Introduktion till Databaser

Mikael Svahnberg*

2024-04-12

1 Introduktion

- Introduktion till Persistent Lagring av Data och Databaser
  - Relationsdatabaser
  - NoSQL-databaser
  - Andra typer av databaser

Notera:

- Databaser handlar om mycket mer än vad vi täcker i den här kursen
- Ni kommer läsa en hel kurs bara om databaser senare.

2 Persistent Lagring

- Data som man vill skall överleva när man startar om programmet och/eller systemet
- Data som man vill enkelt kunna *söka* .
- Till exempel:
  - filer på filsystemet
  - poster i en databas
- Med filer *kan* man implementera sin egen databas men det blir jobbigt.
- Oftare ser man i varje fall skrivning som en enhet - man skriver *hela* filen.
- Om datorn kraschar under skrivning har man förlorat all data.
- Man är själv ansvarig för att skriva program som
  - Letar efter information
  - Ser till att all data har rätt format (heltal, flyttal, strängar, mm)
  - Ser till att alla associationer mellan olika data är uppdaterade och korrekta.

*Mikael.Svahnberg@bth.se

- En *Databas* är ett program som specialiserar sig på *lagring* och *sökning* av data.
- Ställ en fråga mot databasen, hellre än att skriva ett program som letar rätt på information.
- Lätt att skapa nya, mer komplexa frågor.
- Lätt att strukturera data och sätta regler på hur den skall vara formatterad.

3 Transaktioner

- En uppdatering kan bestå av flera delar, till exempel:
  1. Lista avstånden för alla betalningstariffer som finns för en viss stad.
  2. Hitta alla chaufförer inom de olika avstånden och som är tillgängliga
  3. Hitta alla fordon som dessa chaufförer får köra och som är tillgängliga och som är inom rätt avstånd
  4. Skapa en beställning på en körning från punkt A till punkt B med upphämtning klockan T_1 och avlämning klockan T_2.
  5. Välj lämpligt fordon och lägg till i beställningen
  6. Välj lämplig chaufför och lägg till i beställningen
  7. Markera fordonet som upptaget under tiden för beställningen
  8. Markera chauffören som upptagen under tiden för beställningen
  9. Schemalägg en påminnelse till Chauffören
  10. Schemalägg en påminnelse till fordonstvätten att de skall göra i ordning fordonet.
- Enklare exempel:
  1. Kontrollera om det finns mer än X kronor tillgängligt på ditt bankkonto
  2. Plocka bort X kronor från ditt bankkonto
  3. Sätt in X kronor på din hyresvärds bankkonto
- Om någon del av detta inte funkar (t.ex. det finns inga chaufförer lediga) så skall ingenting uppdateras.
- Om systemet kraschar mitt i så skall ingenting uppdateras.
- *Allt måste ske tillsammans* som en enda **Transaktion**

4 ACID

En grundläggande princip som styr hur databaser fungerar:

Atomicity Hela uppdateringen lyckas eller inget alls

Consistency Efter en uppdatering är all data korrekt konsekvent

Isolation Varje uppdatering sker för sig

Durability Resultatet från en lyckad uppdatering är permanent

5 Olika Typer av Databaser

1. *Key-Value stores*
   - Data (av alla möjliga sorter) lagras under en "nyckel", som sedan behövs för att söka fram datan.
   - Användbar bland annat i Big Data-bearbetning
2. *Document-orienterade Databaser*
   - Som *Key-Value*, men med indexerad och sökbar data
   - Man behöver inte ha en bestämd struktur på datan.
3. **Relationsdatabaser**
   - Den "klassiska" databasen
   - Fokus på *tabeller* och *relationer mellan tabeller*
   - Tydlig beskrivning av vilka fält datan består av, och fältens typ
   - Kan ställa komplicerade frågor där man hämtar information från olika tabeller
     - Nära matematikens mängdlära (set theory)
4. *Objekt-orienterade Databaser*
   - Som objektorienterad programmering i stort
   - Objekt kan flaggas som "persistenta"
5. *Graf-Databaser*
   - Data strukturerad som *Noder* och *Bågar*
   - Användbar när relationer mellan liknande noder är i fokus.
   - I en relationsdatabas definieras bågarna av frågorna man ställer,
   - I en graf-databas beskrivs bågarna tillsammans med noderna.

6 Ett exempel på en Relationsdatabas

Först beskriver vi våra tabeller:

- Ett *Fordon* har fälten `{_id, typ:varchar(50), position:point }`
- En *Chaufför* har fälten `{_id, namn:varchar(50),position:point }`

Samlingar får sina egna tabeller:

- Tabellen *ChaufförTillgänglighet* har fälten `{_id, chaufför_id, startTid:Date, slutTid:Date, tillgänglig:boolean}`
- Tabellen *ChaufförFordon* har fälten `{_id, chaufför_id, fordonstyp:varchar(50)}`

6.1 Exempel på tabeller

Exempel *Fordon*

_id	typ	position
0	Herrgårdsvagn	(56.182759,15.591855)
1	Roadster	(56.161405,15.586522)
2	Skåpbil	(56.182759,15.591855)
	...	

(På samma sätt för tabellen Chaufför)

Exempel *ChaufförTillgänglighet*

_id	`chaufför_id`	startTid	slutTid	tillgänglig
0	1	08:00	12:00	false
1	1	12:00	14:00	true
2	1	15:00	17:00	false
3	42	06:00	17:00	true
	...			

(notera att chauffören med id `1` har tre *rader* i tabellen). (PSS för tabellen ChaufförFordon)

6.2 Ställ frågor mot datan

Nu kan vi ställa frågor:

- Från ChaufförTillgänglighet, ge mig alla `chaufför_id` som är tillgängliga klockan 11:00
- Från ChaufförFordon, ge mig alla `fordonstyp` som dessa `chaufför_id` får köra
- Från Fordon, ge mig `positionen` för alla som matchar dessa `fordonstyp`
- Från Chaufför, ge mig `namn` för alla de tidigare `chaufför_id` där `Chaufför.position` är nära `Fordon.position`.

... Och helst vill vi ställa detta som en enda fråga.

7 Relationsdatabasprogram

- Relationsdatabaser har funnits länge, så det finns många leverantörer
- Länge var *Oracle* "det enda rätta", tätt följd av *IBM DB2*
  - *MySQL* är idag också Oracle
- De stora "drakarna" Amazon, Google, Microsoft har sina molnbaserade varianter.
- Microsoft *SQL Server* har faktiskt funnits med ganska länge...
- Open Source- alternativ
  - PostgreSQL
  - MariaDB
  - SQLite
- relationer, tabeller

8 SQL – Structured Query Language

- SQL *Structured (English) Query Language* är ett "programmeringsspråk" för att ställa frågor mot databaser
- Uppfanns redan 1974, har alltså gått hand i hand med utvecklingen av relationsdatabaser.
- Använder engelska hellre än matematisk notation (för relationsalgebra)

Exempel

```
CREATE TABLE weather (
     city    varchar(80),
     temp_lo int,
     temp_hi int,
     prcp    real,
     date    date
);

INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
       VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');

SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
SELECT * FROM weather WHERE city = 'San Francisco' AND prcp > 0.0;
SELECT * FROM weather ORDER BY city;
SELECT DISTINCT city FROM weather ORDER BY city;

DELETE FROM weather WHERE date < now() - interval '5 years';

DROP TABLE cities;
```

9 Relationsalgebra – Joins

- När man vill kombinera data från olika tabeller till mer avancerade frågor
  - t.ex. det tidigare exemplet med chaufförer och fordon
- Gemensamma kolumner, t.ex. `fordonstyp`, `_id`, ...
- För att unikt kunna identifiera varje rad designerar man några kolumner som *Nycklar*

  Primary Key de kolumner som tillsammans unikt identifierar en viss rad i en tabell.

  Foreign Key kolumn(er) som hänvisar till en viss rad i en annan tabell.

  Man sätter upp regler i databasen så att den kan hjälpa till med unikhet.

  Nycklarna används ofta, databasen kommer optimera med hjälp av dessa.

Fordon		ChaufförFordon		Chaufför			Kommenta
position	`fordonstyp`	`fordonstyp`	`chaufför_id`	`chaufför_id`	namn	position	
(X)	Skåpbil						Inga chauff
(Y)	Buss	Buss	8	8	Kalle	(q)	Kalle och
(Y)	Buss	Buss	12	12	Emma	(w)	
(ZZ)	Buss	Buss	8	8	Kalle	(q)	En annan
(ZZ)	Buss	Buss	12	12	Emma	(w)	Som Kalle

```
-- old school:
SELECT chauffeaur.name
       FROM vehicle, chauffeaurvehicle, chauffeur
       WHERE vehicle.type = chauffeaurvehicle.type
       AND chauffeaurvehicle.id = chauffeaur.id
       AND vehicle.position <@> chauffeaur.position;
-- (Cave! <@> -operatorn finns inte alltid)

-- with joins:
SELECT name FROM chauffeaur
       JOIN chauffeaurvehicle ON chauffeaurvehicle.id = chauffeaur.id
       JOIN vehicle ON chauffeaurvehicle.type = vehicle.type
       AND chaffeaur.position <@> vehicle.position;
```

9.1 Olika typer av Joins

INNER JOIN
- Den vanligaste typen av JOIN
- "snittet": Alla rader från båda tabellerna som mathar vilkoret.
- Resultatet har alla kolumner från både FROM-tabellen och från JOIN-tabellen.
- Upprepa posterna om fler rader matchar vilkoret.

LEFT JOIN
- också **LEFT OUTER JOIN**
- Alla rader från FROM och alla från JOIN som matchar

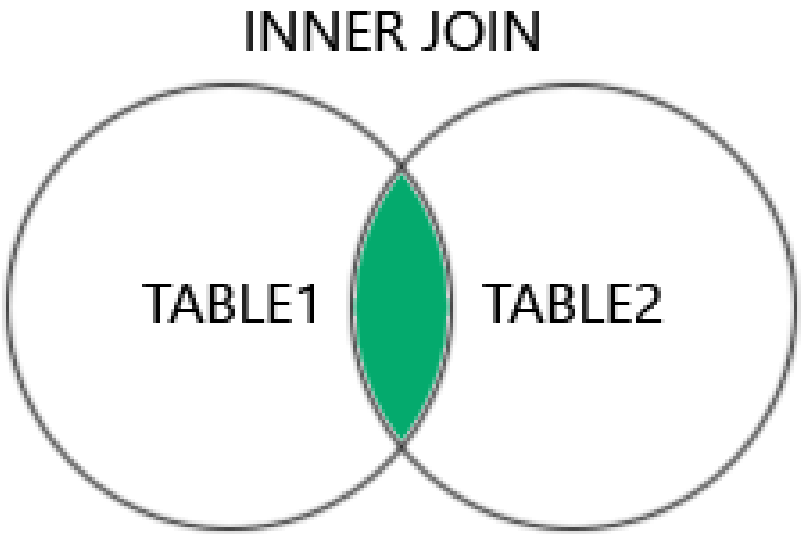
- Resultatet har alla kolumner från både FROM-tabellen och från JOIN-tabellen.
- Kolumnerna från JOIN-tabellen har värdet `NULL` där det inte fanns någon match.
- Upprepa posterna om fler rader matchar vilkoret.

RIGHT JOIN

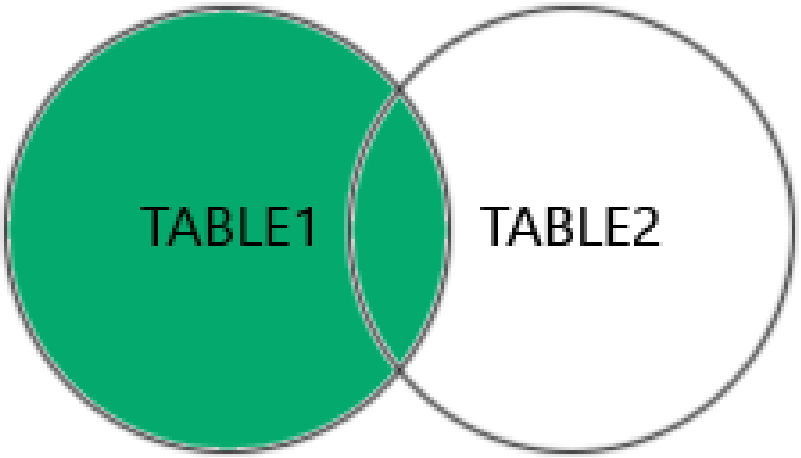
- också **RIGHT OUTER JOIN**
- Alla rader från FROM som matchar och alla rader från JOIN
- Resultatet har alla kolumner från både FROM-tabellen och från JOIN-tabellen.
- Kolumnerna från FROM-tabellen har värdet `NULL` där det inte fanns någon match.
- Upprepa posterna om fler rader matchar vilkoret.

FULL JOIN

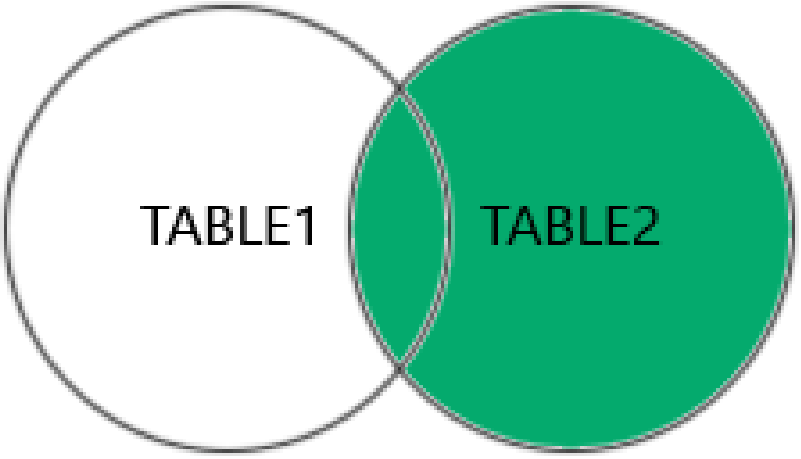
- "Unionen": Alla rader från FROM och alla rader från JOIN
- Resultatet har alla kolumner från både FROM-tabellen och från JOIN-tabellen.
- Kolumnerna från FROM-tabellen har värdet `NULL` där det inte fanns någon match.
- Kolumnerna från JOIN-tabellen har värdet `NULL` där det inte fanns någon match.

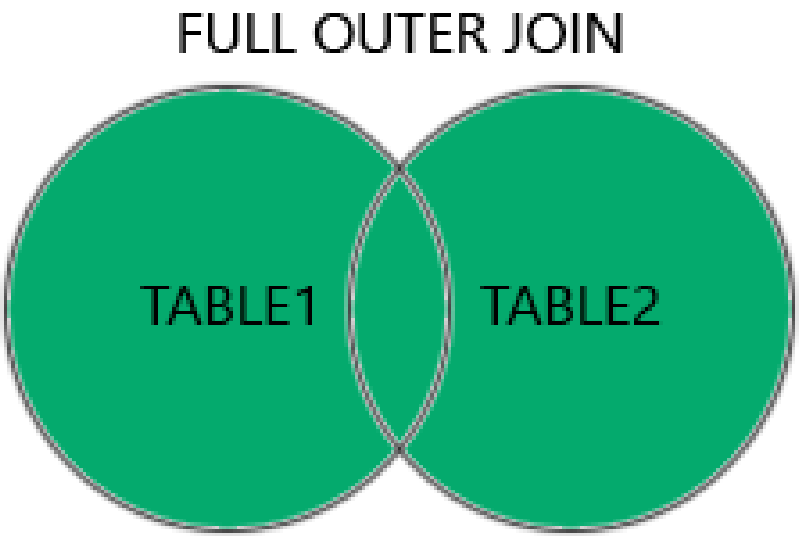


LEFT JOIN



RIGHT JOIN





9.2 När använder man olika Joins?

INNER JOIN
- filter, ta bort alla som inte har *både* X och Y
- samlingar, visa alla element y ∈ Y tillsammans med "ägaren" i X.

LEFT JOIN
- När x ∈ X *kanske* har ett eller flera värden i Y, men du är främst ute efter X.

RIGHT JOIN
- På samma sätt, när du främst är ute efter Y oavsett om det finns matchande x ∈ X eller inte.

FULL JOIN
- *give me **all** the data!*

10 NoSQL

- I en dokument-databas är det inte lika viktigt med relationsalgebra
- Varje dokument skall i stället vara sökbart på egen hand
- Dokumenten *kan* vara strukturerade, t.ex. en JSON-post.
  - ... men de behöver inte vara det. En text, en film, mätdata från en sensor, osv.
- Dokumenten *kan* ha samma eller liknande struktur, t.ex. vissa bestämda fält.
  - ... men de behöver inte ha det. Samma tabell kan innehålla texter, filmer, mm.
  - Min åsikt är att de *bör* ha en liknande struktur.
- Som namnet antyder använder sig NoSQL-databaser inte av SQL

  - Egna språk t.ex. för grafdatabaser
  - Egna API:er.
    * Med SQL förväntas man bygga ihop en textsträng med sin fråga.
    * Många gillar inte detta utan föredrar vanliga objektorienterade metodanrop.

```
await postgreClient.query('SELECT name FROM customers WHERE city=ANY($1)', [ ['London', 'P
await MongoDB.CustomersCollection.find({city: {$in: ['London', 'Paris']}});
```

11 NoSQL och Eventual Consistency

- NoSQL-databaser optimerar ofta för *klustring*
  - Dvs, flera databas-noder som samarbetar
  - De optimerar samtidigt också för att *skriva* ny data, snarare än att *läsa*
- De garanterar inte alltid ACID-konsekvens (Atomicity-Consistency-Isolation-Durability)
- I stället hävdar de *Eventual Consistency*
- Exempel:
  1. Användare A postar till `db-nod-1`
  2. Användare B läser från `db-nod-1` och ser den nya dataposten.
  3. Användare C läser samtidigt från `db-nod-2` och *ser inte* den nya dataposten.
  4. Databaserna synkroniserar, och nu kan användare C också se dataposten.
- I en relationsdatabas hade Användare A fått vänta tills transaktionen skrivits hos alla noder.
  - Varken användare B eller C hade sett dataposten förrän transaktionen var klar.
  - Databasens integritet värnas i alla lägen; ingen får någonsin se en databas mitt i en transaktion.
  - *Skrivaren* av data ser till att hela operationen omfattas av en och samma transaktion.
- I en Eventual Consistent-databas kan man råka läsa data mitt i en transaktion.
  - *Läsaren* av data måste ta hänsyn till att data kan vara inkonsekvent.

12 Med Stor Frihet Kommer Stort Ansvar

- NoSQL-databaser har ökat i popularitet
  - De är enkla att komma igång med
  - Man behöver inte bestämma hur data ser ut (tabeller och kolumner) innan man börjar koda
  - Sökningar skapas efter hand som man ser att de behövs
  - Relationerna kan kodas ihop i sitt favoritprogramspråk snarare än med relationsalgebra.
  - Man slipper planera först; det är bara att "koda loss".

  Tänk dock på:

- I ett stort projekt kan vem som helst skriva vilken data som helst.
- När du läser data vet du inte vilka fält som faktiskt finns.
- Du vet inte heller vad fälten används till.
- Utan plan skapar man en framtida underhållsbörda!

```
// This not efficient programming, it is is a passively aggressive word salad!
return DocumentStore.find(query).sort({time:1})
    .catch( err => { debug('Error while fetching data. \nQuery: %s \nError: %s', query, er
    .then( result => {
      if (result) {
          if (result.field) {
              if (!isNaN(parseFloat(result.field))) {
                  if (0 >= parseFloat(result.field)) {
                      finallyProcessResults(results); // Only here do we know that the dat
                  } else {
                      throw 'result.field is negative';
                  }
              } else {
                  throw 'result.field is not a float';
              }
          } else {
              throw 'result.field does not exist';
          }
      } else {
          throw 'did not receive any result from query';
      }
    })
    .catch( err => { debug('Error while parsing data. \nQuery: %s \nError: %s', query, err
```

13 Mer SQL: Hantera och Begränsa Svaren

WHERE Vilkor som måste stämma för att inkludera en rad i svaren

GROUP BY Samla alla rader som har samma värden till en enda rad

HAVING Filtrera svaren med ytterligare vilkor

SELECT DISTINCT Ge bara unika svar

ORDER BY Vilken kolumn skall användas för att sortera med

LIMIT Returnera bara de första `X` svaren.

OFFSET Hoppa över de första `Y` svaren.

- Används ofta tillsammans, ex. **LIMIT 20 OFFSET 40** ger 20 svar från och med svar 40.

```
SELECT name,address FROM customers
       WHERE customers.city = ANY( SELECT name from cities WHERE population>20000 )
       AND customers.loyaltyStatus = 'premium'
       GROUP BY address,name
       HAVING age>18
       ORDER BY city,address,name;
```

14 Transaktioner

- För mer komplicerade operationer som behöver brytas ner i fler steg.

BEGIN påbörjar en transaktion

COMMIT avslutar och verkställer transaktionen.

- Först då sker alla uppdateringar.
- Sökningar har redan gjorts, och databasen har *förberett* för uppdateringarna.

ROLLBACK Avbryt transaktionen.

- Något gick fel
- Genomför inga uppdateringar.

15 Hantera svaren: Allt på en gång eller Cursors

- Att få alla svaren i t.ex. en array skulle kunna kräva väldigt mycket minne.
- Många databaser returnerar i stället en s.k. *Cursor* till resultaten.
- Man får sedan "hämta ut" lagom många resultat i taget från cursorn.

```
const {Client} = require('pg');
const Cursor = require('pg-cursor');
const connection = { host: 'db', database: 'FleetManagement', user: 'postgres', password:

async function fullSearch() {  // Hämta alla resultaten på en gång
    const client = new Client(connection);
    await client.connect();
```

```
    let result = await client.query('SELECT * FROM chauffeaurs')
    console.log('full', result.rows);

    await client.end();
}

async function cursorSearch() { // Hämta några i taget
    const client = new Client(connection);
    await client.connect();

    const cursor = client.query(new Cursor('SELECT * FROM chauffeaurs')); // Notera ingen

    const MAXRESULTS = 20;
    let done = false;
    while (!done) {
        let rows = await cursor.read(MAXRESULTS); // Await:en kommer här i stället. Varför
        if (0 == rows.length) { done = true; }

        console.log('cursor', rows);
    }

    await cursor.close();
}

fullSearch();
cursorSearch();
```

16 Sammanfattning

- *Skriva* → ***Lagra*** → *Söka*
- Relationsalgebra, mängdlära
- Transaktioner, ACID, Eventual Consistency
- NoSQL vs SQL
- Bli vän med databaser tidigt
  - Både någon relationsdatabas och någon Dokument-databas (NoSQL).
- Ni *kommer* stöta på dem, ni *kommer* finna dem användbara.
- Det finns mycket som inte har nämnts i den här föreläsningen
  - Många "knep" och goda idéer för hur man designar sin databas
  - Månka krav för hur man skall undvika att data dupliceras och att alla relationer fortsätter fungera
  - Många detaljer i hur man skriver frågor
  - Hur man optimerar sin databas för skrivning respektive läsning
  - Hur och varför man skapar procedurer som lagras i databasen
  - ... och mycket, mycket mer.

17 Nästa Föreläsning

- Repetition.

18 Övning: Kom igång med Databaser

18.1 Docker Compose-fil

- Vi behöver inget git-repo den här gången, utan börjar med en enkel docker-compose-fil (se nedan)
  - Vad gör den här filen?
  - Default-användaren heter `postgres`, men man måste ange lösenordet.
- Starta med `docker compose` som vanligt.

```
version: '3.9'
services:
  db:
    image: postgres
    restart: always
    shm_size: 128mb
    environment:
      POSTGRES_PASSWORD: hunter2
  adminer:
    image: adminer
    restart: always
    ports:
      - 8080:8080
```

18.2 Översikt om Adminer

- Öppna en webläsare mot `http://localhost:8080`
- Vid inlogget behöver du ange

  System PostreSQL

  Server db (eftersom det är vad servern heter i docker compose-filen vi skapade)

  Username postgres

  Password hunter2 (eller vad du nu ändrade det till i docker compose-filen)
- Adminer ger dig möjlighet att klicka dig fram i webläsaren för att skapa databaser, tabeller, och värden
- Du kan också skriva SQL-kommandon direkt
  - Det här är tacksamt när man skall göra större eller upprepade operationer.

Tips Om du är God Vän™ med din editor kan du säkert koppla upp dig mot databasen därifrån.

- Du behöver då se till att databasen är tillgänglig på port 5432 även utanför docker-compose-klustret.
- Glöm inte att "stänga in" den igen när du har utvecklat färdigt så att inte någon utifrån kan hacka din databas.

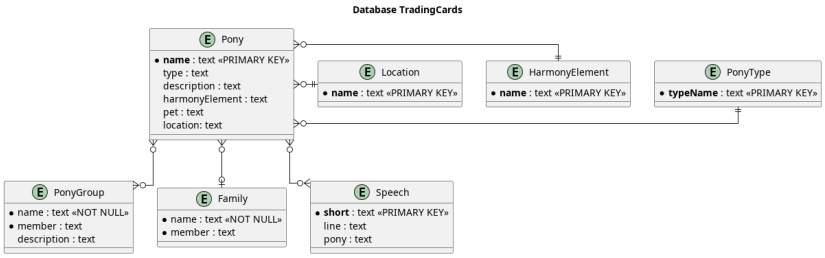
18.3 Skapa En Databas

- Vi fortsätter med pony-extravagansan.
- Den här gången vill vi bygga en databas för att kunna generera "Trading Cards"
- Vi bortser för stunden från bilder, cutie marks, och liknande och håller oss till ren text.
- Databasdesignen är förenklad; det är här ni behöver en hel kurs om databaser bara för att förstå hur och varför.
- **Att göra** Skapa en databas `TradingCards`.


my LITTLE PONY
FRIENDSHIP IS MAGIC
TWILIGHT SPARKLE™

18.4 Skapa Tabeller och Kolumner

1. Skapa följande tabeller och kolumner:



18.5 Fyll på med Data

1. Ladda ner filen `https://codeberg.org/mickesv/gists/raw/branch/main/TradingCards_insert.sql`
   - Titta igenom filen så att du förstår vad den gör.
2. Leta rätt på sidan `"SQL command"` I adminer-gränssnittet.
3. Klistra in filen och tryck på execute.
   - Om du skapade databasen korrekt skall alla `INSERT` fungera.
   - Annars, *läs felmeddelandet*, åtgärda och försök igen.
4. Fyll på med några fler Ponnys, t.ex. härifrån:
   - `https://mlp.fandom.com/wiki/My_Little_Pony_Friendship_is_Magic_Wiki`

18.6 Enkla Sökningar

1. Använd adminer-gränssnittet och gör några enklare sökningar.
   - Till vänster finns det länkar `select` och `tabellnamn` för varje tabell.
   - Välj "select" för rätt tabell, och fyll i fälten för "Select" och "Search" så att du kan hitta:

Visa följande fält	från tabellen	som matchar vilkoret
name, type	pony	name är exakt (=) "Rarity"
name, type	pony	name innehåller (~) "Twilight"
member	family	name är exakt "Apple"
pony, line	speech	pony innehåller 'Rainbow'

18.7 Kombinerade Sökningar

1. Fundera på hur du skulle uttrycka följande frågor:
   - Hitta alla Pony.name och Pony.type för Ponies som är med i en Family.
   - Vad kan alla Pony som befinner sig i Ponyville tänkas säga?
   - Vad kan alla Pony som *inte* befinner sig i Ponyville tänkas säga, och vad heter de?
   - Hitta namnen på alla Ponies som nämns i en PonyGroup men som inte finns i tabellen Pony än.
2. Försök ställa dessa frågor i adminer.
   - Du kan behöva använda "SQL Command" för att lyckas.

18.8 Hantera Sökningar och Resultat från Datorprogram

1. Skapa en Container 'PonyTradingCard' som söker i databasen och listar alla Ponys enligt nedanstående mall.
    - Du väljer själv programspråk. I node.js behövs paketet `"pg"` för PostgreSQL.
2. Uppdatera din docker-compose-fil så att den här containern också körs.

```
--------------------
Pony: Fluttershy
Type: Pegasus
Element of Harmony: Kindness
Pet: Angel
Description: Very shy and scared of dragons.
Location: Everfree Forest

Family Members:
- Mr. Shy
- Mrs. Shy
- Zephyr Breeze

Groups:
- Gen 4
- Main Character

Speech:
- "Oh, my."
- "I don't wanna talk about it."
- "I'd like to be a tree."
```