A-101'
'Johnson'	'A-201'
'Jones'	'A-217'
'Lindsay'	'A-222'
'Smith'	'A-215'
'Turner'	'A-305'

$depositor = \{$
$('Hayes', 'A\text{-}102'),\ ('Johnson', 'A\text{-}101'),$
$('Johnson', 'A\text{-}201'),\ ('Jones', 'A\text{-}217'),$
$('Lindsay', 'A\text{-}222'),\ ('Smith', 'A\text{-}215'),$
$('Turner', 'A\text{-}305')\ \}$

$sch(depositor) =$
$Depositor(CustName, AccNr)$

- There exist alternative definitions of a relation where attributes in a schema and values in a tuple are not ordered (textbooks differ).

Review 2.1

1. Is $r = \{('Tom', 27, 'ZH'), ('Bob', 33, 'Rome', 'IT')\}$ a relation?

2. For $r = \{(1, 'a'), (2, 'b'), (3, 'c')\}$ and $sch(r) = R(X, Y)$ determine:
   - the 2nd attribute of relation r?
   - the 3rd tuple of relation r?
   - the tuple in r with the smallest value for attribute X?

3. What is the difference between a set and a relation? Illustrate with an example.

Database

- A database consists of multiple relations
- Example: Information about an enterprise is broken up into parts, with each relation storing one part of the information
  - *account*: stores information about accounts
  - *customer*: stores information about customers
  - *depositor*: information about which customer owns which account
- Storing all information as a single relation with schema
  - $Bank(AccNr, Balance, CustName, \dots)$

  results in
  - repetition of information: e.g.,if two customers own the same account
  - the need for null values: e.g., to represent a customer without an account

Summary of the Relational Data Model

- A **domain** D is a set of atomic data values.
  - phone numbers, names, grades, birthdates, departments
  - each domain includes the special value `null`
- With each domain a **data type** or format is specified.
  - 5 digit integers, yyyy-mm-dd, characters
- An **attribute** A_i describes the role of a domain in a relation schema.
  - PhoneNr, Age, DeptName
- A **relation schema** $R(A_1, ..., A_n)$ is made up of a relation name R and a list of attributes.
  - $Employee(Name, Dept, Salary)$, $Department(DName, Manager, Address)$
- A **tuple** t is an ordered list of values $t = (v_1, ..., v_n)$ with $v_i \in dom(A_i)$.
  - $t = ('Tom', 'SE', 23K)$
- A **relation** $r \subseteq D_1 \times ... \times D_n$ over schema $R(A_1, ..., A_n)$ is a set of n-ary tuples.
  - $r = \{('Tom', 'SE', 23K), ('Lene', 'DB', 33K)\} \subseteq Names \times Departments \times Integer$
  - $s = \{('SE', 'Tom', 'Boston'), ('DB', 'Lena', 'Tucson')\}$
- A **database** DB is a set of relations.
  - $DB = \{r, s\}$

Review 2.2

1. Illustrate the following relations graphically:
   $r = \{(1, 'a'), (2, 'b'), (3, 'c')\}$, $sch(r) = R(X, Y)$;
   $s = \{(1, 2, 3)\}, sch(s) = S(A, B, C)$

2. What kind of object is $X = \{\{(3)\}\}$ in the relational model?

3. Are DB1 and DB2 identical databases?
   $DB1 = \{\{(1, 5), (2, 3)\}, \{(4, 4)\}\}$
   $DB2 = \{\{(4, 4)\}, \{(2, 3), (1, 5)\}\}$

Constraints

- Constraints are conditions that must be satisfied by all valid relation instances
- There are four main types of constraints in the relational model:
  - Domain constraints: each value in a tuple must be from the domain of its attribute
  - Key constraints
  - Entity constraints
  - Referential integrity constraints

Key Constraints/1

- Let $K \subseteq attr(R)$
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation r
  - By "possible" we mean a relation r that could exist in the mini-world we are modeling.
  - Example: $\{CustName, CustStreet\}$ and $\{CustName\}$ are both superkeys of *Customer*, if no two customers can possibly have the same name.
  - In real life, an attribute such as *CustID* would be used instead of *CustName* to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.

CuName	CuStreet
'N. Jeff'	'Binzmühlestr'
'N. Jeff'	'Hochstr'

CuName cannot be a key

ID	CuName	CuStreet
1	'N. Jeff'	'Binzmühlestr'
2	'N. Jeff'	'Hochstr'

ID can be a key

Key Constraints/2

- K is a **candidate key** if K is minimal
  Example: $\{CustName\}$ is a candidate key for *Customer*, since it is a superkey and no subset of it is a superkey.
- **Primary key:** a candidate key chosen as the principal means of identifying tuples within a relation
  - Should choose an attribute whose value never, or very rarely, changes.
  - E.g. email address is unique, but may change

Entity Constraints

- The entity constraint requires that the primary key attributes of each relation may not have null values.
- The reason is that primary keys are used to identify the individual tuples.
- If the primary key has several attributes none of these attribute values may be null.
- Other attributes of the relation may also disallow null values although they are not members of the primary key.

ID	Name	CuStreet
1	'N. Jeff'	'Binzmühlestr'
	'T. Hurd'	'Hochstr'

ID cannot be primary key

ID	Name	CuStreet
1	'N. Jeff'	'Binzmühlestr'
2	'T. Hurd'	'Hochstr'

ID can be primary key

Referential Integrity

- A relation schema may have an attribute that corresponds to the primary key of a relation. The attribute is called a **foreign key**.
  - E.g. *CustName* and *AccNr* attributes of *Depositor* are foreign keys to *Customer* and *Account* respectively.
  - Only values occurring in the primary key attribute of the **referenced relation** (or null values) may occur in the foreign key attribute of the **referencing relation**.
- In a graphical representation of the schema a referential integrity constraint is often displayed as a directed arc from the foreign key attribute to the primary key attribute.

ID	CuName	CuStrNr
1	'N. Jeff'	2
2	'N. Jeff'	4

StreetNr	Street
2	'Binzmühlestr'
3	'Hochstr'

StreetNr 4 does not exist. CuStrNr = 4 is an invalid reference.

Review 2.3/1

1. Determine the candidate keys of relation *r*:

r(R)

X	Y	Z
1	2	3
1	4	5
2	2	2

Review 2.3/2

2. Determine possible superkeys of relations *r* and *s*. Assume that the possible superkeys indeed are superkeys: determine possible candidate, primary, and foreign keys.

r(R)

A	B	C
'a'	'd'	'e'
'b'	'd'	'c'
'c'	'e'	'e'

s(S)

D	E
'd'	'a'
'e'	'a'
'a'	'a'

possible superkeys:

possible candidate keys:

possible primary keys:

possible foreign keys:

Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - Procedural: specifies **how** to do it; can be used for query optimization
  - Declarative: specifies **what** to do; not suitable for query optimization
- Pure languages:
  - Relational algebra (procedural)
  - Tuple relational calculus (declarative)
  - Domain relational calculus (declarative)
- Pure languages form underlying basis of query languages that people use (such as SQL).

The Basic Relational Algebra

- select σ
- project π
- union $\cup$
- set difference $-$
- Cartesian product $\times$
- rename ρ

Relational Algebra

- The relational algebra is a procedural language
- The relational algebra consists of six basic operators
  - select: σ
  - project: π
  - union: $\cup$
  - set difference: $-$
  - Cartesian product: $\times$
  - rename: ρ
- The operators take one or two relations as inputs and produce a new relation as a result.
- This property makes the algebra **closed** (i.e., all objects in the relational algebra are relations).

Select Operation

- **Notation**: $\sigma_p(r)$
- p is called the **selection predicate**
- **Definition**: $t \in \sigma_p(r) \Leftrightarrow t \in r \wedge p(t)$
- p is a condition in propositional calculus consisting of **terms** connected by : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
- Example: $\sigma_{BranchName=\text{'Perryridge'}}(account)$
- Example: $\sigma_{A=B \wedge D>5}(r)$

r

A	B	C	D
'α'	'α'	1	7
'α'	'β'	5	7
'β'	'β'	12	3
'β'	'β'	23	10

$\sigma_{A=B \wedge D>5}(r)$

A	B	C	D
'α'	'α'	1	7
'β'	'β'	23	10

Project Operation

- **Notation**: $\pi_{A_1,\ldots,A_k}(r)$
- The result is defined as the relation of k columns obtained by deleting the columns that are not listed
- **Definition**: $t \in \pi_{A_1,\ldots,A_k}(r) \Leftrightarrow \exists x(x \in r \wedge t = x[A_1,\ldots,A_k])$
- There are no duplicate rows in the result since relations are sets
- Example: $\pi_{AccNr,Balance}(account)$
- Example: $\pi_{A,C}(r)$

r

A	B	C
'α'	10	1
'α'	20	1
'β'	30	1
'β'	40	2

$\pi_{A,C}(r)$

A	C
'α'	1
'β'	1
'β'	2

Union Operation

- **Notation**: $r \cup s$
- **Definition**: $t \in (r \cup s) \Leftrightarrow t \in r \vee t \in s$
- For $r \cup s$ to be valid r and s must have the same schema (i.e., attributes).
- Example: $\pi_{CustName}(depositor) \cup \pi_{CustName}(borrower)$
- Example: $r \cup s$

r

A	B
'α'	1
'α'	2
'β'	1

s

A	B
'α'	2
'β'	3

$r \cup s$

A	B
'α'	1
'α'	2
'β'	1
'β'	3

Set Difference Operation

- **Notation**: $r - s$
- **Definition**: $t \in (r - s) \Leftrightarrow t \in r \wedge t \notin s$
- Set differences must be taken between (union) compatible relations.
  - r and s must have the same **arity**
  - attribute domains of r and s must be compatible
- Example: $r - s$

r

A	B
'α'	1
'α'	2
'β'	1

s

A	B
'α'	2
'β'	3

$r - s$

A	B
'α'	1
'β'	1

Cartesian Product Operation

- **Notation**: $r \times s$
- **Definition**: $t \in (r \times s) \Leftrightarrow x \in r \wedge y \in s \wedge t = x \circ y$
- We assume that the attribute names of r and s are disjoint. If the attribute names are not disjoint, then renaming must be used.
- Example: $r \times s$

r

A	B
'α'	1
'β'	2

s

C	D	E
'α'	10	'a'
'β'	10	'a'
'β'	20	'b'
'γ'	10	'b'

$r \times s$

A	B	C	D	E
'α'	1	'α'	10	'a'
'α'	1	'β'	10	'a'
'α'	1	'β'	20	'b'
'α'	1	'γ'	10	'b'
'β'	2	'α'	10	'a'
'β'	2	'β'	10	'a'
'β'	2	'β'	20	'b'
'β'	2	'γ'	10	'b'

Rename Operation

- Allows us to name the results of relational algebra expressions by setting relation and attribute names.
- The rename operator is also used if there are name clashes.
- Various flavors:
  - $\rho_r(E)$ changes the relation name to r.
  - $\rho_{r(A_1,...,A_n)}(E)$ changes the relation name to r and the attribute names to $A_1, ..., A_k$.
  - $\rho_{(A_1,...,A_n)}(E)$ changes attribute names to $A_1, ..., A_k$.
- Example: $\rho_{s(X,Y,U,V)}(r)$

r

A	B	C	D
'α'	'α'	1	7
'β'	'β'	23	10

s

X	Y	U	V
'α'	'α'	1	7
'β'	'β'	23	10

Composition of Operations

- Since the relational algebra is **closed**, i.e., the result of a relational algebra operator is always a relation, it is possible to nest expressions.
- Example: $\sigma_{A=C}(r \times s)$

r

A	B
'α'	1
'β'	2

s

C	D	E
'α'	10	'a'
'β'	10	'a'
'β'	20	'b'
'γ'	10	'b'

$r \times s$

A	B	C	D	E
'α'	1	'α'	10	'a'
'α'	1	'β'	10	'a'
'α'	1	'β'	20	'b'
'α'	1	'γ'	10	'b'
'β'	2	'α'	10	'a'
'β'	2	'β'	10	'a'
'β'	2	'β'	20	'b'
'β'	2	'γ'	10	'b'

$\sigma_{A=C}(r \times s)$

A	B	C	D	E
'α'	1	'α'	10	'a'
'β'	2	'β'	10	'a'
'β'	2	'β'	20	'b'

Review 2.4/1

1. Identify and correct syntactic mistakes in the following relational algebra expressions. The schema of relation r is $R(A, B)$.

$\sigma_{r.A>5}(r)$

$\sigma_{A,B}(r)$

$r \times r$

Review 2.4/2

2. Identify and correct syntactic mistakes in the following relational algebra expressions. Relation *pers* has schema *Pers*(*Name*, *Age*, *City*).

$\sigma_{Name=\text{'Name'}}(pers)$

$\sigma_{City=Zuerich}(pers)$

$\sigma_{Age>\text{'20'}}(pers)$

Banking Example

- *Branch(BranchName, BranchCity, Assets)*
- *Customer(CustName, CustStreet, CustCity)*
- *Account(AccNr, BranchName, Balance)*
- *Loan(LoanNr, BranchName, Amount)*
- *Depositor(CustName, AccNr)*
- *Borrower(CustName, LoanNr)*

Review 2.5/1

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all loans larger than $1200.
- Find the loan number for each loan that is larger than $1200.
- Find the names of all customers who have a loan, an account, or both, from the bank.

Review 2.5/2

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Names of all customers who have a loan at the Perryridge branch.
- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

Review 2.5/3

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Give a different relational algebra expressions that determines the names of all customers who have a loan at the Perryridge branch. Compare it to the solution in Review 2.5/2.

Review 2.5/4

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Determine the largest account balance.

Formal Definition of Relational Algebra Expressions

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation (e.g., $\{(1, 2), (5, 3)\}$)
- Let E_1 and E_2 be relational algebra expressions; the following are all relational algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_p(E_1)$, p is a predicate on attributes in E_1
  - $\pi_s(E_1)$, s is a list consisting of some of the attributes in E_1
  - $\rho_x(E_1)$, x is the new name for the result of E_1

Review 2.6/1

Assume the following schemas:

$Train(TrainNr, StartStat, EndStat)$
$Link(FromStat, ToStat, TrainNr, Departure, Arrival)$

1. Sketch an instance of the database.

Review 2.6/2

2. Determine all direct connections (no change of train) from Zürich to Olten.

Additional Relational Algebra Operators

We define additional operations that **do not add expressive power** to the relational algebra, but that simplify common queries. Thus, these are redundant relational algebra operators.

- Set intersection $\cap$
- Join $\bowtie$
- Division $\div$
- Assignment $\leftarrow$

Set Intersection Operation

- **Notation**: $r \cap s$
- **Definition**: $t \in (r \cap s) \Leftrightarrow t \in r \wedge t \in s$
- Precondition: union compatible
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$
- Example: $r \cap s$

r

A	B
'α'	1
'α'	2
'β'	1

s

A	B
'α'	2
'β'	3

$r \cap s$

A	B
'α'	2

Theta Join

- **Notation**: $r \bowtie_\theta s$
- Let r and s be relations on schemas R and S, respectively. θ is a boolean condition on the attributes of r and s.
- $r \bowtie_\theta s$ is a relation on schema that includes all attributes from schema R and all attributes from schema S.
- Example:
  - $R(A, B, C, D)$ and $S(B, D, E)$
  - $r \bowtie_{B<X \wedge D=Y} \rho_{(X,Y,Z)}(s)$
  - Schema of result is (A, B, C, D, X, Y, Z)
  - Equivalent to: $\sigma_{B<X \wedge D=Y}(r \times \rho_{(X,Y,Z)}(s))$

r

A	B	C	D
'α'	1	'α'	'a'
'β'	2	'γ'	'a'
'γ'	4	'β'	'b'
'α'	1	'γ'	'a'
'δ'	2	'β'	'b'

s

B	D	E
1	'a'	'α'
3	'a'	'β'
1	'a'	'γ'
2	'b'	'δ'
3	'b'	'ϵ'

$\sigma_{B<X \wedge D=Y}(r \times \rho_{(X,Y,Z)}(s))$

A	B	C	D	X	Y	Z
'α'	1	'α'	'a'	3	'a'	'β'
'β'	2	'γ'	'a'	3	'a'	'β'
'α'	1	'γ'	'a'	3	'a'	'β'
δ	2	'β'	'b'	3	'b'	ϵ

Natural Join

- **Notation**: $r \bowtie s$
- Let r and s be relations on schemas R and S, respectively.
- Attributes that occur in r and s must be identical.
- $r \bowtie s$ is a relation on a schema that includes all attributes from schema R and all attributes from schema S that do not occur in schema R.
- Example:
  - $r \bowtie s$ with $R(A, B, C, D)$ and $S(E, B, D)$
  - Schema of result is (A, B, C, D, E)
  - Equivalent to: $\pi_{A,B,C,D,E}(\sigma_{B=Y \wedge D=Z}(r \times \rho_{(E,Y,Z)}(s)))$

r

A	B	C	D
'α'	1	'α'	'a'
'β'	2	'γ'	'a'
'γ'	4	'β'	'b'
'α'	1	'γ'	'a'
'δ'	2	'β'	'b'

s

B	D	E
1	'a'	'α'
3	'a'	'β'
1	'a'	'γ'
2	'b'	'δ'
3	'b'	'ϵ'

$r \bowtie s$

A	B	C	D	E
'α'	1	'α'	'a'	'α'
'α'	1	'α'	'a'	'γ'
'α'	1	'γ'	'a'	'α'
'α'	1	'γ'	'a'	'γ'
'δ'	2	'β'	'b'	'δ'

Division Operation

- **Notation**: $r \div s$
- Suited for queries that include the phrase "for all".
- Let r and s be relations on schemas $R(A_1, \ldots, A_m, B_1, \ldots, B_n)$ and $S(B_1, \ldots, Bn)$, respectively
- The result of $r \div s$ is a relation with attributes $R - S = (A_1, \ldots, A_m)$
- **Definition**: $t \in (r \div s) \Leftrightarrow t \in \pi_{R-S}(r) \land \forall u \in s(t \circ u \in r)$
- $t \circ u$ is the concatenation of tuples t and u
- R-S: all attributes of schema R that are not in schema S
- Example: $R = (A, B, C, D), S = (E, B, D), R - S = (A, C)$

Division Operation - Examples

- Example 1

r

A	B
'α'	1
'α'	2
'α'	3
'β'	1
'γ'	1
'ϵ'	6
'ϵ'	1
'β'	2

s

B
1
2

$r \div s$

A
'α'
'β'

- Example 2

r

A	B	C	D	E
'α'	'a'	'α'	'a'	1
'α'	'a'	'γ'	'a'	1
'α'	'a'	'γ'	'b'	1
'β'	'a'	'γ'	'a'	1
'β'	'a'	'γ'	'b'	3
'γ'	'a'	'γ'	'a'	1
'γ'	'a'	'γ'	'b'	1
'γ'	'a'	'β'	'b'	1

s

D	E
'a'	1
'b'	1

$r \div s$

A	B	C
'α'	'a'	'γ'
'γ'	'a'	'γ'

Properties of the Division Operation

- Property
  - Let $q = r \div s$
  - Then q is the largest relation satisfying $q \times s \subseteq r$
- Definition in terms of the basic algebra operation
  Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

  $$r \div s = \pi_{R-S}(r) - \pi_{R-S}((\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r))$$

  To see why
  - R-S,S: all attributes of R that are not in S, followed by all attributes of S
  - Thus, $\pi_{R-S,S}(r)$ reorders attributes of r
  - $\pi_{R-S}(\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r))$ gives those tuples t in $\pi_{R-S}(r)$ such that for some tuple $u \in s, t \circ u \notin r$.

Assignment Operation

- The assignment operation ($\leftarrow$) provides a convenient way to express complex queries by breaking them up into smaller pieces.
  - Write query as a sequential program consisting of
    - a series of assignments
    - followed by an expression whose value is displayed as a result of the query.
  - Assignment must always be made to a temporary relation variable.
- Example: Write $r \div s$ as

  $$temp1 \leftarrow \pi_{R-S}(r)$$
  $$temp2 \leftarrow \pi_{R-S}((temp1 \times s) - \pi_{R-S,S}(r))$$
  $$result \leftarrow temp1 - temp2$$

  - The result to the right of the $\leftarrow$ is assigned to the relation variable on the left of the $\leftarrow$.

Bank Example Queries/1

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all customers who have an account and a loan.

  $\pi_{CustName}(borrower) \cap \pi_{CustName}(depositor)$

- Find the name of all customers who have a loan at the bank and the loan amount

  $\pi_{CustName,Amount}(borrower \bowtie loan)$

Bank Example Queries/2

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all customers who have an account from at least the "Downtown" and the "Uptown" branches.
  - Solution 1

    $\pi_{CustName}(\sigma_{BranchName='\text{Downtown}'}(depositor \bowtie account)) \cap$

    $\pi_{CustName}(\sigma_{BranchName='\text{Uptown}'}(depositor \bowtie account))$
  - Solution 2

    $r \leftarrow \pi_{CustName,BranchName}(depositor \bowtie account))$

    $s \leftarrow \pi_{BranchName}(\sigma_{BranchName='Downtown' \vee BranchName='Uptown'}(account))$

    $Res \leftarrow r \div s$

Review 2.7

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all customers who have an account at all branches located in Brooklyn city.

Extended Relational Algebra Operators

Extended relational algebra operators add expressive power to the basic relational algebra.

- Generalized Projection π
- Aggregate Functions ϑ
- Outer Join ⟕, ⟖, ⟗

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list: $\pi_{F_1,F_2,\ldots,F_n}(E)$
- E is a relational algebra expression
- Each of $F_1, F_2, \ldots, F_n$ are arithmetic expressions involving constants and attributes in the schema of E.
- Example: Given relation $credit_info(CustName, Limit, CredBal)$, find how much more each person can spend:

$\pi_{CustName,Limit-CreditBal}(credit_info)$

Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.
  - **avg** average value
  - **min** minimum value
  - **max** maximum value
  - **sum** sum of values
  - **count** number of values
- **Aggregation operation** in relational algebra

$$_{G_1,G_2,\ldots,G_n}\vartheta_{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}(E)$$

E is any relational algebra expression
  - $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

Aggregate Operation - Example

- Relation r, $res \leftarrow \rho_{Res(SumC)}(\vartheta_{sum(C)}(r))$

r

A	B	C
$'\alpha'$	$'\alpha'$	7
$'\alpha'$	$'\beta'$	7
$'\beta'$	$'\beta'$	3
$'\beta'$	$'\beta'$	10

res

sumC
27

- Balance per branch:
$res \leftarrow \rho_{Res(BName,SumBal)}(_{BranchName}\vartheta_{sum(Balance)}(account))$

account

BranchName	AccNr	Balance
'Perryridge'	'A-102'	400
'Perryridge'	'A-201'	900
'Brighton'	'A-217'	750
'Brighton'	'A-215'	750
'Redwood'	'A-222'	700

res

BName	SumBal
'Perryridge'	1300
'Brighton'	1500
'Redwood'	700

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.
    - We shall study precise meaning of comparisons with nulls later

Outer Join Example/1

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Join**

$loan \bowtie borrower$

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'

Outer Join Example/2

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Left Outer Join** (preserves tuples from left)

$loan ⟕ borrower$

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-260'	'Perryridge'	1700	*null*

Outer Join Example/3

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Right Outer Join** (preserves tuples from right)

$loan ⟖ borrower$

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-155'	*null*	*null*	'Hayes'

Outer Join Example/4

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Full Outer Join** (preserves all tuples)

$loan ⟗ borrower$

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-260'	'Perryridge'	1700	*null*
'L-155'	*null*	*null*	'Hayes'

Modification of the Database

- The content of the database may be modified using the following operations:
  - Deletion
  - Insertion
  - Updating
- All these operations are expressed using the assignment operator.

Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only entire tuples; cannot delete values of particular attributes only.
- A deletion is expressed in relational algebra by:

  $r \leftarrow r - E$

  where r is a relation and E is a relational algebra query.

Deletion Examples

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Delete all account records in the Perryridge branch.

  $accout \leftarrow account - \sigma_{BranchName=\text{'Perryridge'}}(account)$

- Delete all loan records with amount in the range of 10 to 50

  $loan \leftarrow loan - \sigma_{Amount \geq 10 \wedge Amount \leq 50}(loan)$

- Delete all accounts at branches located in Needham.

  $r_1 \leftarrow \sigma_{BranchCity=\text{'Needham'}}(accout \bowtie branch)$

  $r_2 \leftarrow \pi_{AccNr,BranchName,Balance}(r_1)$

  $r_3 \leftarrow \pi_{CustName,AccNr}(r_2 \bowtie depositor)$

  $account \leftarrow account - r_2$

  $depositor \leftarrow depositor - r_3$

Insertion

- To insert data into a relation, we either:
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- In relational algebra, an insertion is expressed by:

  $r \leftarrow r \cup E$

  where r is a relation and E is a relational algebra expression.
- The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.

Insertion Examples

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Insert information into the database specifying that Smith has $1200 in account A-973 at the Perryridge branch.

  $account \leftarrow account \cup \{(\text{'A-973'}, \text{'}Perryridge\text{'}, 1200)\}$

  $depositor \leftarrow depositor \cup \{(\text{'}Smith\text{'}, \text{'A-973'})\}$

- Provide as a gift for all loan customers in the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

  $r_1 \leftarrow \sigma_{BranchName=\text{'Perryridge'}}(borrower \bowtie loan)$

  $account \leftarrow account \cup \pi_{LoanNr,BranchName,200}(r_1)$

  $depositor \leftarrow depositor \cup \pi_{CustName,LoanNr}(r_1)$

Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple; logically this can be expressed by an insertion and deletion; in actual systems updating is much faster than inserting and deleting.
- In relational algebra this can be expressed by replacing r by the result computed by the relational algebra expression E; often the expression is the generalized projection.

  $r \leftarrow E$

  $r \leftarrow \pi_{F_1,F_2,\dots,F_i,\dots}(r)$

- Each F_i is either
  - the ith attribute of r, if the ith attribute is not updated, or,
  - if the attribute is to be updated F_i is an expression, which defines the new value for the attribute

Update Examples

- Make interest payments by increasing all balances by 5%.

  $account \leftarrow \pi_{AccNr,BranchName,Balance*1.05}(account)$

- Pay all accounts with balances over $10,000 6% interest and pay all others 5%.

  $account \leftarrow$

  $\pi_{AccNr,BranchName,Balance*1.06}(\sigma_{Balance>100000}(account))$

  $\cup$

  $\pi_{AccNr,BranchName,Balance*1.05}(\sigma_{Balance\leq 100000}(account))$

Relational Calculus

- First Order Predicate Logic
- Domain Relational Calculus

Relational Calculus

- A **relational calculus** expression creates a new relation, which is specified in terms of variables that range over:
  - tuples of relations (in **tuple relational calculus (TRC)**)
  - attributes of relations (in **domain relational calculus (DRC)**).
- We only consider DRC.
- In a relational calculus expression, there is *no order of operations* to specify how to compute the query result.
- A calculus expression specifies only what information the result should contain; hence relational calculus is a **non-procedural** or **declarative** language.
- Relational calculus is closely related to and a subset of first order predicate logic.

First Order Predicate Logic

Syntax:

- **logical symbols**: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$, ...
- **constant**: string, number, ...; 'abc', 14, ...
- **identifier**: character sequence starting with a letter
- **variable**: identifier starting with capital letter; X, Y, ...
- **predicate symbol**: identifier starting with lower case letter
- **build-in predicate symbol**: $=, <, >, \leq, \geq, \neq$, ...
- **term**: constant, variable
- **atom**: predicate, built-in predicate; $p(t_1, ..., t_n)$, $t_1 < t_2$, ... with terms $t_1, ..., t_n$; predicate symbol p
- **formula**: atom, $A \wedge B$, $A \vee B$, $\neg A$, $A \Rightarrow B$, $\exists XA$, $\forall XA$, (A), ... with formulas A, B; variable X

Review 2.8

Decide which of the following formulas are syntactically correct first order predicate logic formulas.

- $less_than(99, 27)$
- $loves(mother('hans'), france \vee italy)$
- $\forall X(danish(X) \Rightarrow danish('bill_clinton'))$
- $\forall P(P('hans'))$
- $\forall C(neighbour('england', C))$
- $\exists C(neighbour('italien', C))$
- $\forall P(smart(P) \wedge \neg alive(P) \Rightarrow famous(P))$

Selected Properties and Terminology

Terminology

- A variable is **free** if it is not quantified
- A variable is **bound** if it is quantified
- Example: $\forall X(p(X, Y, Y)) \wedge q(X, Y)$

FOPL Equivalences

- $\forall X(A) = \neg\exists X(\neg A)$
- $A \Rightarrow B = \neg A \vee B$
- $A \wedge \exists X(B) = \exists X(A \wedge B)$ if X is not free in A
- $A \wedge \forall X(B) = \forall X(A \wedge B)$ if X is not free in A
- $A \wedge \neg\exists X(B) = A \wedge \neg\exists X(A \wedge B)$ if X is not free in A

Set theory

- $A - B = A - (A \cap B)$

Domain Independence

- Relational calculus only permits expressions that are *domain independent*, i.e., expressions that permit sensible answers (e.g., no infinite results).
- Domain independence is not decidable. There exist various syntactic criteria that ensure domain independence, e.g., safe expressions, range restricted expressions, etc.
- Examples:
  - $emp(X)$ is domain independent
  - $\neg emp(X)$ is not domain independent
  - $stud(X) \land \neg emp(X)$ is domain independent
  - $X > 6$ is not domain independent

Review 2.9/1

Use first order predicate calculus expressions to express the following natural language statements:

- Anyone who is dedicated can learn databases.
- No man is independent.
- Dogs that bark do not bite.

Review 2.9/2

- Not all men can walk.
- Every person owns a computer.
- Lars likes everyone who does not like himself.

Domain Relational Calculus/1

Syntax:

- **logical symbols**: $\land$, $\lor$, $\neg$, $\Rightarrow$, $\exists$, $\forall$, ...
- **constant**: string, number, ...; 'abc', 14, ...
- **identifier**: character sequence starting with a letter
- **variable**: identifier starting with capital letter; X, Y, ...
- **predicate symbol**: identifier starting with lower case letter
- **build-in predicate symbol**: $=, <, >, \leq, \geq, \neq$, ...
- **term**: constant, variable
- **atom**: predicate, built-in predicate; $p(X, ..., 22)$, $X < 5000$, ...
- **formula**: atom, $A \land B$, $A \lor B$, $\neg A$, $A \Rightarrow B$, $\exists X(A)$, $\forall X(A)$, (A), ...
- A domain relational calculus query is of the form

  $\{ X_1, ..., X_n \mid formula \}$

Domain Relational Calculus/2

- The domain relational calculus is based on specifying a number of variables that range over single values from domains of attributes.
- In the domain calculus the position of attributes is relevant. Attribute names are not used.
- Often the anonymous variable _ is used to shorten notation: $r(_, _, X, _) = \exists U, V, W(r(U, V, X, W))$
- Example: To determine first and last names of all employees whose salary is above $50,000, we write the following DRC expression:
  $\{ FN, LN \mid \exists Sal(emp(FN, _, LN, _, Sal) \land Sal > 50000) \}$

Review 2.10/1

branch(BranchName, BranchCity, Assets)
customer(CustName, CustStreet, CustCity)
account(AccNr, BranchName, Balance)
loan(LoanNr, BranchName, Amount)
depositor(CustName, AccNr)
borrower(CustName, LoanNr)

- Find all loans of over $1200.
- Find the loan number for each loan of an amount greater than $1200.
- Find the names of all customers who have a loan, an account, or both, from the bank.

Review 2.10/2

branch(BranchName, BranchCity, Assets)
customer(CustName, CustStreet, CustCity)
account(AccNr, BranchName, Balance)
loan(LoanNr, BranchName, Amount)
depositor(CustName, AccNr)
borrower(CustName, LoanNr)

- Find the names of all customers who have a loan at the Perryridge branch.
- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

Review 2.10/3

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Determine the largest account balance.

Review 2.11

- Consider the following DRC expressions. Formulate equivalent relational algebra expressions. Assume column i of relation r has name rci.
  - $\{X, Y \mid p(X) \wedge q(X, Y)\}$
  - $\{X \mid p(X, 2) \wedge X > 7\}$
  - $\{X \mid p(X) \wedge \neg\exists Y(q(X, Y))\}$
  - $\{X \mid p(X) \wedge \neg\exists Y(p(Y) \wedge Y > X)\}$

Summary/1

- The Relational Model
  - attribute, domain, tuple, relation, database, schema
- Basic Relational Algebra Operators
  - Selection σ
  - Projection π
  - Union $\cup$
  - Difference $-$
  - Cartesian product $\times$
  - Rename ρ
- Additional Relational Algebra Operators
  - Join (theta, natural) $\bowtie$
  - Division $\div$
  - Assignment $\leftarrow$

Summary/2

- Extended Relational Algebra Operators
  - Generalized projection π
  - Aggregate function ϑ
  - Outer joins ⟕, ⟖, ⟗
- Modification of the database
  - insert, delete, update
- Relational Calculus
  - domain relational calculus
- Know syntax of RA and DRC expressions.
- Be able to translate natural language queries into RA and DRC expressions.
- Be able to freely move between RA and DRC.

Database Systems
Spring 2017

SQL
SL03

- Data Definition Language
- Expressions and Predicates
- Table Expressions, Query Specifications, Query Expressions
- Subqueries, Duplicates, Null Values
- Modification of the Database

Literature and Acknowledgments

Reading List for SL03:

- Database Systems, Chapters 4 and 5 (sections 5.1 and 5.4), Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Fundamentals of Database Systems, Fourth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

History/1

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92 (also called SQL2)
    - entry level: roughly corresponds to SQL-89
    - intermediate level: half of the new features of SQL-92
    - full level
  - SQL:2003 (also called SQL3)
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may (will!) work on your particular system.

History/2

- Don Chamberlin holds a Ph.D. from Stanford University.
- He worked at Almaden Research Center doing research on database languages and systems.
- He was a member of the System R research team that developed much of today's relational database technology
- He designed the original SQL database language.
- In 2005 Don Chamberlin received an honory doctoral degree from the University of Zürich for his work on SQL.

http://www.almaden.ibm.com/cs/people/chamberlin/

Model and Terminology

- SQL is based on multisets (or bags) rather than sets. In a multiset an element may occur multiple times.
- We write $\{...\}$ for a set and $\{\{...\}\}$ for a bag.
- SQL uses the terms **table**, **column**, and **row**.
- SQL does not distinguish between upper and lower case in identifiers and keywords.
- Comparison of terminology:

SQL	Relational Algebra	Domain Relational Calculus
table	relation	predicate
column	attribute	argument
row	tuple	-
query	RA expression	formula

Review 3.1

- List reasons for not automatically removing duplicates.

Data Definition Language

- Domain Types
- Create, dropping and altering tables
- Integrity constraints

Data Definition Language

Allows the specification of not only a set of tables but also information about each table, including:

- Conceptual schema:
  - The **schema** for each table.
  - The **domain** associated with each column.
  - **Integrity constraints** that all valid instances must satisfy.
  - **Security** and **authorization** information for each table.
- Physical schema:
  - The **physical storage structure** of each table on disk.
  - The set of **indices** to be maintained for each table.

Domain Types in SQL

- `CHAR(n)` Fixed length character string, with user-defined length *n*.
- `VARCHAR(n)` Variable length character strings, with user-specified maximum length *n*.
- `INTEGER` Integer (a finite subset of the integers that is machine-dependent).
- `SMALLINT` Small integer (a machine-dependent subset of the integer domain type).
- `NUMERIC(p,d)` Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- `REAL, DOUBLE PRECISION` Floating point and double-precision floating point numbers, with machine-dependent precision.
- `FLOAT(n)` Floating point number, with user-specified precision of at least *n* digits.

Create Table Construct

- An SQL table is defined using the **create table** command:

  create table $r($
  $A_1 D_1, A_2 D_2, \ldots, A_n D_n,$
  (integrity-constraint$_1$),
  $\ldots,$
  (integrity-constraint$_k$))

  - r is the name of the table
  - each A_i is an column name in the schema of table
  - D_i is the data type of values in the domain of column A_i

- Example:

```
CREATE TABLE branch (
  BranchName CHAR(15) NOT NULL,
  BranchCity CHAR(30),
  Assets INTEGER )
```


Integrity Constraints in Create Table/1

- **not null**
- **primary key** $(A_1, \ldots, A_n)$
- **foreign key** $(A_1, \ldots, A_n)$ **references** $T(B_1, \ldots, B_n)$

Example: Declare *BranchName* as the primary key for *branch*

```
CREATE TABLE branch (
  BranchName CHAR(15),
  BranchCity CHAR(30),
  Assets INTEGER,
  PRIMARY KEY (BranchName) )
```

In SQL a **primary key** declaration on a column automatically ensures **not null**.

Integrity Constraints in Create Table/2

Example: Define *AccNr* as foreign key in table *depositor*:

```
CREATE TABLE depositor (
  CustName VARCHAR(15),
  AccNr INTEGER,
  FOREIGN KEY (AccNr) REFERENCES account(AccNr))
```

Notation/1

- SQL is a comprehensive language and offers multiple syntactic constructs to define table and integrity constraints.
- The exact syntax depends on the database system and even the version.
- We use a small core of SQL that is general and is mostly independent of the database system and the version.
- Selected peculiarities of the syntax, especially when relevant for the exercises, will be pointed out.
- In case of syntax problems look up the exact syntax (manual, web, forum, etc).

Notation/2

- In SQL small and capital letters are irrelevant (e.g., SELECT, select, seLEct).
- In some systems the capitalization of identifiers is relevant (e.g., table names in MySQL on Linux).
- Small and capital letters are relevant if identifiers are put in quotes ('CustName').
- In Oracle and PostgreSQL single quotes must be used ('abcde').
- In MySQL single ('abcde') and double ("abcde") quotes are permitted.

Notation/3

- On the course slides we use bold face for reserved words (e.g., **select**).
- On the slides code examples are typeset in blue. Keywords are capitalized and bold face.
- In program code reserved words are usually capitalized (e.g., SELECT).
- The end of an SQL statement is often marked by a semicolon:

```
SELECT * FROM account;
```


Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped table from the database.
- The **alter table** command is used to add columns to an existing table:

  alter table *r* **add** *A D*

  where *A* is the name of the column to be added to table *r* and *D* is the domain of *A*.
  - All tuples in the table are assigned *null* as the value for the new column.
- The **alter table** command can also be used to drop columns of a table:

  alter table *r* **drop** *A*

  where *A* is the name of a column of table *r*

Expressions and Predicates

- SQL Expressions
- SQL Predicates

Expressions and Predicates

- Powerful expressions and predicates make computer languages useful and convenient.
- Database vendors compete on the set of expressions and predicates they offer (functionality as well as speed).
- In databases the efficient evaluation of predicates is a major concern.
- Consider a table with 1 billion tuples and the following predicates:
  - `LastName = 'Miller'`
  - `LastName LIKE '%mann'`
- That is one of the reasons why additions of (user defined) functions and predicates to DBMS has been limited.
- We show some SQL expressions and predicates to give a rough idea of common database functionality.

Expressions/1

- SQL provides a number of expressions to manipulate numbers, strings, dates, etc.
- In early versions of SQL virtually all operations were on single values or single columns.
- As of SQL-92 row value constructors (tuples) can be used:
  - `(C1,C2,C3)=(23,234,'a')`
- Thus, in general the result of an expression is a tuple.
- Expressions are built up of
  - columns and constants
  - functions on built-in data types
  - type conversions (**cast**)
  - **case**, **coalesce**, **nullif**, ...
  - aggregates (**min**, **count**, **avg**, ...)

Expressions/2

- Numeric functions: +, - *, /, **abs**(e), **ceil**(e), **tan**(e), **log**(e), **round**(e), **sign**(e), **mod**(e,f), ...
- Character functions:
  - concatenation: 'abc' || 'de' = 'abcde'
  - **position**('48' **in** 'another 48 hours') = 9
  - **substring**('anothe' **from** 1 **for** 3) = 'ano'
  - **upper**(e)
  - **trim**(**leading** ' ' **from** ' test') = 'test'
- date and time functions:
  - **current_date**, **current_time**
  - **current_timestamp**
  - **current_date** + **interval** '1' **day**
  - **current_date** - **date** '1990/3/18'
  - **interval** '6' **day** - **interval** '1' **day** = **interval** '5' **day**
- type conversions:
  - **cast**('48' **as integer**) = 48
  - often "natural" type conversions are done implicitly

Expressions/3

- conditional statement:
  case
  when cond1 **then** result1
  ...
  when condN **then** resultN
  else resultX
  end
- **coalesce**(val1, ..., valN)
  returns the first value that is not null (and null if all values are equal to null)
- **nullif**(e1,e2)
  returns null if e1 and e2 are identical; useful if missing information is not represented with null; **nullif**(cost, 9999)

Predicates

- Predicates evaluate to true, false or unknown
- boolean connectives: **and**, **or**, **not**
- $=, <>, <, >, <=, >=$
- e [**not**] **between** e1 **and** e2
- e **is** [**not**] **null**
- e **in** (e1, ..., eN)
- e1 **like** e2
  example: Name like '_ross%'
  wildcards: % 0-n characters, _ 1 character

Review 3.2

- Rewrite
  1. `(X,Y,Z)=(1,2,3)`
  2. `(X,Y,Z)<(1,2,3)`

  to equivalent expressions without row value constructors.
- Identify the problem with the predicate

  `X>0 AND 1/X>0.1`

  and propose a solution.

Query Expressions

- Table expressions
- Query specifications
- Query Expressions

Structure of SQL Queries/1

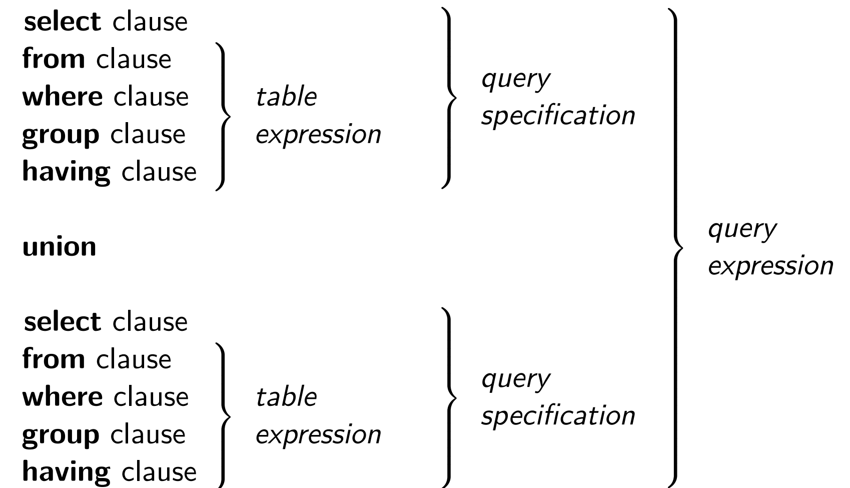
- The main building block of SQL are "queries" (will be defined more precisely below).
- SQL is used very heavily in the real world.
- SQL is much more than a simple select-from-where, such as

```
SELECT *
FROM loan
WHERE LName = 'Bohr'
```

- Many people
  - Underestimate SQL
  - Do not properly understand the concepts of SQL
  - Do not understand how to work with a declarative language and sets (this requires some practice and getting used to)

Structure of SQL Queries/2

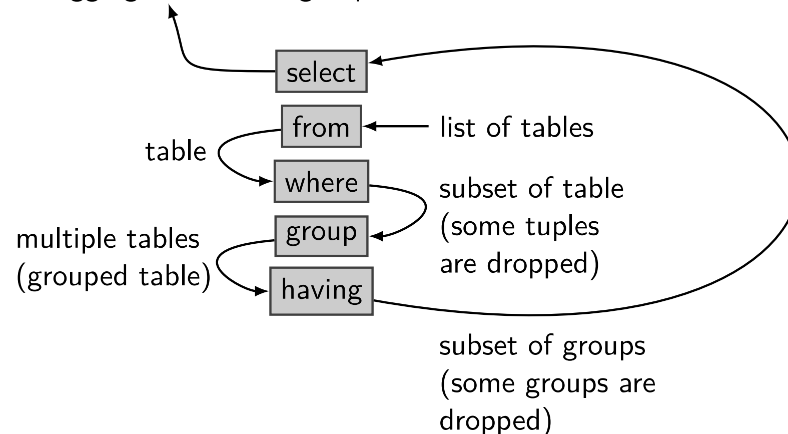
- A typical SQL query has the form:



- The result of an SQL query is a (virtual) table.

Evaluation of Query Specifications

Conceptually, an SQL query specification is processed as follows.



Evaluation of SQL Queries

1. Form the cross product of the tables in the **from clause**
2. Eliminate tuples that do not satisfy the **where clause**
3. Group the remaining tuples according to the **group clause**
4. Eliminate groups that do not satisfy the **having clause**
5. Evaluate the expressions in the **select clause**
6. One result tuple is returned for each group
7. Eliminate duplicates if distinct is specified
8. Compute query specifications independently and apply **set operations** (union, excpet, intersect)
9. Sort all result tuples according to **order clause**

Illustration of SQL Query Evaluation/1

1. FROM: form cross product of all tables in **from** clause

2. WHERE: eliminates tuples that do not satisfy the condition in the **where** clause

3. GROUP BY: groups table according to the columns in the **group** clause

Illustration of SQL Query Evaluation/2

4. HAVING: eliminates groups that do not satisfy the condition of the **having** clause

5. SELECT: evaluates the expressions in the **select** clause and produces a result tuple for each group

Notation/4

- The SQL allows **renaming tables and columns** using the **as** clause:
  old-name **as** *new-name*
- SQL allows **qualified column names** (the relation name is put before the column name):
  relationName.colName
- Keyword **as** is optional and may be omitted
  borrower **as** *t* $\equiv$ *borrower t*
- Example:

```
SELECT borrower.LoanNum AS LoanID, Amount
FROM borrower, loan AS l
WHERE borrower.LoanNum = l.LoanNum
```


The from Clause/1

- The **from** clause lists the tables involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Cartesian product *borrower* $\times$ *loan*

```
FROM borrower, loan
```

- Customer names and loan numbers (all combinations):

```
FROM borrower AS t, loan AS s
```

- Renaming can become necessary if the same table is used multiple times in the from clause.
- In terms of expressiveness nothing more is needed. For convenience, SQL offers many join variants.

The from Clause/2

- SQL supports many different join:
  - `FROM t1 CROSS JOIN t2`
    - Cartesian product
  - `FROM t1 JOIN t2 ON t1.a < t2.b`
    - Theta join
  - `FROM t1 LEFT OUTER JOIN t2 ON t1.a = t2.b`
    - left outer join
  - `FROM t1 NATURAL INNER JOIN t2`
    - Natural join
  - `FROM t1 NATURAL INNER JOIN t2 USING (name)`
    - Limited natural join (not all pair-wise equal columns are used for the natural join; only the ones specified in the using clause)

The from Clause/3

- Table *loan*

LoanNum	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

- Table *borrower*

CustName	LoanNum
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- Note: borrower information missing for L-260 and loan information missing for L-155

The from Clause/4

- `FROM loan INNER JOIN borrower ON loan.LoanNum = borrower.LoanNum`

LoanNum	BranchName	Amount	CustName	LoanNum
'L-170'	'Downtown'	3000	'Jones'	'L-170'
'L-230'	'Redwood'	4000	'Smith'	'L-230'

- `FROM loan LEFT OUTER JOIN borrower ON loan.LoanNum = borrower.LoanNum`

LoanNum	BranchName	Amount	CustName	LoanNum
'L-170'	'Downtown'	3000	'Jones'	'L-170'
'L-230'	'Redwood'	4000	'Smith'	'L-230'
'L-260'	'Perryridge'	1700	*null*	*null*

The from Clause/5

- `FROM loan NATURAL INNER JOIN borrower`

LoanNum	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'

- `FROM loan NATURAL RIGHT OUTER JOIN borrower`

LoanNum	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-155'	*null*	*null*	'Hayes'

The where Clause/1

- The **where** clause specifies conditions that the result tuples must satisfy.
- The where clause takes the virtual table produced by the from clause and filters out those rows that do not satisfy the condition.
- Example: loan number of loans from the Perryridge branch with loan amounts greater than $1200.

```
FROM loan
WHERE BrancheName = 'Perryridge'
AND Amount > 1200
```

LoanNum	BranchName	Amount
'L-260'	'Perryridge'	1700

- The where clause corresponds to the selection predicate.

The where Clause/2

- In SQL predicates can be combined using the logical connectives **and**, **or**, and **not**.
- The where clause can be used to specify join and selection conditions.
- SQL includes a **between** comparison operator
- Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, $\geq$ $90,000 and $\leq$ $100,000)

```
SELECT LoanNum
FROM loan
WHERE Amount BETWEEN 90000 AND 100000
```


Review 3.3

Translate the following RA expressions into equivalent SQL fragments.

1. $R \times S$
2. $(R \times S) \times T$
3. $\sigma_{A>5}(R)$
4. $\sigma_{A>5}(\sigma_{B=4}(R))$
5. $\sigma_{A=X}(R \times S)$
6. $\sigma_{A>5}(R) \times \sigma_{X=7}(S)$

The group Clause/1

- The **group** clause takes the table produced by the where clause and returns a grouped table.
- Example: Accounts grouped by branch.

```
FROM account
GROUP BY BranchName
```

account

AccNr	BranchName	Balance
'A-101'	'Downtown'	500
'A-215'	'Perryridge'	700
'A-102'	'Perryridge'	400
'A-305'	'Perryridge'	350
'A-222'	'Perryridge'	700
'A-201'	'Brighton'	900
'A-217'	'Brighton'	750

The group Clause/2

- The group clause groups multiple tuples together. Conceptually, grouping yields multiple tables.
- The following statements work on the groups (and not on individual tuples).
- For each group **one or zero** result tuples are returned.
- All queries that are not guaranteed to produce at most one result tuple per group are rejected (compile time error)

The having Clause/1

- The having clause takes a grouped table as input and returns a grouped table.
- The having condition is applied to each group.
- Only groups that satisfy the condition are returned.
- The having clause never returns individual tuples of a group.
- The having condition may include grouping columns or aggregated columns.

The having Clause/2

- Consider branches with more than one account only:

```
FROM account
GROUP BY BranchName
HAVING COUNT(AccNr) > 1
```

- This having clause returns all groups with more than one tuple:

account

AccNr	BranchName	Balance
'A-215'	'Perryridge'	700
'A-102'	'Perryridge'	400
'A-305'	'Perryridge'	350
'A-222'	'Perryridge'	700
'A-201'	'Brighton'	900
'A-217'	'Brighton'	750

Review 3.4

- Consider the table expression

  ```
  FROM r1, r2 WHERE x > y AND y > z
  ```

  Which is a join condition and which is a selection condition?

- Give an example where an expression in the group clause (rather than just column names) makes sense.

Review 3.5

account

AccNr	BranchName	Balance
'A-101'	'Downtown'	500
'A-215'	'Perryridge'	700
'A-102'	'Perryridge'	400
'A-305'	'Perryridge'	350
'A-222'	'Perryridge'	700
'A-201'	'Brighton'	900
'A-217'	'Brighton'	750

- Determine which of the following table expressions are correct.

```
FROM account
GROUP BY BranchName
HAVING Balance < 730
```

```
FROM account
GROUP BY BranchName
HAVING BranchName = 'Brighton'
OR BranchName = 'Downtown'
```

```
FROM account
GROUP BY BranchName
HAVING SUM(Balance) < 1000
```


The select Clause/1

- The **select** clause lists the columns that shall be in the result of a query.
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* table:

```
SELECT BranchName
FROM loan
```

- In the relational algebra, the query would be:

$$\pi_{BranchName}(loan)$$

The select Clause/2

- SQL allows duplicates in tables as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the names of all branches in the loan table, and remove duplicates:

```
SELECT DISTINCT BranchName
FROM loan
```

- A * in the select clause denotes "all columns":

```
SELECT *
FROM loan
```


The select Clause/3

- In the select clause aggregate functions can be used.
  - **avg**: average value
  - **min**: minimum value
  - **max**: maximum value
  - **sum**: sum of values
  - **count**: number of values
- The aggregate functions operate on multiple rows. They process all rows of a group and compute an aggregated value for that group.
- Note: Columns in **select** clause outside of aggregate functions must appear in **group by** list

The select Clause/4

- Find the average account balance at the Perryridge branch.

```
SELECT AVG(Balance)
FROM account
WHERE BranchName = 'Perryridge'
```

- Find the number of tuples in the customer table.

```
SELECT COUNT(*)
FROM customer
```

- Find the number of accounts per branch.

```
SELECT COUNT(DISTINCT CustName), BranchName
FROM account
GROUP BY BranchName
```


Review 3.6

account

AccNr	BranchName	Balance
'A-101'	'Downtown'	500
'A-215'	'Perryridge'	700
'A-102'	'Perryridge'	400
'A-305'	'Perryridge'	350
'A-222'	'Perryridge'	700
'A-201'	'Brighton'	900
'A-217'	'Brighton'	750

Formulate the following query in SQL:

1. Determine the largest balance of branches with a at least one account with a balance of less than 600

Derived Tables

- SQL allows a query expression to be used in the **from** clause.
- A derived table is defined by a query expression.
- Find the average account balance of those branches where the average account balance is greater than $1200.

```
SELECT BranchName, AvgBalance
FROM ( SELECT BranchName, AVG(Balance)
       FROM account
       GROUP BY account
     ) AS branchAvg(BranchName, AvgBalance)
WHERE AvgBalance > 1200
```


Review 3.7

- Can an empty/missing from clause be useful?
- Which schema modifications do not change the result of a query specification if no * is used in the select list?
- Why is the following statement strange?

```
SELECT DISTINCT BranchName, SUM(Balance)
FROM account
GROUP BY BranchName
```

Review 3.8

- Consider schema `PC(Model,Speed,RAM,Price)`. Determine
  1. the price of the most expensive PC
  2. the price and model of the most expensive PC

Review 3.9

- Consider schema `Emp(Name, Sal, DName)`. For each department with more than 3 employees determine the number of employees earning more than 40K.

Query Expressions/1

- The set operations **union**, **intersect**, and **except** operate on tables and correspond to the relational algebra operations $\cup, \cap, -$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all**, and **except all**.

  Suppose a tuple occurs m times in r and n times in s, then, it occurs:
  - $m + n$ times in r **union all** s
  - $min(m, n)$ times in r **intersect all** s
  - $max(0, m - n)$ times in r **except all** s

Query Expressions/2

- Find all customers who have a loan, an account, or both:

```
SELECT CustName FROM depositor
UNION
SELECT CustName FROM borrower
```

- Find all customers who have both a loan and an account:

```
SELECT CustName FROM depositor
INTERSECT
SELECT CustName FROM borrower
```

- Find all customers who have an account but no loan:

```
SELECT CustName FROM depositor
EXCEPT
SELECT CustName FROM borrower
```

Notation/5

- Renaming of tables and columns with **as**:
  - **from** depositor **as** d
  - **select** max(balance) **as** HighestBalance
  - PostgreSQL and MySQL: **as** can be omitted in from and select clauses
  - Oracle: **as** must be omitted in the from clause
  - Oracle: **as** can be omitted in the select clause
- In MySQL no set difference (EXCEPT) and no set intersection (INTERSECT) exist.
- In Oracle EXCEPT must be replaced through MINUS.

Subqueries, Duplicates, and Nulls

- Subqueries
- Duplicates
- Nulls
- Ordering

Subqueries/1

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **query expression** that is nested within another query expression.
- Example:

```
SELECT X FROM p WHERE X IN (SELECT Y FROM q)
```

  Semantics/evaluation: For each tuple of p evaluate the subquery and check if X is in the result table computed by the subquery.

- Subqueries are common and used heavily in applications.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- If the inner query uses attributes of the outer query then the queries are **correlated**.

Subqueries/2

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all customers who have both an account and a loan at the bank.

```
SELECT CustName
FROM borrower
WHERE CustName IN ( SELECT CustName
                    FROM depositor )
```

- Find all customers who have a loan at the bank but do not have an account at the bank

```
SELECT CustName
FROM borrower
WHERE CustName NOT IN ( SELECT CustName
                        FROM depositor )
```

Review 3.10

- Give an SQL query that determines customers who have a loan and an account such that the account number is larger than the loan number. Describe the evaluation of the query for relations
  - Borrower(CustName,LoanNr); borrower = { (A,18), (B,19), (C,2) }
  - Depositor(CustName,AccNr); depositor = { (A,17), (B,20), (D,8) }

Subqueries/3

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all customers who have both an account and a loan at the Perryridge branch

```
SELECT CustName
FROM borrower, loan
WHERE borrower.LoanNum = loan.LoanNum
AND BranchName = 'Perryridge'
AND (BranchName, CustName) IN (
       SELECT BranchName, CustName
       FROM depositor, account
       WHERE depositor.AccNr = account.AccNr)
```


Subqueries/4

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

Comparing sets:

- Find all branches that have greater assets than some branch located in Brooklyn.

```
SELECT t.BranchName
FROM branch AS t, branch AS s
WHERE t.Assets > s.Assets
AND s.BranchCity = 'Brooklyn'
```

- Same query using > **some** clause

```
SELECT BranchName
FROM branch
WHERE assets > SOME (
  SELECT assets
  FROM branch
  WHERE BranchCity = 'Brooklyn' )
```


Subqueries/5

- F <comp> **some** $r \Leftrightarrow \exists t \in r$ such that (F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5 < **some** [0, 5, 6]) = true

(read: 5 < some tuple in the table)

(5 < **some** [0, 5]) = false

(5 = **some** [0, 5]) = true

(5 ≠ **some** [0, 5]) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Review 3.11/1

Consider tables

- $p = \{(5), (2)\}, sch(p) = P(X)$
- $q = \{(2), (3)\}, sch(q) = Q(Y)$

Determine the results of the following SQL queries:

1.
```
SELECT *
FROM p
WHERE X IN (
  SELECT Y FROM q )
```

2.
```
SELECT *
FROM p
WHERE X = SOME (
  SELECT Y FROM q )
```


Review 3.11/2

3.
```
SELECT *
FROM p
WHERE X NOT IN (
  SELECT Y FROM q )
```

4.
```
SELECT *
FROM p
WHERE NOT ( X = SOME (
  SELECT y FROM q ))
```

5.
```
SELECT *
FROM p
WHERE X <> SOME (
  SELECT Y FROM q)
```


Subqueries/6

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$
- The exists (and not exists) subqueries are used frequently in SQL.
- Several database systems eliminate (rewrite) all subqueries except subqueries using **exists** and **not exists**.
- Besides **some** there are also **any** (a synonym for **some**) and **all**.
- Avoid **in**, **all**, **any**, **some** and use **exists** instead.

Review 3.12/1

- Translate the following DRC expressions into SQL. Assume column i of table r has name RCi
  - $\{X \mid p(X, 3)\}$
  - $\{X, Z \mid \exists Y(p(X, Y) \land q(X, Y) \land q(Z, 3))\}$

Review 3.12/2

- Translate the following DRC expressions into SQL. Assume column i of table r has name RCi
  - $\{X \mid p(X,3) \land X > 5\}$
  - $\{X,Y,Z \mid p(X,Y,Z) \land \neg q(Y,3)\}$

Review 3.12

- Translate the following DRC expressions into SQL. Assume column i of table r has name RCi
  - $\{X \mid p(X) \land \neg\exists(p(Y) \land Y > X)\}$
  - $\{X \mid p(X) \lor (q(X) \land \neg r(X))\}$

Subqueries/7

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Find all customers who have an account at all branches located in Brooklyn.

```
SELECT DISTINCT s.CustName
FROM depositor AS s
WHERE NOT EXISTS (
   ( SELECT BranchName
     FROM branch
     WHERE BranchCity = 'Brooklyn' )
  EXCEPT
   ( SELECT r.BranchName
     FROM depositor AS t, account AS r
     WHERE t.AccNr = r.AccNr
     AND s.CustName = t.CustName ))
```


Review 3.13

- Assume table r stores for each person the degrees they have got. Determine pairs of persons with exactly the same degrees.

Review 3.13/2

Review 3.13/3

Duplicates/1

- In tables with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators - given multiset relations r_1 and r_2:
  1. $\sigma_\theta(\mathbf{r_1})$: If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
  2. $\pi_{\mathbf{A}}(\mathbf{r})$: For each copy of tuple t_1 in r_1, there is a copy of tuple $\pi_A(t_1)$ in $\pi_A(r_1)$ where $\pi_A(t_1)$ denotes the projection of the single tuple t_1.
  3. $\mathbf{r_1} \times \mathbf{r_2}$: If there are c_1 copies of tuple t_1 in r_1 and r_2 copies of tuple t_2 in r_2, there are $c_1 * c_2$ copies of the tuple t_1, t_2 in $r_1 \times r_2$

Duplicates/2

- Example: Suppose multiset relations $r_1(A, B)$ and $r_2(C)$ are as follows:

$$r_1 = \{(1, a)\ \ (2, a)\} \qquad r_2 = \{(2),\ (3),\ (3)\}$$

- Then $\pi_B(r_1)$ would be $\{(a), (a)\}$, while $\pi_B(r_1) \times r_2$ would be

$$\{(a, 2),\ (a, 2),\ (a, 3),\ (a, 3),\ (a, 3),\ (a, 3)\}$$

- SQL duplicate semantics:

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  is equivalent to the *multiset* version of the expression:

$$\pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ \ldots \ \times r_m))$$

Review 3.14

- Write a SQL statement that eliminates duplicates from a table (without using **distinct**, **unique**, **grouping**).
- Which of the following statements are semantically equivalent?
  - `SELECT X FROM p WHERE X IN (SELECT Y FROM q)`
  - `SELECT X FROM p, q WHERE X = Y`
  - `SELECT DISTINCT X FROM p, q WHERE X = Y`

Null Values/1

- It is possible for tuples to have a null value, denoted by *null*, for some of their columns
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *Amount*.

```
SELECT LoanNum
FROM loan
WHERE Amount IS NULL
```

- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null

Null Values/2

- Any comparison with *null* returns unknown
  - Example: $5 < null$ or $null <> null$ or $null = null$
- Three-valued logic using the truth value *unknown*:
  - **OR**

    (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - **AND**

    (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*
    (*unknown* **and** *unknown*) = *unknown*
  - **NOT**

    (**not** *unknown*) = *unknown*
  - "*P* **is unknown**" evaluates to *true* if predicate *P* evaluates to *unknown*

Null Values/3

- Intuition for semantics of NULL values:

account

AccNr	BranchName	Balance
'A-101'	NULL	500
'A-215'	NULL	NULL
'A-102'	NULL	400
'A-305'	'Perryridge'	350
'A-222'	'Perryridge'	NULL
'A-201'	'Brighton'	900
'A-217'	'Brighton'	750

Neither are these NULL values all identical nor are they all different (cf. above).

Null Values/4

- Any row (or group) that does not evaluate to true is eliminated by the where (having) clause.
- Arithmetic operations return NULL if one argument is NULL: 7 + NULL = NULL.
- Two rows are duplicates if corresponding columns are either equal or both are NULL (thus, NULL values are equal for this purpose!).
- Grouping groups NULL values together.

Null Values/5

- Total of all loan amounts

```
SELECT COUNT(Amount)
FROM loan
```

  - Above statement ignores null amounts
  - Result is 0 if there is no non-null amount
- All aggregate operations, except **count(*)**, ignore tuples with null values on the aggregated columns. (Thus, **sum(X)** is different from summing all values in a column!)
- If all values in a column are NULL then aggregates over this column return NULL (except **count**, which gives 0).

Review 3.15

- What does the query

```
SELECT * FROM pc
WHERE Speed > 1GHz OR Speed < 4GHz
```

  Propose an equivalent but better solution for this query
- What is the result of the following SQL statement:

```
SELECT * FROM r WHERE X <> NULL
```


Review 3.15

- What is the result of the following SQL statement:

```
SELECT X, SUM(Y) FROM r GROUP BY X
```

Review 3.15

- Explain similarities and differences between the following statements over relations with schemas R(X) and S(A):

1. ```
   SELECT * FROM r WHERE X NOT IN (
     SELECT A FROM s )
   ```
2. ```
   SELECT * FROM r WHERE NOT EXISTS (
     SELECT * FROM s WHERE X = A)
   ```


Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan in Perryridge branch

```
SELECT DISTINCT CustName
FROM borrower, loan
WHERE borrower.LoanNum = loan.LoanNum
AND BranchName = 'Perryridge'
ORDER BY CustName
```

- We may specify **desc** for descending order or **asc** for ascending order, for each column; ascending order is the default.
  - Example: `ORDER BY CustName DESC`

Database Modifications

- Insertions
- Deletions
- Updates

Insertions/1

- Add a new tuple to *account* (ordering of values is used to determine columns)

```
INSERT INTO account
  VALUES ('A-9732', 'Perryridge', 1200)
```

or equivalently (name of columns is used to determine columns)

```
INSERT INTO account(BranchName,Balance,AccNr)
  VALUES ('Perryridge', 1200, 'A-9732')
```

- Add a new tuple to account with balance set to null

```
INSERT INTO account
  VALUES ('A-9732', 'Perryridge', NULL)
```

Insertions/2

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

```
INSERT INTO account
  SELECT LoanNum, BranchName, 200
  FROM loan
  WHERE BranchName = 'Perryridge'

INSERT INTO depositor
  SELECT LoanNum, CustName
  FROM loan, borrower
  WHERE BranchName = 'Perryridge'
  AND loan.AccNr = borrower.AccNr
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the table.

Deletions/1

- Delete all account tuples at the Perryridge branch

```
DELETE FROM account
WHERE BranchName = 'Perryridge'
```

- Delete all accounts at every branch located in the city Needham

```
DELETE FROM account
WHERE BranchName IN (
  SELECT BranchName
  FROM branch
  WHERE BranchCity = 'Needham' )
```


Deletions/2

- Delete the record of all accounts with balances below the average at the bank.

```
DELETE FROM account
WHERE Balance < ( SELECT AVG(Balance)
                  FROM account )
```

  - Problem: as we delete tuples from deposit, the average balance changes
  - Solution used in SQL is the *logical update semantics* (guarantees that sequence in which updates to a table are computed does not matter):
    1. First, compute **avg** balance and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or evaluating the condition again)

Updates/1

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

```
UPDATE account
SET Balance = Balance * 1.06
WHERE Balance > 10000

UPDATE account
SET Balance = Balance * 1.05
WHERE Balance <= 10000
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Updates/2

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
UPDATE account
SET Balance = CASE
                WHEN Balance <= 10000
                THEN Balance * 1.05
                ELSE Balance * 1.06
              END
```


Review 3.16

- Consider `p(A) = { (1), (2), (3) }`. Describe the semantics and evaluation of the following statement:

```
UPDATE p t1
SET t1.a = (
  SELECT t1.a + SUM(t2.a) FROM p AS t2 )
```


Summary/1

- DDL (data definition language)
  - Used to **create**, **alter**, and **drop** tables.
  - Table definitions include **constraints** (domain, not null, primary key, foreign key).
  - Constraints ensure **data quality**, which in turn gives competitive advantages to businesses.
  - Enforcing constraints is difficult and requires a focused effort.
  - Data quality must be considered during data acquisition; the database system must enforce it; it cannot be fixed later.
- Expressions and predicates
  - Database systems have been conservative since the efficient evaluation must be considered.
  - The brute force evaluation of a condition on one billion tuples might not be good enough.
  - Modern database systems are extensible with user defined functions.

Summary/2

- SQL
  - SQL is used very heavily in practice; the intergalactic data speak [Michael Stonebraker].
  - The building block of the DML are **query specifications** and **query expressions**.
  - SQL is more than simple select-from-where. The set based semantics of SQL requires some practice.
  - Know the **conceptual evaluation**/semantics of query expressions.
    - Form the cross product of the tables in the from clause
    - Eliminate tuples that do not satisfy the where clause
    - Group the remaining tuples according to the group clause
    - Eliminate groups that do not satisfy the having clause
    - Evaluate the expressions in the select clause
    - With aggregation one result tuple is produced for each group
    - Eliminate duplicates if distinct is specified
    - Compute query expressions independently and apply set operations (union, excpet, intersect)
    - Sort all result tuples of order clause is specified

Summary/3

- The difference between **set** and **multisets** (or bags) is relevant.
- **Subqueries** are natural and help the understanding; it is possible (and advisable) to only use **exists** and **not exists**.
- Large parts of RA, SQL and DRC are equivalent; know how to go from one to the other.
- Modification statements use the **logical update semantics** (compute changes before they are applied).
- The SQL OLAP extensions (cf. Data Warehouse course) further extend SQL to better support report generation (cross tabs, moving averages, etc).

Database Systems
Spring 2017

Database Programming
SL04

- Views
- Recursive Queries
- Integrity Constraints
- Functions and Procedural Constructs
- Triggers
- Accessing Databases

Literature and Acknowledgments

Reading List for SL04:

- Database Systems, Chapters 5 (5.2 and 5.3) and 12, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Fundamentals of Database Systems, Fourth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Views

- Purpose of views
- Creation and use of views
- Handling views in the DBMS
- Temporary views

Views/1

- A view is a table whose rows are not stored in the database. The rows are computed when needed from the view definition.
- This is useful in cases where
  - it is not desirable for all users to see the entire logical model (that is, all the actual tables stored in the database), or
  - the user wants to access computed results (rather than the actual data stored on the disk)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
SELECT CustName, borrower.LoanNr, BranchName
FROM borrower, loan
WHERE borrower.LoanNr = loan.LoanNr
```

- A **view** provides a mechanism to hide data from the view of users, or to give users direct access to the results of (complex) computations.

Views/2

- A view is defined using the **create view** statement which has the form

  CREATE VIEW v **AS** <query expression>

  where <query expression> is any legal SQL expression. The view name is represented by v.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database.
- Any table that is not of the conceptual model but is made visible to a user as a "virtual table" is called a **view**.

Views/3

- A view consisting of branches and their customers:

```
CREATE VIEW allCustomer AS
  ( SELECT BranchName, CustName
    FROM depositor, account
    WHERE depositor.AccNr = account.AccNr )
  UNION
  ( SELECT BranchName, CustName
    FROM borrower, loan
    WHERE borrower.LoanNr = loan.LoanNr )
```

- Find all customers of the Perryridge branch:

```
SELECT CustName
FROM allCustomer
WHERE branche_name = 'Perryridge'
```


Views/4

- The view definition is stored in the meta database.
- The meaning of a query expression that includes views is defined through view expansion.
- The view expansion of an expression repeats the following replacement step:

  repeat
  Find any view v_i in e_1;
  Replace view v_i by the expression defining v_i;
  until no more views are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Views/5

- Tables and views can be used interchangeably in queries.
- Tables and views behave differently wrt modification operations.
- Updates to views are restricted. No broad consensus/standard exists and DBMSs behave differently.
- Roughly, a view shall be **updatable** if the database system can determine the reverse mapping from the view schema to the schema of the underlying base tables.
- The exact definition of updatable views has been enlarged significantly from SQL-1992 to SQL:1999 (cf. below).

Review 4.1

Discuss the behavior of following piece of SQL code.

```
CREATE VIEW goodStudents (sid, gpa) AS
  SELECT SID, GPA
  FROM students
  WHERE GPA > 3.0;

INSERT INTO goodStudents VALUES (51234, 2.8);
```


Views/6

- In SQL-92 a view is updatable if it is defined on a single table using projections and selections with no use of aggregate operations.
- An SQL-92 view is **not updatable** if the defining query expression satisfies one of the following conditions:
  1. the keyword DISTINCT is used in the view definition
  2. the select list contains components other than column specifications, or contains more than one specification of the same column
  3. the FROM clause specifies more than one table reference or refers to a non-updatable view
  4. the GROUP BY clause is used in the view definition
  5. the HAVING clause is used in the view definition

Views/7

- In SQL:1999 primary key constraints are taken into account for defining updatability of views.
- With this also views defined through a join can be updated.
- Intuitively, an column of a view is updatable if it can be traced back to a unique tuple in one of the underlying tables (i.e., **each base tuple is guaranteed to appear at most once in the view**; in Oracle such tables are termed key-preserved tables).
- There are views that can be modified by updating rows and there are views into which tuples can be inserted.
- View defined through set operations (union, except, intersect) cannot be inserted into but might be modifiable.

Review 4.2

Consider the DDL statements:

```
CREATE TABLE account(AccNr INTEGER PRIMARY KEY,
  BrName CHAR(9), Balance INTEGER);

CREATE TABLE depositor(AccNr INTEGER,
  CuName CHAR(9), PRIMARY KEY(AccNr,CuName));

CREATE VIEW v AS
  SELECT CuName, BrName
  FROM depositor NATURAL JOIN account;
```

Explain the behavior of the following statements:

1. `UPDATE v SET BrName = 'Q';`
2. `UPDATE v SET CuName = 'Z';`

Review 4.2

With Clause/1

- The **with** clause provides a way of defining temporary views whose definition is available only to the query in which the **with** clause occurs.
- The with clause is useful to structure complex SQL statements and eliminate code repetitions.
- Find all accounts with the maximum balance

```
WITH
maxBalance(Val) AS (
  SELECT MAX(Balance)
  FROM account
)
SELECT AccNr
FROM account, maxBalance
WHERE account.Balance = maxBalance.Val
```


With Clause/2

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
WITH
braTot(BranchName,Val) AS (
  SELECT BranchName, SUM(Balance)
  FROM account
  GROUP BY BranchName
),
braTotAvg(Val) AS (
  SELECT AVG(Val)
  FROM braTot
)
SELECT BranchName
FROM braTot, braTotAvg
WHERE braTot.Val > braTotAvg.Val
```


Recursion in SQL

- Recursive views in SQL-1999
- Example of a recursive view

Recursion in SQL/1

- SQL:1999 permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

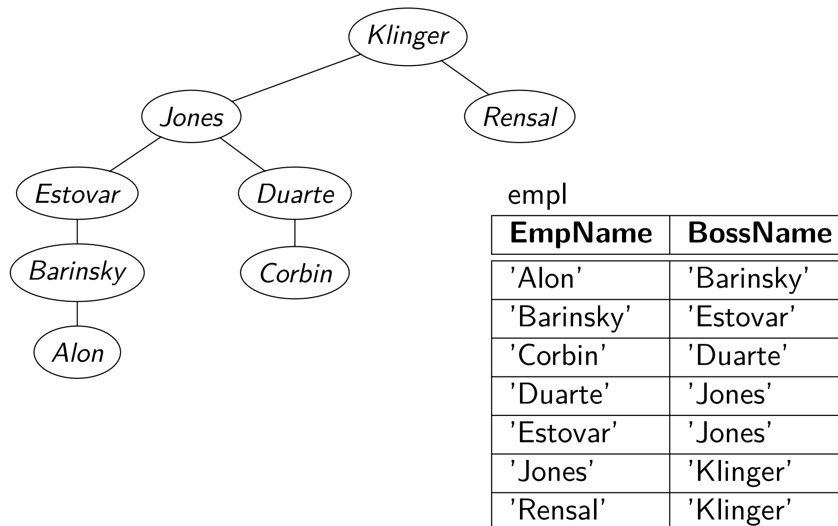
```
WITH RECURSIVE mgr(EmpName,MgrName ) AS (
  SELECT EmpName, BossName
  FROM empl
  UNION
  SELECT empl.EmpName, mgr.MgrName
  FROM empl, mgr
  WHERE BossName = mgr.EmpName
)
SELECT * FROM mgr;
```

View *mgr* is the **transitive closure** of the *empl* relation.

Recursion in SQL/2

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *empl* with itself
    - This can give only a fixed number of levels of managers.
    - We can always construct a database with a greater number of levels of managers than a fixed number.
- Computing transitive closure
  - The next slide shows a *empl* relation
  - Each step of the iterative process constructs an extended version of *mgr* from its recursive definition.
  - The final result is called the *fixpoint* of the recursive view definition.

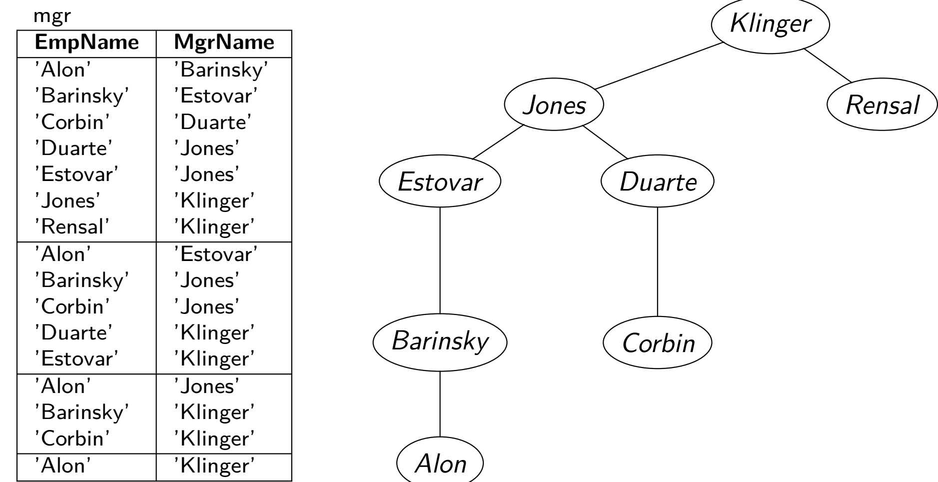
Example of Fixpoint Computation/1



empl

EmpName	BossName
'Alon'	'Barinsky'
'Barinsky'	'Estovar'
'Corbin'	'Duarte'
'Duarte'	'Jones'
'Estovar'	'Jones'
'Jones'	'Klinger'
'Rensal'	'Klinger'

Example of Fixpoint Computation/2



mgr

EmpName	MgrName
'Alon'	'Barinsky'
'Barinsky'	'Estovar'
'Corbin'	'Duarte'
'Duarte'	'Jones'
'Estovar'	'Jones'
'Jones'	'Klinger'
'Rensal'	'Klinger'
'Alon'	'Estovar'
'Barinsky'	'Jones'
'Corbin'	'Jones'
'Duarte'	'Klinger'
'Estovar'	'Klinger'
'Alon'	'Jones'
'Barinsky'	'Klinger'
'Corbin'	'Klinger'
'Alon'	'Klinger'

Recursion in SQL/3

- Recursive views are required to be **monotonic**. That is, if we add tuples to *manager* the view contains all of the tuples it contained before, plus possibly more.
- For representing and querying trees there exist also alternative solutions:
  - Nested set model: techniques for representing nested sets (aka trees or hierarchies) in relational databases.
  - Joe Celko: Trees and Hierarchies in SQL for Smarties
  - Trees (e.g., XML) are shredded and a clever numbering is used that allows to quickly determine subtrees, ancestors, siblings, etc of a node.

Integrity Constraints

- Domain constraints
- Not null constraints
- Primary keys
- Check constraint
- Referential integrity (foreign keys)
- Assertions

Integrity Constraints/1

- Integrity constraints guard against damage to the database, by ensuring that changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000
  - A salary of a bank employee must be at least $4.00 an hour.
  - A customer must have a (non-null) phone number.
  - A customer can only get a loan if she has an account.
- An integrity constraint is a closed first order formula that must be true, i.e., that each database instance must satisfy.
  - Example: $\forall B(account(_, _, B) \Rightarrow B < 10M)$
- The main issue with integrity constraints is how to check them efficiently.
- SQL offers **special-purpose syntax** and **efficient checking mechanisms** for the most important classes of integrity constraints.

Integrity Constraints/2

Integrity constraints on single relations:

- domain constraints
- **not null**
- **primary key**
- **unique**
- **check**(P), where P is a predicate over a single relation

Integrity constraints on multiple relations:

- **foreign key**
- **check**(P), where P is a predicate over multiple relations
- **assertion**

Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraints. They check values inserted in the database, and they check queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  - Example:

```
CREATE DOMAIN Dollars INTEGER
CREATE DOMAIN Pounds INTEGER
```

- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert values of type Dollar as follows:

```
CAST(r.Amnt/1.5 AS Pounds)
```


Not Null Constraint

- Declare *BranchName* to be **not null**:

```
BranchName CHAR(15) NOT NULL
```

- Declare the domain *Dollars* to be **not null**:

```
CREATE DOMAIN Dollars INTEGER NOT NULL
```


Primary Key

- A primary key ensures that an attribute value is not null and unique across all rows of a table. Therefore it can be used to identify a unique row in a table, and is used for query optimization.
- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
- The **primary key** clause lists attributes that form the primary key.
- Example:

```
CREATE TABLE customer (
  CustName CHAR(20),
  CustStreet CHAR(30),
  CustCity CHAR(30),
  PRIMARY KEY (CustName) )
```


The Unique Constraint

- **unique** $(A_1, A_2, \ldots, A_m)$
- The unique specification states that the attributes
  $A_1, A_2, \ldots, A_m$
  form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).

The check Clause/1

- **check** (P), where P is a predicate
  Example: Declare *BranchName* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
CREATE TABLE branch (
  BranchName CHAR(15),
  BranchCity CHAR(30),
  Assets INTEGER,
  PRIMARY KEY (BranchName),
  CHECK (Assets >= 0))
```

- Note that with subqueries (not exists, etc) the check constraint can become very general. Implementations limit the predicates that are allowed in the check clause.

Referential Integrity/1

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Perryridge" is a branch name appearing in one of the tuples in the account relation, then there exists a tuple in the branch relation for branch "Perryridge".
- Foreign keys can be specified as part of the SQL **create table** statement.
- The **foreign key** clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Referential Integrity/2

Examples of integrity constraints:

```
CREATE TABLE customer (
  CustomerName CHAR(20)
  CustStreet CHAR(30),
  CustCity CHAR(30),
  PRIMARY KEY (CustomerName) )


CREATE TABLE branch (
  BranchName CHAR(15),
  BranchCity CHAR(30),
  Assets INTEGER,
  PRIMARY KEY (BranchName) )
```


Referential Integrity/3

Examples of integrity constraints:

```
CREATE TABLE account (
  AccNr CHAR(10),
  BranchN CHAR(15),
  Balance INTEGER,
  PRIMARY KEY (AccNr),
  FOREIGN KEY (BranchN) REFERENCES branch )


CREATE TABLE depositor (
  CustName CHAR(20),
  AccNum CHAR(10),
  PRIMARY KEY (CustName, AccNum),
  FOREIGN KEY (AccNum) REFERENCES account,
  FOREIGN KEY (CustName) REFERENCES customer )
```

Review 4.3

Assume tables p(X) and q(Y). p.X is a primary key. q.Y is a foreign key that references p.X.

1. Use a check constraint to formulate the foreign key constraint.
2. Formulate a query that returns an empty result if the foreign key is satisfied and a non-empty result otherwise.

Assertions/1

- An **assertion** is a predicate expressing a condition that the database must satisfy.
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that might violate the assertion.
  - This testing may introduce a significant amount of overhead; hence assertions should be used with care.
- Asserting
  $\forall X(p(X))$
  is achieved using
  $\neg\exists\neg(p(X))$

Assertion/2

- Every loan has at least one borrower who maintains an account with a minimum balance or $1000

```
CREATE ASSERTION balance_constraint CHECK
  NOT EXISTS (
    SELECT *
    FROM loan
    WHERE NOT EXISTS (
      SELECT *
      FROM borrower b, depositor d, account a
      WHERE loan.LoanNr = b.LoanNr
      AND b.CustName = d.CustName
      AND d.AccNr = a.AccNr
      AND a.Balance >= 1000)))
```


Assertion/3

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
CREATE ASSERTION sum_constraint CHECK (
  NOT EXISTS (
    SELECT *
    FROM branch b
    WHERE
      ( SELECT SUM(Amount)
        FROM loan l
        WHERE l.BranchName = b.BranchName )
      >=
      ( SELECT SUM(Balance)
        FROM account a
        WHERE l.BranchName = b.BranchName )))
```

Review 4.4/1

Consider the tables

```
CREATE TABLE branch(BranchName CHAR(9),
  BranchCity CHAR(9), Assets INTEGER);

CREATE TABLE account(AccNr INTEGER PRIMARY KEY,
  BranchName CHAR(9), Balance INTEGER);
```

Explain how to efficiently check the integrity constraint

$$\forall BN(account(_, BN, _) \Rightarrow branch(BN, _, _))$$

by considering the different database modifications.

Review 4.4/2

Review 4.5

Create a table for cities and a table for states. For each state its capital shall be recorded and for each city the state it is located in shall be recorded. Discuss the properties of the following solution:

```
CREATE TABLE cities(CName CHAR(9) PRIMARY KEY, State CHAR(9));
CREATE TABLE states(SName CHAR(9) PRIMARY KEY, Capital CHAR(9));
ALTER TABLE cities ADD FOREIGN KEY (State) REFERENCES states(SName);
ALTER TABLE states ADD FOREIGN KEY (Capital) REFERENCES cities(CName);
```


User Defined Functions

- PL/pgSQL value functions
- PL/pgSQL table functions
- External language functions

User-Defined Functions (UDF)

- User defined functions or stored procedures allow to execute application logic in the process space of the DBMS.
- This is good for the performance since it reduces the amount of data that is transferred between client and server.
- The SQL standard defines SQL/PSM (SQL/Persistent Stored Modules).
- PostgreSQL supports different kinds of user-defined functions:
  - Query language functions i.e., written in SQL
  - Procedural language functions such as PL/pgSQL
  - C-language functions, dynamically loadable objects (shared libraries)
- UDF functions can
  - Make arithmetic calculations
  - Query tables
  - Manipulate tables
  - Return single values or tables

PL/pgSQL Functions/1

- Structure

  create function *somefunc*()
  returns < *retype* > **as $$**
  [**declare**
      < *declarations* >]
  begin
      < *statements* >
  end;
  $$ language plpgsql;

- < *retype* >
  INTEGER
  RECORD
  TABLE
  ...

PL/pgSQL Functions/2

- < *declarations* >

```
quantity INTEGER := 30;
tbl_row account%ROWTYPE;
```

- < *statements* >

```
quantity := 4;

IF quantity < 5 THEN
  quantity := 5;
END IF;

WHILE quantity < 10 LOOP
  quantity := quantity + 1;
END LOOP;
```


PL/pgSQL Value Functions/1

- A **value function** returns a value (or tuple).
- A value function can be used instead of a value in an SQL statement.
- Returning a single value is fairly straightforward and does not raise new performance issues.
- Example (cf. next slide): Define a function that, given the name of a customer, returns the count of the number of accounts owned by this customer.

PL/pgSQL Value Functions/2

- Count of the number of accounts owned by a customer:

```
CREATE FUNCTION accountCnt (CName VARCHAR(9))
RETURNS INTEGER AS
$$
DECLARE
  accCnt INTEGER;
BEGIN
  SELECT COUNT(*) INTO accCnt
  FROM depositor
  WHERE depositor.CustName = CName;
  RETURN accCnt;
END;
$$ LANGUAGE PLPGSQL;
```

- Usage: `SELECT accountCnt('Bob');`

PL/pgSQL Table Functions/1

- **Table functions** return a table.
- Table functions can be used instead of a table.
- Table functions allow input parameters.
- The type of the return table must be defined.
- In the simplest case a table function returns the result of an SQL query as its result.
- Example (cf. next slide): Define a function that returns all accounts with a balance above an application-specified threshold.

PL/pgSQL Table Functions/2

- All accounts with a balance above a threshold:

```
CREATE FUNCTION highAccnts (limitVal INTEGER)
RETURNS TABLE (AccNum CHAR(10),
               BrName CHAR(15),
               Bal INTEGER) AS
$$
BEGIN
  RETURN QUERY
    SELECT AccNr, BranchName, Balance
    FROM account A
    WHERE A.Balance > highAccnts.limitVal;
END;
$$ LANGUAGE PLPGSQL;
```

- Usage: `SELECT * FROM highAccnts(1000);`

PL/pgSQL Table Functions/3

- Table functions may return large tables.
- The cursor mechanism (similar to iterators) was introduced to deal with result tables.
- A cursor points to the current row and a loop is used to iterate through all rows of a table.
- With a cursor the return type is a record that represents a row in a database table.
- Definition of a rowtype for table account: `account%ROWTYPE`
- Example (cf. next slide): Define a function that returns all accounts with a balance above an application-specified threshold.

PL/pgSQL Table Functions/4

- Accounts with balance above a threshold:

```
CREATE FUNCTION highAccnts (limitVal INTEGER)
RETURNS SETOF account AS
$$
DECLARE
  accrow account%ROWTYPE;
BEGIN
  FOR accrow IN SELECT * FROM account LOOP
    IF accrow.Balance > highAccnts.limitVal
      THEN RETURN NEXT accrow;
    END IF;
  END LOOP;
END;
$$ LANGUAGE PLPGSQL;
```

- Usage: `SELECT * FROM highAccnts(1000);`

PostgreSQL C-Language Functions/1

- User-defined functions can be written in C (or a number of other languages).
- Such external functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand.
- The dynamic loading feature is what sets external functions apart from internal functions.
- The code must convert between PostgreSQL types and C types.

PostgreSQL C-Language Functions/2

- Define a function adding 1 to its argument

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32 arg = PG_GETARG_INT32(0);
    PG_RETURN_INT32(arg + 1);
}
```


PostgreSQL C-Language Functions/3

- Compile as dynamic library func.so
  - gcc -I/usr/include/postgresql/9.3/server/ -fpic -c func.c
  - gcc -shared -o func.so func.o
- Map to a DBMS function

```
CREATE OR REPLACE FUNCTION incr(INTEGER)
RETURNS INTEGER AS '/home/boehlen/func.so', 'add_one'
LANGUAGE C STRICT;
```

- Usage:

```
SELECT incr(5);
```

Triggers

- Purpose of triggers
- Definition of triggers

Triggers/1

- A database trigger is a procedural piece of code that is automatically executed in response to certain events on a particular table in a database.
- Triggers are executed when a specified condition occurs during insert/delete/update.
- Triggers are actions that fire automatically if the condition is satisfied.
- Triggers follow an event-condition-action (ECA) model
  - Event: Database modification (e.g., insert, delete, update)
  - Condition: Any expression that evaluates to true/false
  - Action: Sequence of SQL statements that will be executed

Triggers/2

- Example of a trigger: When a new employees is added to a department, modify the TotSal of the Department to include the new employees salary

```
CREATE TRIGGER TotSal1
  AFTER INSERT ON employee
  FOR EACH ROW
  WHEN (NEW.Dno IS NOT NULL)
    UPDATE department
    SET TotSal = TotSal + NEW.Salary
    WHERE Dno = NEW.Dno;
```

- The above syntax is the one of the SQL standard.

Triggers/3

Explanation of the trigger:

- We **create** a trigger TotSal1
- Trigger TotSal1 will execute **after insert** on employee table.
  - Instead of **after** we could also have **before** or **instead of**.
  - Instead of **insert** we could also have **update** or **delete**.
- The trigger fires (is executed) **for each row** that is inserted.
  - The trigger fires for each statement if **for each statement** is specified instead.
- The **when** condition determines if a trigger is executed or not.
- The trigger will update department by setting the new TotSal to the sum of old TotSal and new.Salary where Dno matches new.Dno

Triggers/4

- In PostgreSQL a trigger consist of two parts:
  - A **trigger function** that defines the action to be performed
  - A **trigger** that defines when the trigger must be fired.
- A trigger function must return NULL or a row with the schema of the table the trigger was fired for.

```
CREATE OR REPLACE FUNCTION checkTemp()
RETURNS TRIGGER AS
$$
DECLARE
BEGIN
  IF NEW.val < -273 THEN
    RAISE EXCEPTION 'invalid value: %', NEW.val;
  END IF
  RETURN NEW;
END;
$$ LANGUAGE PLPGSQL;
```


Triggers/5

PostgreSQL trigger:

- A trigger is associated with a table or view and executes the specified function when certain events occur.
- A trigger can fire before the operation, after the operation has completed, or instead of the operation.
- A trigger can fire for each row or for each statement.

A PostgreSQL example of a trigger definition.

```
CREATE TRIGGER TrigTempCheck
  BEFORE INSERT OR UPDATE
  ON temperature
  FOR EACH ROW
  EXECUTE PROCEDURE checkTemp();
```


Review 4.6

Compare triggers and integrity constraints.

Accessing Databases

- ODBC
- JDBC

Accessing Databases

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- **Embedded SQL**: many languages allow to embed SQL statements in their code. The embedded code can be
  - static (i.e., code is known at compile time)
  - dynamic (i.e., code is unknown at compile time; created at runtime)
- **ODBC** (Open Database Connectivity) is a Microsoft standard works with C, C++, C#, and Visual Basic
- **JDBC** (Java Database Connectivity) is from Sun Microsystems and works with Java

ODBC/1

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a DBMS.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.

ODBC/2

```
int main() {
  SQLAllocEnv(&hEnv);
  SQLAllocConnect(hEnv, &hDbc);
  SQLConnect(hDbc, "pgifi", SQL_NTS, "usr", SQL_NTS, "pwd", SQL_NTS);

  SQLAllocStmt(hDbc,&hstmt);
  SQLPrepare(hstmt, "select tablename from pg_tables", SQL_NTS);

  SQLExecute(hstmt);
  SQLBindCol(hstmt, 1, SQL_C_CHAR, (SQLPOINTER)name, 30, NULL);
  for (;;) {
    if (SQLFetch(hstmt)) break;
    printf("  '%s'\n", name);
  }

  SQLFreeStmt(hstmt, SQL_DROP);

  SQLDisconnect(hDbc); SQLFreeConnect(hDbc); SQLFreeEnv(hEnv);
}
// gcc -c pgODBC.c; gcc -o pgODBC pgODBC.o -lodbc; ./pgODBC
```


JDBC/1

- **JDBC** is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC/2

```
import java.sql.*;

public class pgJDBC {
  public static void main(String[] argv) {

    Class.forName("org.postgresql.Driver");

    Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://pg.ifi.uzh.ch/boehlen?ssl=true" +
      "&sslfactory=org.postgresql.ssl.NonValidatingFactory",
      "boehlen", "xxx");

     Statement stmt = conn.createStatement();

     ResultSet rset = stmt.executeQuery(
       "select tablename from pg_tables where tableowner='boehlen'");

     while (rset.next())
        System.out.println(rset.getString(1));
  }
}
// javac pgJDBC.java
// java -classpath /usr/share/java/postgresql-jdbc4-9.2.jar:. pgJDBC
```


Summary

- **Views** are essential to break up large tasks (SQL statements) in small, independent and manageable blocks.
- **Recursive queries** are used to compute ancestors, descendants, transitive closures, etc.
- **Integrity constraints** ensure a good data quality. Enforcing integrity constraints is not easy: people try to work around.
- **Functions** and **procedural constructs** are used heavily by many applications. For example PostGIS uses functions heavily to add advanced spatial functionality to PostgreSQL.
- Program access is through **ODBC** and **JDBC**. General-purpose graphical tools exist to interact with databases.

Database Systems
Spring 2017

Relational Database Design
SL05

- Relational Database Design Goals
- Functional Dependencies
- 1NF, 2NF, 3NF, BCNF
- Dependency Preservation, Lossless Join Decomposition
- Multivalued Dependencies, 4NF

Literature and Acknowledgments

Reading List for SL05:

- Database Systems, Chapters 14 and 15, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Fundamentals of Database Systems, Fourth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Relational Database Design Guidelines

- Goals of Relational Database Design
- Update Anomalies

Relational Database Design Guidelines/1

- The goal of relational database design is to find a good collection of relation schemas.
- The main problem is to find a good grouping of the attributes into relation schemas.
- We have a good collection of relation schemas if we
  - Ensure a simple semantics of tuples and attributes
  - Avoid redundant data
  - Avoid update anomalies
  - Avoid null values as much as possible
  - Ensure that exactly the original data is recorded and (natural) joins do not generate spurious tuples

Relational Database Design Guidelines/2

- Consider the relation schema:
  - EmpProj(SSN, PNum, Hours, EName, PName, PLoc)
- Update Anomaly:
  - Changing the name of project location "Houston" to "Dallas" for an employee forces us to make this change for all other employees working on this project.
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it (except by using null values).
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.

Relational Database Design Guidelines/3

- Consider relation schema
  EmpProj(SSN, PNum, Hours, EName, PName, Ploc)
  with instance

empproj

SSN	PNum	Hours	EName	PName	PLoc
1234	1	32.5	'Smith'	'ProductX'	'Bellaire'
1234	2	7.5	'Smith'	'ProductY'	'Sugarland'
6688	3	40.5	'Narayan'	'ProductZ'	'Houston'
4567	1	20.0	'English'	'ProductX'	'Bellaire'
4567	2	20.0	'English'	'ProductY'	'Sugarland'
3334	2	10.0	'Wong'	'ProductY'	'Sugarland'
3334	3	10.0	'Wong'	'ProductZ'	'Houston'
3334	10	10.0	'Wong'	'Computerization'	'Stafford'
3334	20	10.0	'Wong'	'Reorganization'	'Houston'

- Relation schema EmpProj is not a good schema and suffers from update anomalies.

Relational Database Design Guidelines/4

- **Guideline 1**: Each tuple in a relation should only represent one entity or relationship instance.
- **Guideline 2**: Design a schema that does not suffer from insertion, deletion and update anomalies.
- **Guideline 3**: Relations should be designed such that their tuples will have as few NULL values as possible; attributes that are NULL shall be placed in separate relations (along with the primary key).
- **Guideline 4**: Relations should be designed such that no spurious (i.e., wrong) tuples are generated if we do a natural join of the relations.

Functional Dependencies

- Definition
- Armstrong's inference rules
- Soundness and completeness
- Closure and minimal cover

Keys (Refresher)

- A **superkey** of a relation schema $R(A_1, A_2, \dots, A_n)$ is a set of attributes $S \subseteq attr(R)$ with the property that no two tuples t1 and t2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
- A **candidate key** K is a *superkey with the additional property* that removal of any attribute from K will cause the reduced K not to be a superkey any more.
- One of the candidate keys is *arbitrarily* chosen to be the **primary key**.
- Notation: We underline the primary key attributes:
  EmpProj(<u>SSN</u>, <u>PNum</u>, Hours, EName, PName, Ploc)
  Thus, $(SSN, PNum)$ is a primary key of EmpProj

Functional Dependencies/1

- Functional dependencies (FDs) are used to specify *formal measures* of the goodness of relational designs.
- Functional dependencies and keys are used to define **normal forms** for relations.
- Functional dependencies are **constraints** that are derived from the *meaning* and *interrelationships* of the attributes.
- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y.
- A functional dependency $X \to Y$ is **trivial** iff $Y \subseteq X$.

Functional Dependencies/2

- $X \to Y$ denotes a functional dependency.
- $X \to Y$ means that X functionally determines Y.
- $X \to Y$ holds if whenever two tuples have the same value for X they have the same value for Y.
  - For any two tuples t_1 and t_2 in any relation instance $r(R)$:
    If $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$
- $X \to Y$ for R specifies a *constraint* on the schema, i.e., on all possible relation instances r(R).
- FDs are derived from the real-world constraints on the attributes.
- Notation: instead of $\{A, B\}$ we write AB (or A, B), e.g., $AB \to BCD$ (instead of $\{A, B\} \to \{B, C, D\}$)

Review 5.1

r

A	B	C
1	1	3
2	1	1
3	2	2
4	1	1

Consider the relation instance $r(R)$ and the statements

1. A is a primary key of R
2. $B \to C$ is a functional dependency that holds for R
3. $C \to B$ is a functional dependency that holds for R
4. $BC \to A$ is a functional dependency that relation instance r satisfies

Which of these statements are true?

Functional Dependencies/3

Examples of FD constraints:

- Social security number determines employee name
  - $SSN \rightarrow EName$
- Project number determines project name and location
  - $PNum \rightarrow PName, PLoc$
- Employee ssn and project number determines the hours per week that the employee works on the project
  - $SSN, PNum \rightarrow Hours$

Functional Dependencies/4

- A FD is a property of the semantics of the attributes.
- A FD constraint must hold on *every* relation instance r(R)
- If K is a candidate key of R, then K functionally determines all attributes in R (since we never have two distinct tuples with $t_1[K] = t_2[K]$)
- Certain FDs can be ruled out based on a given state of the database:

teach

Teacher	Course	Textbook
'Smith'	'Data Structures'	'Bertram'
'Smith'	'Data Management'	'Martin'
'Hall'	'Compilers'	'Hoffman'
'Brown'	'Data Structures'	'Horowitz'

The FD Textbook → Course is possible
The FD Teacher → Course does not hold

Functional Dependencies/5

- Given a set of FDs F, we can **infer** additional FDs that hold whenever the FDs in F hold
- Armstrong's inference rules (aka Armstrong's axioms):
  - Reflexivity: $Y \subseteq X \models X \rightarrow Y$
  - Augmentation: $X \rightarrow Y \models XZ \rightarrow YZ$
  - Transitivity: $X \rightarrow Y, Y \rightarrow Z \models X \rightarrow Z$
- Notation:
  - $A \models B$ means that from A we can infer B
  - XZ stands for $X \cup Z$
- Armstrong's inference rules are **sound** and **complete**
  - These rules hold (are correct) and all other rules that hold can be deduced from these

Review 5.2

1. Prove $W \rightarrow Y \models WX \rightarrow Y$
2. Prove $X \rightarrow Y, Z \subseteq Y \models X \rightarrow Z$
3. Disprove $XY \rightarrow Z, Y \rightarrow W \models XW \rightarrow Z$

Functional Dependencies/6

- Additional inference rules that are useful:
  - Decomposition: $X \to YZ \models X \to Y, X \to Z$
  - Union: $X \to Y, X \to Z \models X \to YZ$
  - Pseudotransitivity: $X \to Y, WY \to Z \models WX \to Z$
- The last three inference rules, as well as any other inference rules, can be deduced from Armstrong's inference rules (because of the completeness property).

Functional Dependencies/7

- The **closure** of a set F of FDs is the set F^+ of all FDs that can be inferred from F.
- The **closure** of a set of attributes X with respect to F is the set X^+ of all attributes that are functionally determined by X.
- F^+ and X^+ can be calculated by repeatedly applying Armstrong's inference rules to F and X, respectively.

Functional Dependencies/8

- Two sets of FDs F and G are **equivalent** if:
  - Every FD in F can be inferred from G, and
  - Every FD in G can be inferred from F
  - Hence, F and G are equivalent if $F^+ = G^+$
- Definition: F **covers** G if every FD in G can be inferred from F (i.e., if $G^+ \subseteq F^+$)
- F and G are equivalent if F covers G and G covers F

Review 5.3

Consider $F = \{A \to C, AC \to D, E \to AD, E \to H\}$ and $G = \{A \to CD, E \to AH\}$. Are F and G equivalent?

Functional Dependencies/9

- A set of FDs is **minimal** if it satisfies the following conditions:
  1. No pair of FDs has the same left-hand side.
  2. We cannot remove any dependency from F and have a set of dependencies that is equivalent to F.
  3. We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where $Y \subset X$ and still have a set of dependencies that is equivalent to F.
- Every set of FDs has an equivalent minimal set
- There can be several equivalent minimal sets
- There is no simple algorithm for computing a minimal set of FDs that is equivalent to a set F of FDs
- The first condition can also be changed to "every FD has a single attribute for its right-hand side" (Elmasri and Navathe does this). Note: $X \rightarrow YZ \equiv X \rightarrow Y, X \rightarrow Z$

Review 5.4

Consider $R(A, B, C)$ and $F = \{A \rightarrow C, A \rightarrow B, B \rightarrow A\}$. Determine the minimal cover.

Normal Forms

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)

Normalization/1

- **Normalization:** The process of decomposing bad relations by breaking up their attributes into smaller relations that satisfy the normal forms.
- The normalization process was proposed by Codd in 1972.
- The normalization process applies a series of tests to a relation schema to verify that the schema qualifies for some normal form.
- A normalized database consists of a good collection of relation schemas.

Normalization/2

- 1NF
  - attribute values must be atomic
- 2NF, 3NF, BCNF
  - based on candidate keys and FDs of a relation schema
- 4NF
  - based on candidate keys, multi-valued dependencies (MVDs)
- 5NF
  - based on candidate keys, join dependencies (JDs)
- Additional properties may be needed to ensure a good relational design:
  - Losslessness of the corresponding join (very important and cannot be sacrificed)
  - Preservation of the functional dependencies (less stringent and may be sacrificed)

Normalization/3

- In practice **normalization** is carried out to guarantee that the resulting schemas are of high quality
- The normalization process provides a deep understanding of relations and attributes.
- The database designers *need not* normalize to the highest possible normal form
  - usually they choose 3NF, BCNF or 4NF
  - controlled redundancy is OK/good
- **Denormalization:**
  - The process of storing the join of higher normal form relations as a base relation (which is in a lower normal form since the join destroys the normal form)

First Normal Form (1NF)/1

- Disallows
  - composite attributes
  - multivalued attributes
  - nested relations: attributes whose values for an *individual tuple* are relations
- Often 1NF is considered to be part of the definition of a relation
- The following instance of schema $Department(DName, \underline{DNum}, DMgrSSN, DLoc)$ is not in 1NF:

department

DName	DNum	DMgrSSN	DLoc
'Research'	5	334455	{'Bellaire', 'Sugarland', 'Houston' }
'Administration'	4	987654	{ 'Stafford' }
'Headquarters'	1	888666	{ 'Houston' }

First Normal Form (1NF)/2

- Remedy to get 1NF: Form new relations for each multivalued attribute or nested relation
- The following instance is the equivalent instance in 1NF:

department

DName	DNum	DMgrSSN	DLoc
'Research'	5	334455	'Bellaire'
'Research'	5	334455	'Sugarland'
'Research'	5	334455	'Houston'
'Administration'	4	987654	'Stafford'
'Headquarters'	1	888666	'Houston'

Second Normal Form (2NF)/1

- A relation schema R is in **second normal form (2NF)** iff each attribute not contained in a candidate key is not partially functional dependent on a candidate key of R.
- An attribute is *partially functional dependent* on a candidate key if it is functionally dependent on a proper subset of the candidate key.
- The following relation is not in 2NF:

empproj

SSN	PNum	Hours	EName	PName	PLoc
1234	1	32.5	'Smith'	'ProductX'	'Bellaire'
1234	2	7.5	'Smith'	'ProductY'	'Sugarland'
6688	3	40.5	'Narayan'	'ProductZ'	'Houston'
4567	1	20.0	'English'	'ProductX'	'Bellaire'
4567	2	20.0	'English'	'ProductY'	'Sugarland'
3334	2	10.0	'Wong'	'ProductY'	'Sugarland'
3334	3	10.0	'Wong'	'ProductZ'	'Houston'
3334	10	10.0	'Wong'	'Computerization'	'Stafford'
3334	20	10.0	'Wong'	'Reorganization'	'Houston'

Second Normal Form (2NF)/2

- Remedy to get 2NF: Decompose and set up a new relation for each partial key with its dependent attributes. Keep a relation with the original key and any attributes that are functionally dependent on it.
- Consider EmpProj(SSN, PNum, Hours, EName, PName, PLoc)
  - Candidate key is SSN and PNum which funtionally determine Hours
  - SSN is a partial key with dependent attributes EName
  - PNum is a partial key with dependent attributes PName and PLoc
- 2NF normalization
  - EmpProj1(SSN, EName)
  - EmpProj2(PNum, PName, PLoc)
  - EmpProj3(SSN, PNum, Hours)

Review 5.5

Consider $R(A, B, C)$ and $F = \{A \rightarrow BC, B \rightarrow C\}$. Is R in 2NF? Is R a good schema?

Third Normal Form (3NF)/1

- A relation schema R is in **third normal form (3NF)** iff for all $X \rightarrow A \in F^+$ at least one of the following holds:
  - $X \rightarrow A$ is trivial
  - X is a superkey for R
  - A is contained in a candidate key of R
- Intuition: "Each non-key attribute must describe the key, the whole key, and nothing but the key." [Bill Kent, CACM 1983]
- A relation that is in 3NF is also in 2NF.

Third Normal Form (3NF)/2

- The following relation with the functional dependencies $SC \rightarrow T$ and $T \rightarrow C$ is in 3NF:

r

Student	Course	Textbook
'Smith'	'Data Structures'	'Bertram'
'Smith'	'Data Management'	'Martin'
'Hall'	'Compilers'	'Hoffman'
'Brown'	'Data Structures'	'Horowitz'

- $SC \rightarrow T$ is OK since SC is a candidate key.
- $T \rightarrow C$ is OK since C is contained in a candidate key.

- This relation is in 3NF but permits redundant information, which can lead to update anomalies.

Third Normal Form (3NF)/3

- Consider adding a tuple to the above relation:

r

Student	Course	Textbook
'Smith'	'Data Structures'	'Bertram'
'Smith'	'Data Management'	'Martin'
'Hall'	'Compilers'	'Hoffman'
'Brown'	'Data Structures'	'Horowitz'
'Jones'	'Data Structures'	'Bertram'

- Assessment of solution:
  - Cons: The fact that Bertram is a textbook for the Data Structures class is stored twice
  - Pros: $SC \rightarrow T$ and $T \rightarrow C$ can be checked by looking at relation r only (dependency preservation, will be discussed later)

Boyce-Codd Normal Form (BCNF)/1

- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** iff for all $X \rightarrow A \in F^+$ at least one of the following holds:
  - $X \rightarrow A$ is trivial
  - X is a superkey for R

- Intuition: "Each attribute must describe the key, the whole key, and nothing but the key." [Chris Date, adaption of Bill Kent for 3NF]

- A relation that is in BCNF is also in 3NF.

- There exist relations that are in 3NF but not in BCNF

Boyce-Codd Normal Form (BCNF)/2

- The following relations with the functional dependencies $SC \rightarrow T$ and $T \rightarrow C$ are in BCNF:

r1

Course	Textbook
'Data Structures'	'Bertram'
'Data Management'	'Martin'
'Compilers'	'Hoffman'
'Data Structures'	'Horowitz'

r2

Student	Textbook
'Smith'	'Bertram'
'Smith'	'Martin'
'Hall'	'Hoffman'
'Brown'	'Horowitz'
'Jones'	'Horowitz'

- $T \rightarrow C$ is OK since T is a candidate key.
- $SC \rightarrow T$ is not considered since it uses attributes from different relations (a functional dependency is a constraint between two sets of attributes in a single relation).

Boyce-Codd Normal Form (BCNF)/3

- With BCNF less redundancy exists but it is no longer possible to check all functional dependencies by looking at one relation only.

r1

Course	Textbook
'Data Structures'	'Bertram'
'Data Management'	'Martin'
'Compilers'	'Hoffman'
'Data Structures'	'Horowitz'

r2

Student	Textbook'
'Smith'	'Bertram'
'Smith'	'Martin'
'Hall'	'Hoffman'
'Brown'	'Horowitz'
'Jones'	'Horowitz'
'Jones'	'Bertram'

- Assessment of solution:
  - Pros: No information is stored redundantly
  - Cons: The fact that Jones uses two textbooks for the Data Structures class and therefore $SC \rightarrow T$ does not hold cannot be checked without joining the relations

Review 5.6

Relation R satisfies BCNF:

r

A	B	C
'a1'	'b1'	'c1'
'a1'	'b2'	

Assume we know that the functional dependency $A \rightarrow C$ holds. What value can we infer for the value that is missing?

Properties of Decompositions and Normalization Algorithm

- Dependency Preservation
- Lossless Join Decomposition
- BCNF Normalization Algorithm

Multiple Relations

- Relational database design by decomposition:
  - Universal Relation Schema: A relation schema $R(A_1, A_2, \ldots, A_n)$ that includes all attributes of the database.
  - Decomposition: decompose the universal relation schema R into a set of relation schemas $D = R1, R2, \ldots, Rm$ by using the functional dependencies.
  - Additional conditions:
    - Each attribute in R will appear in at least one relation schema Ri in the decomposition so that no attributes are lost.
    - Have each individual relation Ri in the decomposition D in 3NF (or higher).
    - **Lossless join decomposition**: ensures that the decomposition does not introduce wrong tuples when relations are joined together.
    - **Dependency preservation**: ensures that all functional dependency can be checked by considering individual relations Ri only.

Dependency Preservation/1

- Given a set of dependencies F on R, the **projection** of F on Ri, denoted by $F|Ri$ where $attr(Ri)$ is a subset of $attr(R)$, is the set of dependencies $X \to Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in $attr(R_i)$.
- Hence, the projection of F on each relation schema Ri in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand side attributes are in $attr(Ri)$.

Dependency Preservation/2

- Dependency Preservation:
  - A decomposition $D = R1, R2, \ldots, Rm$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each Ri in D is equivalent to F; that is

$$(F|R1 \cup \ldots \cup F|Rm)^+ = F^+$$

- It is always possible to find a dependency-preserving decomposition such that each relation is in 3NF.
- It is not always possible to find a dependency-preserving decomposition such that each relation is in BCNF.

Review 5.7

Consider $R(A, B, C, D)$ and $F = \{A \to B, B \to C, C \to D, D \to A, A \to D\}$. Is the decomposition $R1(A, B)$, $R2(B, C)$, and $R3(C, D)$ dependency preserving?

Lossless Join Decomposition

- A decomposition $D = R1, R2, \ldots, Rm$ of R is a **lossless join decomposition** with respect to the set of dependencies F on R if, for every relation instance r of R that satisfies F, the following holds:

$$\pi_{R1}(r) \bowtie \ldots \bowtie \pi_{attr(Rm)}(r) = r$$

- Note: The word loss in lossless refers to loss of information, not to loss of tuples. If a join decomposition is not lossless then new spurious tuples are present in the result of the join.
- R1 and R2 form a lossless join decomposition of R with respect to a set of functional dependencies F iff
  - $(R1 \cap R2) \to (R1 - R2)$ is in F^+ or
  - $(R1 \cap R2) \to (R2 - R1)$ is in F^+

Review 5.8

Consider $R(A, B, C)$, $F = \{AB \rightarrow C, C \rightarrow B\}$, $R1(A, C)$, $R2(B, C)$.

1. Is $R1, R2$ a lossless decomposition of R?
2. Illustrate your answer for $r = \{(x, 0, a), (y, 2, b), (z, 1, c), (x, 2, c)\}$.
3. Discuss what happens if we replace tuple $(x, 2, c)$ by $(x, 2, b)$.

Algorithm for BCNF Normalization/1

teach

Student	Course	Textbook
'Smith'	'Data Structures'	'Bertram'
'Smith'	'Data Management'	'Martin'
'Hall'	'Compilers'	'Hoffman'
'Brown'	'Data Structures'	'Horowitz'

- Three possible decompositions for relation *teach*
  - (Student, Textbook) and (Student, Course)
  - (Course, Textbook) and (Course, Student)
  - (Textbook, Course) and (Textbook, Student)
- All three decompositions will loose fd1 ($SC \rightarrow T$).
  - We have to settle for sacrificing dependency preservation. We cannot sacrifice the lossless join decomposition.
- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join (and, thus, is lossless).

Algorithm for BCNF Normalization/2

```
Set D := { R };
while a relation schema Q in D is not in BCNF do
    find a functional dependency X → Y in Q that violates BCNF;
    replace Q in D by two relation schemas (Q − Y) and (X ∪ Y);
```

Assumption: No null values are allowed for the join attributes.

- The result is a lossless join decomposition of R.
- The resulting schemas do not necessarily preserve all dependencies.

Review 5.9

Consider $R(Course, Teacher, Hour, Room, Student, Grade)$ and the following functional dependencies:

- $C \rightarrow T$ each course has only one teacher
- $HR \rightarrow C$ one course in one room at one time
- $HT \rightarrow R$ a teacher can only teach in one room at one time
- $CS \rightarrow G$ students get one grade in one course
- $HS \rightarrow R$ students can be in one room at one time

Decompose the schema into a lossless BCNF.

Discussion of BCNF Normalization

- It is valuable to construct a good schema that is in BCNF.
- The normalization process gives important insights into the properties of the data.
- A potential difficulty is that the database designer must first specify *all* the relevant functional dependencies among the database attributes.
- The normalization algorithms are **not deterministic** in general (e.g., not a unique minimal cover).
- It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF.

3NF versus BCNF

- It is possible to construct a decomposition that is in BCNF and is lossless
- It is possible to construct a decomposition that is in 3NF, is lossless, and preserves dependency.
- It is not always possible to construct a decomposition that is in BCNF, is lossless, and is dependency preserving.

- 3NF allows redundancies that BCNF does not allow.
- BCNF cannot check all functional dependencies efficiently since multiple relations must be considered for the check.
- The application needs to determine if a BCNF or 3NF decomposition should be chosen.

Multivalued Dependencies

- Definition
- Fourth Normal Form (4NF)

Multivalued Dependencies/1

Definition:

- A **multivalued dependency (MVD)** $X \twoheadrightarrow Y$ on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation instance r of R:

  If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote $(R - (X \cup Y))$:
  - $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
  - $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
  - $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

- A MVD $X \twoheadrightarrow Y$ for R is called a **trivial MVD** if $Y \subset X$ or $X \cup Y = attr(R)$.

Review 5.10

Consider schema $R(Brand, Product, Country)$. Show an instance that represents the following facts:

- Nike produces shoes and socks
- Nike produces in Taiwan and China
- Ecco produces shoes
- Ecco produces in Denmark and China

Determine the multivalued dependencies on the resulting instance. How must the instance be changed so that the multivalued dependency no longer holds?

Multivalued Dependencies/2

Inference Rules for Functional and Multivalued Dependencies:

- reflexivity FDs: $X \supseteq Y \models X \rightarrow Y$.
- augmentation FDs: $X \rightarrow Y \models XZ \rightarrow YZ$.
- transitivity FDs: $X \rightarrow Y, Y \rightarrow Z \models X \rightarrow Z$.
- complementation: $X \twoheadrightarrow Y \models X \twoheadrightarrow (R - (X \cup Y))$.
- augmentation MVDs: $X \twoheadrightarrow Y, W \supseteq Z \models WX \twoheadrightarrow YZ$.
- transitivity MVDs: $X \twoheadrightarrow Y, Y \twoheadrightarrow Z \models X \twoheadrightarrow (Z - Y)$.
- replication: $X \rightarrow Y \models X \twoheadrightarrow Y$.
- coalescing: $X \twoheadrightarrow Y, \exists W(W \cap Y = \emptyset, W \rightarrow Z, Y \supseteq Z) \models X \rightarrow Z$.

Fourth Normal Form (4NF)/1

Definition:

- A relation schema R with a set of functional and multivalued dependencies F is in **4NF** iff, for every multivalued dependency $X \twoheadrightarrow Y$ in F^+ at least one of the following holds:
  - $X \twoheadrightarrow Y$ is trivial
  - X is a superkey for R
- F^+ is called the **closure** of F and is the complete set of all dependencies (functional or multivalued) that will hold in every relation state r of R that satisfies F.

Fourth Normal Form (4NF)/2

Example of decomposing a relation that is not in 4NF:

1. Relation emp is not in 4NF.
2. Relations *emp_projects* and *emp_dependents* are in 4NF.

emp

EName	PName	DName
'Smith'	'X'	'John'
'Smith'	'Y'	'Anna'
'Smith'	'X'	'Anna'
'Smith'	'Y'	'John'
'Brown'	'W'	'Jim'
'Brown'	'X'	'Jim'
'Brown'	'Y'	'Jim'
'Brown'	'Z'	'Jim'
'Brown'	'W'	'Joan'
'Brown'	'X'	'Joan'
'Brown'	'Y'	'Joan'
'Brown'	'Z'	'Joan'
'Brown'	'W'	'Bob'
'Brown'	'X'	'Bob'
'Brown'	'Y'	'Bob'
'Brown'	'Z'	'Bob'

emp_projects

EName	PName
'Smith'	'X'
'Smith'	'Y'
'Brown'	'W'
'Brown'	'X'
'Brown'	'Y'
'Brown'	'Z'

emp_dependents

EName	DName
'Smith'	'John'
'Smith'	'Anna'
'Brown'	'Jim'
'Brown'	'Joan'
'Brown'	'Bob'

Fourth Normal Form (4NF)/3

- If a relation is not in 4NF because of the MVD $X \twoheadrightarrow Y$ we decompose R into $R1(X \cup Y)$ and $R2(R - Y)$.
- Such a decomposition is lossless.
- R1 and R2 form a lossless join decomposition of R with respect to a set of functional and multivalued dependencies iff
  - $(R1 \cap R2) \twoheadrightarrow (R1 - R2)$ or
  - $(R1 \cap R2) \twoheadrightarrow (R2 - R1)$

Summary/1

- Relational database design goal: eliminate redundancy
- Main concept: functional dependencies
- Functional Dependencies (FDs)
  - Definition
  - Armstrong's inference rules: reflexivity, augmentation, transitivity
  - equivalence of sets of FDs
  - minimal sets of FDs
- Approach: Start with all attributes in a single relation and decompose it vertically until all functional dependencies are acceptable

Summary/2

- Normal forms based on candidate keys and FD
  - 1NF, 2NF, 3NF, BCNF
- BCNF normalization algorithm
- Dependency Preservation
  - always possible for 3NF; not always possible for BCNF
- Lossless Join Decomposition
  - always required
- Multivalued dependencies, 4NF

Database Systems
Spring 2017

Conceptual Database Design
SL06

- Entities, Entity Types, Entity Sets, Attributes
- Relationships, Relationship Types, Relationship Sets
- Weak Entities, N-ary Relationships
- Subclasses and Superclasses
- ER-to-Relational Mapping Algorithm

Literature and Acknowledgments

Reading List for SL06:

- Fundamentals of Database Systems, Chapters 7 and 8, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010. [optional: EWD696, EWD1036]

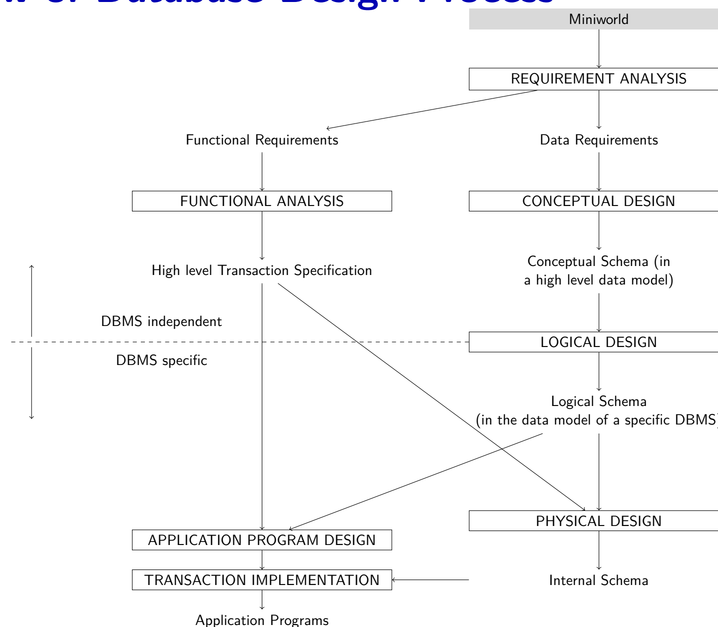
These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Fundamentals of Database Systems, Fourth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Overview of Database Design Process



Example COMPANY Database

- We want to create a database schema design based on the following **requirements** of the COMPANY database:
  - The company is organized into departments. Each department has a name, number and an employee who manages the department. We keep track of the start date of the department manager. A department may have several locations.
  - Each department controls a number of projects. Each project has a unique name, unique number and is located at a single location.
  - We store each employee's social security number, address, salary, sex, and birthdate. Each employee works for one department but may work on several projects. We keep track of the number of hours per week that an employee currently works on each project. We also keep track of the direct supervisor of each employee.
  - Each employee may have a number of dependents. For each dependent, we keep track of their name, sex, birthdate, and relationship to the employee.

Entities and Attributes

- Entities, entity types, entity sets
- Attributes

Entities and Attributes

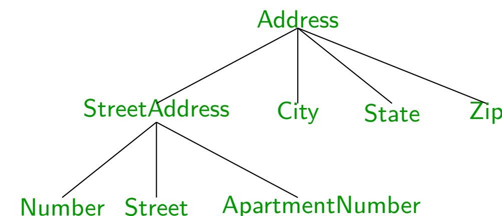
- **Entities** are specific objects or things in the mini-world that are represented in the database.
  - For example employee John Smith, the Research department, project ProductX
- **Attributes** are properties used to describe an entity.
  - For example an employee entity may have the attributes Name, SSN, Address, Sex, BirthDate
- A specific entity will have a value for each of its attributes.
  - For example a specific employee entity may have
    Name = 'John Smith',
    SSN = '123456789',
    Address = '731, Fondren, Houston, TX',
    Sex = 'M',
    BirthDate = '09-JAN-55'
- Each attribute has a *domain* (value set, data type) associated with it, e.g., enumerated type, integer, string, subrange, . . .

Types of Attributes/1

- **Simple attribute**
  - Each entity has a single atomic value for the attribute. For example, SSN or Sex.
- **Composite attribute**
  - The attribute may be composed of several components. For example:
    - Address(Apt#, House#, Street, City, State, ZipCode, Country), or
    - Name(FirstName, MiddleName, LastName).
- **Multi-valued attribute**
  - An entity may have multiple values for that attribute. For example, colors of a car or degrees of a student.
    - Denoted as {Color} or {PreviousDegree}.
- **Derived attribute**
  - Attributes can be derived (computed) rather than stored. Attribute NumEmployees is a derived attribute.

Types of Attributes/2

- Composite and multi-valued attributes may be nested arbitrarily to any number of levels.
- For example, PreviousDegree of a student is a composite multi-valued attribute denoted by {PreviousDegree(College, Year, Degree, Field)}
  - Multiple PreviousDegree values can exist
  - Each has four subcomponent attributes:
    - College, Year, Degree, Field
- A hierarchy of composite attributes:

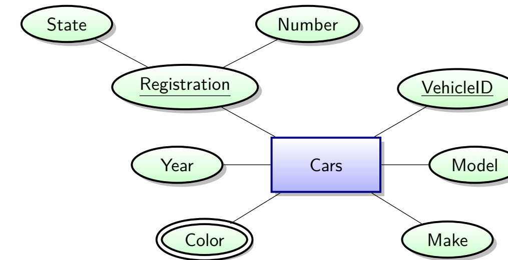


Entity Types and Key Attributes

- Entities with the same basic attributes are grouped or typed into an **entity type**.
  - For example, the entity types Employees and Projects.
- An attribute of an entity type for which each entity must have a unique value is called a **key attribute** of the entity type.
  - For example, SSN of Employees.
- A key attribute may be composite.
  - VehicleTagNumber is a key of the Cars entity type with components (Number, State).
- An entity type may have more than one key.
  - The Cars entity type may have two keys:
    - VehicleIdentificationNumber (popularly called VIN)
    - VehicleTagNumber(Number, State), aka license plate number.
- Each key is underlined.

Displaying an Entity Type

- In ER diagrams, an **entity type** is displayed in a rectangular box
- Attributes are displayed in ovals
  - Each attribute is connected to its entity type
  - Components of a composite attribute are connected to the oval representing the composite attribute
  - Each key attribute is underlined
  - Multivalued attributes are displayed in double ovals
  - Derived attributes are displayed in dashed ovals
- Entity type Cars with attributes Registration(Number, State), VehicleID, Make, Model, Year, {Color}



Review 6.1

Consider schema $R(A, B, C, D, E)$ with candidate keys $\{A, BE, C\}$. Represent R as an entity type.

Entity Set

- The collection of all entities of a particular entity type in the database is called the **entity set**.
- The same name (e.g., Cars) is used to refer to both the entity type and the entity set.
- The entity set is the current *state* of the entities of that type that are stored in the database.
- Entity set Cars:

Car_1
((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 2004, {red, black})

Car_2
((ABC 123, NEW YORK), WP9872, Nissan Maxima, 4-door, 2005, {blue})

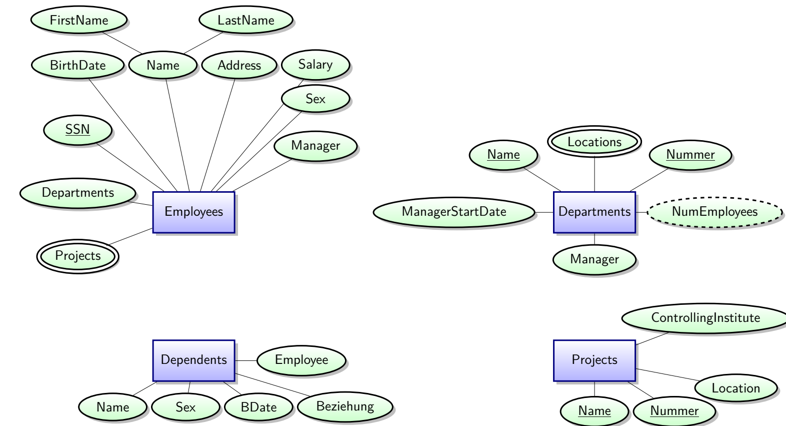
Car_3
((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4-door, 2002, {white, blue})

.
.
.

Entity Types for the COMPANY Database/1

- Based on the requirements, we can identify four initial entity types in the COMPANY database:
  - Departments
  - Projects
  - Employees
  - Dependents
- The initial attributes are derived from the requirements description
- Guidelines for determining entity types and attributes:
  - The *nouns* in a descriptions translate to entity types
    - "organized into departments"
  - *Nouns that describe nouns* of entity types translate to attributes
    - "department has a name, number"

Entity Types for the COMPANY Database/2



Relationships

- Relationships, Relationship Types, Relationship Sets
- Structural Constraints on Relationships
- Recursive Relationships
- Attributes of Relationships

Relationships and Relationship Types/1

- ER model has three main concepts:
  - Entities (and their entity types and entity sets)
  - Attributes (simple, composite, multivalued)
  - Relationships (and their relationship types and relationship sets)
- The initial design is typically not complete.
- The schema design process is an **iterative refinement process**. The initial design is created and iteratively refined.
- Some aspects in the requirements will be represented as **relationships**.

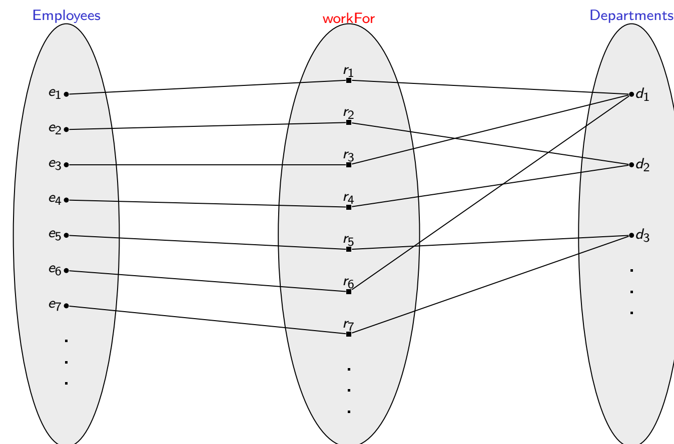
Relationships and Relationship Types/2

- A **relationship** relates two or more distinct entities with a specific meaning. For example,
  - employee John Smith *works on* project ProductX, or
  - employee Franklin Wong *manages* the research department.
- Relationships of the same type are grouped or typed into a **relationship type**.
  - For example, the workOn relationship type in which employees and projects participate, or the manage relationship type in which employees and departments participate.
- The degree of a relationship type is the number of participating entity types.
  - Both manage and workOn are *binary* relationships.
- In ER diagrams, we represent a *relationship type* as follows:
  - Diamond-shaped box is used to display a relationship type
  - Connected to the participating entity types via straight lines

Relationships and Relationship Types/3

- Relationship type:
  - Schema description of a relationship
  - Identifies the relationship name and the participating entity types
  - Identifies relationship constraints
- **Relationship set**:
  - The current set of relationship instances represented in the database
  - The current *state* of a relationship type
- Each instance in the set relates individual participating entities – one from each participating entity type

The workFor Relationship

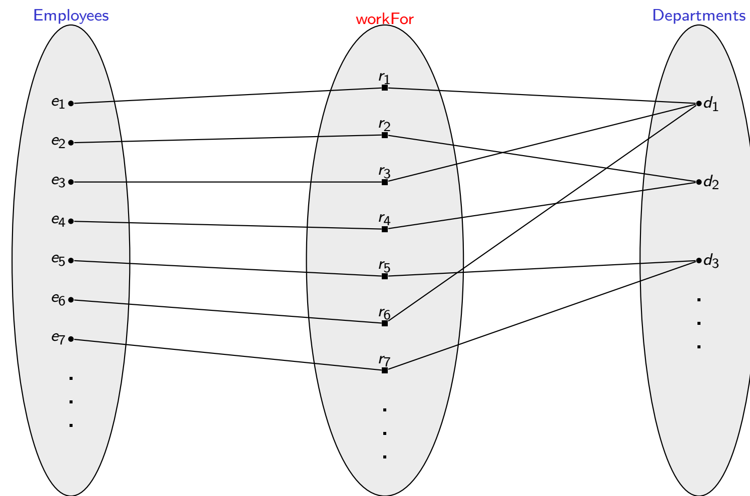


- entity: e_1
- relationship: $r_1 =$ workFor(e_1,d_1)
- entity set: Employees
- relationship set: workFor

Structural Constraints on Relationships

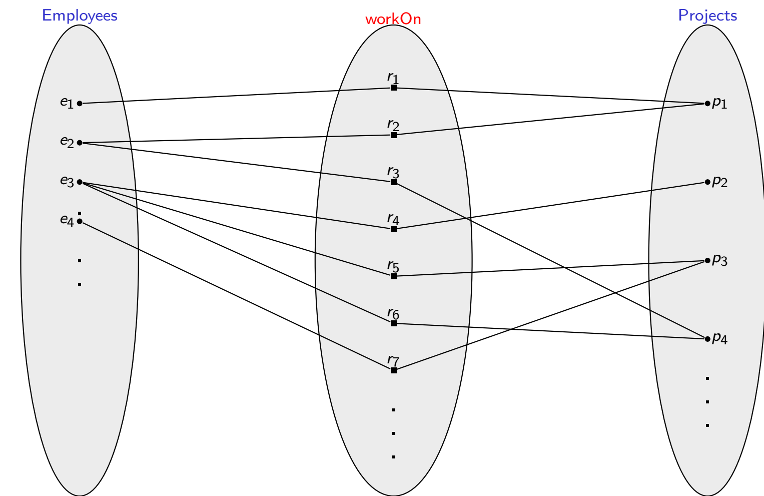
- Structural constraints on relationship types limit the possible combinations of entities in relationship sets.
- A **cardinality constraint** specifies *maximum* participation
  - One-to-one (1:1)
  - One-to-many (1:N) or Many-to-one (N:1)
  - Many-to-many (M:N)
- A **participation constraint** specifies *minimum* participation (also called existence dependency)
  - zero (optional participation, not existence-dependent)
  - one or more (mandatory participation, existence-dependent)

Many-to-one (N:1) Relationship



- Employees:Departments = N:1
- An employee works for at most 1 department
- A department employs at most N employees

Many-to-many (M:N) Relationship

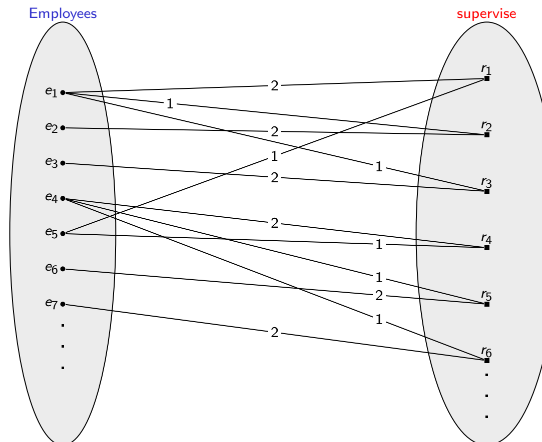


- Employees:Projects = M:N
- An employee works on at most N projects
- A project is assigned to at most M employees

Recursive Relationships/1

- In a recursive relationship type:
  - The same entity type participates in different roles.
  - For example, the supervise relationships between
    - an employee in role of supervisor (or boss) and
    - another employee in role of subordinate (or worker).
- In ER diagrams we need to display role names to distinguish participations.

Recursive Relationships/2



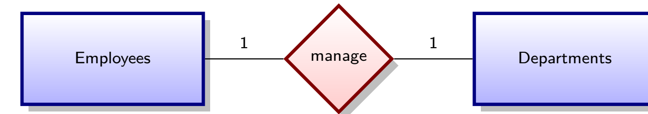
- label 1 stands for *supervisor* role
- label 2 stand for *subordinate* role
- e_1 is the supervisor of e_2
- e_1 is supervised by e_5

Attributes of Relationship Types

- A relationship type can have attributes:
  - For example, HoursPerWeek of workOn
  - Its value for each relationship instance describes the number of hours per week that an employee works on a project.
    - A value of HoursPerWeek depends on a particular (employee, project) combination
  - Most relationship attributes are used with M:N relationships
    - In 1:N relationships, they can be transferred to the entity type on the N-side of the relationship

Notation for Constraints on Relationships/1

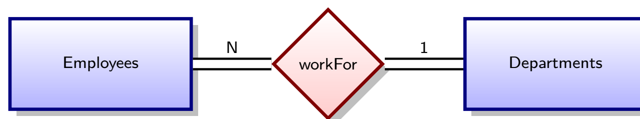
- **Cardinality constraints** (of a binary relationship): 1:1, 1:N, N:1, or M:N
- Cardinality constraints are shown by placing appropriate numbers on the relationship edges.



- Reading is from left-to-right and top-down (usually; there are always exceptions) and the terms are chosen accordingly.
- Reading: An employee manages 1 department
- Inverse: A department is managed by 1 employee

Notation for Constraints on Relationships/2

- **Participation constraint** on each participating entity type: total (called existence dependency) or partial.
- Total participation is shown by double line, partial participation by single line.

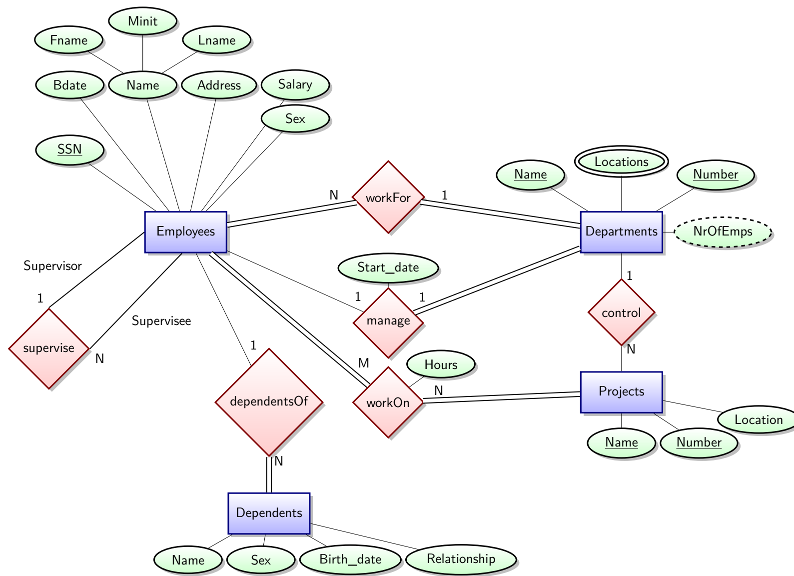


- A total participation constraint specifies a minimum participation:
  - An employee must workFor a department
  - A department must giveWorkTo N employees
- Structural constraints are easy to specify for binary relationship types but get more subtle for higher order relationship types.

Relationships for the COMPANY Database/1

- By examining the requirements, six relationship types are identified
- All are *binary* relationships (degree 2)
- Relationship types with participating entity types:
  - workFor (between Employees and Departments)
  - manage (also between Employees and Departments)
  - control (between Departments and Projects)
  - workOn (between Employees and Projects)
  - supervise (between Employees (as subordinate) and Employees (as supervisor))
  - dependentsOf (between Employees and Dependents)

Relationships for the COMPANY Database/2



Discussion of Relationship Types

- In the refined design, some attributes from the initial entity types are refined into relationships:
  - Manager of Departments -> manage
  - Projects of Employees -> workOn
  - Department of Employees -> workFor
  - etc
- More than one relationship type can exist between the same participating entity types:
  - manage and workFor are distinct relationship types between Employees and Departments.
  - These relationship types have different meanings and different relationship instances.

Review 6.2/1

Assume a company employs agents to sell products to customers. Agents are based in a city and they work on commission. Customers negotiate individual discounts. For products we record production location, price, and sales quantity. Design an appropriate ER schema.

Review 6.2/2

Review 6.2/3

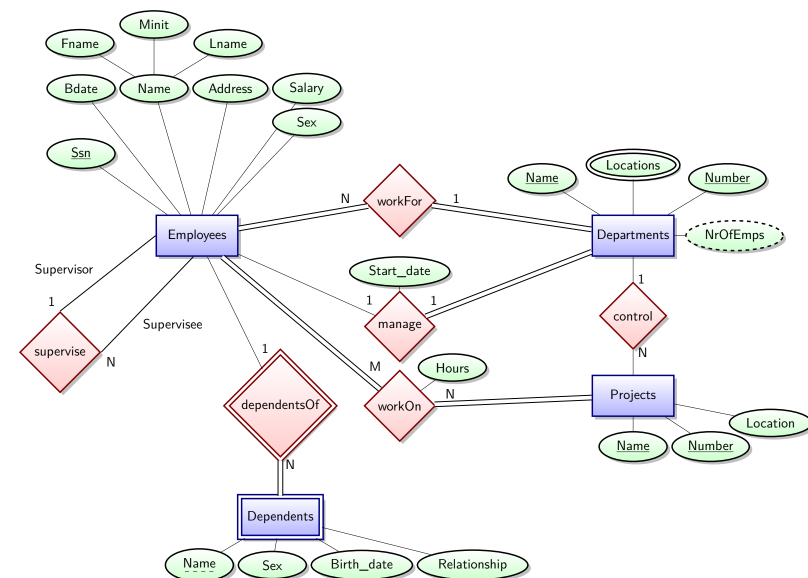
Weak Entities and N-ary Relationships

- Weak Entities
- N-ary Relationships

Weak Entity Types

- A **weak entity type** is an entity type that does not have a key attribute.
- A weak entity must participate in an **identifying relationship type** with an owner or identifying entity type.
- Entities are identified by the combination of:
  - A partial key of the weak entity type
  - The particular entity they are related to in the identifying entity type
- Example:
  - A Dependent entity is identified by the dependent's first name, and the specific Employee with whom the dependent is related.
  - Name of Dependent is the *partial key*
  - Dependent is a *weak entity type*
  - Employee is its identifying entity type via the identifying relationship type dependentOf.

The COMPANY Database



Review 6.3

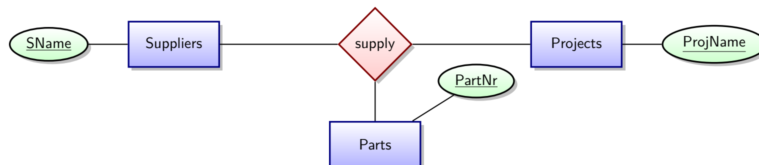
Construct an ER schema for a hospital with patients and medical doctors. For each patient we keep a log of various tests and examinations.

Review 6.4

Assume we want to schedule classrooms for the final exams at a university. The examination unit are sections of courses. Propose an ER diagram.

Relationships of Higher Degree/1

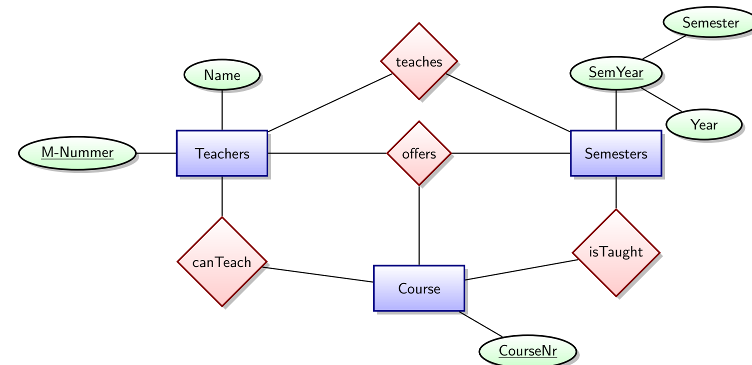
- Relationship types of degree 2 are called binary.
- Relationship types of degree 3 are called ternary and of degree n are called n-ary.



- Constraints are harder to specify for higher-degree relationships (n>2) than for binary relationships.
- In an n-ary relationship a cardinality constraint specifies how often an entity may occur for one specific instance of all other entities.
- In general, an n-ary relationship is not equivalent to n binary relationships.
- If needed, the binary and n-ary relationships can all be included in the schema design.

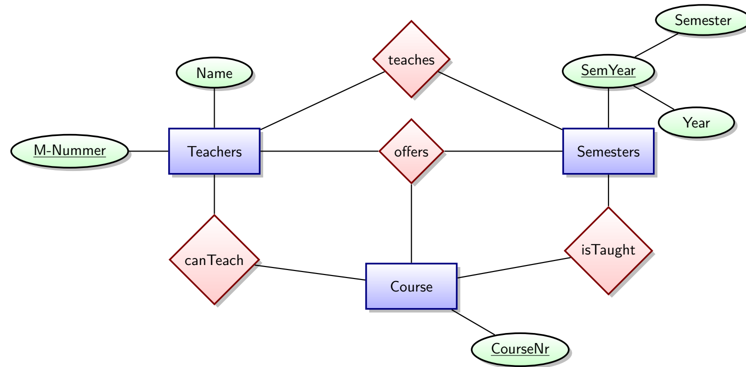
Relationships of Higher Degree/2

- An n-ary relationship is not equivalent to n binary relationships. The schema can include binary as well as n-ary relationships.
- The relationship offers cannot be derived from teaches, canTeach, isTaught.



Relationships of Higher Degree/3

- If a particular binary relationship can be derived from a higher-degree relationship at all times, then it is *redundant*.
- The teaches binary relationship can be derived from the ternary relationship offers (based on the meaning of the relationships).

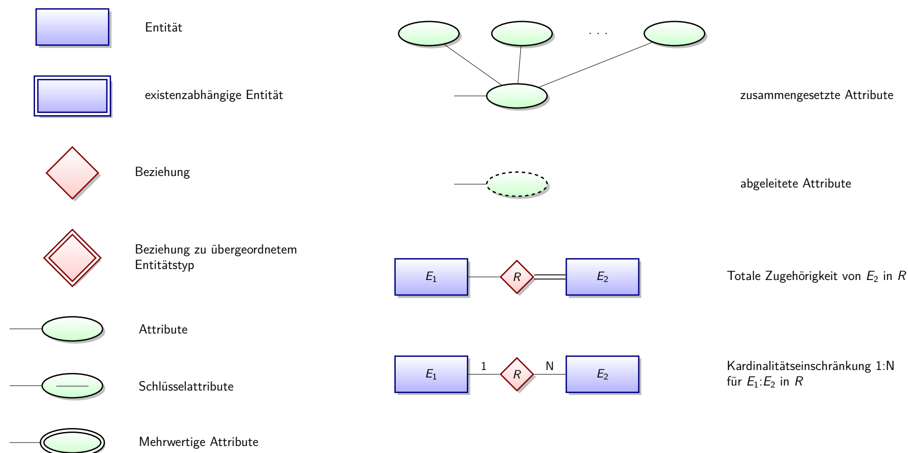


Alternative Diagrammatic Notation

- ER diagrams is one popular example for displaying database schemas.
- Many other notations exist in the literature and in various database design and modeling tools.
- UML class diagrams are another way of displaying ER concepts.

- Choose a systematic naming (there is no standard), e.g.,
  - Plural names for entity types
  - Upper case for entity type and lower case for relationship type
  - Initial letter capitalized for attribute names

Summary of Notation for ER Diagrams



Data Modeling Tools

- A number of popular tools that conceptual modeling and mapping into relational schema design.
  - Examples: ERWin, S-Designer (Enterprise Application Suite), ER-Studio, etc.
- Pros:
  - Serves as documentation of application requirements
  - Easy user interface: mostly graphics editor support
  - Simple graphical models are very intuitive
- Cons:
  - Graphical models easily get complex and ambiguous
  - Mostly represent a relational design in a diagrammatic form rather than a conceptual ER-based design

Subclasses and Superclasses

- Sub- and Superclasses
- Disjointness Constraint (disjoint, overlapping)
- Completeness Constraint (total, partial)

Subclasses and Superclasses/1

- An important extension that was not present in the initial ER model are subgroupings.
- An entity type may have additional meaningful subgroupings of its entities
- Example: Employees may be further grouped into:
  - Secretaries, Engineers, Technicians, ...
    – Based on the job of an employee
  - Managers
    – Employees who are managers
  - SalariedEmps, HourlyEmps
    – Based on the method of pay
- Extended ER diagrams represent these additional subgroupings, called *subclasses*.

Subclasses and Superclasses/2

- Each subgrouping is a subset of Employees
- Each subgrouping is a subclass of Employees
- Employees is the superclass for each of these subclasses
- These are called superclass/subclass relationships:
  - Employees/Secretaries
  - Employees/Technicians
  - Employees/Managers
  - ...
- Superclass/subclass relationships are also called IS-A relationships
  - Secretaries IS-A Employees
  - Technicians IS-A Employees
  - ...

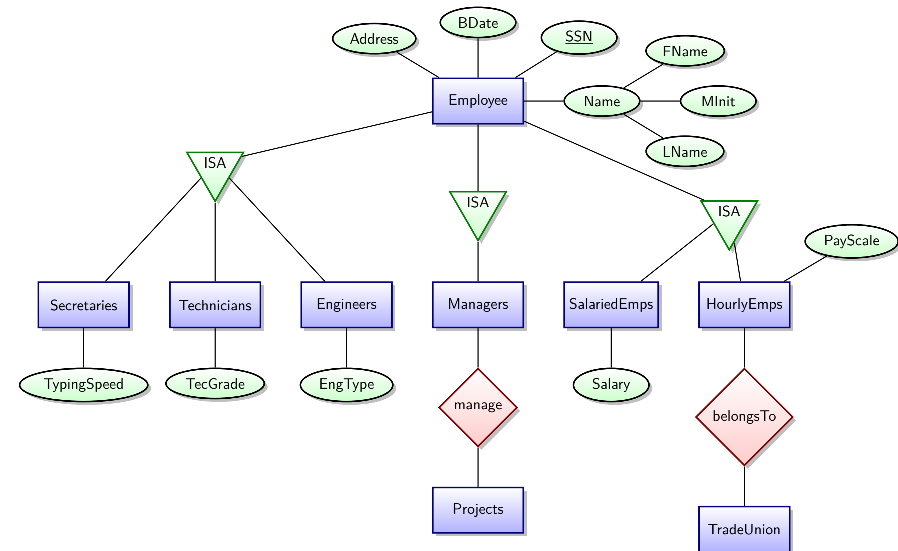
Subclasses and Superclasses/3

- An entity that is member of a subclass represents the same real-world entity as some member of the superclass:
  - The subclass member is the same entity in a *distinct specific role*
  - An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass
- A member of the superclass can be optionally included as a member of any number of its subclasses
- Examples:
  - A salaried employee who is also an engineer belongs to two subclasses:
    - Engineers, and
    - SalariedEmps

Attribute Inheritance

- An entity that is member of a subclass *inherits*
  - All attributes of the entity as a member of the superclass
  - All relationships of the entity as a member of the superclass
- Example:
  - Secretaries (as well as Technicians and Engineers) inherit the attributes Name, SSN, ..., from Employees
  - Every Secretaries entity will have values for the inherited attributes

Example of Subclasses and Superclasses



Specialization and Generalization

- Specialization is the process of defining a set of subclasses of a superclass
- The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass
- Example: SECRETARY, ENGINEER, TECHNICIAN is a specialization of EMPLOYEE based upon *job type*.
- We may have several specializations of the same superclass
- Generalization is the reverse of the specialization process
- Several classes with common features are generalized into a superclass.
- Example: CAR, TRUCK generalized into VEHICLE;
  - CAR, TRUCK become subclasses of the superclass VEHICLE.
  - We can view {CAR, TRUCK} as a specialization of VEHICLE
  - Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK

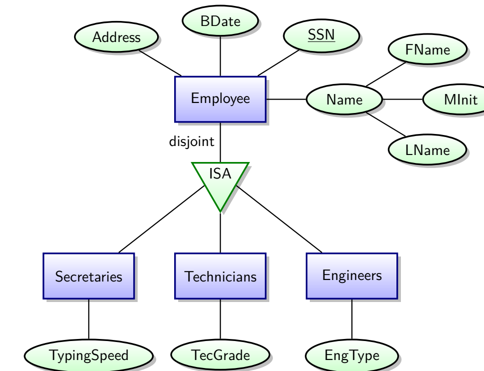
Constraints on Specialization/1

- Two basic constraints can apply to a specialization/generalization:
  - Disjointness Constraint
  - Completeness Constraint
- **Disjointness Constraint**:
  - Specifies that the subclasses of the specialization must be *disjoint*:
    - an entity can be a member of at most one of the subclasses of the specialization
    - Annotate edge in ER diagram with *disjoint*
  - If not disjoint, specialization is *overlapping*:
    - that is the same entity may be a member of more than one subclass of the specialization
    - Annotate edge in ER diagram with *overlapping* (or no annotation)

Constraints on Specialization/2

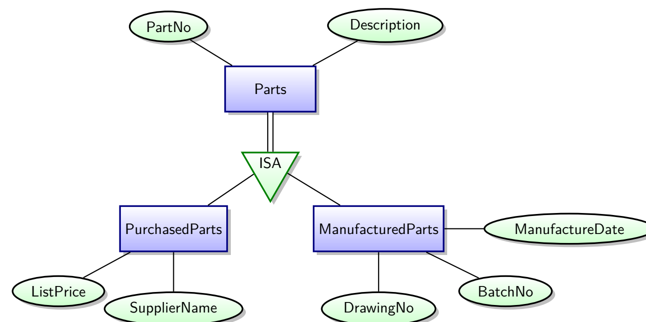
- **Completeness Constraint**:
  - *Total* specifies that every entity in the superclass must be a member of some subclass in the specialization/generalization
  - Shown in ER diagrams by a **double line**
  - *Partial* allows an entity not to belong to any of the subclasses
  - Shown in ER diagrams by a single line
- Hence, we have four types of specialization/generalization:
  - Disjoint, total
  - Disjoint, partial
  - Overlapping, total
  - Overlapping, partial

Example of Disjoint Partial Specialization



- An employee my be neither a secretary nor a technician nor an engineer (partial specialization).
- An employee cannot be a secretary and technician or secretary and engineer or technician and engineer (disjoint specialization).

Example of Overlapping Total Specialization



- Every part must either be a manufactured part or a purchased part (total specialization).
- A part may be a manufactured part and a purchased part (overlapping specialization).

Review 6.5

Assume employees work on projects. Employees who work on projects are allowed to use machines. Use an ER diagram to model this.

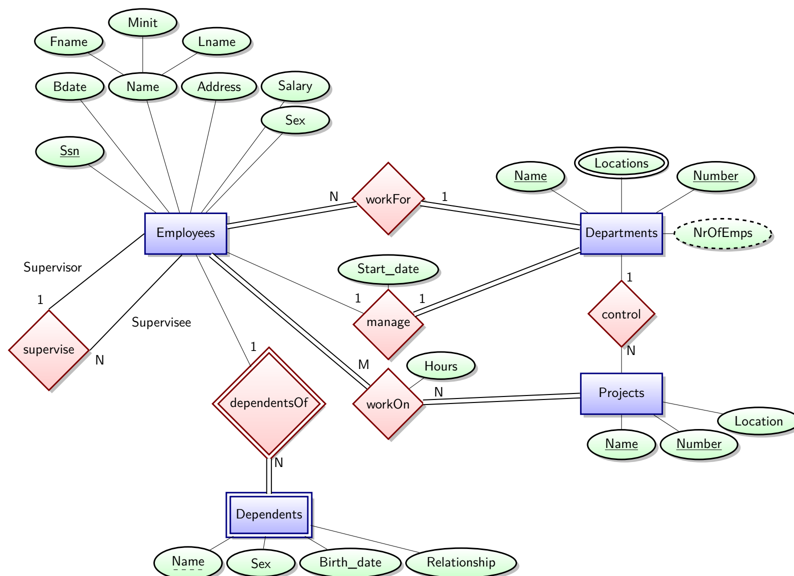
ER-to-Relational Mapping Algorithm

- Mapping an ER Model to a Relational Model

ER-to-Relational Mapping Algorithm/1

- Algorithm to map a conceptual database design to a relational database design.
- ER-to-Relational Mapping Algorithm
  - Step 1: Mapping of Regular Entity Types
  - Step 2: Mapping of Weak Entity Types
  - Step 3: Mapping of Binary 1:1 Relation Types
  - Step 4: Mapping of Binary 1:N Relationship Types
  - Step 5: Mapping of Binary M:N Relationship Types
  - Step 6: Mapping of Multivalued attributes
  - Step 7: Mapping of N-ary Relationship Types
  - Step 8: Mapping Specialization or Generalization

The ER schema for the COMPANY database

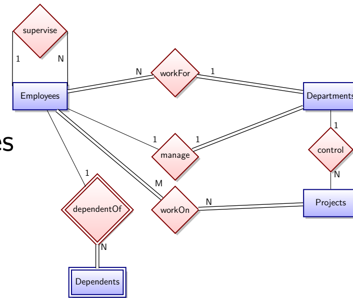


ER-to-Relational Mapping Algorithm/2

- **Step 1: Mapping of Regular Entity Types.**
  - For each regular (strong) entity type E in the ER schema, create a relation schema *R* that includes all the simple attributes of E.
  - A composite attribute is flattened into a set of simple attributes.
  - Choose one of the key attributes of E as the primary key for *R*.
  - If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of *R*.

ER-to-Relational Mapping Algorithm/3

- **Example:** We create relation schemas *Employee*, *Department* and *Project*. . They correspond to the regular entities in the ER diagram.
  - *SSN*, *DNumber*, and *PNumber* are the primary keys for the relations *employee*, *department*, and *project* as shown.



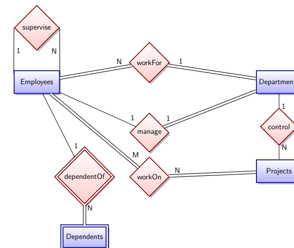
Employee(FName, MInit, LName, <u>SSN</u>, BDate, Address, Sex, Salary)
Department(DName, <u>DNumber</u>)
Project(PName, <u>PNumber</u>, PLocation)

ER-to-Relational Mapping Algorithm/4

- **Step 2: Mapping of Weak Entity Types**
  - For each weak entity type W in the ER diagram with owner entity type E, create a relation schema *R* and include all simple attributes (or simple components of composite attributes) of W as attributes of *R*.
  - Include as foreign key attributes of *R* the primary key attributes of the relations that correspond to the owner entity types.
  - The primary key of *R* is the *combination of* the primary keys of the owners and the partial key of the weak entity type W, if any.

ER-to-Relational Mapping Algorithm/5

- **Example:** Create relation schema *Dependent* for the weak entity type Dependent.
  - Include the primary key *SSN* of *Employee* as a foreign key attribute of *Dependent* (renamed to *ESSN*).
  - The primary key of *Dependent* is the combination {*ESSN*, *DepName*} because *DepName* is the partial key of *Dependent*.



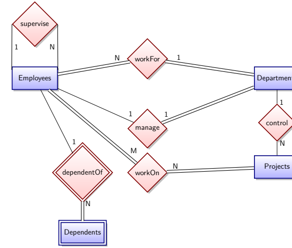
Employee(FName, MInit, LName, <u>SSN</u>, BDate, Address, Sex, Salary)
Department(DName, <u>DNumber</u>)
Project(PName, <u>PNumber</u>, PLocation)
Dependent(<u>ESSN</u>, <u>DepName</u>, Sex, BDate, Relationship)

ER-to-Relational Mapping Algorithm/6

- **Step 3: Mapping of Binary 1:1 Relation Types**
  - For each binary 1:1 relationship type r in the ER schema, identify the relation schemas *S* and *T* that correspond to the entity types participating in r.
- There are three possible approaches:
  1. **Foreign Key approach:** Choose one of the relations, say *S*, and include as foreign key in *S* the primary key of *T*. It is better to choose an entity type with total participation in R in the role of *S*.
  2. **Merged relation option:** An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when both participations are total.
  3. **Cross-reference or relationship relation option:** The third alternative is to set up a third relation schema *R* for the purpose of cross-referencing the primary keys of the two relation schemas *S* and *T* representing the entity types.

ER-to-Relational Mapping Algorithm/7

- **Example**: The 1:1 relation manage is mapped by choosing the participating entity type Department to serve in the role of S, because its participation in the manage relationship type is total.



Employee(FName, MInit, LName, <u>SSN</u>, BDate, Address, Sex, Salary)
Department(DName, <u>DNumber</u>, MgrSSN, MgrStartDate)
Project(PName, <u>PNumber</u>, PLocation)
Dependent(<u>ESSN</u>, <u>DepName</u>, Sex, BDate, Relationship)

Review 6.6

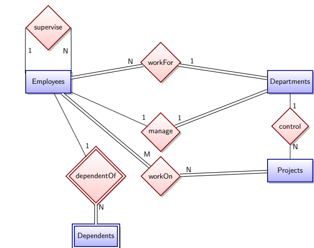
Illustrate the problems that occur if the 1:1 relationship manage is mapped through a foreign key in relation schema *Employee*.

ER-to-Relational Mapping Algorithm/8

- **Step 4: Mapping of Binary 1:N Relationship Types**
  - For each binary 1:N relationship type r, identify the relation schemas S and T that correspond to the entity types participating in r. S is the N-side.
  - Include as foreign key in S the primary key of T that represents the other entity type participating in r.
  - Include simple attributes of the 1:N relation type as attributes of S.

ER-to-Relational Mapping Algorithm/9

- **Example:** 1:N relationship types workFor, control and supervise.
  - For example for workFor we include the primary key *DNumber* of the *Department* relation schema as foreign key in the *Employee* relation schema and call it *DNo*.

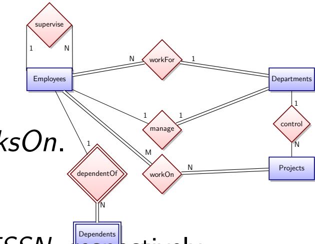


Employee(FName, MInit, LName, <u>SSN</u>, BDate, Address, Sex, Salary, SuperSSN, DNo)
Department(DName, <u>DNumber</u>, MgrSSN, MgrStartDate)
Project(PName, <u>PNumber</u>, PLocation, DNum)
Dependent(<u>ESSN</u>, <u>DepName</u>, Sex, BDate, Relationship)

ER-to-Relational Mapping Algorithm/10

- **Step 5: Mapping of Binary M:N Relationship Types.**
  - For each regular binary M:N relationship type r, *create a new relation schema* S to represent r.
  - Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; *their combination will form the primary key* of S.
  - Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S.

ER-to-Relational Mapping Algorithm/11



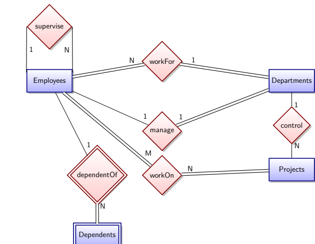
- **Example:** The M:N relationship type worksOn from the ER diagram is mapped by creating a relation schema *WorksOn*.
  - The primary keys of *Project* and *Employee* are included as foreign keys in *WorksOn* and renamed *PNo* and *ESSN*, respectively.
  - Attribute *Hours* in *WorksOn* represents the Hours attribute in the ER schema. The primary key of *WorksOn* is the combination of the foreign key attributes *ESSN*, *PNo*.

Employee(FName, MInit, LName, SSN, BDate, Address, Sex, Salary, SuperSSN, DNo)
Department(DName, DNumber, MgrSSN, MgrStartDate)
Project(PName, PNumber, PLocation, DNum)
Dependent(ESSN, DepName, Sex, BDate, Relationship)
WorksOn(ESSN, PNo, Hours)

ER-to-Relational Mapping Algorithm/12

- **Step 6: Mapping of Multivalued attributes.**
  - For each multivalued attribute A, create a new relation schema R.
  - R includes an attribute A, plus the primary key attribute K of the relation that represents the entity or relationship type that has A as an attribute.
  - The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.

ER-to-Relational Mapping Algorithm/13



- **Example:** The relation *DeptLoc* is created.
  - The attribute *DLocation* represents the multivalued attribute Locations of *Department*, while *DNumber*, as foreign key, represents the primary key of the *Department* relation.
  - The primary key of R is the combination of *DNumber*, *DLocation*.

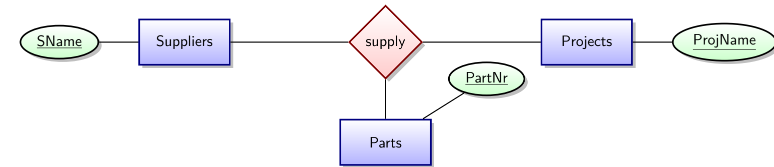
Employee(FName, MInit, LName, SSN, BDate, Address, Sex, Salary, SuperSSN, DNo)
Department(DName, DNumber, MgrSSN, MgrStartDate)
Project(PName, PNumber, PLocation, DNum)
Dependent(ESSN, DepName, Sex, BDate, Relationship)
WorksOn(ESSN, PNo, Hours)
DeptLoc(DNumber, DLocation)

ER-to-Relational Mapping Algorithm/14

- **Step 7: Mapping of N-ary Relationship Types.**
  - For each n-ary relationship type r, where n>2, create a relation schema *S*.
  - Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types.
  - Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of *S*.

ER-to-Relational Mapping Algorithm/15

- **Example:** The relationship type supply



- This can be mapped to the relation schema *Supply* shown in the relational schema, whose primary key is the combination of the three foreign keys *SName*, *PartNo*, *ProjName*.

Supplier(SName, ...)
Project(ProjName, ...)
Part(PartNo, ...)
Supply(SName, ProjName, PartNo, Quantity)

Review 6.7

Use the ER model to model job applications. Candidates submit applications to companies and possibly get invited to an interview.

ER-to-Relational Mapping Algorithm/16

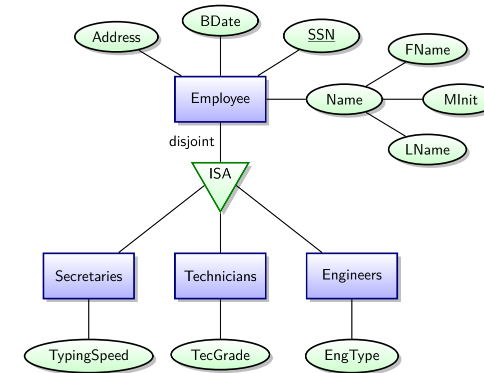
- **Step8: Options for Mapping Specialization or Generalization.**
  - Convert each specialization with m subclasses $S_1, S_2, \ldots, S_m$ and superclass C, where the attributes of C are $k, a_1, \ldots, a_n$ and k is the primary key, into relational schemas using one of the four following options:
    - Option 8A: Multiple relations for superclass and subclasses
    - Option 8B: Multiple relations for subclass relations only
    - Option 8C: Single relation with one type attribute
    - Option 8D: Single relation with multiple type attributes

ER-to-Relational Mapping Algorithm/17

- **Option 8A: Multiple relations for superclass and subclasses**
  - Create a relation schema L for C with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\}$ and $PK(L) = k$.
  - Create a relation schema L_i for each subclass $S_i, 1 < i < m$, with attributes $Attrs(Li) = \{k\} \cup \{attributes\ \ of\ \ S_i\}$ and $PK(L_i) = k$.
  - This option works for any specialization (total or partial, disjoint of overlapping).

ER-to-Relational Mapping Algorithm/18

- **Example:** Specialization of EMPLOYEE



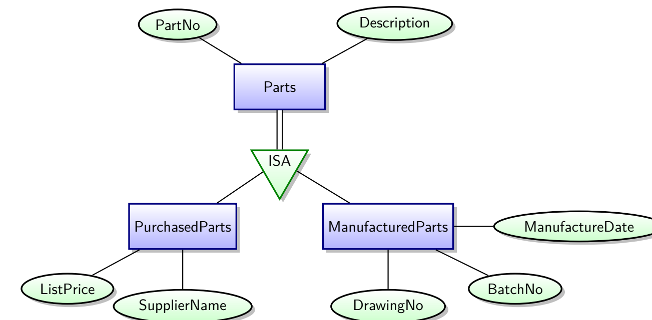
Employee(SSN, FName, MInit, LName, BirthDate, Address, JobType)
Secretary(SSN, TypingSpeed)
Technician(SSN, TGrade)
Engineer(SSN, EngType)

ER-to-Relational Mapping Algorithm/19

- **Option 8B: Multiple relations for subclass relations only**
  - Create a relation L_i for each subclass $S_i, 1 < i < m$, with attributes $Attr(L_i) = \{attributes\ \ of\ \ S_i\} \cup \{k, a_1, \ldots, a_n\}$ and $PK(L_i) = k$.
  - This option only works for a specialization whose subclasses are total (i.e., every entity in the superclass must belong to at least one of the subclasses).

ER-to-Relational Mapping Algorithm/20

- **Example:** Specialization of Parts



Manufactured(PartNo, Desc, DrawNo, BatchNo, ManufDate)

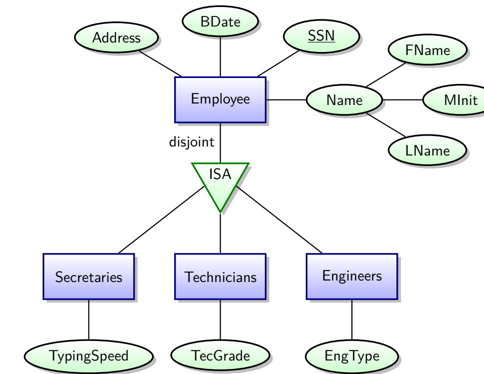
Purchased(PartNo, Desc, SuppName, ListPrice)

ER-to-Relational Mapping Algorithm/21

- **Option 8C: Single relation with one type attribute**
  - Create a single relation L with attributes $Attr(L) = \{k, a_1, \ldots, a_n, t\}$ $\cup$ $\{attributes\ \ of\ \ S_1\}$ $\cup$ $\{attributes\ \ of\ \ S_m\}$ and $PK(L) = k$.
  - The attribute t is called a type attribute that indicates the subclass to which each tuple belongs.
  - This option only works for a specialization whose subclasses are disjoint and might generate many null values for attributes of subclasses.

ER-to-Relational Mapping Algorithm/22

- **Example**: Use *JobType* as a type attribute in *Employee* to distinguish between secretary, technician, and engineer.



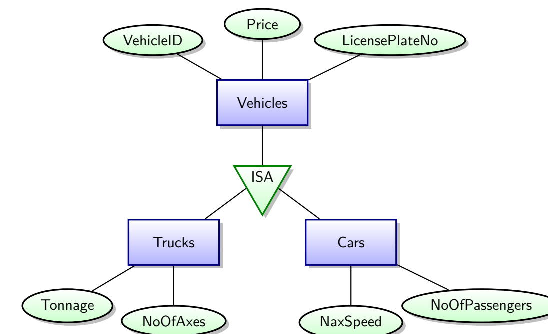
Employee(SSN, FName, MInit, LName,..., JobType, TypingSpeed, TGrade, EngType)

ER-to-Relational Mapping Algorithm/23

- **Option 8D: Single relation with multiple type attributes**
  - Create a single relation L with attributes $Attr(L) = \{k, a_1, \ldots, a_n,$ $t_1, ..., t_m\}$ $\cup$ $\{attributes\ of\ S_1\}$$\cup$ $\{attributes\ \ of\ \ S_m\}$ and $PK(L) = k$.
  - Each t_i is a Boolean attribute that indicates whether a tuple belongs to subclass S_i.
  - This option works for a specialization whose subclasses are overlapping (but will also work for disjoint specialization).

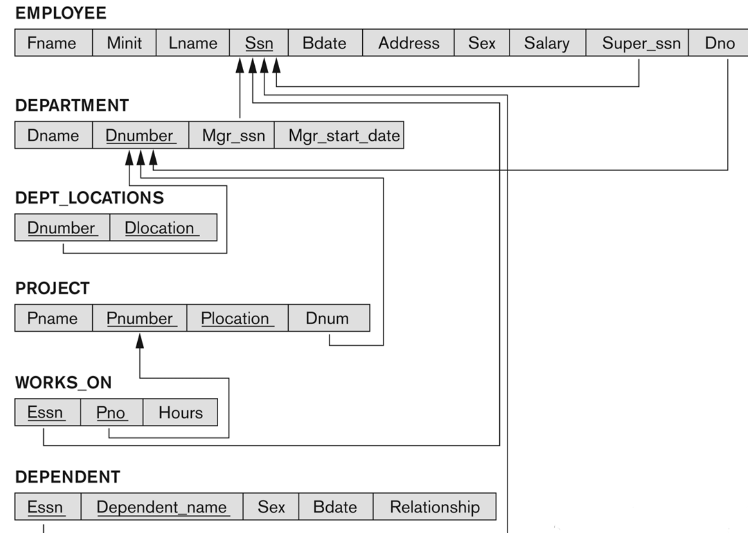
ER-to-Relational Mapping Algorithm/24

- **Example**:



VEHICLE(
    VehicleID, Price, LicensePlateNo,
    CarFlag, MaxSpeed, NoPassengers,
    TruckFlag, NoAxes, Tonnage)

ER-to-Relational Mapping Algorithm/27



Summary of Mapping Constructs and Constraints

Correspondence between ER and Relational Models

ER Model	**Relational Model**
Entity type	Entity relation
1:1 or 1:N relationship type	Foreign key (or relationship relation)
M:N relationship type	Relationship relation and two foreign keys
n-ary relationship type	Relationship relation and n foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Key attribute	Primary (or secondary) key

Summary/1

- ER model concepts:
  - entity type, entity set, entity
  - attribute, attribute value
  - relationship type, relationship set, relationship
  - structural constraints: cardinality, participation
  - subclasses and superclasses
- During the conceptual modeling process a number of **clarifying decisions/assumptions** must be made
- It is important to make all necessary decisions (even if alternatives exist) and go on. **Make decisions explicit** (i.e., write them down).
- There is not a unique ER model.
- In order to clarify the semantics it can hep to consider relation instances

Summary/2

- ER-to-Relational Mapping Algorithm
  - Step 1: Mapping of Regular Entity Types
  - Step 2: Mapping of Weak Entity Types
  - Step 3: Mapping of Binary 1:1 Relation Types
  - Step 4: Mapping of Binary 1:N Relationship Types.
  - Step 5: Mapping of Binary M:N Relationship Types.
  - Step 6: Mapping of Multivalued attributes.
  - Step 7: Mapping of N-ary Relationship Types.
  - Step 8: Mapping Specialization or Generalization.
- The **mapping algorithm is mechanical** and without conceptual difficulty. Make sure you know how to apply it.

Database Systems
Spring 2017

Physical Database Design
SL07

- Disk Storage and Files
  - Physical Storage Media
  - Accessing the Storage
  - Organization of Files
- Index Structures
  - Types of Indexes
  - B+ Tree
  - Hashing
  - Index Definition in SQL

Literature and Acknowledgments

Reading List for SL07:

- Database Systems, Chapters 16 and 17, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Database Systems, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Disk Storage and Files

- Physical Storage Media
- Storage Access
- File Organization

Physical Storage Media/1

- Several types of storage media exist in computer systems and are relevant for DBMS
- The storage media can be organized into a **storage hierarchy**.
- **Classification** of storage media
  - **Speed** with which data can be accessed
  - **Cost** per unit of data
  - **Reliability**
    - data loss on power failure or system crash
    - physical failure of the storage device
  - **Volatile** vs. **non-volatile** storage
    - Volatile storage: Loses contents when power is switched off
    - Non-Volatile storage: Contents persist when power is switched off

Physical Storage Media/2

- **Cache**
  - Volatile
  - Fastest and most costly form of storage
  - Managed by the computer system hardware

- **Main memory**
  - Volatile
  - Fast access (x0 to x00 of nanosecs; 1 nanosec $= 10^{-9}$ secs)
  - Generally only a part of a database is loaded into memory
    - Capacities of up to a few Gigabytes (or even Terabytes) widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
    - If entire database is kept in memory we have a main memory database

Physical Storage Media/3

- **Flash memory (SSD)**
  - Non-volatile
  - Reads are roughly as fast as main memory
  - Writes are slow (few microseconds) and more complicated
    - Data cannot be overwritten, but a block must be erased and written over simultaneously
  - Cost per unit of storage roughly similar to main memory
  - Widely used in embedded devices such as digital cameras
  - Also known as EEPROM (Electrically Erasable Programmable Read-Only Memory)

Physical Storage Media/4

- **Magnetic disk**
  - Non-volatile
  - Data is stored on spinning disk, and read/written magnetically
  - Much slower access than main memory
  - Much larger capacities than main memory; typically up to roughly x00 GB - 2 TB currently
    - Growing rapidly with technology improvements (factor 2 to 3 every 2 years)
  - Primary medium for the long-term storage of data; typically stores entire DB.
  - Data must be moved from disk to main memory for access, and written back for storage
  - Direct data access, i.e., data on disk can be read in any order, unlike magnetic tape
  - Hard disks vs. floppy disks

Physical Storage Media/5

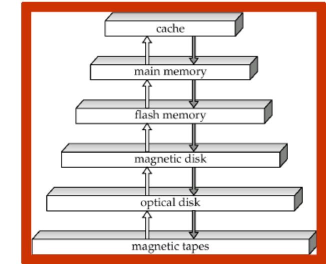
- **Optical disk**
  - Non-volatile
  - Data is read optically from a spinning disk using a laser
  - Reads and writes are slower than with magnetic disk
  - Different types
    - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
    - Write-one, read-many (WORM) optical disks used for archival storage
    - Multiple write versions also available (CD-RW, DVD-RW, and DVD-RAM)
  - Juke-box systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data.

Physical Storage Media/6

- **Tape storage**
  - Non-volatile
  - Much slower than disk due to sequential access only
  - Very high capacity (up to tens of terabytes)
  - Used primarily for backup and for archival data
  - Tape can be removed from drive
  - Tape storage costs are much cheaper than disk storage costs.
  - Tape juke-boxes available for storing massive amounts of data
    - Hundreds of terabytes (1 terabyte $= 10^{12}$ bytes) to even a petabyte (1 petabyte $= 10^{15}$ bytes)

Physical Storage Media/7

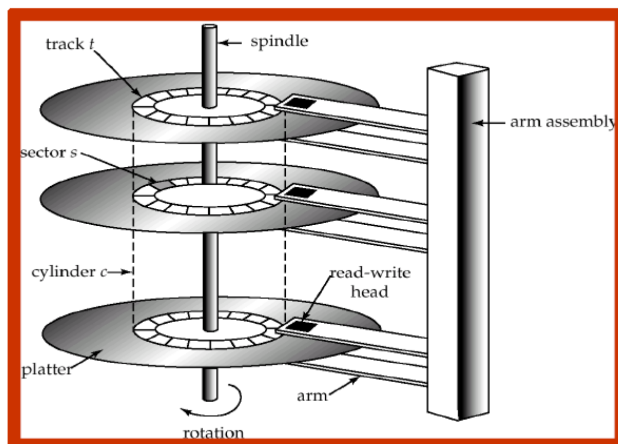
- The storage media can be organized in a hierarchy according to their speed and cost
- **Primary storage**: Fastest media, but volatile
  - e.g., cache, main memory
- **Secondary storage**: Non-volatile,moderately fast access
  - e.g., flash memory, magnetic disks
  - also called on-line storage
- **Tertiary storage**: Non-volatile, slow access time
  - e.g., magnetic tape, optical storage
  - also called off-line storage



- DBMS must explicitly deal with storage media at all levels of the hierarchy

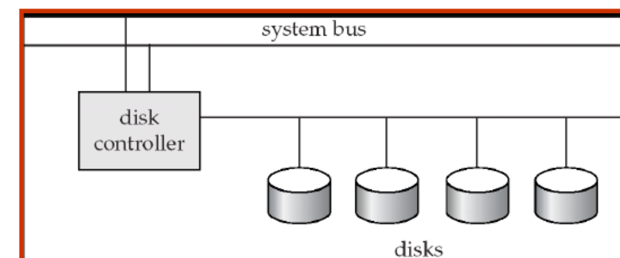
Magnetic Hard Disks/1

- Most DBs are stored on magnetic disks for the following reasons:
  - Generally, DBs are too large to fit entirely in main memory
  - Data on disks is non-volatile
  - Disk storage is cheaper than main memory
- Simplified and schematic structure of a magnetic disk



Magnetic Hard Disks/2

- **Disk controller:** Interface between the computer system and the HW of the disk drive. Performs the following tasks:
  - Translates high-level commands, such as read or write a sector, into actions of the disk HW, such as moving the disk arm or reading/writing the sector.
  - Adds a checksum to each sector
  - Ensures successful writing by reading back a sector after writing it



Magnetic Hard Disks/3

- **Performance measures** of hard disks
  - **Access time:** the time it takes from when a read or write request is issued to when the data transfer begins. Is composed of:
    - **Seek time:** time it takes to reposition the arm over the correct track
    - Avg. seek time is 1/2 the worst case seek time (2-10 ms on typical disks)
    - **Rotational latency:** time it takes for the sector to be accessed to appear under the head
    - Avg. latency is 1/2 the worst case latency (e.g., 4-11 ms for 5400-15000 rpm)
  - **Data-transfer rate:** rate at which data can be retrieved from or stored to disk (e.g., 25-100 MB/s)
    - Multiple disks may share a single controller
  - **Mean time to failure (MTTF)**: average time the disk is expected to run continuously without any failure
    - Typically several years

Storage Access Through Blocks/1

- A **block** is a contiguous sequence of sectors from a single track.
- Blocks are separated by **interblock gaps**, which hold control information created during disk initialization.
- Logically, a block is a unit of storage allocation and data transfer.
  - Data between disk and main memory is transferred in blocks.
  - A database file is partitioned into fixed-length blocks.
  - Typical block sizes range from 4 to 16 kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks

Review 7.1

Consider relations $r(A)$ and $s(A)$. r is ordered, s is unordered. Block size $B = 2KB$. Tuple size $t = 100$Bytes. $|r| = |s| = 800'000$ tuples. The values of A are uniformly distributed between 5M and 10M. The time for 1 IO is 0.025 sec. Determine the execution times for the following queries where $x = r$ or $x = s$: $\sigma_{A=6M}(x)$, $\sigma_{A<5'000'500}(x)$, $\sigma_{A\neq 6M}(x)$.

Storage Access Through Blocks/2

- A major **goal** of DBMSs: Make the transfer of data between disk and main memory as efficient as possible by
  - Optimizing/Minimizing the disk-block access time
  - Minimizing the number of block transfers
  - Keeping as many blocks as possible in memory ($\rightarrow$ buffer manager)
- Techniques to optimize disk-block access:
  1. Disk arm scheduling
  2. Appropriate file organization
  3. Write buffers and log disks

Storage Access Through Blocks/3

- **Disk arm scheduling algorithms:** Order pending accesses to tracks so that disk arm movement is minimized
- **Elevator algorithm**
  - Disk controller orders the requests by track (from outer to inner or vice versa)
  - Move disk arm in that direction, processing the next request in that direction, till no more requests in that direction
  - Then reverse the direction (i.e., inner to outer) and repeat the previous two steps

Storage Access Through Blocks/4

- **File organization:** Optimize block access time by organizing the blocks to correspond to how data will be accessed
  - e.g., store related information on the same or nearby cylinders.
  - Files may get fragmented over time
    - e.g., if data is inserted to or deleted from the file
    - e.g., if free blocks on disk are scattered, which means that a newly created file has its blocks scattered over the disk
    - Sequential access to a fragmented file results in increased disk arm movement
  - Some systems have utilities to defragment the file system, in order to speed up file access

Storage Access Through Blocks/5

Updated blocks can be written asychnronously to increase the write speed.

- **Non-volatile write buffers:** Speed up disk writes by writing blocks to a non-volatile, battery backed up RAM or flash memory immediately; the controller then writes to disk whenever the disk has no other requests or request has been pending for some time.
  - Even if power fails, the data is safe.
  - Writes can be reordered to minimize disk arm movement.
  - Database operations that require data to be safely stored before continuing can continue immediately.
- **Log disk:** A disk devoted to write a sequential log of block updates
  - Used exactly like non-volatile RAM
  - Write to log disk is very fast since no seeks are required
  - No need for special hardware.

Review 7.2/1

Consider a disk as follows: block size B = 512 Bytes, interblock gap size G = 128 Bytes, blocks per track B/T = 20, tracks per surface T/S = 400, double-sided disks D = 15, seek time st = 30 msec, 2400 rotations per minute. Determine the following values:

1. total capacity per track
2. useful capacity per track
3. number of cylinders
4. useful capacity per cylinder

Review 7.2/2

5. transfer rate tr
6. block transfer time btt
7. rotational delay rd
8. bulk transfer rate btr
9. block read time
10. time for 20 random reads
11. time for 20 sequential reads

Buffer Manager/1

- **Buffer:** Portion of main memory available to store copies of disk blocks.
- **Buffer Manager:** Subsystem that is responsible for buffering disk blocks in main memory.
  - The overall goal is to minimize the number of disk accesses.
  - Buffer manager is similar to a virtual-memory manager of an operating system.

Buffer Manager/2

- Programs call the buffer manager when they need a block from disk.
- Buffer manager **algorithm**
  1. Programs call the buffer manager when they need a block from disk.
     - The requesting program is given the address of the block in main memory.
  2. If the block is not in the buffer:
     - The buffer manager allocates space in the buffer for the new block (replacing/throwing out some other block, if required).
     - The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     - Once space is allocated in the buffer, the buffer manager reads the block from the disk to the buffer, and passes the address of the block in memory to the requesting program.
- There exist different strategies/policies to replace buffers

Buffer Replacement Policies/1

- **LRU strategy:** Replace the block least recently used
  - Idea: Use past pattern of block references to predict future references
  - Applied successfully by most operating systems
- **MRU strategy:** Replace the block most recently used
- LRU can be a bad strategy in DBMS for certain access patterns involving repeated scans of data
- Queries in DBs have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references

Buffer Replacement Policies/2

- **Example:** compute a join with nested loops

```
for each tuple tr of r do
    for each tuple ts of s do
        if the tuples tr and ts match then ...;
```

- Different access pattern for r and s
  - An r-block is no longer needed, after the last tuple is processed (even if it has been used recently), thus should be removed immediately
  - An s-block is needed again after all other s-blocks are processed, thus MRU is the best strategy
  - A mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

Review 7.3

Assume a relation *r* with 3 tuples and a relation *s* with 3 tuples. Assume a block can fit 2 tuples. Illustrate how a nested loop join processes tuples and how to use blocks if 2 blocks are available for the join.

Buffer Replacement Policies/3

- **Pinned block:** Memory block that is not allowed to be written back to disk.
  - e.g., the r-block before processing the last tuple tr
- **Toss immediate strategy:** Frees the space occupied by a block as soon as the final tuple of that block has been processed
  - e.g., the r-block after processing the last tuple tr
- MRU + pinned block is the best choice for the join

Buffer Replacement Policies/4

- **Buffer replacement policies in DBMS can use various information**
  - Queries have **well-defined access patterns** (e.g., sequential scans)
  - **Information in a query** to predict future references
  - **Statistical information** regarding the probability that a request will reference a particular relation.
    - e.g., the data dictionary is frequently accessed.
    - Heuristic: keep data dictionary blocks in main memory buffer

File Organization

- **File:** A file is logically a **sequence of records**, where
  - a record is a sequence of fields;
  - the file header contains information about the file.
- Usually, a relational table is mapped to a file and a tuple to a record.
- A DBMS has the choice to
  - Use the file system of the operating system (reuse code).
  - Manage disk space on its own (OS independent, better optimization, e.g., Oracle)
- Two approaches to represent files (or records) on disk blocks:
  - **Fixed length** records (fixed-length records are simple, inflexible, and inefficient in terms of memory)
  - **Variable length** records (variable-length records are complex, flexible, and efficient in terms of memory)

Fixed-Length Records/1

- Store record i starting from byte $m * (i - 1)$, where m is the size of each record.
- Record access is simple but records may cross blocks
  - **spanned** organization: records can be split and span accross block boundaries using pointers
  - **unspanned** organization: records may not cross block boundaries; leave free space in blocks if records do not fit
- Deletion of record i is more complicated. Several alternatives exist:
- Move records $i + 1, ..., n$ to $i, ..., n - 1$
- Move record n to i
- Do not move records, but link all free records in a free list

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixed-Length Records/2

- **Free list**
  - Store the address of the first deleted record in the file header.
  - Use this first record to store the address of the second deleted record, and so on
  - Note the additional field to store pointers.
  - More space efficient representations are possible: No pointers need to be stored in records that contain data.

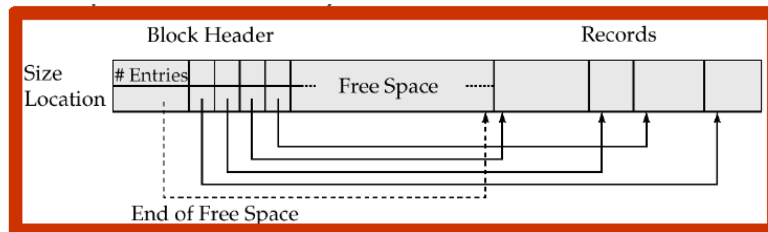
header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

Variable-Length Records/1

- Variable-length records arise in DBMS in several ways:
  - Records types that allow variable lengths for one or more fields.
  - Storage of multiple record types in a file.
  - Record types that allow repeating fields (used in some older models).
- There exist different methods to represent variable-length records:
  - Slotted page structure is the most flexible organization of variable-length records.
  - A slotted page structure maintains a directory of slots for each page.

Variable-Length Records/2

- **Slotted page structure**
  - Slotted page header contains:
    - number of record entries
    - end of free space in the block
    - location and size of each record
  - Records can be moved around in a page to keep them contiguous with no empty space between them; entry in the header must be updated.
  - Pointers should not point directly to record - instead they should point to the entry for the record in header.



Organization of Records in Files/1

There are different ways to logically organize records in a file (this is called the primary file organization):

- **Heap file organization:** A record can be placed anywhere in the file where there is space; there is no ordering in the file.
- **Sequential file organization:** Store records in sequential order based on the value of the search key of each record.
- **Hash file organization:** A hash function is computed on some attribute of each record; the result specifies in which block of the file the record is placed.
- Generally, each relation is stored in a separate file.

Organization of Records in Files/2

- **Sequential file:** The records in the file are ordered by a search key (one or more attributes)
  - Records are chained together by pointers
  - Suitable for applications that require sequential processing of the entire file
  - To be efficient, records should also be stored physically in search key order (or close to it).
  - **Example:** account(account-number,branch-name,balance)

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Organization of Records in Files/3

- It is difficult to maintain the physical order as records are inserted and deleted.
  - Deletion: Store a deletion marker with each record; use pointer chains to build a free list
- Insertion:
  - Locate the position where the record is to be inserted
  - If there is free space insert there
  - If no free space, insert the record in an overflow block
- Need to reorganize the file from time to time to restore (physical) sequential order

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350
A-888	North Town	800

Review 7.4/1

Assume a disk with the following characteristics: block size B = 512 Bytes, blocks per track = 20, tracks per surface = 400, number of double-sided disks = 15, rotations per minute = 2400 rpm, seek time = 30 msec.

Assume a relation Emp(N 30 Bytes, SSN 9 Bytes, A 40 Bytes, P 9 Bytes) with 20'000 tuples.

Determine the following values:

1. HD capacity

2. size of 1 Emp tuple

Review 7.4/2

3. blocking factory (bfr) of Emp (= number of tuples per block)

4. number of blocks used by Emp (unspanned organization)

5. number of blocks used by Emp (spanned organization)

6. average time for a linear search in Emp (contiguous file)

Index Structures for Files

- Basic Concepts, Types of Indexes
- B+ Tree
- Hashing
- Ordered Indexing versus Hashing
- Index Definition in SQL

Basic Concepts/1

- Indexing mechanism are used to speed up access to data
  - e.g., author catalog in library, book index
- **Index file:** Consists of records (called **index entries**) of the form (*search key, pointer*) where
  - **search key** is an attribute or set of attributes used to look up records in a data file
  - **pointer** is a pointer to a record (database tuple) in a data file
- Duplicated search keys in an index file are allowed
- Index files are typically much smaller than the original file

Basic Concepts/2

- **Evaluation of an index** must include
  - Access Time
  - Insertion Time
  - Deletion Time
  - Space overhead
  - Access Type supported efficiently, e.g.,
    - Records with a **specific value** in the attribute, e.g., persons who were born 1970
    - Reocrds with an attribute value falling in a **specific range of values**, e.g., persons who were born between 1970 and 1976

Basic Concepts/3

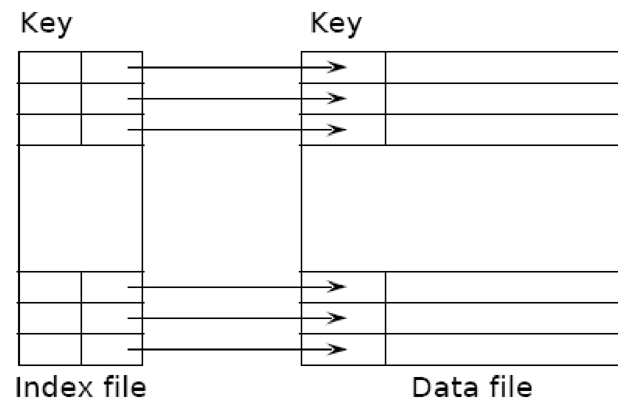
- Depending on the ordering of the data and the index file we can have a
  - **clustering index** (same order of data and index)
  - **non-clustering index** (different order of data and index)
- Depending on what we put into the index we have a
  - **sparse index** (index entry for some tuples only)
  - **dense index** (index entry for each tuple)

- A clustering index is usually sparse
- A non-clustering index must be dense

- Note: terminology is not consistent across textbooks

Clustering Index/1

- **Clustering index**
  - In a clustering index the search key order corresponds to the sequential order of the records in the data file.
  - If the search key is a candidate key (and therefore unique) it is also called a **primary index**.



Clustering Index/2

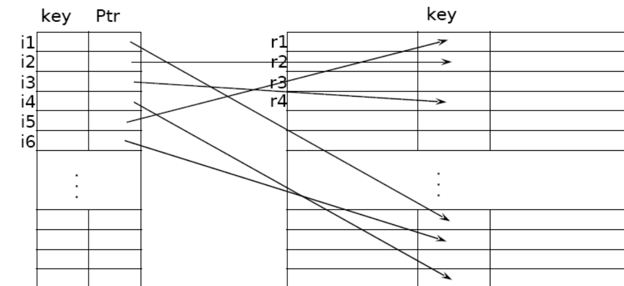
- **Index file:**
  - The index file is a sequential (ordered) file.
  - For a clustering index both index and data are stored on sequential files (index file and data file)
  - Designed for both efficient sequential access and random access
  - One of the oldest indexing techniques in DB

Non-Clustering Indexes/1

- Non-Clustering indexes are used to quickly find all records whose values in a certain field satisfy some condition.
- **Example:** Consider an account relation that is stored sequentially by account number
  - Find all accounts in a particular branch
  - Find all accounts with a specified balance or range of balances
- The above query can only be answered by retrieving and checking all records (very inefficient).
- An additional (non-clustering) index is needed to answer such queries efficiently.

Non-Clustering Indexes/2

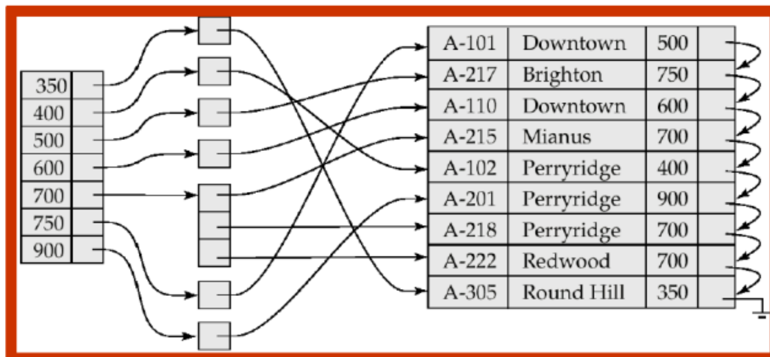
- **Non-clustering index:** Index whose search key specifies an order **different** from the sequential order of the file



- Non-clustering indexes are also called **secondary indexes**.
- Non-clustering indexes **must be dense**, i.e., they must include an index entry for every search key value and a pointer to every record in the data file.

Non-Clustering Indexes/3

- Two options for data pointers
  - Duplicate index entries: an index record for every data record
  - Buckets: An index record for each search key value; index record points to a **bucket** that contains pointers to all the actual records with that particular search key value
- **Example:** Non-Clustering index on the balance field of the account relation using buckets

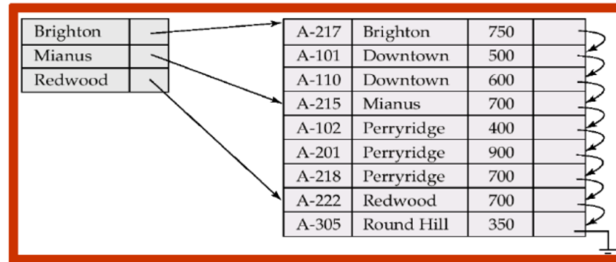


Review 7.5

- Relation $R(A, B, ...)$ with 6M tuples.
- Clustering index on A. Non-clustering index on B.
- 200 index entries per block. 50 tuples per block.
- Values of A and B are uniformly distributed in $[0, 100M]$.
- Use indexes to answer $Q1 = \sigma[A > 75M](r)$ and $Q2 = \sigma[B > 75M](r)$. Determine the number of IOs. Interpret the result.

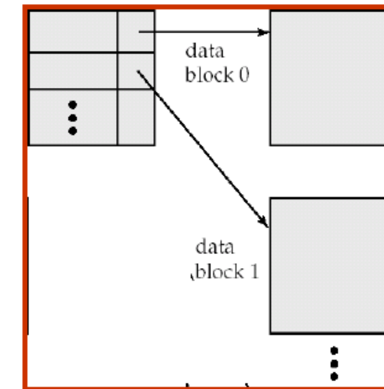
Sparse Index/1

- **Sparse index**
  - Contains index records for only some search key values.
  - Sparse index has (much) fewer entries than records in a table.
  - Applicable when records are sequentially ordered on the search key



Sparse Index/2

- Often a sparse index contains an index entry for every block in file.
- The index entry stores the least search key value of the block it points to.



Dense Index/1

- **Dense index:**
  - An index record appears for **each record** in the data file.
  - Dense index can get large (but still is much smaller than the data file).
  - Handling gets easier if there is exactly one entry for each record.
  - Alternative definition (used in some textbooks/systems: the index contains a record for each search key; the index record points to the first data record with that search key value; the remaining data records with that search key are stored sequentially

Clustering versus Non-Clustering Indexes

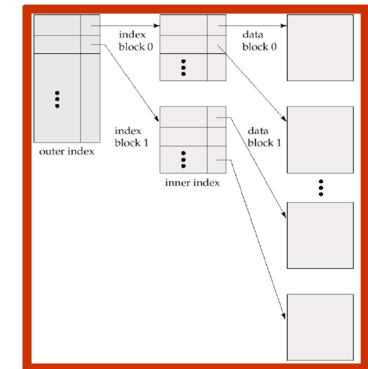
- Indexes offer substantial benefits for lookups
- Updating indexes imposes overhead on DB modifications: whenever data are modified, all index on this data must be updated too
- Clustering indexes can be dense or sparse
- Non-clustering indexes must be dense
- Sequential scan using clustering index is efficient
- A sequential scan using a non-clustering index is expensive since each record access may fetch a new block from disk
- A sparse index uses less space than a dense index
- The maintenance overhead for insertion and deletion is less for a sparse index than for a dense index
- In general a sparse index is slower than a dense index for locating records.

Multilevel Index/1

- If an index grows the handling becomes more expensive
  - A search for a data record requires several disk block reads from the index file
  - Binary search might be used on index file: $log_2 b$ disk block reads, where b is the total number of index blocks
  - If overflow blocks are used in the index file, binary search is not applicable, and sequential scan is required: b disk block reads are required
- To reduce the number of index block I/Os, treat clustering index kept on disk as a sequential file (like any other data file) and construct a sparse index on it
  - $\Rightarrow$ multilevel index

Multilevel Index/2

- **Multilevel index**
  - Inner index: The main index file for the data
  - Outer index: A sparse index on the index
- If even outer index is too large to fit in main memory, yet another level of index can be created, etc.



Multilevel Index/3

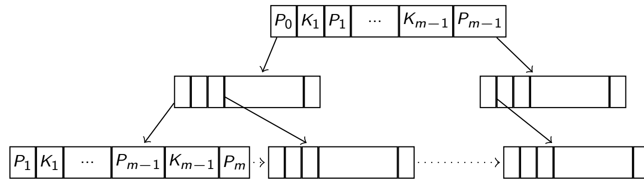
- **Index search: querying**
  - Start at the root
  - Check all entries (the entries are sorted) and follow the appropriate pointer
  - Repeat until you arrive at a leaf where the pointer points to the tuple
- **Index update: deletion and insertion**
  - Indexes at all levels must be updated on insertion and deletion in the data file
  - Update starts with the inner index
  - Algorithms are extensions of the single-level algorithms

B+ Tree/1

- The **B+ tree** is a multi-level index and is an alternative to sequential index files
  - Advantage of B+ tree index files
    - Automatically maintains as many levels of index as appropriate
    - Automatically reorganizes itself with small, local changes in the face of insertions and deletions
    - reorganization of entire file is not required to maintain performance
  - Disadvantage of B+ tree
    - Extra insertion and deletion overhead as well as space overhead
  - Advantages of B+ trees by far outweigh disadvantages, and they are used extensively

B+ Tree/2

- **B+ tree:** a rooted tree with the following properties



  - **Balanced tree**, i.e., all paths from root to leaf are of the same length (at most $\lceil log_{\lceil m/2 \rceil}(K) \rceil$ for K search key values)
  - A **node** contains up to $m-1$ search key values and m pointers, and the search key values within a node are sorted
  - Nodes are between half and completely full
  - **Internal nodes** have between $\lceil m/2 \rceil$ and m children
  - **Leaf nodes** have between $\lceil (m-1)/2 \rceil$ and m - 1 search key values
  - **Root node**: If it is a leaf, it can have between 0 and $m-1$ search key values; otherwise, it has at least 2 children

Terminology and Notation

- A pair (P_i, K_i) in a leaf node is an **entry**
- A pair (K_i, P_i) in an internal (i.e., non-leaf) node is an **entry**
- $L[i]$ denotes the value of the ith entry in node L
- Data pointers are stored at leaf nodes only
- Leaf nodes are linked together: the last pointer in a node points to the next leaf node.

- Note that there are many small variations of B+ tree; textbooks differ; stick to approach described on these slides.

B+ Tree Node Structure/1

- **Leaf nodes**

P_1	K_1	P_2	K_2	···	P_{m-1}	K_{m-1}	P_m

  - $K_1, \dots, K_{m-1}$ are the search key values
  - $P_1, ..., P_{m-1}$ are pointers to records or buckets of records (for leaf nodes)
  - The search keys in a node are ordered: $K_1 < K_2 < K_3 < ... < K_{m-1}$
  - P_i points to the database tuples with search keys X equal to K_i
  - $P_1, \dots, P_{m-1}$ either point to a file record with search key value K_i (unique) or to a bucket of pointers to file records with search key value K_i (non-unique)
  - Bucket structure is only needed if search key does not form a primary key
  - Pointer P_m points to next leaf node in search key order

B+ Tree Node Structure/2

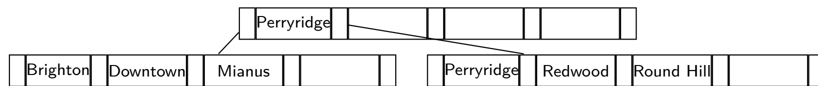
- **Internal nodes**

P_0	K_1	P_1	K_2	P_2	···	K_{m-1}	P_{m-1}

  - Form a multi-level sparse index on the leaf nodes
  - $P_0, ..., P_{m-1}$ are pointers to children (for non-leaf nodes)
  - P_i points to a subtree with search keys X such that $K_i \leq X < K_{i+1}$
    - P_0 points to the subtree where all search key values are less than K_1
    - For $1 \leq i < m-1$: Pointer P_i points to the subtree where all search key values are greater than or equal to K_i and less than K_{i+1}
    - Pointer P_{m-1} points to the subtree where all search key values are greater than or equal to K_{m-1}

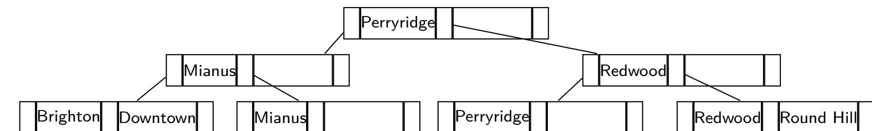
Example of B+ Tree/1

- **B+ tree for account file (m=5 pointers per node)**
  - Leaf nodes: between 2 and 4 search key values ($\lceil (m-1)/2 \rceil$ and $m-1$)
  - Non-leaf nodes other than root: between 3 and 5 children ($\lceil (m/2 \rceil$ and m))
  - Root node: at least 2 children



Example of B+ Tree/2

- **B+ tree for account file (m=3)**
  - Leaf nodes: between 1 and 2 search key values ($\lceil (m-1)/2 \rceil$ and $m-1$)
  - Non-leaf nodes other than root: between 2 and 3 children ($\lceil (m/2 \rceil$ and m))
  - Root node: at least 2 children



Observations about B+ Trees/1

- **B+ tree for account file**
  - Since the inter-node connections are done by pointers, logically close blocks need not be physically close
    - gives flexibility
    - increases times for seeks and latency
  - The non-leaf levels of the B+ tree form a hierarchy of sparse indexes (= multilevel index on leaf nodes)
  - The B+ tree contains a relatively small number of levels
    - $\lceil log_{\lceil m/2 \rceil}(K) \rceil$ for K search key values in the file

Observations about B+ Trees/2

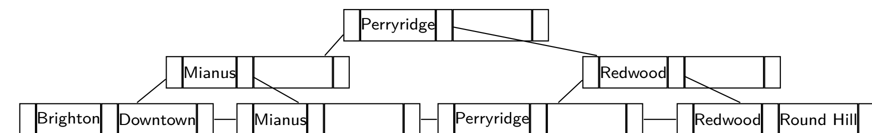
- **Search is efficient, since only a small number of index blocks need to be read**
  - Compare to the $log_2(b)$ disk block reads for binary search in sequential index files
  - Typically the root node and perhaps the first level nodes are kept in main memory, which further reduces the disk block reads.
- **Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time**

Queries on B+ Trees/1

- **Steps to find all records with a search key value of *k* (we assume a dense index):**
  1. Set C = root node
  2. **while** C is not a leaf node **do**
     Search for the largest search key value $\leq k$
     if such a value exists, assume it is K_i
     then set C = the node pointed to by P_i
     else set C = the node pointed to by P_0
  3. If there is a key value K_i in C such that $K_i = k$
     then follow pointer P_i to the desired record or bucket
     else no record with search key value k exists

Queries on B+ Trees/2

- **Example:** Find all records with a search key value equal to Mianus
  - Start from the root node
  - No search key $\leq$ Mianus exists, thus follow P_0
  - Mianus is the largest search key $\leq$ Mianus, thus follow P_1
  - search key = Mianus exists, thus follow the first data pointer to fetch record



Queries on B+ Trees/3

- In processing a query, a path is traversed in the tree from the root to some leaf node
- For K search key values in the data file, the path length is at most $\lceil log_{\lceil m/2 \rceil}(K) \rceil$
- A node generally corresponds to a disk block, typically 4KB, and m is typically ≈ 400 (10 bytes per index entry)
- With 1 million search key values and m = 400, at most $log_{200}(1,000,000) = 3$ nodes are accessed in a lookup
  - Contrast this with a balanced binary tree (or binary search) with 1 million search key values: around 20 nodes are accessed in a lookup
  - This difference is significant since every node access may need a disk I/O, costing around 20 milliseconds.

Intuition for B+ Tree Insertions/1

- **Insert a record with search key value of *k***
  1. Find the leaf node in which the search key value would appear
  2. **If** the search key value is already there **then**
     - Add record to the data file
  3. **If** the search key value is not there and leaf is not full **then**
     - Add record to the data file
     - Insert (pointer, key-value) pair in the leaf node such that the search keys are still in order
  4. **If** search value is not there and leaf is full **then**
     4.1 Take all entries (including the new one being inserted) in sorted order; place the first half in the original node and the rest in a new node
     4.2 Insert the smallest entry of the new node into the parent of the node being split
     4.3 **If** the parent is full **then** split it and propagate the split further up
- **Splitting proceeds upwards until a node that is not full is found**
  - In the worst case the root node may be split, increasing the height of the tree by 1

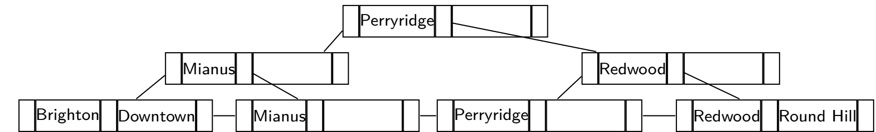
B+ Tree Insertion Algorithm

Algo: B+TreeInsert(L,k,p)

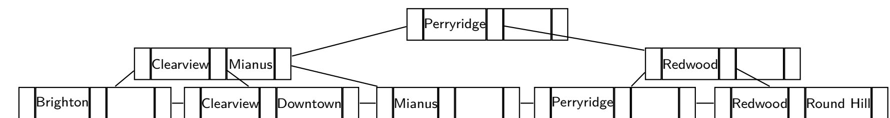
if *L is not yet full* **then**
  insert (k,p) into L
else
  create new node L';
  if *L is a leaf* **then**
    $L := L + (k, p); k' := L\lceil(m+1)/2\rceil$;
    move entries greater or equal to k' from L to L';
  else
    $L := L + (k, p); k' := L\lceil m/2\rceil$;
    move entries greater or equal to k' from L to L';
    delete entry with value k' from L'
  if *L is not the root* **then** B+TreeInsert(parent(L),k',L') ;
  else create new root with children L and L' and value k' ;

B+ Tree Insertions/3

- **Example:**
- B+ tree before insertion of Clearview



- B+ tree after insertion of Clearview



Review 7.6/1

Assume an empty B+ tree of order 4. Show the B+ tree after the following insertions: +2 +3 +5 ; +7 ; +11 +17 ; +19 +23 ; +29 +31 ; +8 +9 ; Show the B+ tree at the points indicated by a semicolon.

Review 7.6/2

+19 +23:

+29 +31

+8 +9

Intuition for B+ Tree Deletions/1

- **Deletion of a record with search key *k***
  1. Find leaf node with (pointer, key-value) entry; remove entry
  2. **If** the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node **then**
     - **Coalesce** siblings, i.e., insert all search key values in the two nodes into a single node (the one on the left if it exists; the right otherwise) and delete the other node
     - Delete the entry in parent node that is between the two nodes by applying the deletion procedure recursively
  3. **If** the node has too few entries due to the removal, and the entries in the node and a sibling do not fit into a single node **then**
     - **Redistribute** the pointers between the node and a sibling such that both have more than the minimum number of entries
     - Update the corresponding search key value in the parent of the node
- Node deletions may cascade upwards till a node with $\lceil m/2 \rceil$ or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the child becomes the root.

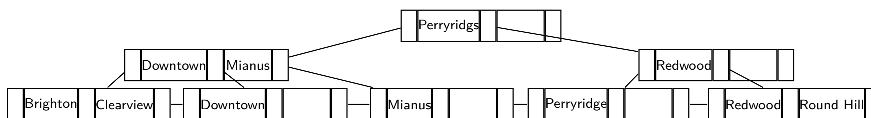
B+ Tree Deletion Algorithm

Algo: B+TreeDelete(L,k,p)

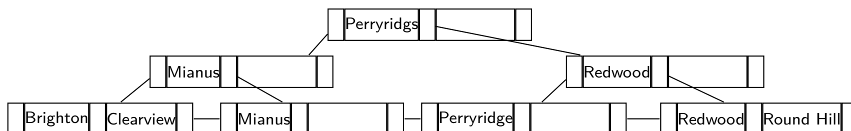
```
delete (p,k) from L;
if L is root with one child then root := child;
else if L has too few entries then
    L' is previous sibling of L [next if there is no previous] ;
    k' is value in parent that is between L and L';
    if entries L and L' fit on one page then
        if L is leaf then move entries from L to L';
        else move k' and all entries from L to L';
        B+TreeDelete(parent(L),k',L)
    else
        if L is leaf then
            move last [first] entry of L' to L;
            replace k' in parent(L) by value of first entry in L [L'];
        else
            move [first] last entry of L' to L;
            replace k' in parent(L) by value of first entry of L [L'];
            replace value of first entry in L [L'] by k';
```


B+ Tree Deletions/3

- **Example:** Before deleting Downtown



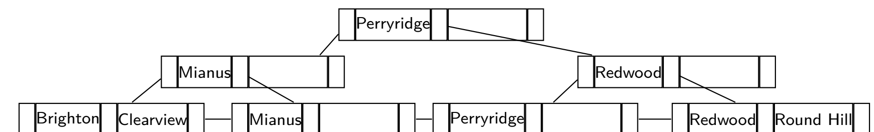
- After deleting Downtown



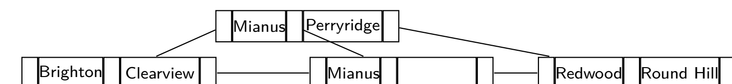
- The removal of the leaf node containing Downtown do not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent.

B+ Tree Deletions/4

- **Example:** Before deleting Perryridge



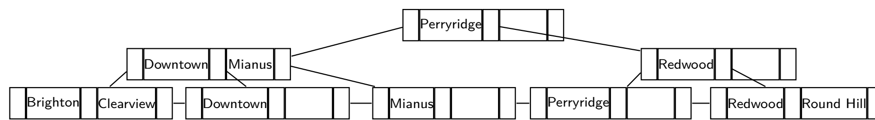
- After deleting Perryridge



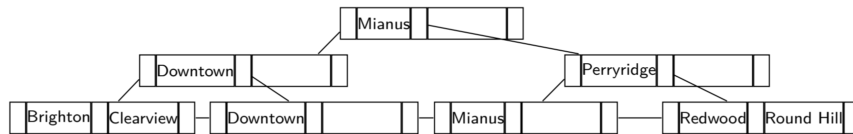
- Node with Perryridge becomes underfull and is merged with its sibling.
- As a result Perryridge node's parent becomes underfull, and is **coalesced** with its sibling (and an entry is deleted from their parent).
- Root node then has only one child and is deleted.

B+ Tree Deletions/5

- **Example:** Before deleting Perryridge



- After deleting Perryridge



- Parent of leaf containing Perryridge became underfull and borrowed a pointer from its left sibling (**redistribute** entries).
- Search key value in the parent's parent changes as a result.

Review 7.7/1

Consider the final B+ tree from review 7.6. Show the B+ trees after the following operations: -19 ; -17 -11 ; -9 ; -2 ; Show the B+ tree at the points indicated by a semicolon.

-19:

Review 7.7/2

Consider the final B+ tree from review 7.6. Show the B+ trees after the following operations: -19 ; -17 -11 ; -9 ; -2 ; Show the B+ tree at the points indicated by a semicolon.

-17, -11:

-9:

-2:

Static Hashing/1

- **Disadvantage of sequential and B+ tree index file organization**
  - B+ tree: index structure must be accessed to locate data
  - Sequential file: binary search on large file might be required
  - This leads to additional block IO
- **Hashing**
  - provides a way to avoid index structures and to access data directly
  - provides also a way of constructing indexes
- A **bucket** is a unit of storage containing one or more records (typically a disk block; possibly multiple contiguous disk blocks).

Static Hashing/2

- **Hash file organization**
  - We obtain the bucket where a record is stored directly from its search key value using a hash function.
    - Constant access time
    - Avoids the use of an index
  - **Hash function h:** A function from the set of all search key values K to the set of all bucket addresses B.
  - Function h is used to locate records for access, insertion, and deletion.
  - Records with different search key values may map to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Static Hashing/3

- **Example:** Hash file organization of account file, using branch-name as key
- 10 buckets
- Binary representation of the ith character is assumed to be i, e.g. binary(B) = 2
- Hash function h
  - Sum of the binary representations of the characters modulo 10, e.g.,
  - h(Perryridge) = 5
  - h(Round Hill) = 3
  - h(Brighton) = 3

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash Functions/1

- Worst hash function maps all search key values to the same bucket
  - This makes access time proportional to the number of search key values in the file.
- An **ideal hash function** has the following properties:
  - The distribution is **uniform**, i.e., each bucket is assigned the same number of search key values from the set of all possible values.
  - The distribution is **random**, so in the average case each bucket will have the same number of records assigned to it irrespective of the actual distribution of search key values in the file.

Hash Functions/2

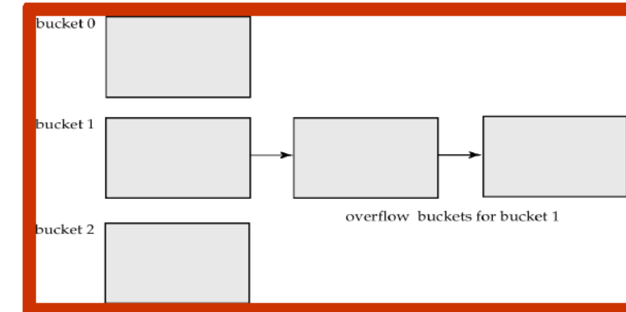
- **Example:** 26 buckets and a hash function that maps branch names beginning with the i-th letter of the alphabet to the i-th bucket
  - Simple, but not a uniform distribution, since we expect more branch names to begin, e.g., with B and R than Q and X.
- **Example:** Hash function on the search key balance by splitting the balance into equal ranges: 1 - 10000, 10001 - 20000, etc.
  - Uniform but not random distribution
- **Typical hash function:** Perform computation on the internal binary representation of the search key.
  - e.g., for a string search key, add the binary representations of all characters in the string and return the sum modulo the number of buckets

Bucket Overflow/1

- **Bucket overflow:** If a bucket has not enough space, a bucket overflow occurs; two reasons for bucket overflow
  - **Insufficient buckets**: the number of buckets n_B must be chosen to be $n_B > n/f$, where n = total number of records and f = number of records in bucket
  - **Skew in distribution of records:** A bucket may overflow even when other buckets still have space. This can occur due to two reasons:
    - multiple records have same search key value
    - hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it **cannot** be eliminated!
  - Handled by using overflow buckets

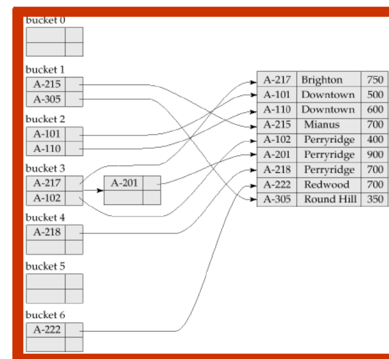
Bucket Overflow/2

- **Overflow chaining (closed hashing)**
  - If a record is inserted into bucket b, and b is already full, an **overflow bucket** is provided, where the record is inserted
    - The overflow buckets of a given bucket are chained together in a list.



Hash Indexes

- **Hash index:** organizes the search key values with their associated record pointers into a hash file structure.
  - Buckets contain search keys and pointers to the data records
  - Multiple (search key, pointer)-pairs might be required (different from index-sequential file)
- **Example:** Index on account
  - h: Sum of digits in account-number modulo 7



Deficiencies of Static Hashing/1

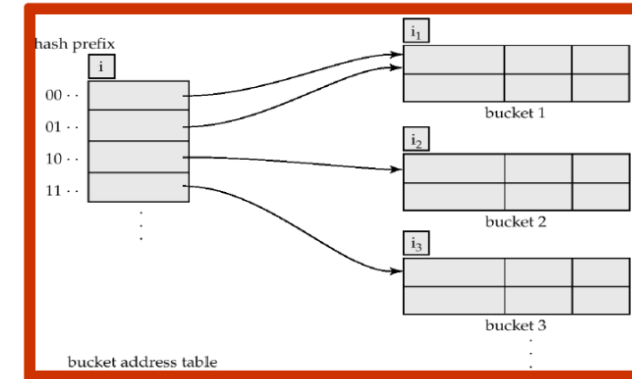
- In static hashing, the **fixed** set B of bucket addresses presents a serious problem
  - Databases grow and shrink with time
  - If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space is wasted initially.
  - If database shrinks, again space will be wasted
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically
  - $\Rightarrow$ dynamic hashing

Deficiencies of Static Hashing/2

- **Dynamic hashing:** Allows the hash function to be modified dynamically.
- **Extendable hashing:** one form of dynamic hashing
  - Hash function h generates values over a large range - typically b-bit integers, with $b = 32$.
  - At any time use only a prefix of h to index into the bucket address table
  - Let the size of the prefix be i bits, $0 \leq i \leq 32$
    - Bucket address table has size $= 2^i$
    - Value of i grows and shrinks as the size of DB grows and shrinks; initially $i = 0$
  - The actual number of buckets is $\leq 2^i$
    - Multiple entries in the bucket address table may point to the same bucket.
    - All such entries have a common hash prefix, $i_j \leq i$, which is stored with each bucket j
    - The number of buckets changes dynamically due to coalescing and splitting of buckets.

Extendable Hashing

- **General structure of extendable hashing**
  - i indicates the number of bits that are used from the hash value.
  - Consecutive entries may point to the same bucket (leads to a smaller prefix associated with this bucket).
  - In this structure, $i_2 = i_3 = i = 2$, whereas $i_1 = i - 1 = 1$ (thus, two entries point to bucket 1)



Lookup in Extendable Hashing

- **Lookup:** Locate the bucket containing search key value K_j
  1. Compute $h(K_j) = X$
  2. Use the first i (hash prefix) high order bits of X as a displacement into the bucket address table, and follow the pointer to the appropriate bucket

Updates in Extendable Hashing/1

- **Insertion** of a record with search key value K_j
  1. Use lookup to locate the bucket, say bucket j
  2. **If** there is room in bucket j **then**
     - Insert the record in the bucket.
  3. **Else**
     - The bucket must be split and insertion re-attempted

Updates in Extendable Hashing/2

- **Split a bucket j when inserting search key value K_j**
  - **If** $i > i_j$ (more than one pointer to bucket j) **then**
    - Allocate a new bucket z, and set i_j and i_z to the old $i_j + 1$.
    - Update bucket address table entries that point to j according to prefix (some will now point to z)
    - Remove and reinsert each record in bucket j.
    - Recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full).
  - **If** $i = i_j$ (only one pointer to bucket j) **then**
    - Increment i and double the size of the bucket address table.
    - Replace each entry in the table by two entries that point to the same bucket.
    - Recompute new bucket address table entry for K_j
- Overflow buckets needed instead of splitting (or in addition) in some cases, e.g., too many records with same hash value.

Updates in Extendable Hashing/3

- **Deletion** of a key value K
  1. Locate K in its bucket and remove it (search key from bucket and record from the file).
  2. The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  3. Coalescing of buckets can be done (can coalesce only with a buddy bucket having same value of i_j and same i_j -1 prefix, if it is present).
  4. Decreasing bucket address table size is also possible.
- **Note:** Decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Review 7.8/1

Consider the following hash function: h(Brighton) = 0010, h(Downtown) = 1010, h(Mianus) = 1100, h(Perryridge) = 1111, h(Redwood) = 0011. Assume a bucket size of two and extendable hashing with an address table size of 1. Show the hash table after the following modifications:

- insert 1 Brighton and 2 Downtown records
- insert 1 Mianus record
- insert 1 Redwood record
- insert 3 Perryridge records

Review 7.8/2

Extendable Hashing: Discussion

- **Benefits** of extendable hashing
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
  - No buckets are reserved for future growth, but are allocated dynamically.
- **Disadvantages** of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Need a tree structure to locate desired record in the structure
  - Changing size of bucket address table is expensive

Ordered Indexing versus Hashing

- Cost of periodic re-organization
  - Hashmaps (e.g., Google's sparse and dense hash maps) do not provide constant insert/lookup time because of reorganization
- Relative frequency of insertions and deletions
  - B+ trees are better than hashing if there are many database updates
- Is it desirable to optimize average access time at the expense of worst-case access time?
  - Hashing has a better average time but no worst case guarantees
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indexes are to be preferred
    - There is no ordering in hash organization, and hence there is no notion of "next record in sort order".

Berkeley DB

o Database access methods:
- B+tree
- Hash (Extended Linear Hashing)
- Fixed/Variable Length Records
- Duplicate records per key in the B+tree and Hash access methods.
- Retrieval by record number in the B+tree access method.
- Keyed and sequential (forward and reverse) retrieval, insertion, modification and deletion.
- Memory-mapped read-only databases.
- Retrieval into user-specified or allocated memory.
- Partial-record data storage and retrieval.
- Architecture independent databases.
- Maximum B+tree depth of 255.
- Individual database files up to 2^{48} bytes, individual key or data elements up to 2^{32} bytes (or available memory).

Berkeley DB

The Berkeley Database Package (DB) is being used by many different organizations in many different applications! Here are a few of which you've probably heard:

Netscape SuiteSpot, an integrated suite of intranet and Internet server software, lets you communicate, access, and share information throughout your organization. The Enterprise, Catalog, Directory and Mail servers all use Berkeley DB.

The Isode Ltd. LDAP/X.500 Enterprise Directory Server uses Berkeley DB as its primary database. Berkeley DB is also used in its X.400/Internet Message Switch and X.400 Message Store products. Here's the press release from Isode about Berkeley DB.

Sendmail is the program that routes electronic mail throughout the Internet. Sendmail is used on almost every UNIX-like system, and it uses Berkeley DB.

The OSF Distributed Computing Environment (DCE) is an industry-standard, vendor-neutral set of distributed computing technologies. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. The DCE backing store library (Open Group RFC #45) is a subset of Berkeley DB.

Index Definition in SQL

- SQL-92 does not define syntax for indexes because these are not considered part of the logical data model
- All DBMSs (must) provide support for indexes
- Create an index:

  create index <IdxName> **on** <RelName> (<AttrList>)

  E.g,: **create index** BrNaIdx **on** branch (branch-name)
- **Create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index: drop index <index-name>

  E.g,: drop index BrNaIdx

Indexes in PostgreSQL

- **CREATE [UNIQUE] INDEX** name **ON** table_name "(" col [**DESC**] { "," col [**DESC**] } ")" [...]

- `CREATE INDEX MjIdx ON enroll (Major)`
- `CREATE INDEX MjIdx ON enroll USING HASH (Major)`
- `CREATE INDEX MjMnIdx ON Enroll (Major, Minor)`

- Properties of indexes:
  - Indexes are automatically maintained as data are inserted, deleted, and updated.
  - Indexes slow down database modification statements.
  - Creating an index can take a long time.

Indexes in Oracle

- B+ tree indexes in Oracle
  CREATE [UNIQUE] INDEX name **ON** table_name "(" col [**DESC**] { "," col [**DESC**] } ")" [**pctfree** n] [...]
  - pct_free specifies how many percent of a index page are left unfilled initially (dafult to 10%)
  - In index definitions UNIQUE should not be used because it is a logical concept.
  - Oracle creates a B+ Tree index for each unique (and primary key) declaration.

```
CREATE TABLE BOOK
  ISBN INTEGER, Author VARCHAR2(30), ...);
CREATE INDEX book_auth ON  book(Author);
```

- Creating a hash-partitioned global index:

```
CREATE INDEX CustLNameIX ON customers(LName)
GLOBAL PARTITION BY HASH (LName)
PARTITIONS 4;
```


Summary/1

- Physical storage media
  - storage hierarchy: cache, RAM, flash, disk, optical disk, tape, ...
- Accessing the storage
  - block-based access:
    - know characteristics of disks
    - compute number of IOs
    - compute execution time
  - buffer manager
- Organization of files
  - fixed-length record, variable-length record
  - heap file (unordered), sequential file (ordered), hash file

Summary/2

- Definition of and differences between index types
  - clustering and non-clustering index
  - dense and sparse index
- B+ tree
  - universal database access structure; also for range predicates
  - definition (node, leaf, non-leaf, entry)
  - insertion and deletion
- Hashing
  - static and extendable hashing
  - no index structure needed for primary index (hash function gives record location directly)
  - good for equality predicates (used heavily in applications)
- Index definition in SQL

Database Systems
Spring 2017

Query Processing and Query Optimization
SL08

- Query Processing
  - Sorting, Partitioning
  - Selection, Join
- Query Optimization
  - Cost estimation
  - Rewriting of relational algebra expressions
  - Rule- and cost-based query optimization

Literature and Acknowledgments

Reading List for SL08:

- Database Systems, Chapter 18, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Database Systems, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

PostgreSQL Example/1

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *

File Edit Query Favourites Macros View Help

boehlen on local socket

```sql
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 7000000;
```

Scratch pad

Output pane

Data Output | Explain | Messages | History

i1 → r1 → Nested Loop → Aggregate

i3

OK. Unix Ln 4 Col 23 Ch 85 8 rows. 8 ms

PostgreSQL Example/2

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *

File Edit Query Favourites Macros View Help

boehlen on local socket

```sql
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 7000000;
```

Scratch pad

Output pane

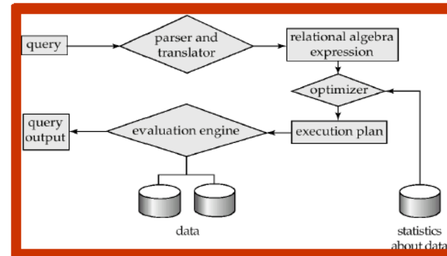
Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=1224.35..1224.36 rows=1 width=0)
2	-> Nested Loop (cost=5.14..1224.10 rows=100 width=0)
3	-> Bitmap Heap Scan on r1 r (cost=5.14..388.89 rows=100 width=4)
4	Recheck Cond: (unique1 > 7000000)
5	-> Bitmap Index Scan on i1 (cost=0.00..5.11 rows=100 width=0)
6	Index Cond: (unique1 > 7000000)
7	-> Index Scan using i3 on r2 s (cost=0.00..8.34 rows=1 width=4)
8	Index Cond: (s.unique1 = r.unique1)

OK. Unix Ln 4 Col 23 Ch 85 8 rows. 8 ms

Query Processing and Optimization

- One of the most important tasks of a DBMS is to figure out an efficient **evaluation plan** (also termed **execution plan** or **access plan**) for high level statements.
  - It is particularly important to have evaluation strategies for:
  - Selections (search conditions)
  - Joins (combining information in relational database)
- Query processing is a 3-step process:
  1. Parsing and translation (from SQL to RA)
  2. Optimization (refine RA expression)
  3. Evaluation (exec RA operators)



Query Processing

- Measuring the query costs
- Sorting
- Optimizing selections
- Optimizing joins

Measuring the Query Costs/1

- **Query cost** is generally measured as the **total elapsed time** for answering a query.
- Many factors contribute to time cost and are considered in real DBMS, including
  - CPU cost and network communication
  - Disk access
    - Difference between sequential and random I/O
  - Buffer Size
    - Having more memory reduces need for disk access
    - Amount of real memory available for buffers depends on other concurrent OS procsesses, and is difficult to determine ahead of actual execution.
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Measures of Query Cost/2

- Typically **disk access is the predominant cost**, which is relatively easy to estimate. The cost of disk accesses is measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block, since data is read back after being written to ensure that the write was successful
- For simplicity
  - we just use **number of block transfers from disk** as the cost measure, and
  - we do not include cost of **writing output to disk**

Sorting

- **Sorting** is important for for several reasons:
  - SQL queries can specify that the output is sorted
  - Several relational operations can be implemented efficiently if the input relations are first sorted, e.g., joins
  - Often sorting is a crucial first step for efficient algorithms
- We may build an index on the relation, and then use the index to read the relation in sorted order.
  - With an index sorting is only logical and not physical. This might lead to one disk block access for each tuple (can be very expensive)
    - It may be desirable/necessary to order the records physically.
- Relation fits in memory: Use techniques like **quicksort**
- Relation does not fit in main memory: Use external sorting, e.g., **external sort-merge** is a good choice

External Sort-Merge/1

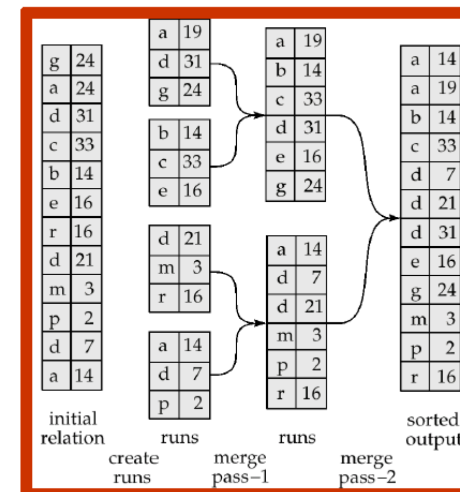
- **Step 1: Create N sorted runs** (M is # blocks in buffer)
  1. Let i be 0 initially.
  2. Repeatedly do the following until the end of the relation
     - 2.1 Read M blocks of the relation (or the rest) into memory
     - 2.2 Sort the in-memory blocks
     - 2.3 Write sorted data to run file R_i;
     - 2.4 Increment i.
- **Step 2: Merge runs (N-way merge)** (assume N < M)
  (Use N blocks in memory to buffer input runs, and 1 block to buffer output)
  1. Read the first block of each run R_i into its buffer page
  2. **Repeat until** all input buffer pages are empty
     - 2.1 Select the first record (in sort order) among all buffer pages
     - 2.2 Write the record to the output buffer. If the output buffer is full write it to disk.
     - 2.3 Delete the record from its input buffer page.
     - 2.4 **If** the buffer page becomes empty **then** read the next block (if any) of the run into the buffer

External Sort-Merge/2

- If $N \geq M$, **several merge passes** (step 2) are required:
  - In each pass, contiguous groups of $M - 1$ runs are merged
  - A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor.
    - E.g. If $M = 11$, and there are 90 runs, one pass reduces the number of runs to 9, each run being 10 times the size of the initial runs
  - Repeated passes are performed until all runs have been merged into one.

External Sort-Merge/3

- **Example**: M = 3, 1 block = 1 tuple



External Sort-Merge/4

- **Cost analysis**
  - b_r = number of blocks in r
  - Initial number of runs: b_r/M
  - Total number of merge passes required: $\lceil log_{M-1}(b_r/M) \rceil$
    - The number of runs decreases by a factor of *M-1* in each merge pass
  - Disk accesses for initial run creation and in each pass is $2b_r$
    - Exception: For final pass there is no write cost
  - Thus total number of disk accesses for external sorting:
    $Cost = b_r \ (2 \lceil log_{M-1}(b_r/M) \rceil + 1)$
- **Example:** Cost analysis of previous example
  - 12 (2 * 2 + 1) = 60 disk block transfers

Selection Evaluation Strategies/1

- The selection operator:
  - **select** * **from** r **where** θ
  - $\sigma_\theta(r)$

  is used to retrieve those records that satisfy the selection condition
- The strategy/algorithm for the evaluation of the selection operator depends
  - on the type of the selection condition
  - on the available index structures

Review 8.1

Assume a B+ tree index on (BrName, BrCity). What would be the best way to evaluate the query:

$$\sigma_{BrCity<'Brighton' \ \wedge \ Assets<5000 \ \wedge \ BrName='Downtown'}(branch)$$

Selection Evaluation Strategies/2

Types of selection conditions:

- **Equality queries**: $\sigma_{a=v}(r)$
- **Range queries**: $\sigma_{a\leq v}(r)$ or $\sigma_{a\geq v}(r)$
  - Can be implemented by using
    - linear file scan
    - binary search
    - using indices
- **Conjunctive selection**: $\sigma_{\theta_1 \wedge \theta_2 \cdots \wedge \theta_n}(r)$
- **Disjunctive selection**: $\sigma_{\theta_1 \vee \theta_2 \cdots \vee \theta_n}(r)$

Selection Evaluation Strategies/3

Basic search methods for selection operator:

- **File scan**
  - Class of search algorithms that **read the file line by line** to locate and retrieve records that fulfill a selection condition, i.e., $\sigma_\theta(r)$
  - Lowest-level operator to access data
- **Index scan**
  - Class of search algorithms that **use an index**
  - Assume B+ tree index and equality conditions, i.e., $\sigma_{a=v}(r)$

Selection Evaluation Strategies/4

- **A1 Linear search:** Scan each file block and test all records to see whether they satisfy the selection condition.
  - Fairly expensive, but always applicable (regardless of indexes, ordering, selection condition, etc)
  - Fetching a contiguous range of blocks from disk has been optimized by disk manufacturers and is cheap in terms of seek time and rotational delay (pre-fetching)
  - Cost estimate (b_r = number of blocks in file):
    - Worst case: $Cost = b_r$
    - If the selection is on a key attribute: $Average\ cost = b_r/2$ (stop when finding record)

Selection Evaluation Strategies/5

- **A2 Binary search:** Apply binary search to locate records that satisfy selection condition.
  - Only applicable if
    - the blocks of a relation are stored contiguously (very rare), and
    - the selection condition is a comparison on the attribute on which the file is ordered
  - Cost estimate for $\sigma_{A=v}(r)$:
    - $\lceil log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the blocks
    - Plus number of blocks containing records that satisfy selection condition

Selection Evaluation Strategies/6

- **A3 Primary index + equality on candidate key**
  - Retrieve a single record that satisfies the equality condition
  - $Cost = HT_i + 1$ (height of B+ tree + 1 data block)
- **A4 Primary index + equality on non-candidate key**
  - Retrieve multiple records, where records are on consecutive blocks
  - $Cost = HT_i + \#$ blocks with records with given search key
- **A5 Secondary index + equality on search-key**
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = HT_i + 1$
  - Retrieve multiple records if search-key is not a candidate key
    - $Cost = HT_i + \#$ buckets with search-key value $+ \#$ retrieved records
    - Can be very expensive, since each record may be on a different block
    - Linear file scan may be cheaper if many records have to be fetched

Selection Evaluation Strategies/7

- **A6 Primary index on A + non-equality condition**
  - $\sigma_{A \geq v}$: Use index to find first tuple $\geq v$; then scan relation sequentially
  - $\sigma_{a \leq v}$: Scan relation sequentially until first tuple $> v$; do not use index.
- **A7 Secondary index on A + non-equality condition**
  - $\sigma_{A \geq v}$: Use index to find first index entry $\geq v$; scan index sequentially from there, to find pointers to records.
  - $\sigma_{A \leq v}$: Scan leaf pages of index finding record pointers until first entry $> v$
  - Requires in the worst case one I/O for each record; linear file scan may be cheaper if many records are to be fetched

Review 8.2

Consider relations $r1(\underline{A}, B, C)$, $r2(\underline{C}, D, E)$, $r3(\underline{E}, F)$ with keys underlined and cardinalities $|r1| = 1000$, $|r2| = 1500$, $|r3| = 750$.

- Estimate the size of $r1 \bowtie r2 \bowtie r3$
- Give an efficient strategy for computing the result and compute its cost

Join Evaluation Strategies

- There exist several different algorithms for the evaluation of join operations:
  - Nested loop join
  - Block nested loop join
  - Indexed nested loop join
  - Merge join
  - Hash join
- Choice based on cost estimate
- Examples use the following relations:
  - customer = (CustName, CustStreet, CustCity)
    - Number of records: $n_c = 10'000$
    - Number of blocks: $b_c = 400$
  - depositor = (CustName, AccNumber)
    - Number of records: $n_d = 5'000$
    - Number of blocks: $b_d = 100$

Nested Loop Join/1

- Compute the theta join: $r \bowtie_\theta s$

```
for each tuple t_r in r do
    for each tuple t_s in s do
        if pair (t_r,t - s) satisfies join condition θ then
            add t_r ∘ t_s to result
```

- r is called the outer relation, s the inner relation of the join.
- Always applicable. Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples.

Nested Loop Join/2

- Order of r and s important: Relation r is read once, relation s is read up to $|r|$ times
  - **Worst case:** Only one block of each relation fits in main memory
    $Cost = n_r * b_s + b_r$
  - If the smaller relation fits entirely in memory, use that as the inner relation.
    $Cost = b_s + b_r$
- **Example:**
  - Depositor as outer relation $d \bowtie c$:
    5'000 * 400 + 100 = 2'000'100 block accesses
  - Customer as outer relation $c \bowtie d$:
    10'000 * 100 + 400 = 1'000'400 block accesses
  - Smaller relation *(depositor)* fits into memory:
    400 + 100 = 500 blocks

Block Nested Loop Join/1

- Simple nested loop algorithm is not used directly since it is not block-based.
- Variant of nested loop join in which every block of the inner relation (s) is paired with every block of the outer relation (r).

```
for each block B_r in r do
  for each block B_s in r do
    for each tuple t_r in r do
      for each tuple t_s in s do
        if pair (t_r,t - s) satisfies join condition θ then
          add t_r ∘ t_s to result
```


Block Nested Loop Join/2

- $r \bowtie s$
- Worst case: $Cost = b_r * b_s + b_r$
  - Each block in the inner relation s is read once for each block in the outer relation (instead of once for each tuple in the outer relation)
- Best case: $Cost = b_s + b_r$
- **Example:** Compute depositor ⋈ customer, with depositor as the outer relation.
  - Block nested loop join:
    Cost = 100 * 400 + 100 = 40'100 blocks (worst case)

Block Nested Loop Join/3

- Improvements to nested loop and block nested loop algorithms (M is the number of main memory blocks):
  - Block nested loop: Use M-2 disk blocks for outer relation and two blocks to buffer inner relation and output; join each block of the inner relation with M-2 blocks of the outer relation.
    - $Cost = \lceil b_r/(M-2) \rceil * b_s + b_r$
  - If equi-join attribute forms a key on inner relation, stop inner loop on first match.
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement).

Indexed Nested Loop Join/1

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - index is available on the inner relation's join attribute
  - index can be constructed just to compute a join
- For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.
- Worst case: Buffer has space for only one page of r, and, for each tuple in r perform an index lookup on s.
  - $Cost = n_r * c + b_r$
    - c is the cost of traversing the index and fetching all matching s tuples for one tuple of r
    - c can be estimated as cost of a single selection on s using the join condition.
- If indexes are available on join attributes of both r and s, use relation with fewer tuples as the outer relation.

Indexed Nested Loop Join/2

- **Example:** Compute depositor x customer, with depositor as the outer relation.
  - Let customer have a primary B+ tree index on the join attribute CustName, which contains 20 entries in each index node.
  - Since customer has 10'000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
  - *depositor* has 5'000 tuples and 100 blocks
  - Indexed nested loops join:
    Cost = 5'000 * 5 + 100 = 25'100 disk accesses.

Review 8.3/1

Consider $e \bowtie_{SSN=MgrSSN} d$ with $r_d = 50$ (number of tuples in relation d), $r_e = 5000$, $b_d = 10$ (number of blocks for relation d), $b_e = 2000$, $n_b = 6$ (number of available buffer blocks).
Compute the number of IOs for the following evaluation strategies:

1. Block NL, $e \bowtie d$, 4 blocks for e (1 block for d, 1 block for result)

2. Block NL, $e \bowtie d$, 4 blocks for D

Review 8.3/2

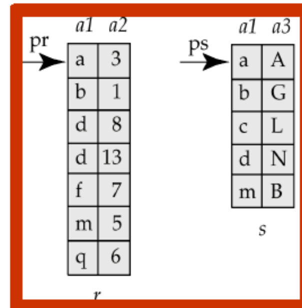
3. Block NL, $d \bowtie e$, 4 blocks for D

4. Indexed NL, $e \bowtie d$

5. Indexed NL, $d \bowtie e$

Merge Join/1

- Basic idea of **merge join**: Use two pointers *pr* and *ps* that are initialized to the first tuple in *r* and *s* and move in a synchronized way through the sorted relations.
- Algorithm
  1. Sort both relations on their join attributes (if not already sorted on the join attribute).
  2. Scan *r* and *s* in sort order and return matching tuples.
  3. Move the tuple pointer of the relation that is less far advanced in sort order (more complicated if the join attributes are not unique - every pair with same value on join attribute must be matched).

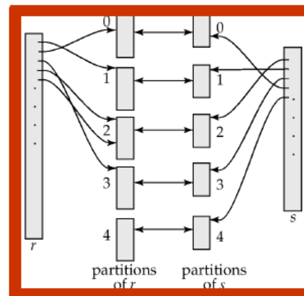


Merge Join/2

- Applicable for equi-joins and natural joins only
- If all tuples for any given value of the join attributes fit in memory
  - One file scan of *r* and *s* is enough
  - Cost $= b_r + b_s$ (+ the cost of sorting if relations are not sorted)
- Otherwise, a block nested loop join must be performed between the tuples with the same attributes
- If the relation are not sorted appropriately we first have to sort them. The combined operator is called a **sort-merge join**.

Hash Join/1

- Applicable for equi-joins and natural joins only.
- Partition tuples of r and s using the **same** hash function *h*, which maps the values of the join attributes to the set 0, 1, ..., n
  - Partitions of r-tuples: $r_0, r_1, ..., r_n$
    - All $t_r \in$ r with h(t_r[JoinAttrs]) $= i$ are put in r_i
  - Partitions of s-tuples: $s_0, s_1, .., s_n$
    - All $t_s \in$ s with h(t_s[JoinAttrs]) $= i$ are put in s_i
- *r*-tuples in r_i need only to be compared with *s*-tuples in s_i
  - an *r*-tuples and *s*-tuples that satisfy the join condition have the same hash value *i*, and are mapped to r_i and s_i, respectively.



Hash Join/2

- **Algorithm** for the hash join of *r* and *s*
  1. Partition the relation s using hash function h. (When partitioning a relation, one block of memory is reserved as the output buffer for each partition.)
  2. Partition *r* similarly.
  3. For each *i*:
     - 3.1 Load s_i into memory and **build** an in-memory hash index on it using the join attribute. This hash index uses a **different** hash function than the earlier one *h*.
     - 3.2 Read the tuples in r_i from the disk (block by block). For each tuple t_r **probe** (locate) each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes as result tuple.
- Relation *s* is called the **build input** and *r* is called the **probe input**.

Hash Join/3

- **Cost analysis** of hash join
  - Partitioning of the two relations: 2 * $(b_r + b_s)$
    - Complete reading of the two relations plus writing back
  - The build and probe phases read each of the partitions once: $b_r + b_s$
  - Cost $= 3 * (b_r + b_s)$
- **Example:** *customer* ⋈ *depositor*
  - Assume that memory size is 20 blocks
  - $b_d = 100$ and $b_c = 400$.
  - *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
  - Similarly, partition customer into five partitions, each of size 80. This is also done in one pass.
    - Partition size of probe relation needs not to fit into main memory!
  - Therefore total cost = 3 * (100 + 400) = 1500 block transfers
    - Ignores cost of writing partially filled blocks

Review 8.4

Consider $b_c = 400$, $n_c = 10'000$, $b_d = 100$, $n_d = 5'000$, disk IO time = 10 msec, memory access time = 60 nsec. Compare the execution times for NL and sort merge (best case).

Query Optimization

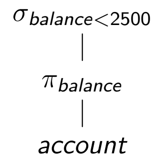
- Cost estimation
- Transformation of relational algebra expressions (rewrite rules)
- Rule-based (aka heuristic) query optimization
- Cost-based query optimization

Query Optimization/1

- Alternative ways of evaluating a query because of
  - Equivalent expressions
  - Different algorithms for each operation
- A **query evaluation plan** (query plan) is an annotated RA expression that specifies for each operator how to evaluate it.
- The cost difference between a good and a bad query evaluation plan can be enormous
  - e.g., performing $r \times s$ followed by a selection r.A = s.B is much slower than performing a join on the same condition
- The query optimizer needs to estimate the cost of operations
  - Depends critically on statistical information about relations
  - Estimates statistics for intermediate results to compute cost of complex expressions

Query Optimization/2

- **Step 1: Parsing and translation**
  - Translate the query into its internal form (query tree)
  - The query tree corresponds to a **relational algebra (RA)** expression
  - Each RA expression can be written as a **tree** where the algebra operator is the root and the argument relations are the children.
- **Example:**
  - SQL query: **select** balance **from** account **where** balance $<$ 2500
  - RA expression: $\sigma_{balance<2500}(\pi_{balance}(\text{account}))$
  - Tree:

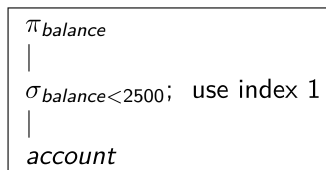


Query Optimization/3

- **Step 2: Optimization**
  - An RA expression may have many (semantically) equivalent expressions
  - The following two RA expressions are equivalent:
    - $\sigma_{balance<2500}(\pi_{balance}(\text{account}))$
    - $\pi_{balance}(\sigma_{balance<2500}(\text{account}))$
  - Each RA operation can be evaluated using one of several different algorithms.
  - Thus, an RA expression can be evaluated in many ways.

Query Optimization/4

- **Step 2: Optimization**
  - **Evaluation plan:** Annotated RA expression that specifies for each operator detailed instructions on how to evaluate it.
    - use index on balance to find accounts with balance $<$ 2500
    - can perform complete relation scan and discard accounts with balance $\geq$ 2500



  - **Goal of query optimization:** Among all equivalent evaluation plans choose the one with lowest cost.
    - Cost is estimated using statistical information from the database catalog, e.g., number of tuples in each relation, size of tuples, etc.
- **Step 3: Evaluation**
  - The query-execution engine takes an evaluation plan, executes that plan, and returns the answers.

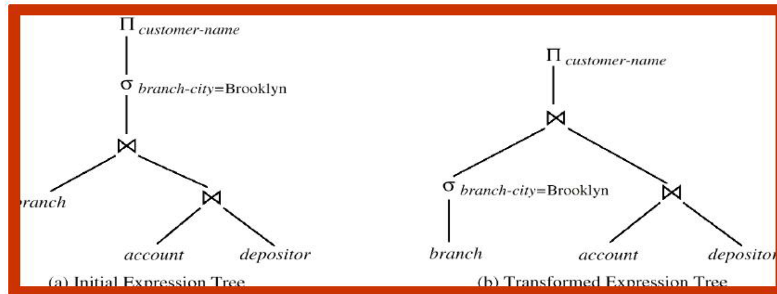
Review 8.5

Display the trees that correspond to the following algebra expressions:

- $RA1 = \pi_A(R1 \bowtie \sigma_{X=Y}(R2 \bowtie \pi_{B,C}(R3 - R4) \bowtie R5))$
- $RA2 = \pi_A(R1) \cup \sigma_{X>5}(R2)$

Query Optimization/5

- **Example:** Find the names of all customers who have an account at any branch located in Brooklyn.
  - $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$
    - Produces a large intermediate relation
    - Transformation into a more efficient expression $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie (account \bowtie depositor))$



Query Optimization/6

- **Goal of query optimizer:** Find the most efficient query evaluation plan for a given query.
- **Cost-based** optimization:
  1. Generate logically equivalent expressions by using equivalence rules to rewrite an expression into an equivalent one
  2. Annotate resulting expressions with information about algorithms/indexes for each operator
  3. Choose the cheapest plan based on **estimated cost**
- **Rule-based/heuristic** optimization:
  1. Generate logically equivalent expressions, controlled by a set of heuristic query optimization rules
- In general, it is not possible to identify the optimal query tree since there are too many. Instead, a reasonably efficient one is chosen.

Statistical Information/1

- The cost of an operation depends on the size and other statistics of its inputs, which is partially stored in the database catalog and can be used to estimate statistics on the results of various operations.
  - n_r: number of tuples in a relation r.
  - b_r: number of blocks containing tuples of r.
  - s_r: size of a tuple of r.
  - f_r: blocking factor of r, i.e., the number of tuples of r that fit into one block.
  - V(A, r): number of distinct values that appear in r for attribute A; same as the size of $\pi_A(r)$.
  - SC(A, r): selection cardinality of attribute A of relation r; average number of records that satisfy equality on A.

Statistical Information/2

- f_i: average fan-out of internal nodes of index i, for tree-structured indexes such as B+ trees.
- HT_i: number of levels in index i, i.e., the height of i.
  - For a B^+-tree on attribute A of relation r, $HT_i = \lceil log_{f_i}(V(A, r)) \rceil$
  - For a hash index, HT_i is 1.
  - LB_i: number of lowest-level index blocks in i, i.e, the number of blocks at the leaf level of the index.
- For accurate statistics, the catalog information has to be updated every time a relation is modified.
  - Many systems update statistics only during periods of light system load (or when requested explicitly), thus statistics is not completely accurate.
  - Plan with lowest estimated cost might not be the cheapest
  - PostgreSQL: run ANALYZE once a day

Rewriting Relational Algebra Expressions

- Two relational algebra expressions are **equivalent** if on every legal database instance the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant
- Two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples
- An equivalence rule states that two different expressions are equivalent and can replace each other

Equivalence Rules/1

- $E, E_1, \ldots$ = RA expressions
  $\theta, \theta_1, \ldots$ = predicates/conditions
  - **ER1** Conjunctive selection operations can be deconstructed into a sequence of individual selections.
    $$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$
  - **ER2** Selection operations are commutative.
    $$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$
  - **ER3** Only the last in a sequence of projections is needed, the others can be omitted (Li are lists of attributes).
    $$\pi_{L1}(\pi_{L2}(\ldots(\pi_{Ln}(E))\ldots)) = \pi_{L1}(E)$$
  - **ER4** Selections can be combined with Cartesian product and theta joins
    (a) $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
    (b) $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules/2

- **ER5** Theta joins (and natural joins) are commutative.
  $$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$
- **ER6** Associativity
  (a) Natural join operations are associative:
  $$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$
  (b) Theta joins are associative in the following way:
  $$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$
  where θ_2 involves attributes from only E_2 and E_3.
  Any of these conditions might be empty, hence, the Cartesian product operation is also associative
- Commutativity and associativity of join operations are important for join reordering.

Equivalence Rules/3

- **ER7** The selection operation distributes over the theta join operation under the following conditions:
  - (a) When all attributes in θ_o involve only the attributes of one of the expressions (E1) being joined:
    $$\sigma_{\theta_o}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_o}(E_1) \bowtie_\theta E_2$$
  - (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2:
    $$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta 1}(E_1) \bowtie_\theta \sigma_{\theta 2}(E_2)$$

Equivalence Rules/4

- **ER8** The projection operation distributes over the theta join operation as follows:
  - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
  - (a) if θ involves only attributes from $L_1 \cup L_2$:
    $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1}(E_1) \bowtie_\theta \pi_{L_2}(E_2)$
  - (b) Consider a join $E_1 \bowtie_\theta E_2$.
    - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
    - Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
      $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1 \cup L_2}(\pi_{L_1 \cup L_3}(E_1) \bowtie_\theta \pi_{L_2 \cup L_4}(E_2))$

Equivalence Rules/5

- **ER9** The set operations union and intersection are commutative
  $$E_1 \cup E_2 = E_2 \cup E_1$$
  $$E_1 \cap E_2 = E_2 \cap E_1$$
  Set difference is not commutative
- **ER10** Set union and intersection are associative.
  $$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
  $$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

Equivalence Rules/6

- **ER11** The selection operation distributes over $\cup, \cap$ and $-$.
  $$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$$
  $$\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$$
  $$\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$$

  Also $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$
  and similarly for $\cap$ in place of $-$, but not for $\cup$
- **ER12** The projection operation distributes over union
  $$\pi_L(E_1 \cup E_2) = \pi_L(E) \cup \pi_L(E_2)$$

Review 8.6

Determine the equivalences that hold. Give counterexamples for the false ones.

1. $\sigma_\theta({}_X\vartheta_F(A)) = {}_X\vartheta_F(\sigma_\theta(A))$, $attr(\theta) \subseteq attr(X)$
2. $\pi_X(A - B) = \pi_X(A) - \pi_X(B)$
3. $A ⟕ (B ⟕ C) = (A ⟕ B) ⟕ C$
4. $A \cap B = A \cup B - (A - B) - (B - A)$

Rewrite Examples/1

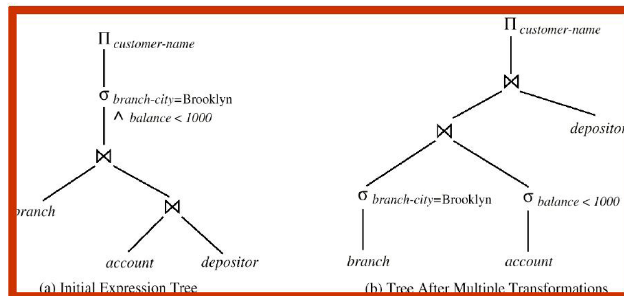
- **Example 1:** Bank database
  - *branch = (BranchName, BranchCity, Assets)*
  - *account = (AccNumber, BranchName, Balance)*
  - *depositor = (CustName, AccNumber)*
- **Query:** Find the names of all customers who have an account at some branch located in Brooklyn.
  $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$
- Transformation using rule **ER7**(a):
  $\pi_{CustName}$
  $(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie (account \bowtie depositor))$
- Performing the selection as early as possible reduces the size of the intermediate relation to be joined.

Rewrite Examples/2

- **Example 2:** Multiple transformations are often needed
  - **Query:** Find the names of all customers with an account at Brooklyn whose balance is below $1000.
    $\pi_{CustName}(\sigma_{BranchCity='Brooklyn' \wedge Balance<1000}$
    $(branch \bowtie (account \bowtie depositor)))$
- Rewrite using rule **ER6**(a) (join associativity):
  $\pi_{CustName}(\sigma_{BranchCity='Brooklyn' \wedge balance<1000}$
  $(branch \bowtie account) \bowtie depositor)$
- Rewrite using rule **ER7**(b) (perform selection early)
  $\sigma_{BranchCity='Brooklyn'}(branch) \bowtie \sigma_{Balance<1000}(account)$

Rewrite Examples/3

- **Example 2** (continued)
  - Tree representation after multiple transformations



Rewrite Examples/4

- **Example 3:** Projection operation
  - Query:
    $\pi_{CustName}((\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account) \bowtie depositor)$
- When we compute
  $\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account$
  we obtain an intermediate relation with schema
  (BranchName, BranchCity, Assets, AccNumber, Balance)
- Push projections using equivalence rules **ER8**(a) and **ER8**(b); thus, eliminate unneeded attributes from intermediate results:
  $\pi_{CustName}$
  $(\pi_{AccNumber}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account)$
  $\bowtie depositor)$

Rewrite Examples/5

- **Example 4:** Join ordering
- For all relations r_1, r_2, and r_3:
  $$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$
- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose
  $$(r_1 \bowtie r_2) \bowtie r_3$$
  so that we compute and store a smaller temporary relation.

Rewrite Examples/6

- **Example 5:** Join ordering
- Consider the expression
  $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account \bowtie depositor)$
- Could compute $account \bowtie depositor$ first, and join result with
  $\sigma_{BranchCity='Brooklyn'}(branch)$
  but $account \bowtie depositor$ is likely to be a large relation.
- Since it is more likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn, it is better to compute first
  $\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account$

Review 8.7

Show how to rewrite and optimize the following SQL query:

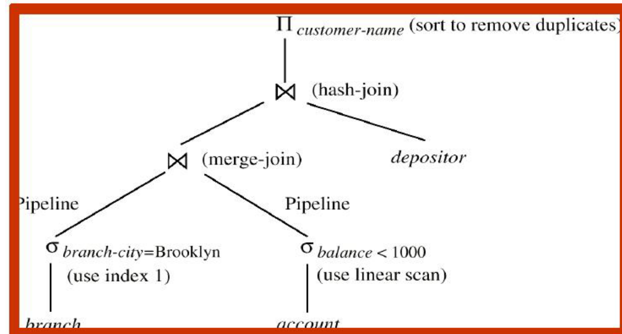
```
SELECT e.LName
FROM employee e, worksOn w, project p
WHERE p.PName = 'A'
AND p.PNum = w.PNo
AND w.ESSN = e.SSN
AND e.BDate = '31.12.1957'
```


Enumeration of Equivalent Expressions

- **Query optimizers** use the equivalence rules to systematically generate expressions that are equivalent to the given expression
- **repeat**
  For each expression found so far, use all applicable equivalence rules, and add newly generated expressions to the set of expressions found so far
  until no more expressions can be found
- This approach is very expensive in space and time
- Reduce space requirements by sharing common subexpressions:
  - When E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared (e.g. when applying join associativity)
- Time requirements are reduced by not generating all expressions (e.g. take cost estimates into account)

Evaluation Plan

- **Evaluation plan (query plan/query tree):** Defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Choosing Evaluation Plans

- When choosing the best evaluation plan, the query optimizer must consider the interaction of evaluation techniques:
  - Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm, e.g.
    - merge join may be costlier than hash join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested loop join may provide opportunity for pipelining
  - Practical query optimizers combine elements of the following two broad approaches:
    1. **Cost-based optimization:** Search all plans and choose the best plan in a cost-based fashion.
    2. **Rule-based optimization:** Uses heuristics to choose a plan.

Heuristic Optimization/1

- Heuristic optimization transforms the query-tree by using a set of heuristic rules that typically (but not in all cases) improve execution performance.
- Overall goal of heuristic rules:
  - Try to **reduce the size of (intermediate) relations** as early as possible
- Heuristic rules
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
- Some (old) systems use only heuristics
- Modern database systems combine heuristics (consider some plans only) with cost-based optimization (determine database specific cost of each plan).

Heuristic Optimization/2

- Example: Consider the expression $\sigma_\theta(r \bowtie s)$, where θ is on attributes in s only.
  - Selection early rule would push down the selection operator, producing $r \bowtie \sigma_\theta(s)$.
  - This is not necessarily the best plan if
    - relation r is extremely small compared to s,
    - and there is an index on the join attributes of s,
    - but there is no index on the attributes used by θ.
  - The early select would require a scan of all tuples in s, which is probably more expensive than the join

Heuristic Optimization/3

- Steps in typical heuristic optimization
  1. Break up conjunctive selections into a sequence of single selection operations (rule **ER1**).
  2. Move selection operations down the query tree for the earliest possible execution (rules **ER2**, **ER7**(a), **ER7**(b), **ER11**).
  3. Execute first those selection and join operations that will produce the smallest relations (rule **ER6**).
  4. Replace Cartesian product operations that are followed by a selection condition by join operations (rule **ER4**(a)).
  5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (rules **ER3**, **ER8**(a), **ER8**(b), **ER12**).
  6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

Cost-Based Optimization/1

Basic working of a cost-based query optimizer:

- **Algorithm**
  1. Use transformations (equivalence rules) to generate multiple candidate evaluation plans from the original evaluation plan.
  2. Cost formulas estimate the cost of executing each operation in each candidate evaluation plan.
     - Cost formulas are parameterized by
       - statistics of the input relations;
       - dependent on the specific algorithm used by the operator;
       - CPU time, I/O time, communication time, main memory usage, or a combination.
  3. The candidate evaluation plan with the **least total cost** is selected for execution.

Cost-Based Optimization/2

- Cost-based optimization can be used to determine the best join order.
- A good ordering of joins is important for reducing the size of temporary results ($|r|, ..., |r|^n$).
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie ... r_m$
- There are $(2(m-1))!/(m-1)!$ different join orders for above expression.
  - With $m = 3$, the number is 12
  - With $m = 7$, the number is 665,280
  - With $m = 10$, the number is greater than 17.6 billion

Cost-Based Optimization/3

- Cost-based optimization is expensive, but worthwhile for queries on large datasets
- Typical queries have a small number m of operations; generally $m < 10$
- With dynamic programming time complexity of optimization with bushy trees is $O(3^m)$.
  - With $m = 10$, this number is 59000 instead of 17.6 billion!
- Space complexity is $O(2^m)$

Review 8.8

Consider a DB with the following characteristics:

- $|r1(A, B, C)| = 1000$, $V(C, r1) = 900$
- $|r2(C, D, E)| = 1500$, $V(C, r2) = 1100$, $V(E, r2) = 50$
- $|r3(E, F)| = 750$, $V(E, r3) = 100$

Estimate the size of $r1 \bowtie r2 \bowtie r3$ and determine an efficient evaluation strategy.

Cost-Based Optimization Example/1

- **Example:** $\sigma_{SSN=0810643773}(Emp)$
- Statistics:
  - $|Emp|$ = 10'000 tuples
  - 5 tuples per block
  - Secondary B^+-tree index of depth 4 on SSN
  - SSN is primary key
- Plan p1: full table scan
  - cost(p1) = (10'000/5)/2 = 1'000 blocks
- Plan p2: B^+-tree lookup
  - cost(p2) = 4 + 1 = 5 blocks

Cost-Based Optimization Example/2

- Example: $\sigma_{DNo>15}(Emp)$
- Statistics:
  - $|Emp|$ = 10'000 tuples
  - 5 tuples per block
  - Primary index on DNo of depth 2
  - 50 different departments
- Plan p1: full table scan
  - cost (p1) = 10'000/5 = 2'000 blocks
- Plan p2: index search
  - cost(p2) = 2 + (50-15)/50*(10'000/5) = 1'400 blocks

Cost-Based Optimization Example/3

- Emp $\bowtie_{DNo=DNum}$ Dept
- Statistics:
  - $|Emp|$ = 10'000 tuples ; 5 Emp tuples per block
  - $|Dept|$ = 125; 10 Dept tuples per block
  - Hash index on Emp(DNo)
  - 4 EmpDept result tuples per block
- Plan p1: Block nested loop with Emp as outer loop
  - cost(p1) = (10'000/5) + (10'000/5) * (125/10) + (10'000/4)
= 30'500 IOs
  - (10.000/4 is cost of writing final output)
- Plan p2: Indexed nested loop with Dept as outer loop and hashed lookup in Emp
  - cost(p2) = (125/10) + 125 * (10'000/125/5) + (10'000/4)
= 4'513 IOs
  - 10'000/125/5 is the average number of blocks/department

PostgreSQL Query Optimization/1

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *

```
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 70000;
```

Data Output | Explain | Messages | History

r1, r2, Hash, Hash Join, Aggregate

OK. | Unix | Ln 4 Col 15 Ch 77 | 7 rows. | 6 ms

PostgreSQL Query Optimization/2

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *

```
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 70000;
```

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=125743.81..125743.82 rows=1 width=0)
2	-> Hash Join (cost=55330.18..123406.99 rows=934725 width=0)
3	Hash Cond: (r.unique1 = s.unique1)
4	-> Seq Scan on r1 r (cost=0.00..41911.12 rows=934725 width=4)
5	Filter: (unique1 > 70000)
6	-> Hash (cost=39412.08..39412.08 rows=1000008 width=4)
7	-> Seq Scan on r2 s (cost=0.00..39412.08 rows=1000008 width=4)

OK. | Unix | Ln 4 Col 15 Ch 77 | 7 rows. | 6 ms

PostgreSQL Query Optimization/3

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *

```
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 700000;
```

Data Output | Explain | Messages | History

r2, Hash Join, Aggregate, i1, r1, Hash

OK. | Unix | Ln 4 Col 22 Ch 84 | 9 rows. | 8 ms

PostgreSQL Query Optimization/4

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *

```
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 700000;
```

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=103409.52..103409.53 rows=1 width=0)
2	-> Hash Join (cost=43633.16..102659.08 rows=300177 width=0)
3	Hash Cond: (s.unique1 = r.unique1)
4	-> Seq Scan on r2 s (cost=0.00..39412.08 rows=1000008 width=4)
5	-> Hash (cost=38854.95..38854.95 rows=300177 width=4)
6	-> Bitmap Heap Scan on r1 r (cost=5690.73..38854.95 rows=300177 width=4)
7	Recheck Cond: (unique1 > 700000)
8	-> Bitmap Index Scan on i1 (cost=0.00..5615.69 rows=300177 width=0)
9	Index Cond: (unique1 > 700000)

OK. | Unix | Ln 4 Col 22 Ch 84 | 9 rows. | 8 ms

Summary/1

- Query evaluation techniques:
  - Physical sorting:
    - Physical sorting is a basic and important technique
    - The same sort order should be useful to many operators and not just one (global optimization versus local optimization)
  - Evaluation techniques for selections:
    - Use primary index if available; secondary index is much worse
    - Equality conditions are selective and should be optimized
    - Linear scan with sequential IO is the base line for selections
  - Evaluation techniques for joins:
    - nested loop: base line; avoid whenever possible
    - sort merge: robust and fast
    - hash join: fastest; only for equality

Summary/2

- Query optimization techniques
  - **Equivalence rules** for relational algebra expressions (must hold for multisets)
  - **Rule-based query optimization** is based on heuristics (usually the goal is to keep intermediate results as small as possible)
  - **Cost-based query optimization** uses statistical information to find the cheapest (or reasonably cheap) plan

Database Systems
Spring 2017

Transaction Processing
SL09

- Transactions
  - ACID Properties, Schedules, Serializability, Recoverability
- Concurrency Control
  - Lock-based Protocols, Transactions in SQL, Deadlock Handling
- Recovery System
  - Log-Based Recovery, Deferred/Immediate Modifications

Literature and Acknowledgments

Reading List for SL09:

- Database Systems, Chapters 20, 21, and 22, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Database Systems, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Addison Wesley, 2004.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Transaction Concept

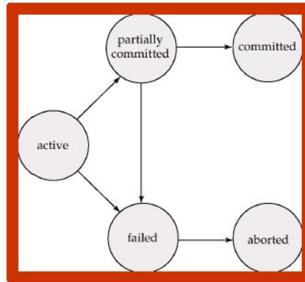
- **Transaction:** A logical unit of program execution (i.e., a sequence of actions) that accesses and possibly updates various data items. A transaction is never executed partially and it includes one or more DB access operations (insertion, deletion, modification, retrieval)
- For a transaction the following must hold:
  - A transaction must see a consistent database
  - During transaction execution the DB may be inconsistent.
  - When the transaction is committed, the DB must be consistent
- Two main issues to deal with:
  - Concurrent execution of multiple transactions
  - Various failures, e.g., hardware failures and system crashes

Transactions

- Transaction States
- ACID Properties
- Schedules
- Serializability
- Recoverability

Transaction States

- **Active:** (initial state) The transaction stays in this state during execution.
- **Partially committed:** After the final statement has been executed.
- **Committed**: Transaction has successfully completed and changes are permanent.
- **Failed:** After the discovery that normal execution can no longer proceed.



- **Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Two possible options after a transaction has been aborted:
  - Restart the transaction
  - Kill the transaction

ACID Properties/1

- To preserve integrity of data, a transaction must meet the **ACID** properties:
  - **Atomicity:** A transaction's changes to the state are atomic, i.e., either all operations of the transaction are properly reflected in the DB or none are. (⇒ recovery manager)
  - **Consistency**: A transaction is a correct transformation of a state.The actions (taken as a group) do not violate any of the integrity constraints associated with the state. (⇒ application programs and integrity checker)
  - **Isolation:** Although multiple transactions may execute concurrently, it appears to each transaction that all other transactions are either executed before or after. (⇒ concurrency manager)
  - **Durability:** After a transaction completes (commits) successfully, the changes it has made to the database persist, even if there are system failures. (⇒ recovery manager)

ACID Properties/2

- Example: Transaction to transfer \$50 from account A to account B:
  1. ***read*** *(A)*
  2. *A := A - 50*
  3. ***write*** *(A)*
  4. ***read*** *(B)*
  5. *B := B + 50*
  6. ***write*** *(B)*
- ACID properties:
  - ***Consistency requirement:*** the sum of A and B is unchanged by the execution of the transaction.
  - ***Atomicity requirement:*** if the transaction fails after step 3 and before step 6, the system should ensure that the updates are not reflected in the DB, else an inconsistency will result.

ACID Properties/3

- Example (contd.)
  - **Durability requirement:** once the user has been notified that the transaction has completed (i.e., the \$50 are transferred), the updates to the DB by the transaction must persist despite failures.
  - **Isolation requirement:** if between steps 3 and 6, another transaction is allowed to access the partially updated DB, it will see an inconsistent DB (the sum $A + B$ will be less than it should be). This might result in an inconsistent DB state after the completion of both transaction, e.g., if the second transaction performs updates on A and B.
    - These problems can be avoided trivially by running transactions serially, i.e., one after the other.
    - However, executing multiple transactions concurrently has significant benefits in performance.

ACID Properties/4

- The **recovery manager** of a DBMS implements the support for atomicity and durability.
- The **concurrency control system** restricts the interactions between concurrent transactions in order to ensure isolation.
- Advantages of running multiple transactions concurrently in the system:
  - **increased processor and disk utilization**, leading to better transaction throughput: one transaction can use the CPU while another reads from or writes to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- It is the task of **application programs** to ensure consistency:
  - Each transaction preserves DB consistency.
  - Serial execution of a set of transactions preserves DB consistency.

Schedules/1

- **Schedule (or history):** Sequence of instructions from a set of concurrent transactions that indicate the chronological order in which these instructions are executed.
  - Must consist of all instructions of all transactions.
  - Must preserve the order of instructions within each individual transaction.
- **Serial schedule:** Transactions execute one after the other.
  - One transaction is completely finished before another transaction starts.

Schedules/2

- **Example:** Consider the following transactions:
  - T_1 transfers \$50 from *A to B*
  - T_2 transfers 10% of the balance from *A to B*
- The following is a serial schedule, i.e., $< T_1; T_2 >$, in which T_1 is followed by T_2.
- Integrity constraint
  - Sum of A + B is preserved

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedules/3

- **Example:** (contd.)
- The following schedule is not a serial schedule, but it is **equivalent** to the previous one (i.e., gives the same result).
- In both schedules the sum of A + B is preserved.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedules/4

- **Example:** (contd.)
- The following concurrent schedule does not preserve the value of the sum A + B.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- **Serializable schedule:** A schedule is serializable if it is equivalent to a serial schedule.
- There exist different forms of **schedule equivalence**.
- Examples are *conflict equivalent* and *view equivalent*.
- We consider conflict equivalence.
- In order to reason about transactions we model transactions as sequences of basic **read** and **write** operations:
- We assume that transactions may perform arbitrary computations on data in local buffers in between **reads** and **writes**.

Conflict Serializability/1

- Instructions I_i and I_j of transactions T_i and T_j **conflict** iff there exists a data item Q accessed by I_i and I_j and at least one of these instructions is a write operation on Q, i.e.,
  - $I_i = \textbf{write}(Q)$ and $I_j = \textbf{write}(Q)$
  - $I_i = \textbf{read}(Q)$ and $I_j = \textbf{write}(Q)$
  - $I_i = \textbf{write}(Q)$ and $I_j = \textbf{read}(Q)$
- $I_i = \textbf{read}(Q)$ and $I_j = \textbf{read}(Q)$ do not conflict since the order in which the two instructions are executed does not matter.

Conflict Serializability/2

- The goal is to determine **transformations** of schedules that generate equivalent schedules.
- Assume a schedule S and two consecutive instructions I_i and I_{i+1} from different transactions. Instructions I_i and I_{i+1} can be **swapped** if they are **non-conflicting**, i.e., if
  - both are read instructions, or
  - they refer to different DB items, or
  - one of them is not a DB operation (i.e., not a read or write)

Conflict Serializability/3

- **Conflict equivalent schedules:** If a schedule S can be transformed into a schedule S' by a series of nonconflicting swaps of consecutive instructions then S and S' are conflict equivalent.
- **Conflict serializable schedule:** A schedule S is conflict serializable iff it is conflict equivalent to a serial schedule.

Conflict Serializability/4

- **Example:** A schedule that is not conflict serializable

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain the serial schedule $< T_3; T_4 >$ or the serial schedule $< T_4; T_3 >$.

Conflict Serializability/5

- Example: The following schedule is conflict serializable, since it can be transformed into $< T_1; T_2 >$ by nonconflicting swaps.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Beyond Conflict Serializability/1

- **Example:** The following schedule is not conflict serializable by non-conflicting swaps.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- The above schedule is equivalent to the serial schedule $< T_3, T_4, T_6 >$.
- The schedule includes **blind writes**, i.e., write operations without having performed a read operation.

Beyond Conflict Serializability/2

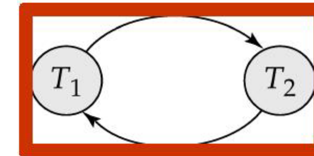
- The schedule given below produces the same outcome as the serial schedule $< T_1, T_5 >$, yet it is not conflict equivalent to it.

T_1	T_5
read(A)	
A:= A - 50	
write (A)	
	read(B)
	B:= B - 10
	write (B)
read(B)	
B:= B + 50	
write (B)	
	read(A)
	A:= A + 10
	write (A)

- Determining such equivalence requires to analyse operations other than read and write.

Testing for Conflict Serializability/1

- Consider a schedule S of transactions $T_1, T_2, ..., T_n$ (we use $r_i(X)$ to denote that transaction T_i reads item X).
- **Precedence graph (conflict graph):** A directed graph with a node T_i for each transaction and with an edge $T_i \rightarrow T_j$ iff one of the following conditions holds:
  - T_i executes write(Q) before T_j executes read(Q)
  - T_i executes read(Q) before T_j executes write(Q)
  - T_i executes write(Q) before T_j executes write(Q)
- A schedule is **conflict serializable** if and only if its precedence graph is **acyclic**
- Example of cyclic precedence graph:



Review 9.1

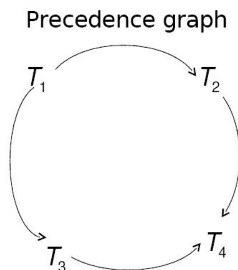
Draw conflict graphs for the following schedules:

1. $w_1(A); w_2(B)$
2. $r_1(A); r_2(A)$
3. $r_1(A); r_2(A); w_1(A); w_2(A)$
4. $r_1(A); w_1(A); r_2(A); w_2(A)$

Testing for Conflict Serializability/2

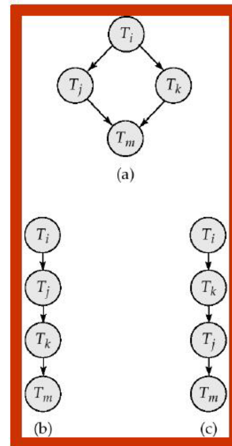
- **Example:** A conflict serializable schedule with 5 transactions and precedence graph

T1	T2	T3	T4	T5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				



Testing for Conflict Serializability/3

- Cycle-detection algorithms exist which take $O(n^2)$ time, where n is the # of vertices in the graph.
- If the precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. This is a linear order consistent with the partial order of the graph
  - e.g., a serializability order for the schedule in the previous example would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
- Topological sorting example



Concurrency Control versus Serializability Tests

- Testing a schedule for serializability after it has executed is too late.
- **Goal:** Develop concurrency control protocols that will assure serializability.
- **Concurrency control protocols** will generally not examine the precedence graph as it is being created; instead a **protocol** will impose a discipline (i.e., a set of rules) that avoids non-seralizable schedules.
- Tests for serializability help to understand why a concurrency control protocol is correct.
- Examples of concurrency control protocols are lock-based protocols and multiversion concurrency control.

Recoverability/1

- Need to address the effect of transaction failures on concurrently running transactions.
- **Recoverable schedule:** For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i must appear before the commit operation of T_j.
- DBMS must ensure that schedules are **recoverable**.
- **Example**: The following schedule is not recoverable if T_9 commits immediately after the read operation.
  - If T_8 aborts, T_9 would have read an inconsistent DB state.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Recoverability/2

- **Cascading rollback:** A single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)
  - If T_{10} fails, T_{11} and T_{12} must also be rolled back.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- **Main problem:** Can lead to the undoing of a significant amount of work.

Recoverability/3

- **Cascadeless schedules:** For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j. This avoids cascading rollbacks.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Purpose of Transaction Manager

- Ensure ACID properties (A, I, and D)
- If only one transaction can execute at a time we get serial schedules, which provides a poor throughput (number of transactions per time unit).
  - Transaction acquires a lock on the entire DB before it starts and releases the lock after it has committed.
- Concurrency control schemes are a tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- **Desirable properties** of a schedule:
  - serializable
  - recoverable
  - preferably cascadeless

Concurrency Control

- Lock-Based Protocols
- Pitfalls of Lock-Based Protocols
- Two-Phase Locking (2PL)
- Transactions in SQL
- Transaction in DBMSs

Lock-Based Protocols/1

- One way to **ensure serializability** is to require that data items be accessed in a mutually exclusive manner, i.e., while one transaction is accessing a data item, no other transaction can modify it.
- Locking is the most common mechanism to implement this requirement.
- **Locking**: Mechanism to control concurrent access to a data item.
- Lock requests are made to concurrency control manager.
  - Transaction can proceed only after request is granted.

Lock-Based Protocols/2

- Data items can be locked in two modes:
  - **exclusive mode (X)**: Data item can be both read as well as written. X-lock is requested using **X-lock(A)** instruction.
  - **shared mode (S)**: Data item can only be read. S-lock is requested using **S-lock(A)** instruction.
- Locks can be released: **U-lock(A)**
- **Locking protocol**: A set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols restrict the set of possible schedules.
  - Ensure serializable schedules by delaying transactions that might violate serializability.

Lock-Based Protocols/3

- **Lock-compatibility matrix** tells whether two locks are compatible or not.
  - Any number of transactions can hold shared locks on a data item
  - If any transaction holds an exclusive lock on a data item no other transaction may hold any lock on that item.

lock 1		lock 2	
		S	X
	S	TRUE	FALSE
	X	FALSE	FALSE

- **Locking Rules/Protocol**
  - A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
  - If a lock cannot be granted, the requesting transaction is made to wait until all incompatible locks held by other transactions have been released. The lock is then granted.

Pitfalls of Lock-Based Protocols/1

- Too early unlocking can lead to non-serializable schedules.
- Too late unlocking can lead to deadlocks.
- **Example**
  - Transaction T1 transfers $50 from account B to account A.
  - Transaction T2 displays the total amount of money in accounts A and B, that is, the sum A + B.

Pitfalls of Lock-Based Protocols/2

- **Example** (contd.): Early unlocking can cause **non-serializable schedules**, and therefore potentially incorrect results.
  - e.g., A = $100, B = $200
  - displaying A + B shows $250
  - $< T1; T2 >$ and $< T2; T1 >$ display $300

T1	T2
1. X-lock(B)	
2. read B	
3. B := B-50	
4. write B	
5. U-lock(B)	
6.	S-lock(A)
7.	read A
8.	U-lock(A)
9.	S-lock(B)
10.	read B
11.	U-lock(B)
12.	display A + B
13. X-lock(A)	
14. read A	
15. A := A+50	
16. write A	
17. U-lock(A)	

Pitfalls of Lock-Based Protocols/3

- **Example** (contd.): Late unlocking can lead to **deadlocks** (transactions block each other)

T1	T2
1. X-lock(B)	
2. read B	
3. B := B-50	
4. write B	
5.	S-lock(A)
6.	read A
7.	S-lock(B)
8. X-lock(A)	

- Neither T_1 nor T_2 can make progress:
  - **S-lock(B)** causes T_2 to wait for T_1 to release its lock on B.
  - **X-lock(A)** causes T_1 to wait for T_2 to release its lock on A
- To handle a deadlock one of T_1 or T_2 must be rolled back and its locks released.

Two-Phase Locking Protocol/1

- **Two-Phase Locking Protocol**: A locking protocol that ensures conflict-serializable schedules. It works in two phases:
  - Phase 1: **Growing Phase**
    - transaction may obtain locks
    - transaction may not release locks
  - Phase 2: **Shrinking Phase**
    - transaction may release locks
    - transaction may not obtain locks
- **Lock point**: Transition point form phase 1 into phase 2, i.e., when the first lock is released.

Two-Phase Locking Protocol/2

- **Example**: Schedule with locking instructions following the Two-Phase Locking Protocol

T1	T2
1. X-lock(B)	
2. read B	
3. B := B-50	
4. write B	
5. X-lock(A)	
6. U-lock(B)	
7.	S-lock(B)
8.	read(B)
9. read(A)	
10. A := A+50	
11. write(A)	
12. U-lock(A)	
13.	S-lock(A)
14.	read(A)
15.	display A + B
16.	U-lock(B)
17.	U-lock(A)

Two-Phase Locking Protocol/3

- **Properties** of the Two-Phase Locking Protocol
  - Ensures serializability
    - It can be shown that the transactions can be serialized in the order of their **lock points** (i.e., the point when a transaction acquired its final lock).
  - Does not ensure freedom from deadlocks
  - Cascading rollback is possible
- Modifications of the two-phase locking protocol
  - **Strict two-phase locking** (S2PL)
    - A transaction must hold **all its exclusive locks** until it commits/aborts
    - Avoids cascading rollback
  - **Rigorous two-phase locking** (SS2PL)
    - **All locks** are held till commit/abort.
    - Transactions can be serialized in the order in which they commit.

Review 9.2

Use Venn diagrams to relate

1. serial schedules (SS)
2. schedules (S)
3. conflict serializable schedules (CSS)
4. correct schedules (CS)
5. two-phase locking schedules (2PL)

and give representative examples.

Review 9.3

Consider transactions
T1: r(A); r(B); if A=0 then B:=B+1; w(B)
T2: r(B); r(A); if B=0 then A:=A+1; w(A)
Add lock and unlock instructions that follow 2PL. Show a 2PL schedule that leads to a deadlock?

Transactions in SQL/1

- The SQL standard defines three **undesired phenomena** of transactions:
  - **dirty read**: a transaction sees changes from other uncommitted transactions
  - **nonrepeatable read**: if a transactions retrieves a row twice it gets different answers
  - **phantom reads**: if a transaction retrieves a range of rows twice it retrieves a different answer

Transactions in SQL/2

- The SQL standard uses the undesired phenomena to define four **isolation levels** as follows:

Phenomena / Isolation level	Dirty read	Nonrepeatable read	Phantom read
read uncommitted	yes	yes	yes
read committed	no	yes	yes
repeatable read	no	no	yes
serializable	no	no	no

Review 9.4/1

Assume $r(a) = \{(1),(2),(3)\}$ and isolation levels read committed, read uncommitted, and serializability. What is the behavior of schedule:

T1: **UPDATE** p **SET** a=5 **WHERE** a=1

T2: **SELECT** ⋆ **FROM** p;

T1: **COMMIT;**

T2: **SELECT** ⋆ **FROM** p;

Review 9.4/2

Assume relation $r(a) = \{(1),(2),(3)\}$ and isolation level read committed. What is the behavior of the following schedule:

T1: **UPDATE** p **SET** a=5 **WHERE** a=1;
T2: **UPDATE** p **SET** a=5 **WHERE** a=1;
T1: **COMMIT;**

Transactions in DBMSs/1

- Database systems do not always make *serializable* the default isolation level.
- The SQL commands **COMMIT** and **ROLLBACK** finish the running transaction.
- Some database systems issue *autocommits* after each statement or at the end of sessions.
- PostgreSQL has a command **BEGIN** to start a transaction.
- Applications programs (graphical tools, etc) might implement their own transaction handling by issuing **COMMIT** and **ROLLBACK**.
- SQL does not permit to mix DDL and DML statement inside a transaction.

Transactions in DBMSs/2

- In order to increase performance (throughput) more and more DBMSs offer MVCC (multiversion concurrency control) rather than 2PL.
- Systems that use MVCC (Oracle, PostgreSQL) and 2PL (DB2, MySQL) behave differently.
- With MVCC the isolation level *serializable* does not permit dirty reads, nonrepeatable reads, and phantom reads (as required by the SQL standard), but true serializability is not guaranteed.
- With MVCC explicit locks must be used by the application to get serializability: **SELECT** ⋆ **FROM** p **FOR UPDATE;**

Transactions in DBMSs/3

- DB2:
  - DB2 uses 2PL
  - To rule out phantom reads tables are locked.
  - With 2PL the isolation level serializable corresponds to true serializability
  - DB2 deviates from the SQL terminology and offers the following isolation levels: repeatable read (RR), read stability (rs), cursor stability (cs), and uncommitted read (ur)
  - Example:

```
UPDATE command options USING C off;
SET current ISOLATION LEVEL RR;
SELECT SUM(a) FROM p;
INSERT INTO p VALUES (6);
SELECT SUM(a) FROM p;
COMMIT;
```


Review 9.5/1

Assume relation $r(a) = \{(1), (2), (3)\}$ and isolation level serializable with 2PL. What is the behavior of the following schedule:

```
T1: INSERT INTO p VALUES(5);
T2: SELECT * FROM p;
T1: COMMIT;
T2: COMMIT;
```


Review 9.5/2

Assume relation $r(a) = \{(1), (2), (3)\}$ and 2PL with isolation level set to serializability. What is the behavior of the following schedule:

```
T1: INSERT INTO p VALUES (5);
T2: SELECT * FROM p;
T1: COMMIT;
T2: SELECT * FROM p;
T1: INSERT INTO p VALUES (3);
T2: COMMIT;
T1: COMMIT;
```


Review 9.5/3

Assume relation $r(a) = \{(1), (2), (3)\}$ and 2PL with isolation level set to serializability. What is the behavior of the following schedule:

```
T1: SELECT SUM(a) FROM p;
T1: INSERT INTO p VALUES (6);
T2: SELECT SUM(a) FROM p;
T2: INSERT INTO p VALUES (6);
T1: SELECT SUM(a) FROM p;
T2: SELECT SUM(a) FROM p;
T2: COMMIT;
T1: SELECT SUM(a) FROM p;
T1: COMMIT;
T1: SELECT SUM(a) FROM p;
T2: SELECT SUM(a) FROM p;
```

Deadlock Handling/1

- Consider the following two transactions:

  T1: write (A) T2: write(B)
  write(B) write(A)

- Schedule with deadlock

T_1	T_2
X-lock(A)	
write(A)	
	X-lock(B)
	write(B)
	waits for X-lock on A
waits for X-lock on B	

Deadlock Handling/2

- **Deadlock**: A system is in a deadlock state if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- A deadlock has to be resolved by rolling back some of the transactions involved in the deadlock.
- Deadlocks are addressed in two ways:
  - Deadlock prevention protocols are used
  - Deadlocks are detected and resolved

Deadlock Prevention Protocols

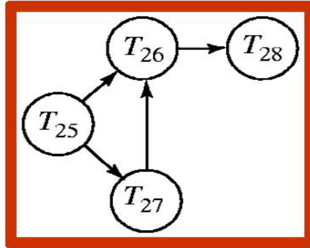
- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state.
- Two deadlock prevention protocols are *wait-die* and *wound-wait*. They use transaction timestamps to prevent deadlocks.
- With both protocols rolled-back transactions are restarted with their original timestamp. Since older transactions have precedence over newer ones starvation is avoided.
- Timeout-based protocols can be used to avoid deadlocks:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Thus deadlocks are not possible.
  - Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection and Recovery/1

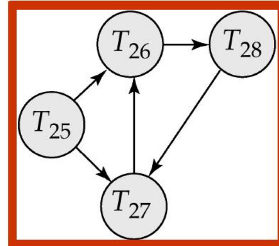
- Deadlocks can be described as a **wait-for graph**, which consists of a pair G = (V,E),
  - V is a set of vertices representing all the transactions
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- If T_i requests a data item being held by T_j, edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection and Recovery/2

- Wait-for graph without a cycle



- Wait-for graph with a cycle



Deadlock Detection and Recovery/3

- When a deadlock is detected, the system must **recover** from the deadlock.
- The most common solution is to roll back one or more transactions to break the deadlock. Three actions are required:
  1. **Selection of a victim**: For rollback select the transaction(s) that will incur minimum cost.
  2. **Rollback**: Determine how far to roll back transaction
     - Total rollback: Abort the transaction and then restart it.
     - Partial rollback: roll back transaction only as far as necessary to break deadlock.
  3. **Check Starvation**: happens if same transaction is always chosen as victim.
     - Include the number of rollbacks in the cost factor to avoid starvation

Recovery System

- Log-Based Recovery
- Deferred DB Modifications
- Immediate DB Modifications
- Checkpoints

Recovery System/1

- **Recovery system**: Ensures atomicity and durability of transactions in the presence of failures (and concurrent transactions).
- Atomicity and durability properties of transactions:
  - A transaction either completes fully with a permanent result (i.e., committed transaction)
  - or does not happen at all and has no effect on the DB (i.e., aborted/rolled-back transaction if some error occurs)
- Transactions are aborted or rolled-back if some error occurs

Recovery System/2

- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the DB contents to a state that ensures atomicity, consistency and durability
- Problems of recovery procedures
  - The DBMS does not know which instruction was last executed.
  - Buffers may not have been written to the disk yet.
  - Observable (external) writes cannot be undone, e.g., writes to the screen or the printer. Possible solutions include:
    - Delay external writes until the end of the transaction (if possible).
    - forbid external writes.
    - Relax atomicity.

Recovery System/3

- To ensure atomicity the DBMS must first output information describing the modifications to “stable storage” without modifying the DB itself.
- A log is the most popular structure for recording DB modifications on stable storage
  - Consists of a sequence of **log records** that record all the update activities in the DB
  - Each log record describes a significant event during transaction processing

Log-Based Recovery/1

- Types of **log records**
  - $< T_i, \textbf{start} >$: if transaction T_i has started
  - $< T_i, X_j, V_1, V_2 >$: before T_i executes a write(X_j), where V_1 is the old value before the write and V_2 is the new value after the write
  - $< T_i, \textbf{commit} >$: if T_i has committed
  - $< T_i, \textbf{abort} >$: if T_i has aborted
  - $< \textbf{checkpoint} >$

Log-Based Recovery/2

- A log allows us to
  - write DB modifications to the disk
  - undo DB modifications (using the old value)
  - redo DB modifications (using the new value)
- Properties of logs
  - Logs must be placed on stable storage
  - Logs are large because they record all DB activities
  - Checkpoints are used to reduce the size of logs
    - Transactions that committed before a checkpoint don't have to be redone

Log-Based Recovery/3

- When a **failure occurs** the following two operations can be executed
  - **Undo**: restore DB to state prior to execution
    - **undo**(T_i) restores the value of all data items updated by transaction T_i to the old values.
    - undo must be idempotent, i.e., executing it several times must be equivalent to executing it once
  - **Redo**: perform the changes to the DB over again
    - **redo**(T_i) (re)executes all actions of transaction T_i, i.e., sets the value of all data items updated by T_i to the new values.
    - redo must be idempotent.
- Two approaches using logs
  - Deferred database modifications
  - Immediate database modifications

Deferred DB Modifications/1

- **Deferred DB Modification Scheme**: All DB modifications are recorded in the log but are deferred until the transaction is ready to commit (i.e., after partial commit)
- A transaction is ready to commit if the commit log-record has been written to stable storage, i.e., when transitioning to the committed state
- This schema is also known as NOUNDO/REDO

Deferred DB Modifications/2

- Actions after a **rolled back transaction**
  - The log is ignored; nothing has to be undone
- Actions after a **crash**
  - A transaction T_i needs to be redone if and only if a $< T_i, start >$ and a $< T_i, commit >$ record is in the log
  - To redo transactions the log has to be scanned forward.
- The old value in the log record is not needed for deferred DB updates.
- Any failure that does not result in the loss of information on non-volatile storage can be handled.

Deferred DB Modifications/3

- Example: Transactions T_0 and T_1 (T_0 executes before T_1)

T_0: **read**(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)

T_1: **read**(C)
C = C- 100
write(C)

Deferred DB Modifications/4

- Possible order of actual outputs to the log and the DB

Log	DB
$< T_0, \textbf{start} >$	
$< T_0, A, 950 >$	
$< T_0, B, 2050 >$	
$< T_0, \textbf{commit} >$	
	A = 950
	B = 2050
$< T_1, \textbf{start} >$	
$< T_1, C, 600 >$	
$< T_1, \textbf{commit} >$	
	C = 600

Deferred DB Modifications/5

- **Example (contd.)**: Consider the log after some system crashes and the corresponding recovery actions

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

(a) No redo actions need to be taken

(b) **redo**(T_0) must be performed since $< T_0, \textbf{commit} >$ is present

(c) **redo**(T_0) must be performed followed by **redo**(T_1) since $< T_0, \textbf{commit} >$ and $< T_1, \textbf{commit} >$ are present

Immediate DB Modifications/1

- **Immediate DB Modification Scheme**: DB modifications can be written to disk before a transaction commits. However, before doing so the modifications have to be written to the log first.
- Known as UNDO/REDO.

Immediate DB Modifications/2

- Actions after a **rolled back transaction**
  - The effects on the DB have to be undone.
- Actions after a **crash**
  - Transaction T_i needs to be **undone** if the log contains a $< T_i, \textbf{start} >$ record, but does not contain a $< T_1, \textbf{commit} >$ record
    - for undo the log must be scanned backwards
  - Transaction T_i needs to be **redone** if the log contains the record $< T_1, \textbf{start} >$ and $< T_1, \textbf{commit} >$
    - for redo the log must be scanned forwards
  - undo must be done before redo
- Any failure that does not result in the loss of information on non-volatile storage can be handled.

Immediate DB Modifications/3

- Example (contd.): Consider the log after some system crashes and the corresponding recovery actions

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

(a) **undo**(T_0): B is restored to 2000 and A to 1000

(b) **undo**(T_1) and **redo**(T_0): C is restored to 700, and then A and B are set to 950 and 2050, respectively

(c) **redo**(T_0) and **redo**(T_1): A and B are set to 950 and 2050, respectively; then C is set to 600

Review 9.6

List advantages and disadvantages of respectively, deferred and immediate database updates.

Checkpoints/1

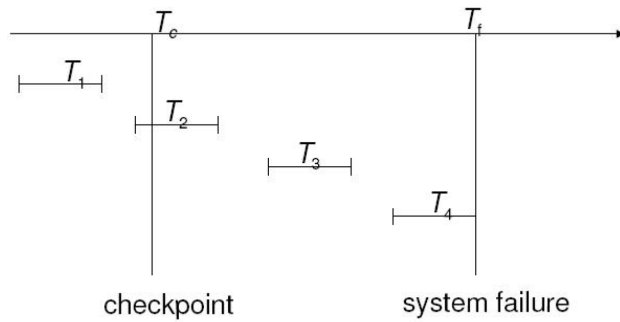
- Problems in recovery procedure
  - Searching the entire log is time-consuming
  - We might unnecessarily redo transactions which have already output their updates to the DB
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record $<$ **checkpoint** $>$ onto stable storage
- Any transaction T_i with a $<T_1,$ **commit** $>$ record before a $<$ **checkpoint** $>$ record in the log need not be considered after a system crash

Checkpoints/2

- **Recovery procedure**: Only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i need to be considered.
  1. Scan backwards from end of log to find the most recent $<$ **checkpoint** $>$ record
  2. Continue scanning backwards till a record $<T_i,$ **start** $>$ is found.
  3. Need only consider the part of log following $<T_i,$ **start** $>$ record. Earlier part of log can be ignored and can be erased.
  4. Scan forward the log (starting from T_i).
  5. For all transactions T_j with no $<T_j,$ **commit** $>$ record, execute **undo**(T_j).
     - Done only in case of immediate modification
  6. For all transactions T_j with $<T_j,$ **commit** $>$ record, execute **redo**(T_j).

Checkpoints/3

- **Example:**



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Summary

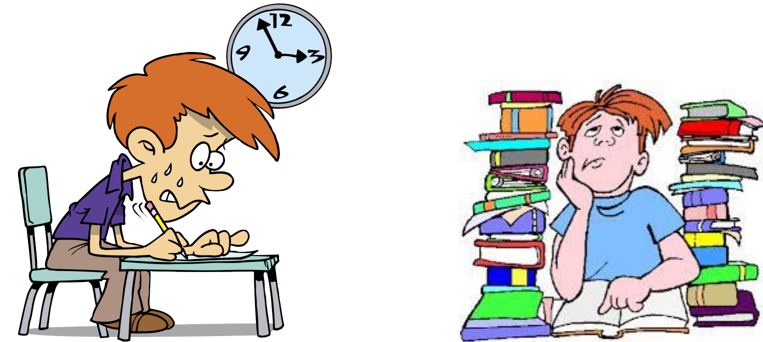
- Transactions
  - ACID properties of transactions:
    A(tomicity), C(onsistency), I(soltaion), D(urability)
  - Transaction states
  - Schedules (correct; serializable; conflict serializable; serial)
  - Conflict (or precedence) graph
  - Recoverability, cascading rollback
- Concurrency Control
  - Lock-based protocols
  - Two-phase locking (2PL, strict, rigorous)
  - Transactions (SQL, DBMSs)
  - Deadlocks (prevention; detection and recovery)
- Recovery
  - Log-based recovery
  - Deferred DB modifications
  - Immediate DB modifications
  - Checkpoints

Database Systems

Spring 2017

1. Exam and syllabus
2. DBS courses at IfI
3. BSc theses, etc.
4. Beyond the DBS course

Exam and Syllabus



The Exam

- The final exam is written and takes place
  Tuesday, June 20, 10:15 - 12:00 in BIN 1.B.01 and BIN 0.K.02
  (see VVZ web page for details).
- Auxiliary material during exam: 1 A4 sheet with notes.
- Course web page: http://www.ifi.uzh.ch/dbtg/
- The textbook is Database Systems by Elmasri and Navathe, 6th edition.

What is Important

- **Being precise** is important.
- **Solving relevant examples** is important.
- Understanding material in detail; **apply** to new examples.
- It is not sufficient to "know about it" or to "reproduce".
- You must understand and apply techniques learned during the course.
- Exercises are representative for exam and are the best preparation.
- Simple and readable solutions are important (this has an impact on your grade).

Preparation Material

- Exercise with solutions (practice to do exercises yourself before you look at solution)
- Lectures with reviews
- Slides
- Textbook
- Exams of the last five years (note that the material has evolved; e.g., definition of B+ tree; mapping from ER to relational DB)
- Open door policy in the database group

Syllabus/1

- **Relational model, algebra, and calculus**
  - Elmasri and Navathe: chapters 3 and 6
  - relational model
  - relational algebra (RA):
    σ, π, $\cup$, $-$, $\times$
    ρ, $\leftarrow$
    $\bowtie$, $\bowtie_{\theta}$, ⟕, ⟖, ⟗, ϑ, $\div$,
  - domain relational calculus (DRC), FOPL

  - practice Cartesian product
  - practice quantifiers
  - move between RA, DRC, natural language
  - be precise (e.g., qualified names do not exist in RA; use renaming)

Syllabus/2

- **SQL**
  - Elmasri and Navathe: chapters 4 and 5
  - data definition language, data manipulation language
  - query expressions, query specifications, orthogonality
  - subqueries
  - duplicates
  - null values
  - logical update semantics

  - practice formulation of declarative queries
  - consider effects of duplicates and NULL values
  - solve and try out SQL solutions with PostgreSQL
    - important is systematic plan: input, output, modular SQL code
    - avoid trial and error
  - be conservative with SQL features

Syllabus/3

- **Constraints, triggers, views, DB programming**
  - Elmasri and Navathe: chapters 5, 12 and 25
  - views, with clause
  - column constraints, table constraints, assertions, referential integrity
  - functions, triggers, stored procedures
  - expressiveness, recursion

  - know key concepts and their properties
  - know when and how to use these concepts; use as appropriate; helps to solve specific problems or break down solutions into smaller parts
  - details of extended SQL syntax are not the crucial part

Syllabus/4

- **Relational database design**
  - Elmasri and Navathe: chapters 14 and 15
  - design goals, redundancy, keys
  - functional dependencies, Armstrong's inference rules
  - 1NF, 2NF, 3NF, BCNF, 4NF
  - normalization algorithm
  - dependency preservation, lossless join decompositions
  - closure, equivalence, minimal cover

  - definition of FDs and MVDs
  - definition of normal forms, dependency preservation and losless join decomposition
  - application of inference rules and normalization algorithm

Syllabus/5

- **Conceptual database design**
  - Elmasri and Navathe: chapters 7 and 8
  - conceptual design process: ER model, entities, attributes, relationships
  - weak entities, specializations
  - 8 step ER-to-relational mapping

  - ER diagrams: construct ER diagrams from real world descriptions
  - no unique solution; make clarifying assumptions during design
  - analyze strengths and weaknesses of an ER diagram
  - extend and modify ER diagrams
  - use and apply 8 step mapping algorithm

Syllabus/6

- **Physical database design**
  - Elmasri and Navathe: chapters 16 and 17
  - seek time, latency, block read time
  - file and buffer manager
  - indexing: secondary and primary index
  - B+ tree
  - extendable hashing

  - compute basic characteristics (nr of blocks, nr of IOs, etc)
  - know definitions of B+ tree and extendable hashing
  - apply algorithms on concrete examples
  - use B+ tree definitions and algorithms from slides (other solutions are not valid)

Syllabus/7

- **Query processing and optimization**
  - Elmasri and Navathe: chapters 18
  - measures of query cost
  - sorting (external sort merge)
  - selection (scan, binary search, index)
  - join (nested loop, block nested loop, sort merge, hash join)
  - algebra trees, evaluation plans
  - heuristic and cost-based query optimization

  - compute cost of operations
  - algorithms for sorting, selection and join
  - transformation (rewriting) of relational algebra expressions
  - interpretation of query plans

Syllabus/7

- **Transaction processing**
  - Elmasri and Navathe: chapters 20, 21, 22
  - transactions, schedules, serializability, recoverability, ACID properties
  - concurrency control, locking, protocols, deadlocks
  - transactions in SQL
  - recovery, database log

  - schedules, deadlocks, recoverability
  - conflict, conflict serializability, conflict graphs
  - lock-based protocols, 2PL and variants
  - immediate and deferred database, redo, undo

Database System Courses at IfI

Database Systems Courses @IfI

- **Database Systems**, Spring (sem4)
- **Praktikum Datenbanksysteme**, Fall (sem5)
- **Distributed Databases**, Fall (sem5)
- **Seminar in Database Systems**, Spring (sem6, sem8, sem12)
- **XML and Databases**, Spring (sem8, sem6)
- **Data Warehousing**, Spring (sem8, sem6; even years only)
- **Nonstandard Databases**, Fall (sem9, sem7)

- **BSc thesis**, **MSc thesis**, **independent studies** (Vertiefung, Facharbeit, etc)

Andreas Geppert: Praktikum Datenbanksysteme

- Since 2016 Andreas Geppert is with SwissRe.
- From 2001 until 2016 Andreas Geppert was with Credit Suisse (as database architect).
- Before this he was a senior researcher in the Database Technology Research Group.
- He received his diploma in Computer Science from the University of Karlsruhe (Germany) in 1989 and his PhD in computer science from the University of Zurich (1994).
- From August 1998 to August 1999 he was a visiting scientist at the IBM Almaden Research Center.
- Praktikum Datenbanksysteme
  - Apply/practice your SQL knowledge on a case study
  - Dienstag 16:00 - 18:00

Andreas Geppert: Praktikum Datenbanksysteme

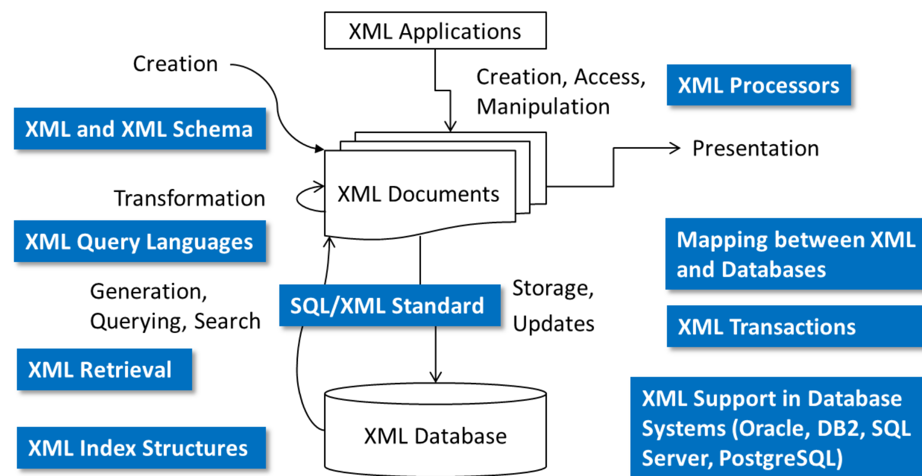
- Syllabus
  - relational database systems
  - conceptual and logical design
  - query languages
  - triggers, stored procedures
  - application development in Java (JDBC)
- Characteristic
  - Application development with PostgreSQL
  - Independent work with guidance
- Application (use case)
  - reservation system for car sharing (from the database to the web)
  - management of the car park, members and stations

Can Türker: XML and Databases

- Dr. Can Türker heads the Data Integration Group of the Functional Genomics Center Zurich (FGCZ). He holds a Ph.D. degree in computer science (1999) and is (co-)author of the several lecture books in the area of databases, among others 'Object-Relational Databases' and 'SQL:1999 & SQL:2003'.
- Lecture Hours: Thursday 8-10am
- Course content: XML is introduced with related technologies and it is shown how XML can be used for storing, accessing, querying, and updating data. The mapping between XML and databases as well as specific requirements arising from the usage of XML for data management are elaborated not only conceptually but are also demonstrated practically using todays major database systems.
- Prerequisite: The course expects background knowledge in database systems (especially in SQL).

Can Türker: XML and Databases

Goal: This lecture deals with the interplay of two essential technologies, namely XML and databases.



Topics for BSc Theses, MSc Theses, etc in the DBS Area

BSc and MSc Theses in the DBS Area

- We are happy to supervise BSc and MSc theses in the DBS area
- Relevant for all specializations and degrees
- Typically close interaction with an assistant with weekly meetings
- Algorithms, software development, scientific writing
  - Extension of PostgreSQL (parser, analyzer, query optimizer)
  - Incorporation of a load balancing algorithm into Apache Hadoop
  - Work with real world data (e.g., feedbase.ch)
  - Matrix factorization
- Work goes into details
- Contact assistant by email and ask for a meeting to discuss theses

Beyond the DBS Course

Beyond the DBS Course

- Jennifer Widom's (Stanford) recipe for database research:
  1. pick a fundamental assumption underlying database systems
  2. drop it
  3. solve resulting problem
- drop: fixed structure (schema) of the data
  - leads to: **semistructured databases, XML databases**
- drop: data sets are persistent and disk-resident
  - leads to: **stream processing systems**
- drop: tuple presence is certain
  - leads to: **probabilistic databases**

Beyond the DBS Course

- drop: banking applications with short update transactions
  - leads to: **data warehouses, OLAP**
- drop: sophisticated SQL queries
  - leads to: **key values stores, NoSQL**
- drop: centralized processing of entire database
  - leads to: **MapReduce, Hadoop**
- drop: data is stored row by row
  - leads to: **column stores**