Physical database design

- The learning objectives for this week are:
  - Knowing the purpose of **physical database design**
  - Knowing the basic types of **integrity constrains**
  - Knowing how to create a basic set of tables with integrity constraints in SQL Server
  - Knowing the advantages and disadvantages of **database indexes**
  - Knowing how to create indexes on database tables in SQL Server
  - Knowing what is meant by **database performance** and **database security**

Physical database design

- Once we have designed a logical database schema based on a specific data model, we can start designing the **physical implementation of the database**
- The **physical database design** is concerned with the **target DBMS product** and all **physical-level details**
- The target DMBS product could be for example SQL Server, PostgreSQL, or MySQL
- The designer should be fully aware of the functionality of the target DBMS and must know how the computer system hosting the DBMS operates
- For example in RDMSs, the common SQL syntax and functionality is the same, but there are still important differences
- In practice, physical design **must be tailored to a particular DBMS**

Creating tables

- In a SQL database the data is stored into **tables** that have **columns** with different **data types**, such as `VARCHAR(n)`, `INTEGER`, and `DATE`
- Different DMBS support mostly the same data types, but many of them also support their own, **non-standard data types**, such as the PostgreSQL's `JSONB` data type for storing JSON data
- A database table can be created with the `CREATE TABLE` statement:

```
CREATE TABLE Customer (
  customer_id INTEGER,
  name VARCHAR(50)
)
```

Database integrity

- When we create a table with the `CREATE TABLE` stamement, we can define different **constraints** for the table
- Constraint enforce the database integrity in different ways:
  - The `PRIMARY KEY` constraint enforces uniqueness of values of a primary key
  - The `UNIQUE` constraint enforces uniqueness of values of an alternate key
  - The `FOREIGN KEY` constraint enforces referential integrity
  - The `NOT NULL` constraint does not allow missing values
  - The `CHECK` constraint enforces a user-defined business rule

An example of constraints

```
CREATE TABLE Student (
  student_number INTEGER NOT NULL,
  -- the first_name column must have a value
  first_name VARCHAR(50) NOT NULL,
  surname VARCHAR(50) NOT NULL,
  -- the ssn column must have a unique value
  ssn VARCHAR(20) NOT NULL UNIQUE,
  email VARCHAR(50),
  study_advisor INTEGER,

  -- the student_number is the primary key columnc
  CONSTRAINT pk_Student PRIMARY KEY (student_number),
  -- the study_advisor is a foreign key column referencing the Teacher table
  CONSTRAINT fk_TeacherStudent FOREIGN KEY (study_advisor)
  REFERENCES Teacher(teacher_number),
  -- the email column must have a value that is in an email format
  CONSTRAINT chk_email_format CHECK (email LIKE '%_@_%._%')
)
```

Domain constraint

- In addition to these constraints, a **domain constraint** can be used to specificy the data type and a set of allowed values in a column
- A new domain can be created with the `CREATE DOMAIN` statement:

```
CREATE DOMAIN SizeDomain AS CHAR(2) CHECK (VALUE IN ('S', 'M', 'L', 'XL'))
```

- The new domain name can be used in column definitions instead of a built-in data type name
- The `CREATE DOMAIN` statement is support by DBMSs such as PostgreSQL, but **not** by SQL Server

Column autonumbering

- **Autonumbering** allows a unique number to be automatically generated when a new row is inserted into a table
- This is a useful option for **generating column values for surrogate primary keys**
- We can create a **autonumber column** for a table by using an extra option in the column definition
- In SQL Server, an autonumber column is defined with the `IDENTITY` property and it can be used with `TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `DECIMAL`, and `NUMERIC` data types

```
CREATE TABLE Customer (
  -- the customer_id column is a autonumber column
  customer_id INTEGER NOT NULL IDENTITY,
  name VARCHAR(50) NOT NULL,
  CONSTRAINT PK_Customer PRIMARY KEY (customer_id)
)
```

Example of column autonumbering

- When we insert a new row into the table, the **DBMS assigns automatically a value to the identity column**
- We **cannot** insert a value to the `IDENTITY` column explicitly:

```
INSERT INTO customer (name) VALUES ('Kalle Ilves')
INSERT INTO customer (name) VALUES ('Kari Silpiö')
```

customer_id	name
1	Kalle Ilves
2	Kari Silpiö

Database security

- **Database security** is are the mechanisms that **protect the database against intentional or accidental threats**
- These threats include for example **loss of confidentiality** and **loss of privacy**
- Organizations need to maintain secrecy over the data that is critical to them (confidentiality)
- At least as important is the need to protect data about individuals (privacy). For example individual's private information (such as social security number, email, phone number) should be accessible limitedly
- Database security is accomplished by **verifying the identity of the database users** (authentication) and **controlling what these users are permitted to do** (authorization)

User authentication

- The typical **user authentication** approaches are the following:
  - **SQL authentication**: a virtual username is registered in the DBMS instance and protected by a password. Anybody who knows the username and password can log in into the DBMS instance
  - **Operating system authentication**: selected users or user groups in an operating system domain are allowed to connect to the DBMS instance. That is, the DBMS trusts the authentication service of the operating system
- Out of these two the operating system authentication is considered more secure, because it uses more secure security protocol and enforces stricter password policies

User authorisation

- The typical **user authorisation** mechanism is called **discretionary access control** (DAC)
- The access control is based on the **privileges** that the DBMS system administrator or a database administrator grants to users:
  - On the **DBMS instance**: for example log in, create or drop databases, create users and roles **within the DBMS instance**
  - On the **database**: for example create tables, users, and other database objects **within the database**
  - On a **database object**: for example select, insert, update and delete **on a table**

Database performance

- A database commonly has certain **performance requirements**, for example a user don't want to wait for several seconds so that their discussion history is loaded in a messaging application
- The two main aspects of **database performance** are **response time** and **throughput**
- **Response time** is the time it takes for a user to receive the result for a certain SQL query (for example the result table of a `SELECT` query) they send to the DMBS
- Response time includes the CPU time, queuing within the operating system, disk access, lock waits in multi-user environment, network traffic and other required operations
- **Throughput** describes the overall capacity of the system to process data. It is measured in **database transactions** per second (TPS)
- A single database transaction may involve several database operations

Improving database performance

- There are many ways to improve the database performance, for example:
  - Creating indexes on tables
  - Tuning or rewriting individual SQL queries. For example replacing certain subqueries with `JOIN` operations
  - Distributing data access on disks
  - Increasing main memory of the database server

Database indexes

- Suppose that you are librarian in a library that has hundreds of books and you need to find the book by title "Dune"
- If you don’t have any information on the locations of different books, you have to go through **every single book** to find the book you are looking for
- If you have an ordered list of book titles and their locations you can quickly skip to letter "D" and find the location of the book you are looking for quite quickly
- In a database such "ordered list of book titles and their locations" resembles an **index**

Database indexes

- **Indexes** are critical for database performance, due to the fact that in data processing (reading and writing data) **disk access** is extremely slow compared to **main memory access**
- The whole database won't be able to fit into the main memory, but index is implemented with a compact data structure that at least partly fit into the main memory
- Based on a database query, index "points" to the locations of the result rows on the disk, which makes it possible for the DBMS to find data in the database with fewer disk accesses
- Indexes are created on **columns** or a **group of columns** to speed up queries that are referencing the columns in a `WHERE` or `JOIN` clause
- For example the following query would benefit from an index on the `title` column:

```
SELECT title FROM Book WHERE title = 'Dune'
```

Database indexes

- **Without an index**, the DBMS has to go through each row in the table (full table scan), which requires many **disk accesses** (slow 🐢)
- **With an index**, the DBMS just has to find the corresponding **index entry** in the index data structure. In most cases the index entry can be found in the **main memory** (fast ⚡)
- The index entry will point to data on the disk, which minimizes the required disk accesses

Creating an index

- We can create an index in the SQL Server using the `CREATE INDEX` statement:

```
CREATE INDEX ix_title ON Book(title)
```

- In this example, we create an index with name `ix_title` for the `Book` table's `title` column
- We can delete an index with a specific name in a table using the `DROP INDEX` statement:

```
DROP INDEX ix_title ON Book
```

Which queries benefit from the index?

- Index on a column makes it faster to query rows **based on that specific column**
- However, **other columns don't benefit from the index**

```
-- ⚡ This query is fast, it can use the index on the title column
SELECT title FROM Book WHERE title = 'Dune'
-- 🐢 This query is slow, it can't use the index on the title column
SELECT title FROM Book WHERE publish_year = 1965
-- ⚡ Also UPDATE and DELETE operations benefit from an index
UPDATE Book SET publish_year = 1965 WHERE title = 'Dune'
DELETE Book WHERE title = 'Dune'
```

Advantages of indexes

- If indexes are the magical way to search for rows based on a certain column value faster, then **why not add a index for every column on table?**
- Indexes have many **advantages**, such as:
  - Minimise the number of disk accesses needed to locate data in the database
  - Enforce `PRIMARY KEY` and `UNIQUE` constraints. If a suitable index exists, then the DBMS can use the index to find out if a duplicate value already exists for the key
  - Enforce `FOREIGN KEY` constraints. If a suitable index exists, then the DBMS can use the index to find out if any child row exists for a row that is being deleted
  - Accelerate `JOIN` operations
  - Avoid sorts for `DISTINCT`, `GROUP BY`, `ORDER BY` and `UNION` clauses

Disadvantages of indexes

- But there are also **disadvantages**, such as:
  - More main memroy and disk space is needed for indexes
  - The DBMS has to update all indexes on the table when a new row is inserted
  - Also, when an existing row is updated the DBMS has to update related indexes accordingly
  - That is, if there are **unnecessary indexes** on a table they might start decreasing performance on inserts, updates, and deletes

When to use indexes?

- In general, indexes are important for tables with a **large number of rows** and **frequently repeated queries**
- If the number of rows in a table is very small, then an index does not improve performance
- Indexes are not useful for columns that have **large values** (for example long strings) or a very **low selectivity** of values (for example a `is_completed` column that has only two possible values, `'Y'` or `'N'`)

Which columns should have an index?

- Typically, basic indexes are created on:
  - **Primary keys**: SQL Server **automatically** creates a **unique clustered index** on the primary key
  - **Foreign keys**: Use `CREATE INDEX` to create an index on a foreign key
  - **Alternate keys**: In SQL Server, `UNIQUE` constraint is physically implemented as a **unique non-clustered index**, which SQL Server creates **automatically**
- In addition, we can consider indexes on columns that are referenced in a `WHERE` or `JOIN` clause:

```
-- to improve the performance of this query,
-- we could create an index on the city column
SELECT first_name, surname FROM Student WHERE city = 'Helsinki'
```

Summary

- The purpose of the **physical database design** is to produce a description of all the database's **physical-level implementation details** for a **certain DBMS product**
- The physical database design should provide **adequate performance** and insure database **integrity**, **security** and **recoverability**
- **Database integrity** is ensured with **constraints** (for example `PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL` and `UNIQUE` constraints)
- **Database security** is are the mechanisms that **protect the database against intentional or accidental threats**
- Database security is accomplished by user **authentication** and **authorization**
- Database indexes are data structures which **speed up queries** at the cost of **main memory and disk space** and **slower write operations**