# Modifying Views

Deleting view:

```
DROP VIEW ParamountMovies;
DROP TABLE Movie;
```

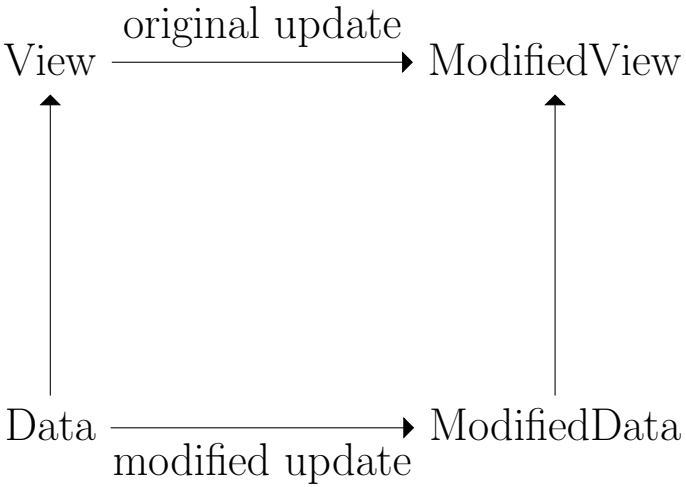
The first deletes the view, but does not delete the data

The second deletes the data

Is it possible to update views?

Assuming no security reasons to ban this, can one define an update to a view?

We can't actually modify the data in the view, so we have to convert the update into an update on the data in the database




Problems

- Can we find a way to modify the data so that the effect will be to modify the view
- Can we find a *unique way*? If not, can we select one that makes sense?

## Insertion

```
INSERT INTO ParamountMovies
   VALUES ('Star Trek',1979);
```

`Movies(title,year,length,genre,studioName,producerC#)`

We must insert values for `length`, `genre` etc.

We know (from the definition of the view) that `studioName` is equal to `'Paramount'`

There are many possibilities for the other attributes, but in this case it makes sense to use `NULL`s, i.e., to insert

`Movies('Star Trek',1979,NULL,NULL,'Paramount',NULL)`

So the insert on the view should be converted into

```
INSERT INTO Movies(studioName,title,year)
  VALUES ('Paramount', 'Star Trek', 1979);
```

## Another example

```
DELETE FROM ParamountMovies
WHERE title LIKE '%Trek%';
```

Becomes

```
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

If the first query included both title and year, adding the `studioName` is not really needed, but doesn't hurt

## Another example

```
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

Becomes

```
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
      studioName = 'Paramount';
```

## View with join

Two relations with `codiceFiscale` as key

```
Person (codiceFiscale, name)
Address (codiceFiscale, city)
```

View

```
CREATE View R(name, city) AS
   SELECT name, city
   FROM Person NATURAL JOIN Address;
```

Update: Insert the tuple `(Paolo, Trento)`

We must insert two tuples

`(Paolo, x)` into `Person`

and

`(x, Trento)` into `Address`

But we can't invent the Codice Fiscale!

Inserting `NULL`s is not allowed, as this is a key (it wouldn't work anyway, for reasons that we'll see when we study `NULL`s)

Therefore: This update is not allowed on the view

## Rules for view update

The rules in the SQL standard are complicated, but the key points are:

- The view must select attributes (not `SELECT DISTINCT`) from a single relation $R$
- The `WHERE` clause cannot use $R$ in a subquery
- The `FROM` clause can only have one occurence of $R$, and no other relation
- The list of attributes in the `SELECT` clause must contain enough attributes to complete the tuple

Note that $R$ itself may be a view, as long as it is allowed to update $R$

# Indexes

- Data structure used for efficient access to data
- Most database systems use *B-trees*
- Similar to search trees you have probably studied, but with special features to permit efficient update
- In principle, this is an implementation detail that should be decided by the system
- In practice, users can declare indexes themselves

```
CREATE INDEX YearIndex ON Movies(year);
```

Queries that search for movies from a specific year will probably be faster now

Most systems automatically create indexes for keys

Other indexes may be created automatically by the system

- DBMS logs all queries, so can determine what are common queries
- System can also analyze the performance and see where indexes may help
- The designer could be offered candidate indexes and decide which should be implemented

# Materialized Views

- Similar to views, but are physically stored ("materialized")
- Unlike new relations, materialized views are automatically updated when the base relations change
- Queries are more efficient
- Updates are less efficient

```
CREATE MATERIALIZED VIEW MovieProd AS
  SELECT title, year, name
  FROM Movies, MovieExec
  WHERE producerC# = cert#;
```

## `NULL` values

What do `NULL` values mean?

- Unknown value: The value exists, but I don't know it (birthdate, gender)
- Value inapplicable: No value makes sense here (spouse attribute for unmarried person)
- Value hidden: Unlisted phone number
- Other meanings

How the DB handles NULL values could depend on their semantics, but attempts to design languages that take this into account have failed

## Some history

What should a query mean if the relation contains NULL values

One attempt for a general framework

Relation $R$:

| *student* | *grade* |
|---|---|
| a | 30 |
| a | NULL |
| b | 18 |

Interpret this relation as meaning that the NULL could stand for any value from 0 to 31.

A query can be answered if we get the same answer for every possible assignment of values for the NULLs

## Examples

- `SELECT grade WHERE student = 'b';`

  Answer: 18

- `SELECT grade WHERE student = 'a';`

  No answer. Could be $\{0, 30\}$, $\{1, 30\}$ etc

- ```
  SELECT student
  FROM R
  GROUP BY student
  HAVING AVG (grade) > 10
  ```

  Answer: $\{a, b\}$.
  ```

Can this be implemented?

Practical problem: In general we have to check the whole domain, not just 32 values (all integers...)

- Good news: There exist more efficient algorithms whose complexity depend on the size of the data, not the size of the domains
- Bad new: Complexity is exponential or worse

## The SQL approach

- Much simpler to describe and use
- Sometimes unintuitive

Consider a condition:

`A = 5`

If `A` is `NULL`, what is the truth value of this expression?

- `A` equals 5: `TRUE`
- `A` equals 6: `FALSE`
- We don't know the value of `A`: `UNKNOWN`

*3-valued logic*: $\{\texttt{T}, \texttt{F}, \texttt{U}\}$

## Conditions

In SQL any expression with `NULL` values has truth value `U`

So, if $A$ and $B$ are equal to `NULL` the following all have truth value `U`

- `A = 5`
- `A > 5`
- `A = B`
- `A = A` (!)
- `A*2 = 6` for integer `A`
- `A*0 = 0` (!)

What about `A = NULL`?

If $A$ is NULL, value is $U$

To test whether $A$ is a NULL value, there is a special syntax

```
A IS NULL
```

```
A IS NOT NULL
```

## Combining conditions

We can now evaluate single conditions

What about logical connectives?

`A = 5 AND B = 4`

- $A = 5$, $B = 3$: $\texttt{T} \wedge \texttt{F} = \texttt{F}$
- $A =$ `NULL`, $B = 3$: $\texttt{U} \wedge \texttt{F} = ?$

`U could be either F or T`

U $\wedge$ F could be

- U stands for T: F
- U stands for F: F

So: U $\wedge$ F = F

But: U $\vee$ F could be

- U stands for T: T
- U stands for F: F

So: U $\vee$ F = U

Truth tables

| p | q | $p \wedge q$ |
|---|---|---|
| T | T | T |
| T | U | U |
| T | F | F |
| U | T | U |
| U | U | U |
| U | F | F |
| F | T | F |
| F | U | F |
| F | F | F |

| p | q | $p \vee q$ |
|---|---|---|
| T | T | T |
| T | U | T |
| T | F | T |
| U | T | T |
| U | U | U |
| U | F | U |
| F | T | T |
| F | U | U |
| F | F | F |

| p | $\neg p$ |
|---|---|
| T | F |
| U | U |
| F | T |

WHERE C

Evaluate C using these rules

Select *only* those tuples for which the condition is true

## Strange consequence

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

Is equivalent to

```
SELECT *
FROM Movies
WHERE length IS NOT NULL;
```

Why?

# View update, again

Person (codiceFiscale, name)
Address (codiceFiscale, city)

View

```
CREATE View R(name, city) AS
   SELECT name, city
   FROM Person NATURAL JOIN Address;
```

Update: Insert the tuple (Paolo, Trento)

Suppose we could insert (Paolo,NULL) into Person and insert (NULL,Trento) into Address

Why would this not work?

What about aggregates?

| *student* | *grade* |
|---|---|
| a | 30 |
| a | NULL |
| b | 18 |

```
SELECT student, AVG(grade) AS avg,
                COUNT(grades) AS count
FROM R
GROUP BY student
```

Result: (a,30,1),(b,18,1)

NULLs are ignored

Exception COUNT(*)

$R$:

| A | B |
|---|---|
| NULL | 5 |
| 6 | NULL |
| NULL | NULL |

```
SELECT COUNT(A) AS X, SUM (B) AS Y, COUNT(*) AS Z
FROM R;
```

Result: $(1, 5, 3)$

# Recursion

Example: Train connections

$R(S, D)$:

| S | D |
|---|---|
| Trento | Verona |
| Verona | Bologna |
| Bologna | Firenze |
| Firenze | Roma |
| Cagliari | Olbia |

Query: Can we get by train from Trento to Olbia?

## One connection

Consider

```
SELECT T1.S AS S, T2.D AS D
FROM R T1, R T2
WHERE T1.D = T2.S;
```

| S | D |
|---|---|
| Trento | Bologna |
| Verona | Firenze |
| Bologna | Roma |

But we want 0 or 1 connections

```
(SELECT * FROM R)
UNION
(SELECT T1.S AS S, T2.D AS D)
 FROM R T1, R T2
 WHERE T1.D = T2.S);
```

Result

| S | D |
|---|---|
| Trento | Verona |
| Trento | Bologna |
| Verona | Bologna |
| Verona | Firenze |
| Bologna | Firenze |
| Bologna | Roma |
| Firenze | Roma |
| Cagliari | Olbia |

0,1, or 2 connections

```
(SELECT * FROM R)
UNION
(SELECT T1.S AS S, T2.D AS D)
 FROM R T1, R1 T2
 WHERE T1.D = T2.S);
```

Result $R2$:

| S | D |
|---|---|
| Trento | Verona |
| Trento | Bologna |
| Trento | Firenze |
| Verona | Bologna |
| Verona | Firenze |
| Verona | Roma |
| Bologna | Firenze |
| Bologna | Roma |
| Firenze | Roma |
| Cagliari | Olbia |

Up to 3 connections

```
(SELECT * FROM R)
UNION
(SELECT T1.S AS S, T2.D AS D)
 FROM R T1, R1 T2
 WHERE T1.D = T2.S);
```

Result $R3$:

| S | D |
|---|---|
| Trento | Verona |
| Trento | Bologna |
| Trento | Firenze |
| Trento | Roma |
| Verona | Bologna |
| Verona | Firenze |
| verona | Roma |
| Bologna | Firenze |
| Bologna | Roma |
| Firenze | Roma |
| Cagliari | Olbia |

- Still no connection from Trento to Olbia
- Try again? But how do we know when to stop?
- We get $R4 = R3$
- And in fact $R3 = R4 = R5 = \cdots$
- So no matter how often we do this, $R3$ is all we need
- But how do we know how often to do this?

## A more formal approach

$R$ is a relation with schema $(A, B)$

$\Phi(U)$ an operator on a relation with schema $(B, C)$

$$\phi(U) = R \cup \pi_{A,C}(R \bowtie U)$$

(we omit the renaming of the first column to $B$)

Define

$$\begin{aligned} R_0 &= \emptyset \\ R_1 &= \Phi(R_0) \\ R_2 &= \Phi(R_1) \\ &\vdots \end{aligned}$$

These are the same relations we created above

If $R_n = R_{n-1}$ then $R_{n+1} = \Phi(R_n) = \Phi(R_{n-1}) = R_n$

So once we find a *fixpoint* $S$, i.e. a relation with $\Phi(S) = S$ we are done.

We can evaluate the query $(\text{Trento}, \text{Olbia})$ on this *fixpoint* and get a negative answer

But why should a fixpoint exist?

Our original database uses 7 different *constants*: Trento, Verona, ...

- There are 49 different tuples we can build with these constants
- $\Phi$ does not construct new values (with arithmetic, queries on other relations etc)
- $U \subseteq \Phi(U)$: we never lose tuples
- Assuming we haven't reached a fixpoint $R_0$ has 5 tuples, $R_1$ at least 6 tuples, ...
- So after 42 iterations we have all 49 tuples and we have finished
- In theory complexity is terrible, in practice usually much better

Terminology: Given $R(x, y)$ the fixpoint is called the ‘‘transitive closure’’

Transitive relation

$$R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$$

*Transitive closure*: Smallest transitive extension of $R$

# SQL

WITH seen before as an abbreviation

For recursion it is essential: WITH defines a relation that depends on the relation it is defining

New keyword RECURSIVE

Initial relation R

Transitive closure Reaches

```
WITH RECURSIVE Reaches(from,to) AS
     (SELECT frm,to FROM R)
   UNION
     (SELECT R1.frm, R2.to
      FROM Reaches R1, Reaches R2
      WHERE R1.to = R2.frm)
SELECT * FROM Reaches;
```

Last line is the query

SELECT * FROM Reaches;

WITH constructs Reaches

Meaning is: Start with Reaches empty, then reiterate until
we reach fixpoint

Actual construction is slightly different: it finds all routes
with 1,2,4,8,...connections, so coverges faster

## What can go wrong

Artificial example: Negation. R(x) unary relation

Also shows use of double recursion

```
WITH
  RECURSIVE P(x) AS
    (SELECT * FROM R)
      EXCEPT
    (SELECT * FROM Q)

  RECURSIVE Q(x) AS
    (SELECT * FROM R)
      EXCEPT
    (SELECT * FROM P)

SELECT * FROM P;
```

Let $R$ initially contain only the tuple $(0)$

Initially $P = Q = \emptyset$

After one iteration, $P = Q = \{(0)\}$

After two iterations $P = Q = \emptyset$

After $2n + 1$ iteration, $P = Q = \{(0)\}$

After $2n$ iterations $P = Q = \emptyset$

No fixpoint

# Creation of new values

A more realistic example.

$R(S, D, T)$:

| S | D | T |
|---|---|---|
| Trento | Verona | 1 |
| Verona | Bologna | 2 |
| Bologna | Firenze | 1 |
| Firenze | Roma | 1 |
| Cagliari | Olbia | 4 |

Define $U(S, D, T)$ recursively using

```
(SELECT * FROM R)
UNION
(SELECT T1.S AS S, T2.D AS D, T1.T+T2.T as T)
 FROM R T1, U T2
 WHERE T1.D = T2.S);
```

Result

| S | D | T |
|---|---|---|
| Trento | Verona | 1 |
| Trento | Bologna | 3 |
| Trento | Firenze | 4 |
| Trento | Roma | 5 |
| Verona | Bologna | 2 |
| Verona | Firenze | 3 |
| Verona | Roma | 4 |
| Bologna | Firenze | 1 |
| Bologna | Roma | 2 |
| Firenze | Roma | 1 |
| Cagliari | Olbia | 4 |

Result depends on the data

Suppose we had (note the additional tuple at the end)

| S | D | T |
|---|---|---|
| Trento | Verona | 1 |
| Verona | Bologna | 2 |
| Bologna | Firenze | 1 |
| Firenze | Roma | 1 |
| Cagliari | Olbia | 4 |
| Verona | Trento | 1 |

Result includes:

| S | D | T |
|---|---|---|
| Trento | Verona | 1 |
| Verona | Trento | 1 |
| Trento | Trento | 2 |
| Trento | Verona | 3 |
| Verona | Trento | 3 |
| Trento | Trento | 4 |
| ⋮ | ⋮ | ⋮ |

No fixpoint!

Can be fixed by

```
(SELECT * FROM R)
UNION
(SELECT T1.S AS S, T2.D AS D, T1.T+T2.T as T)
 FROM R T1, U T2
 WHERE T1.D = T2.S AND T1.S <> T2.D);
```

How to we decide one works and the other doesn't?

In practice: Recursion only used for simple transitive closure as seen above