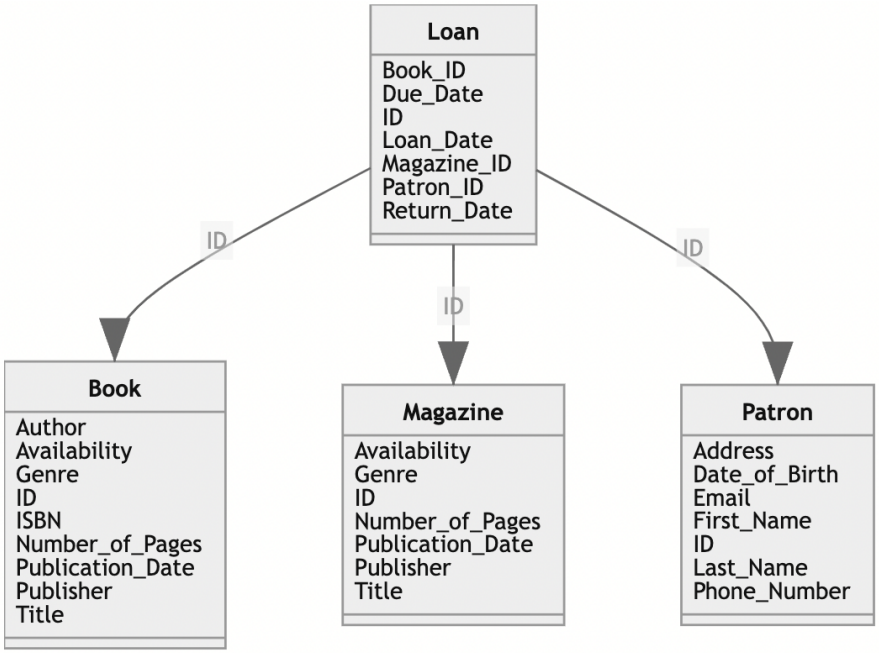
Databases - SP1

Github Repo: https://github.com/FrederikBA/DatabaseSP1

Opgave 1 (Database design)

A (Normalize the database to the 3.5 normal form (3.5NF)

Vi startede med at designe databasen ud fra scenariet beskrevet i opgaven. Det vi endte op med var fire tabeller, henholdsvis: Loan, Book, Magazine og Patron:



Opgave 1.A lyder på at normalisere databasen til normalform 3,5 (Boyce-Codd).

Vi har opfyldt dette ved at dele ´Loan´ tabellen op i to yderligere tabeller ved navn ´BookLoan´ & ´MagazineLoan.´ Dette er gjort da både ´Magazine´ og ´Book´ deler attributterne 'Due_Date', 'Loan_Date' og 'Return_Date.'

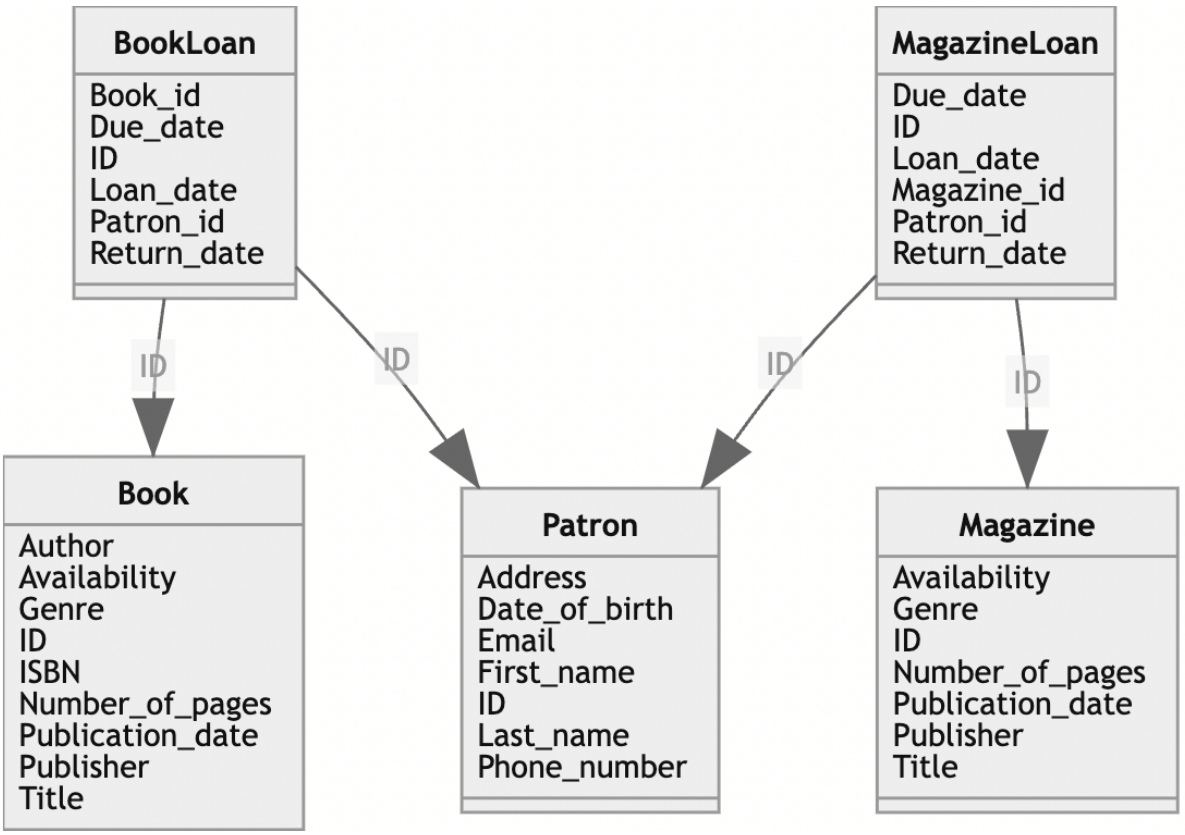
B (Evaluate the design, is denormalization necessary)

Vi mente, at databasen ikke havde behov for denormalisering for at minimere brugen af joins, således at man ville få et performance gain ved reads. De fleste logiske selects vi kunne tænke på, kræver ikke joins på tværs af tabellerne. Eksempler på scenarier vil være:

- At tjekke om en bog eller et magasin er ledig
- At slå et lån op ud fra enten bog eller låner.

C (Final database design diagram)

Herunder kan vi se et diagram af databasen, efter den er blevet normaliseret til 3.5nf også kaldet BCNF.



I både **Book** og **Magazine** tabellerne er hver kolonne funktionelt afhængig af primærnøglen, som er *ID*. I **Patron** tabellen er hver kolonne også funktionelt afhængig af primærnøglen, som er *ID*. Derudover har **BookLoan** og **MagazineLoan** tabellerne fremmednøgler, der henviser til primærnøglerne i andre tabeller.

Der er ingen tegn på nogen transitive afhængigheder i nogle af tabellerne, hvilket er hoved bekymringen i 3.5NF. Derfor opfylder disse tabeller kravene i BCNF.

D (Optional: Expand database with further tables and normalize to 5NF)

Vi valgte ikke at lave denne valgfrie opgave.

Opgave 2 (Stored Procedures + Transactions)

A (Stored procedure that inserts multiple records)

Denne stored procedure er lavet således at der er lavet en type ved navn **BookListType** som table, der indeholder alle parameterne. Herefter er proceduren lavet, hvor man referere til **BookListType** med en ny variabel ved navn **@BooksToAdd**. Man laver derefter transaktionen, hvor man tilføjer alle parameter fra **@BooksToAdd**.
Efter transaktionen, er der lavet en declare som indeholder de antal bøger, i vores tilfælde tre bøger, som vi ønsker at lægge ind i tabellen.
Således er der blevet tilføjet flere end én bog i en enkelt stored procedure.

B (Stored procedure that updates multiple records)

Denne stored procedure **sp_UpdateBooks** kan opdatere titlen på en eller flere bøger. Proceduren tager imod en varchar bestående af book ID'er separeret med komma samt en enkelt titel. Alle bøgerne der har et ID tilsvarende til de ID'er man har smidt med som parameter, vil få deres titel opdateret til denne nye titel.

C (Stored procedure that deletes multiple records)

Den gemte procedure *DeleteBooksAndBookLoans* er en transaktionsbaseret *delete procedure.* Proceduren er designet til at slette rækker fra to tabeller, **Book** og **BookLoan**, i en enkelt transaktion. Proceduren tager en kommasepareret liste over bog-ID'er, som skal slettes, som input. Vi forsøger først at slette rækkerne i tabellen **BookLoan** inden vi forsøger at slette de tilsvarende rækker i **Book** tabellen.
Hvis der ikke er nogen konflikter med foreign keys, slettes de tilsvarende rækker i **BookLoan** tabellen. Herefter slettes de ønskede rækker i **Book** tabellen.
Proceduren gør brug af transactions for at sikre, at hele processen udføres som en enkelt enhed. Hvis der opstår en fejl under sletningen af rækkerne, vil proceduren rulle transaktionen tilbage og returnere en fejlmeddelelse. Hvis sletningen af rækkerne lykkes, vil transaktionen blive bekræftet, og ændringerne vil blive gemt i databasen. På denne måde opretholder proceduren ACID-egenskaberne (Atomicity, Consistency, Isolation og Durability) i en relationel database, hvilket sikrer, at databasen altid forbliver i en konsistent og gyldig tilstand.

D (Stored procedure that retrieves data based on a certain criteria)

E (Dummy data)

Vi har brugt mockaroo.com til at genere dummy data i et query script for hver tabel. Her kunne man på hjemmesiden definere attribut navnene, deres datatyper og forskellige kriterier for genereringen. Disse scripts har vi gemt i query filer, således at vi i management studio kan populere databasen når nødvendigt.

Link: https://mockaroo.com/

F (indexing, partitioning, and caching)

Der er forskellige områder i vores database, hvor de forskellige teknikker kan anvendes til at forbedre ydeevnen:

1. Indeksering:
    - Book-tabellen kan indekseres efter ISBN-kolonnen, da denne kolonne sandsynligvis vil blive brugt til at søge efter bøger.
    - BookLoan- og MagazineLoan-tabellerne kan indekseres efter Patron_id og Book_id/Magazine_id kolonnerne, da disse kolonner sandsynligvis vil blive brugt til at finde lån, der er foretaget af en bestemt bruger eller til en bestemt bog/magasin.
2. Partitionering:
    - Hvis vores Book-tabellen vokser i størrelse, kan vi opdele den i flere partitioner baseret på publikationsdatoen. Dette vil gøre det lettere at søge efter bøger i en bestemt tidsperiode og også forbedre ydeevnen ved at reducere mængden af data, der skal søges igennem.
    - Hvis vores BookLoan- og MagazineLoan-tabeller også vokser i størrelse, kan vi også overveje at partitionere dem efter lånedatoen for at gøre søgninger mere effektive.
3. Caching:
    - Hvis vores database bliver brugt meget ofte af mange brugere, kan vi overveje at implementere caching. Dette indebærer at gemme ofte anvendte data i hukommelsen i stedet for at hente dem fra databasen hver gang. Vi kan for eksempel gemme populære bøger eller de seneste lån i hukommelsen, så de kan hentes hurtigere. Vi kan også bruge cache-teknikker til at gemme ofte anvendte forespørgsler i hukommelsen, så de kan besvares hurtigere.

Vores forsøg på optimering:

Vi har valgt at implementere indexing i vores database. Når man opretter en indeks på en kolonne i en tabel, opretter databasen en separat struktur, der sorterer værdierne i den kolonne på en bestemt måde og peger på placeringen af disse værdier i tabellen. Når man søger efter en værdi i denne kolonne, kan databasen derefter bruge indekset til at finde rækken, hvor værdien findes, hurtigere end hvis den skulle gennemgå alle rækkerne i tabellen. Dette kan føre til betydelige forbedringer i ydeevnen for forespørgsler, der involverer søgning efter data i store tabeller.

I tilfælde af vores database kunne indekseringen af ISBN-kolonnen i Book-tabellen føre til en hurtigere søgning efter bøger ved hjælp af ISBN-nummer, da databasen kunne bruge indekset til at finde de relevante rækker i tabellen hurtigere, i stedet for at skulle gennemgå hele tabellen hver gang. I billederne nedenunder kan det ses at efter vi implementerede indeksering på ISBN kolonnen gik CPU demanded fra 80.000 ned til 27.000 og duration gik fra 85.699 til 27.101.

Proflier på vores stored procedure: GetBooksLoansByISBNs uden indeksering:

Label	Value
EventClass	sql_batch_completed
StartTime	2023-02-18T12:47:27.653Z
TextData	EXEC GetBookLoansByISBNs '294435408-6,549034588-8,798098042-5,324599895-3,...
CPU	80000
Duration	85699
Reads	469
Writes	7
HostName	Frederiks-Air
SPID	54

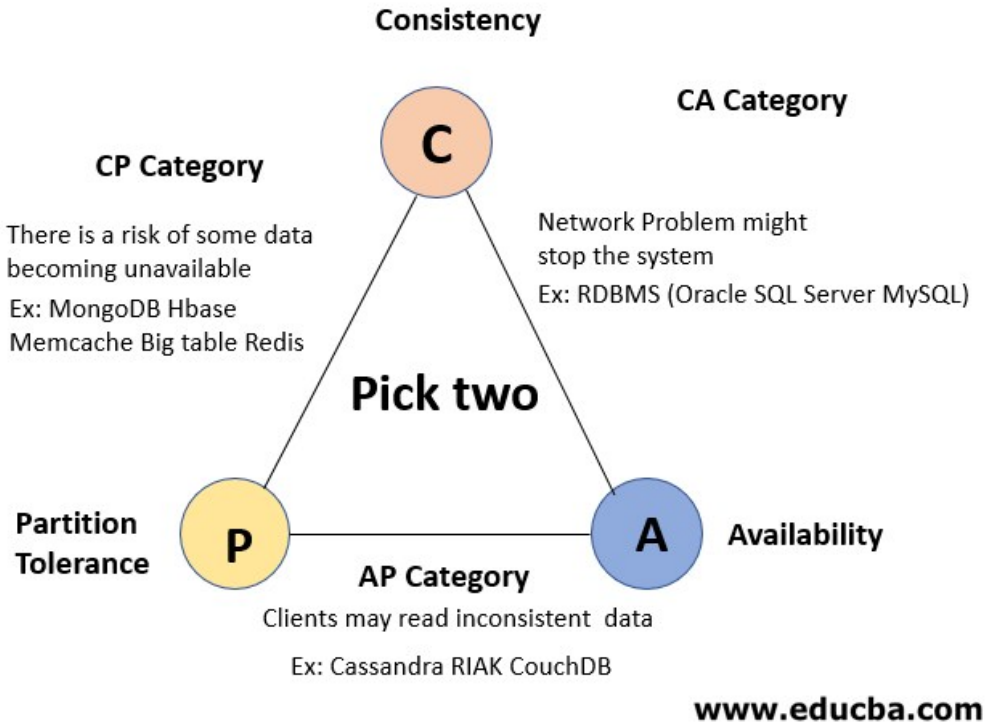
Proflier på vores stored procedure: GetBooksLoansByISBNs efter indeksering:

Label	Value
EventClass	sql_batch_completed
StartTime	2023-02-18T12:49:51.616Z
TextData	EXEC GetBookLoansByISBNs '294435408-6,549034588-8,798098042-5,324599895-3,...
CPU	27000
Duration	27101
Reads	104
Writes	0
HostName	Frederiks-Air
SPID	54

Opgave 3

A (Explain CAP theory and ACID properties and how they were applied to our design)

CAP Theory:



CAP Theorem, er den database model du ønsker din database er designet omkring. Der er tre muligheder: Consistency, Partition Tolerance og Availability. Ifølge CAP teori kan der kun vælges to af disse strukturer.

Consistency: Hvis en bruger opdaterer en stykke data på en node, vil alle andre noder straks se den samme opdaterede data. Dette er vigtigt i systemer, hvor dataintegritet er afgørende, såsom finansielle systemer.

Availability: Hvis en bruger sender en anmodning til en node i systemet, bør de altid modtage et svar, selv hvis andre noder er nede. Dette er vigtigt i systemer, hvor høj tilgængelighed er afgørende, såsom e-handelssystemer.

Partition tolerance: hvis en netværks afbrydelse opstår, bør systemet stadig være i stand til at fungere. Dette er vigtigt i systemer, der er distribueret på tværs af flere geografier eller datacentre.

Fordi at vi har lavet opgaven i MSSQL som er en relationel database så er den valgte CAP teori CA, som er det relational database er baseret på.

ACID Properties

ACID Properties er en betegnelse for en række egenskaber, der almindeligvis anvendes til at sikre pålideligheden og konsistensen af database-transaktioner.

De forskellige egenskaber i ACID blev brugt således
design and implementation

- **Atomicity:** har brugt ved at lave rollback, i tilfælde af at en transaktion fejler. Så vil den rollback til det tidligere stadie, så intet går tabt.

- **Consistency:** Et eksempel på, hvor vi bruger consistency i vores database, er ved brug af constraints såsom foreign key constraints, unique constraints og check constraints. Disse begrænsninger sikrer, at dataene i vores database overholder visse regler og krav til integritet. Hvis en transaktion forsøger at indsætte eller opdatere data, der bryder en af disse begrænsninger, vil transaktionen blive afvist, og databasen vil forblive i en consistent tilstand.

- **Isolation:** I MSSQL Server kan man opnå isolation ved at bruge "Read Committed" isolationsniveauet. Dette betyder, at en transaktion kun kan læse data, der er blevet bekræftet af en anden transaktion. Således kan en transaktion ikke læse data, der endnu ikke er blevet bekræftet, hvilket kan føre til unøjagtigheder i resultaterne. For eksempel, hvis en bruger låner en bog og en anden bruger forsøger at låne samme bog på samme tid, vil transaktionen fra den anden bruger blive blokeret indtil den første transaktion er færdig. Dette sikrer, at begge transaktioner har adgang til korrekte data og ikke overskriver hinandens handlinger.

- **Durability:** I vores database er durability normalt en implicit egenskab ved DBMS'en og påvirkes ikke direkte af vores database design eller implementation.

B (Reflections on the proces)

Denormalisering

Vi mente, at databasen ikke havde behov for denormalisering for at minimere brugen af joins, således at man ville få et performance gain ved reads. De fleste logiske selects vi kunne tænke på, kræver ikke joins på tværs af tabellerne. Det kan selvfølgelig være, at det på flere punkter havde været smart at denormalisere databasen, men ud fra databasens begrænsede størrelse og vores manglende viden på området, tog vi et valg om, ikke at denormalisere databasen.

Yderligere normalisering og design

Grundet begrænset forståelse af normaliserings formene efter Boyce-codd(3.5). Har vi haft svært ved at lave de valgfrie opgaver, og derfor sprunget dem over.

Optimering

I forsøget på at optimere ydeevnen af vores database, implementerede vi indeksering på vores ISBN kolonne i Book tabellen. Vi lærte og erfarede at indeksering kan markant forbedre hastigheden af søgningen ved at reducere antallet af poster, som skal behandles under en forspørgelse. Vi valgte ikke at implementere partitionering og caching, da det ville kræve mere planlægning og konfiguration end indeksering.

C (Conclusion and recommendations for future improvements)

Som gruppe er vi overordnet tilfreds med de tilgange og resultater vi har fået, gennem fx normalisering og optimering af databasen. Ved for eksempel indeksering opnåede vi et 4-dobbelt performance gain observeret på profileren.

Vi valgte i gruppen at tage nogle friheder og springe de valgfrie opgaver over. I en lignende situation i fremtiden, vil det måske være smart at give disse et skud for netop at forbedre vores begrænsede viden om normalisering på de højere normalformer.