

## Data Storage and Formats

### SQL part II

Lecture 5  
Fall 2008  
Anna Pagh

Some figures are borrowed from the ppt slides from the book used in the course, Database systems by Kiefer, Bernstein, Lewis Copyright © 2006 Pearson, Addison-Wesley, all rights reserved.



## Today's lecture

Based on Chapter 5.

- Fast summary of SQL from last week
- Subqueries in WHERE
  - Uncorrelated
  - Correlated
- Subqueries in FROM clause
- More on aggregates
- Views
- Materialized views



## SQL: SELECT-FROM-WHERE

```
SELECT C.Color
FROM Car C
WHERE C.Regnr=42
```

Use tuple variables

Select

Where

From

$$\pi_{Color}(\sigma_{Regnr=42}(Car))$$


## Join

```
SELECT C.Color, O.Id
FROM Car C, Owner O
WHERE C.Ownerid=O.Id AND C.Regnr=42
```

A join query

$$\pi_{Color, Id}(\sigma_{Regnr=42}(Car) \bowtie_{Ownerid=Id} Owner)$$


## Another join example

```
SELECT T1.Regnr, T2.Regnr
FROM Cartax T1, Cartax T2
WHERE T1.Amount=T2.Amount AND
      T1.Regnr<T2.Regnr
```

All pairs of cars with the same tax.



## Set operations

- UNION ( $\cup$ ), INTERSECT ( $\cap$ ), and EXCEPT ( $-$ )

```
(SELECT *
FROM Car C
WHERE C.Color='green')
UNION
(SELECT *
FROM Car C
WHERE C.Color='blue')
```

```
(SELECT C.Regnr, C.Color
FROM Car C
WHERE C.Color='green')
EXCEPT
(SELECT *
FROM Car C
WHERE NOT C.Regnr=1234)
```



## Aggregation and grouping

Aggregation: Max, sum, count,...  
Grouping: Compute several aggregates

```
SELECT C.Ownerid AS Id, SUM(T.Amount) AS TotalTax
FROM Cartax T, Car C
WHERE T.Regnr=C.Regnr
GROUP BY C.Ownerid
```

## Subqueries

NOT IN

WHERE R.a IN (*select\_query*)

The condition in the WHERE-clause can use a subquery to form a condition.

R.a IN (*select\_query*) is true when the value of R.a is in the table that the *select\_query* results in.

Note that the subquery *must* produce a relation with **one** attribute only.

## Example: subquery

```
SELECT C.Regnr
FROM Car C
WHERE C.Ownerid IN
  (SELECT O.Id
   FROM Owner O
   WHERE O.Lastname = 'Sørensen')
```

All registration numbers for cars owned by a person named Sørensen.

```
SELECT C.Regnr
FROM Car C, Owner O
WHERE C.Ownerid=O.Id AND O.Lastname='Sørensen'
```

## Subquery to define a value

A subquery producing a single value can be used as any other value (constant or attribute):

Which cars have the same tax as the car with registration number AB12345?

```
SELECT T.Regnr
FROM Cartax T
WHERE T.Amount =
  (SELECT T2.Amount
   FROM Cartax T2
   WHERE T2.Regnr='AB12345')
```

Note that the same table is used twice with two different tuple variables

## Problem session

Which cars have the same tax as the car with registration number AB12345?

```
SELECT T.Regnr
FROM Cartax T
WHERE T.Amount =
  (SELECT T2.Amount
   FROM Cartax T2
   WHERE T2.Regnr='AB12345')
```

The above query can also be written without a subquery. How?  
Which way do you believe is most efficient?

## Correlated subqueries

A subquery is said to be **correlated** when a variable in the outer query is used in the subquery:

```
SELECT R.Studid, P.Id, R.CrsCode
FROM TRANSCRIPT T, PROFESSOR P
WHERE R.CrsCode IN
  (SELECT T1.CrsCode
   FROM TEACHIN T1
   WHERE T1.ProfId=P.Id AND T1.Semester='S2009')
```

The inner query is evaluated **for each** P.Id.

Often **expensive** to evaluate correlated nested subqueries.

## NOT EXISTS

```
SELECT O.Id
FROM Owner O
WHERE NOT EXISTS
  (SELECT C.Regnr
   FROM Car C
   WHERE C.Color LIKE '%green%' AND
        C.Ownerid=O.Id)
```

True when subquery returns an empty relation

O is a **global variable** for the entire query, C is a **local variable** for the subquery.

Subquery "is" evaluated for each value of O.Id.

## > ALL

Which owners has a car with higher tax than all my cars (Id=1234) and what is the tax?

```
SELECT C.Id, T.Amount
FROM Car C, Cartax T
WHERE T.Amount > ALL
  (SELECT T2.Amount
   FROM Cartax T2, Car C2
   WHERE C2.OwnerId=1234 AND
        T2.Regnr=C2.Regnr)
```

> ALL compares a value V to a set of values, S. It is True if V is greater than all values in S.

## = ANY

Which owners has car with the same tax as one of my (Id=1234) cars and what is the tax?

```
SELECT C.Id, T.Amount
FROM Car C, Cartax T
WHERE T.Amount = ANY
  (SELECT T2.Amount
   FROM Cartax T2, Car C2
   WHERE C2.OwnerId=1234 AND
        T2.Regnr=C2.Regnr)
```

= ANY compares a value V to a set of values, S. It is True if V is equal to one value in S.

## IN, ALL, ANY, NOT EXIST

**IN**: True if the value is in the set

**ALL**: Can be combined with all comparison, e.g. > ALL, <= ALL. True if the comparison is true for all members of the set.

**ANY**: Can be combined with all comparisons, e.g. < ANY, >= ANY. True if the comparison is true for one value in the set.

**NOT EXIST**: True if the set is empty.

Note in previous slide:  
= ANY is the same as IN

## Problem session

Given the following relations write:

1. A query with an uncorrelated subquery
2. A query with a correlated subquery
3. A query using NOT EXISTS
4. A query using ANY or ALL

```
PERSON(Cpr,Name,Birthday)
ADDRESS (Id,Street,Number,Zip,City)
LIVESAT(Cpr,AddressId)
PHONE(SubCpr,Number,Type,AddressId)
```

## Subqueries in FROM clause

A relation in the FROM clause can be defined by a subquery:

```
SELECT O.FirstName, O.LastName, TPO.TotalTax
FROM Owner O,
  (SELECT Sum (T.Amount) AS TotalTax, OwnerId AS Id
   FROM Cartax T, Car C
   WHERE T.Regnr=C.Regnr
   GROUP BY C.Ownerid) AS TPO
WHERE TPO.Id=O.Id
```

## FOR ALL

```
SELECT DISTINCT C.OwnerId
FROM Car C
WHERE FOR ALL
  (SELECT T.Amount
   FROM Cartax T
   WHERE C.Regnr=T.Regnr )
  (Amount>1000)
```

Computes a table with all car owners where **all** his/her cars have a tax exceeding 1000 kr.

General form:

**FOR ALL table (condition):** True if for all rows in *table* the *condition* is true.



## FOR SOME

```
SELECT DISTINCT C.OwnerId
FROM Car C
WHERE FOR SOME
  (SELECT T.Amount
   FROM Cartax T
   WHERE C.Regnr=T.Regnr )
  (Amount>1000)
```

Computes a table with all car owners owning **at least one** car with tax exceeding 1000 kr.

General form:

**FOR SOME table (condition):** True if for at least one row in *table* the *condition* is true



## Aggregation and grouping revisited

### Aggregate functions:

- COUNT ([DISTINCT] attr): Number of rows
- SUM ([DISTINCT] attr): Sum of attr values
- AVG ([DISTINCT] attr): Average over attr
- MAX (attr): Maximum value of attr
- MIN (attr): Minimum value of attr
- DISTINCT: only one unique value for attr is used

### Grouping:

GROUP BY is used to compute aggregate for sets of rows, defined by the value of some attribute(s).



## HAVING

```
SELECT C.OwnerId, SUM(T.Amount)
FROM Car C, Cartax T
WHERE C.Regnr=T.Regnr
GROUP BY C.OwnerId
HAVING SUM(T.Amount)<=1000
```

HAVING is a condition on the group.

Note that not any condition makes sense on a group, e.g. Regnr=1234.



## ORDER BY

```
SELECT C.OwnerId, SUM(T.Amount) AS TotalTax
FROM Car C, Cartax T
WHERE C.Regnr=T.Regnr
GROUP BY C.OwnerId
HAVING SUM(T.Amount)<=1000
ORDER BY TotalTax
```

In the ORDER BY clause, attribute names that appear in the resulting table must be used:  
**ORDER BY SUM(T.Amount) is wrong**

ORDER BY gives the result in ascending order.  
ORDER BY DESC gives the descending order.



## Evaluation algorithm

**Algorithm** for evaluating a SELECT-FROM-WHERE statement:

1. **FROM** clause is evaluated. Cartesian product of relations is computed.
2. **WHERE** clause is evaluated. Rows not fulfilling condition are deleted.
3. **SELECT** clause is evaluated. All columns not mentioned are removed.

Now we have learned about other clauses. How are they integrated in the above algorithm?



## Evaluating aggregates

**Algorithm** for evaluating a SELECT-FROM-WHERE statement:

1. **FROM:** Cartesian product of tables is computed. Subqueries are computed recursively.
2. **WHERE:** Rows not fulfilling condition are deleted. Note that aggregation is evaluated after WHERE, i.e. aggregate values can't be in the condition.
3. **GROUP BY:** Table is split into groups.
4. **HAVING:** Eliminates groups that don't fulfill the condition.
5. **SELECT:** Aggregate function is computed and all columns not mentioned are removed. One row for each group is produced.
6. **ORDER BY:** Rows are ordered.

## Problem session

What does the following query compute?

```
SELECT C1.Color, AVG(T.Amount)
FROM (SELECT O.Id AS Id
      FROM Owner O, Car C2
      WHERE O.Id=C2.Ownerid
      GROUP BY O.Id
      HAVING COUNT(*)>8) AS Bigshots,
      Cartax T,
      Car C1
WHERE T.Regnr=C1.Regnr AND C1.Ownerid=Bigshots.Id
GROUP BY C1.Color
```

## Subroutines in SQL

**Views** are used to define queries that are used several times as subqueries:

```
CREATE VIEW OwnerColor AS
SELECT O.Id, C.Color
FROM Owner O, Car C
WHERE O.Id=C.Ownerid
```

The view can be used in different queries:

```
SELECT COUNT(*)          SELECT O.Color,COUNT(*)
FROM OwnerColor O        FROM OwnerColor O
WHERE O.Color='pink'     GROUP BY O.Color
                        HAVING COUNT(*)<200
```

## Views

- A view defines a **relation** and can be used as any other relation.
- A view's relation is **not stored physically**.
- When a view is used the **code** from the CREATE VIEW statement is **copied** into the query (as a subquery).

```
CREATE VIEW OwnerColor AS      SELECT COUNT(*)
SELECT O.Id, C.Color          FROM OwnerColor OC
FROM Owner O, Car C           WHERE OC.Color='pink'
WHERE O.Id=C.Ownerid
-----
SELECT COUNT(*)
FROM (SELECT O.Id, C.Color FROM Owner O, Car C WHERE
O.Id=C.Ownerid) AS OC
WHERE OC.Color='pink'
```

## Usage of views

Views can be used for:

1. Defining queries used as subqueries
2. Access control
3. Simplifying access and application development by customizing views for different users/developers
4. Logical data independence

## Access control

Views can be used to limit the access to data, the right to update data, etc. Example:

```
GRANT SELECT ON OwnerColor TO ALL
```

Meaning: All users can see the table OwnerColor, but not the underlying relations Car and Owner.

Other options:

- GRANT INSERT, GRANT ALL, and more
- TO ALL, TO user, TO group

## Updating using a view

```
CREATE VIEW ProfNameDept(Name,DeptId) AS
SELECT P.Name, P.DeptId
FROM Professor P
```

What are the results of the following 2 updates?

```
INSERT INTO ProfNameDept
VALUES (Hansen, 'CS')
```

```
DELETE FROM ProfNameDept
WHERE Name=Hansen and DeptId='CS'
```

## Updating using a view

**Insertion:** For unspecified attributes, use NULL or default values if possible.

**Deletion:** May be unclear what to delete. Several restrictions, e.g. exactly one table can be mentioned in the FROM clause.

NOT ALL VIEWS ARE UPDATABLE!

## Levels of data independence

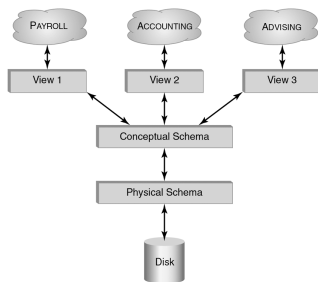


Figure: Copyright © 2006 Pearson Addison-Wesley. All rights reserved.

## Logical data independence

When all applications are written using views instead of **base tables** the application is independent of the conceptual schema.

**Base table:** the tables defined by the conceptual schema. Tables that are stored physically.

If the conceptual schema is changed the application can be **unchanged** if

- the data is still in the database, and
- the definitions of the views are updated

## Materialized views

**Views** are computed **each time** they are accessed – possibly inefficient

**Materialized views** are computed and stored physically for faster access.

When the base tables are updated the view changes and must be recomputed:

- May be inefficient when many updates
- Main issue – when and how to update the stored view

## Updating materialized views

**When** is the view updated

- **ON COMMIT** – when the base table(s) are updated
- **ON DEMAND** – when the user decides, typically when the view is accessed

**How** is the view updated

- **COMPLETE** – the whole view is recomputed
- **FAST** – some method to update only the changed parts.
  - For some views the incremental way is not possible with the available algorithms.)

## Updating the database

```
INSERT INTO TableName(a1,...,an)  
VALUES (v1,...,vn)
```

```
INSERT INTO TableName(a1,...,an)  
SelectStatement
```

```
DELETE FROM TableName  
WHERE Condition
```

```
UPDATE TableName  
SET a1=v1,...,ai=vi  
WHERE Condition
```



## Standard SQL vs MySQL

The book describes SQL as defined by the standards, however MySQL does not have all the standard constructions implemented:

- Set operations INTERSECT and EXCEPT not implemented
- FOR SOME and FOR ALL not supported
- Aggregates: More functions defined
- Materialized views not supported



## Problem session

- DBS June 2006, question 3

