Øving 08 - Egen databaseløsning

Av: John I. Eriksen

1.1 - Definere behov

Case: Database for å registrere idrettsutøvere og -arrangement, med fokus på løpere og løp. Løpstype er innledningsvis terrengløp, fortrinnsvis ultraløp, men så får man se hva det blir til. Kanskje terrengløp generelt.

De kategoriene som må dekkes er:

- Utøver, eller atleten (*athlete*)
- Løpsarrangement (*event*)
- Resultat (*result*)
- Skader (*injury*)

Målet med databasen er å lage en oversikt over registrerte utøvere, hvilke løp de har deltatt i, resultatene deres og eventuelle skader de har pådratt seg. Skadene kan deles opp i de som ble pådratt under et arrangement, og de som eksisterte fra før dvs oppsto under trening.

Utøveren registreres med navn og alder, i tillegg til egne koblinger til tabeller som viser løp de har deltatt i og resultatet derfra. En tabell som viser skadehistorikken, enten i løp eller utenfor løp, må også kobles på.

Arrangementet, eller "eventen", registreres med antall plasser/påmeldte, teknisk data som distanse, høydemeter, og muligens løyperekord. Her kan også skader kobles inn. Siden skader en løper pådro seg ved et arrangement blir registrert på den aktuelle løperen, kan disse også bli koblet inn mot det aktuelle løpet.

Resultatene for en løper vil telle med på både løperens historikk og historikken og "leaderboard" på et spesifikt løp. Siden ett terrengløp ikke er likt et annet terrengløp, regner man alle resultater for å være på det spesifikke løpet. Dette til forskjell fra f.eks et maraton, der et hvilket som helst offisielt godkjent maraton skal være tilnærmet likt et hvilket som helst annet. Derfor kan man si at "Navn Navnesen har verdensrekord i maraton", men man kan ikke si at "Ola Olsen har verdensrekord i 170 km terrengløp", fordi to terrengløp kan være vidt forskjellige på antall høydemeter, terreng og værforhold. 170 km med 8000 høydemeter i motvind og 2 grader celsius er ikke det samme som 170 km med 1500 høydemeter i 16 grader og overskyet.

Skader legges inn under en utøver, med detaljer om skaden, dato, og hvorvidt den inntraff under trening eller et arrangement. Siden skader kan kobles mot arrangement, om relevant, kan databasen vise en oversikt over skade både basert på den enkelte løpers historikk og hvilke skader som har inntruffed ved spesifikke løp.

Finnes det begrensninger for hva som er mulig å oppnå? Ikke egentlig, men det kan veldig fort bli veldig mange tabeller. Slike ting har en tendens til å balle på seg. F.eks vil det være fort å lage undertabeller for skader, og undertabeller til undertabellene, fordi det er svært mange faktorer som spiller inn.

1.2 - Egen relasjonsdatabase med eksempler i SQL

Relasjoner og forklaring

Databasediagram

Relasjonsform

`athlete(athlete_id{PK}, name, surname, sex, age, nationality)`

`result(result_id{PK}, athlete_id{FK}, event_id{FK}, time, placement)`

`event(event_id{PK}, name, type, distance, ascent, location, country, date)`

`injury(injury_id{PK}, athlete_id{FK}, event_id{FK}, description, date)`

PK- og FK-notasjon

Jeg har valgt å brukt {PK} for å markere primærnøkkel, og {FK} for fremmednøkkel siden dette er mye tydeligere i teksten enn å bruke understrek og stjerne. Har gjort dette også på tidligere øvinger, uten at det har vært noe problem i rettingen, så for lesbarhetens skyld gjør jeg det her også. Hvis dette ikke er "innafor", gi beskjed så skal jeg oppdatere besvarelsen.

Forklaring, "athlete" og "event":

`athlete----result`:

- Relasjonstype: 1-til-mange, `1..*`
- En enkelt utøver kan ha mange resultater, men hvert resultat er knyttet til nøyakltig én utøver.

`result----athlete`

- Relasjonstype: Mange-til-1, `*..1`
- Mange resultater kan være knyttet til en enkelt utøver, men hvert enkelt resultat refererer til én spesifikk utøver.

`event----result`

- Relasjonstype: 1-til-mange, `1..*`
- Et enkelt arrangement kan ha mange resultater, men hvert resultat er knyttet til ett spesifikt arrangement.

`result----event`

- Relasjonstype: Mange-til-1, `*..1`
- Flere resultater kan være knyttet til et enkelt arrangement, men hvert enkelt resultat refererere spesifikt til ett arrangement

Dette reflekterer realiteten av løpsarrangementer, hvor hver utøver kan delta i flere løp, og ha flere resultater tilknyttet seg. Hvert løp kan også ha flere deltaktere, med mange resultater hvor hvert resultat er unikt og tilknyttet en spesifikk deltaker.

Relasjonen mellom `athlete` og `event` går via `result`, siden dette er en fellesnevner for begge. An utøver må ha et resultat for å ha vært med på et løp, og et løp må ha et resultat registrert på en gitt utøver.

Forklaring, "athlete" og "injury"

`athlete----injury`

- Relasjonstype: 1-til-mange, `1..*`
- En enkelt utøver ha ha flere registrerte skader, men hver skade er unikt tilknyttet én spesifikk utøver.

`injury----athlete`

- Relasjonstype: Mange-til-1, `*..1`
- Flere skader kan være registrert for hver utøver, men hver enkeltskade er tilknyttet en spesifikk utøver.
- Om man søker på en spesifikk skade (`injury_id`), vil denne identifisere hvilken utøver den tilhører.

`event----injury`

- Relasjonstype: 1-til-mange, `1..*`
- Valgfri. En skade trenger ikke koblet mot et arrangement.
- Et enkelt arrangement kan være tilknyttet flere skader, men hver skade er registrert for et spesifikt arrangement (eller på ingen arrangement).

`injury----event`

- Relasjonstype: Mange-til-1, `*..1`
- Valgfri. En skade trenger ikke koblet mot et arrangement.
- Hver skade kan være tilknyttet et enkelt arrangement, men mange skader kan referere spesifikt til ett arrangement.

Dette vil gjøre det mulig å spore skader som oppsto ved forskjellige løpsarrangement. Hensikten kan være å se på vanskelighetsgraden av terrenget, og på hvilke utøvere som pådro seg skade. Det vil gi informasjon om f.eks aldersgruppe og plassering for de skadde. F.eks kan det være de som er raskest og egentlig stiller sterkest som får skader fordi de er mindre forsiktige, eller eldre aldersgrupper pga. redusert reaksjonsevne i veldig ulendt terreng.

Relasjonen mellom `event` og `injury` er valgfri. Dermed kan man la være å registre skade om ingen oppstå, eventuelt regisrere en skade som oppsto utenfor et løpsarrangement, f.eks under trening.

SQL-spørringer

Hente ut alle resultater for en bestemt utøver

```sql
SELECT
  result.time,
  result.placement,
  event.name AS event_name,
  event.type
FROM result
INNER JOIN athlete ON result.athlete_id = athlete.athlete_id
INNER JOIN event ON result.event_id = event.event_id
WHERE athlete.name = 'Reodor Såle';
```

Finne alle utøvere som har deltatt i et spesifikt arrangement

```sql
SELECT
  athlete.name,
  athlete.surname,
  result.time,
  result.placement
FROM athlete
INNER JOIN result ON athlete.athlete_id = result.athlete_id
INNER JOIN event ON result.event_id = event.event_id
WHERE event.name = 'Lofoten Ultra-trail';
```

Finne alle skader for utøvere i et spesifikt arrangement

```sql
SELECT
  athlete.name,
  injury.description,
  injury.date
FROM injury
INNER JOIN athlete ON injury.athlete_id = athlete.athlete_id
INNER JOIN event ON injury_event_id = event.event_id
WHERE event.name = 'Lofoten Ultra-trail'
```

Oppsummering

Bruk av denne databseløsningen var ikke spesielt innviklet. Siden databasen ble laget med en enkel struktur, og unngikk å la kompleksiteten skli ut i form av mange undertabeller for å håndtere alle mulige variasjoner av f.eks skader, ble tabellene forholdsvis enkle og oversiktlige.

Å skrive spørringer ble følgelig ikke spesielt komplisert siden strukturen var såpass enkel. Bruk av `INNER JOIN` er en enkel måte å koble sammen de aktuelle tabellene og kolonnene, hvilket fungerer ganske bra når tabellene ikke er spesielt store.

Videre utvikling av databasen ville vært f.eks sko (merke, modell, stack, drop, dempegrad, form) og treningsregime. Disse ville knyttet veldig fint inn i skade-tabellen, men ville gjort tabellen mye mer innviklet enn hva oppgaven ber om. Om databsen hadde inkludert disse, kunne den blitt brukt til å se på statistisk korrelasjon mellom forskjellige typer sko og skadefreksvens i løp, eller hvordan treningsmetoder påvirket ytelse og skadefrekvens. Det sagt, ville det blirr svært mye arbeid å skulle føre detaljert treningsdata i en

databse for hånd, og det måtte dermed bli satt opp en form for integrasjon med eksterne datakilder, som f.eks utøverenes egne treningslogger, gjerne i form av API-tilkobling mot Strava, Garmin, TrainingPeaks, osv. Men, resultatet av dét ville i grunnen bare vært enda en tjeneste som gjør det mange andre tjenester allerede gjør, eller kan gjøre.

1.3 - Løsning med XML evt. JSON i MySQL

Et eksempel på bruk av JSON i tabellen `athlete`:

```
CREATE TABLE athlete (
  athlete_id INT AUTO_INCREMENT PRIMARY KEY
  name VARCHAR(255),
  surname VARCHAR(255),
  sex CHAR(1) CHECK (sex IN ('M', 'F')),
  age INT CHECK (age >= 18),
  nationality VARCHAR(100),
  additional_info JSON
);
```

JSON-kolonnen `additional_info` kan her brukes til å lagre informasjon som ikke passer i de andre, f.eks utstyr, treningsopplegg, ernæring, pårørende, personlige rekorder (PR), osv.

Eksempel på data til JSON-kolonnen:

```
INSERT INTO athlete (
  name,
  surname,
  sex,
  age,
  nationality,
  additional_info
)
VALUES (
  'Reodor',
  'Såle',
  'M',
  33,
  'Norge',
  '{
  "personal_records": {"180 km": "34:14:09", "21,1 km": "1:20:32"},
  "equipment": {"shoes": "Altra King MT", "watch": "Garmin fenix 6X"}
  "next_of_kin": {"name": Ludvig von Pinn", "tel": "81549300"}
  }'
);
```

Dermed kan JSON-infoen hentes ut ved følgende eksempelspørring:

```sql
SELECT athlete_id, name, additional_info->'$.personal_records' as personal_records
FROM athlete
WHERE name = 'Reodor Såle';
```

Her viser spørringen personlige rekorder for utøveren `Reodor Såle`.

Fordeler og ulemper, JSON

Bruk av JSON i en relasjonsdatabase er noe som bør veies nøye opp mot behovene og kravene til til løsningen. Det bør kun brukes når man ser sikker på at fleksibiliteten til å lagre såkalt "semistrukturell" data veier tyngre enn den kompleksiteten i spørringer og vedlikehold som tilføres.

Fordeler

- **Fleksibilitet:** JSON gjør at man kan lagre semistrukturert data, hvilket kan være nyttig for data som ikke nødvendigvis har en naturlig plass i en rigid tabellstruktur, hvor man ikke ønsker massevis av undertabeller for alt mulig rart.
- **Redusert behov for mange tabeller:** Som nevnt over, vil dette redusere behovet for antallet tabeller som kreves for å lagre komplekse og varierte data, og man slipper massevis av "småtabeller" som kanskje skaper mer rot enn orden.
- **Utvidbarhet:** Nye dataelementer kan legges til som JSON-objekter uten at det endrer databasens struktur.

Ulemper

- **Kompleksitet i spørringer:** Spørringene kan bli mer komplekse og mindre intuitive når man må ta høyde for mange mer eller mindre strukturerte JSON-kolonner.
- **Indeksering og søk:** Indeksering av JSON-kolonner er genrelt mer begrenset og mindre effektivt enn "vanlige" kolonner. Dessuten er det ikke alle databaseløsninger som støtter indeksering av JSON-felt.
- **Dataintegritet:** Siden JSON-kolonner fort kan bli felt for fritekst, vil dette påvirke dataintegriteten og validering av innhold, sammenlignet med vanlige databasetabeller hvor struktur og innholdt er lagt mer opp til å kunne følge en gitt oppskrift.

1.4 - NoSQL-løsning

Siden NoSQL-databaser har større fleksibilitet i utforming av skjema, er den bedre egnet til håndtering av variert og semistrukturell data, som f.eks idrettsutøverprofiler, treningslogger og utstyrsinformasjon. NoSQL ble laget for å prosessere store mengder data på kort tid, og gjør dette på beskostning av ACID-egenskapene (ACID = Atomicity, Consistency, Isolation, Durability).

- ACID, BASE og CAP
- SQL vs NoSQL: A Performance Comparison
- SQL vs NoSQL: 5 Main Differences

Derimot er de mindre egnet for komplekse spørringer og analyser som krever omfattende JOIN-operasjoner og avansert transaksjonsbehandling. Her kommer relasjonsdatabasen til sin rett. I tillegg har de aller fleste NoSQL-løsninger utfordringer med å opprettholde ACID-egenskaper for transaksjoner

Vedrørende MongoDB, gjør det at den er godt egnet til å håndtere semistrukturell data at den er et godt alternativ, spesielt om man greier å holde databasen forholdsvis ryddig samtidig, slik at man ikke støter på problemer på grunn av dårligere støtte for komplekse transaksjoner. Den har også god skalerbarhet, hvilket betyr at den kan håndtere en raskt voksende mengde data og sanntidsdata. Hadde databasen skulle bli brukt til "live tracking", hadde MongoDB vært et veldig godt alternativ.

Det sagt, og gitt databasen formulert her i oppgaven, er "vanlig" SQL mer enn godt nok. En NoSQL-løsning kunne vært aktuelt om den skulle håndtert enorme mengder data, og er antageligvis brukt hos både Garmin og Strava.