Gennemgang af opgaverne om subqueries

Opgave 9

```
SELECT email_address, MAX(order_total) AS max_order_total

FROM

(

  SELECT email_address, o.order_id, SUM((item_price - discount_amount) *
quantity) AS order_total

  FROM customers c

    JOIN orders o ON c.customer_id = o.customer_id

    JOIN order_items oi ON o.order_id = oi.order_id

  GROUP BY email_address, o.order_id

) t

GROUP BY email_address
```

Opgaven går ud på at man først skal lave et result set med kolonnerne email_address, order_id og en order_total beregnet ud fra order_items tabellen. Dette 'indre' result set er altså en table med kolonner og rækker – det man også kalder et 'inline view'. Man skal joine 3 tabeller for at få kolonnerne: customers, orders og order_items. Det der er lidt anderledes end I har set før er at man laver GROUP BY på 2 kolonner, og ikke bare en enkelt. Det der sker når man grupperer på to kolonner er at en gruppe ikke bare er fx email, men samtlige mulige kombinationer af email og order_id. Prøv at køre den indre query alene i WorkBench – så kan man se at fx kunden med emailen 'allan.sherwood@yahoo.com' har to ordrer, med henholdsvis order_id 1 og 3.

I denne query er man nødt til at lave GROUP BY på både email og order_id, fordi de står oppe i SELECT sammen med en aggregate function, og de derfor SKAL indgå i en GROUP BY hvis ikke man skal få et underligt og ubrugeligt resultat. Prøv fx at fjerne order_id nede I GROUP BY, og kør statementet – så kan man se at der kun kommer en enkelt række for 'allan.sherwood@yahoo.com', og at SUM nu lægger alle hans ordrers pris sammen, men at der står order_id 1, som jo er helt vilkårligt, for ordre nr 1 har jo ikke den pris der er beregnet i SUM.

Den indre query giver altså et overblik over alle de ordrer en kunde har, og hver ordres samlede pris (beregnet ud fra antal order_item, rabatter, mv).

Den ydre query behandler den indre som om den faktisk er en tabel der eksisterer i database. Den har fået aliaset t – det kunne lige så godt have været et andet alias der var valgt (fx sub) – det vigtige er at huske at der skal være et alias når en subquery resulterer i en tabel.

Resultatet af den samlede query er alså et overblik over hver kundes dyreste ordre.

NB! Jeg har rettet en sjuskefejl jeg kom til at lave i den første løsning jeg lagde op ved middagstid i dag: den ydre query har kun 2 kolonner (og altså ikke noget med order_id oppe i SELECT, som havde sneget sig med i min første eksempelløsning!). Jeg har lagt en korrigeret fil op på Fronter nu.

Opgave 10

```
SELECT product_name, discount_percent

FROM products

WHERE discount_percent NOT IN (

   SELECT discount_percent

   FROM products

   GROUP BY discount_percent

   HAVING count(discount_percent) > 1)

ORDER BY product_name;
```

I denne opgave skal der kun bruges én enkelt tabel, nemlig products. Prøv at starte med at lave en SELECT * på products tabellen, for at få et overblik. Products tabellen har en kolonne der hedder discount_percent hvor man kan se hvor mange procents rabat der er på produktet – man kan fx få 52 % rabat på en Gibson SG, et vældig godt tilbud ☺. Nogle produkter har den samme rabat %, andre har forskellige. Opgaven beder om at få et overblik over alle de produkter der har forskellige procenter i rabat, dvs alle dem der har samme % rabat skal ikke med i resultatet.

For at forstå hvad den indre query gør kan det være en god ide at eksperimentere lidt med nogle dele, eller trin, af den. Hvis vi nu tænker på SELECT * FROM products som udgangspunktet, så kan vi allerede se at der er flere produkter med 30% rabat. Hvad hvis vi gerne vil ha talt hvor mange forskellige produkter der har den samme %? Det kan vi gøre ved at vælge discount_percent kolonnen i SELECT, og også bruge den i GROUP BY. Hvis vi i første omgang laver den indre query uden linjen med HAVING, og samtidig laver

en COUNT på discount_percent oppe i SELECT, så bliver det nemmere at forstå hvad der foregår. Prøv altså at lave en query der ser sådan her ud:

```
SELECT discount_percent, COUNT(discount_percent)

  FROM products

GROUP BY discount_percent
```

Når man kører denne query kan man se at der er 4 produkter der har en discount_percent på 30. Det er denne række vi kan få fat på ved at rykke COUNT(discount_percent) ned i vores HAVING – det er faktisk præcis samme beregning der bliver lavet som i det lille eksperiment ovenfor, med den forskel at vi får sorteret alle de andre products fra som ikke har samme discount_percent som nogle af de andre, ved at sige HAVING count(discount_percent) > 1.

Resultatet af den indre query er i dette tilfælde kun en enkelt række med discount_percent, nemlig 30%, men der kunne lige så vel have været flere rækker – fx hvis der var andre guitarer der også have 52% lige som Gibson SG.

Den ydre query laver så en NOT IN i forhold til discount_percent, for at finde alle dem der IKKE har samme discount_percent som andre (altså i dette tilfælde alle dem der ikke har en discount_percent på 30).

Opgave 11

```
SELECT email_address, order_id, order_date

FROM customers c

 JOIN orders o ON c.customer_id = o.customer_id

WHERE order_date =

 (SELECT MIN(order_date)

  FROM orders

  WHERE customer_id = o.customer_id)
```

Denne opgave bruger en correlated query, som er en slags loop. Det vil sige at subquerien kører én gang for hver række i den ydre query. Vi har 2 tabeller i spil: orders og customers, som bliver joined i den ydre query. I den indre query bliver der returneret en enkelt værdi af den function der bliver kørt, nemlig summery querien MIN(order_date), som giver den ældste ordredato for hver kunde. Det der gør at denne query er correlated er at der i den indre query bliver henvist til den ydre tabel orders med alias o, WHERE customer_id = o.customer.id.

o.customer_id skifter altså til en ny værdi for hver række i den ydre query, ligesom man kender det fra loops i programmering. Den samlede query kører altså igennem alle rækker i orders, hvilket man også ser i resultatsættet.