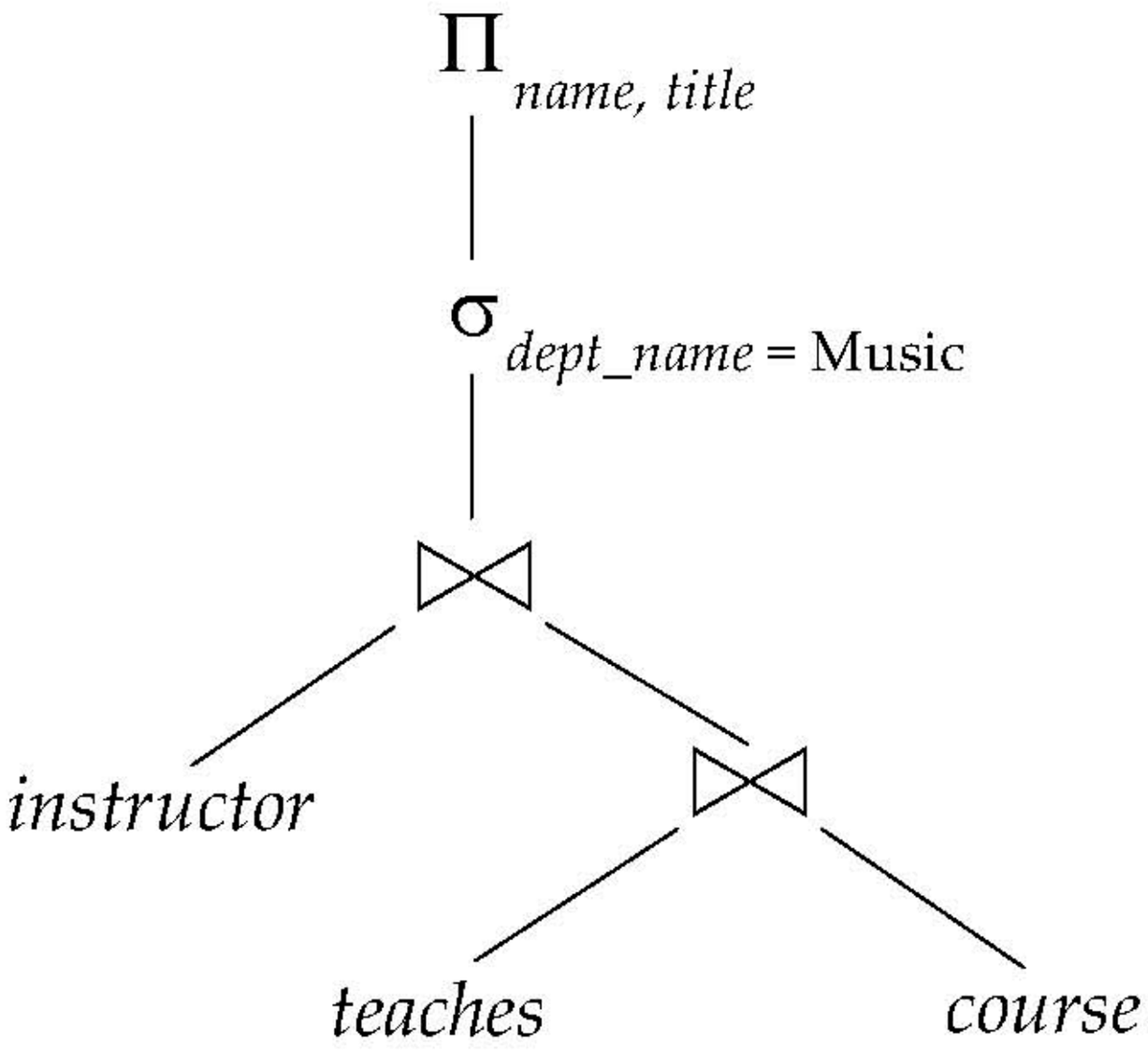
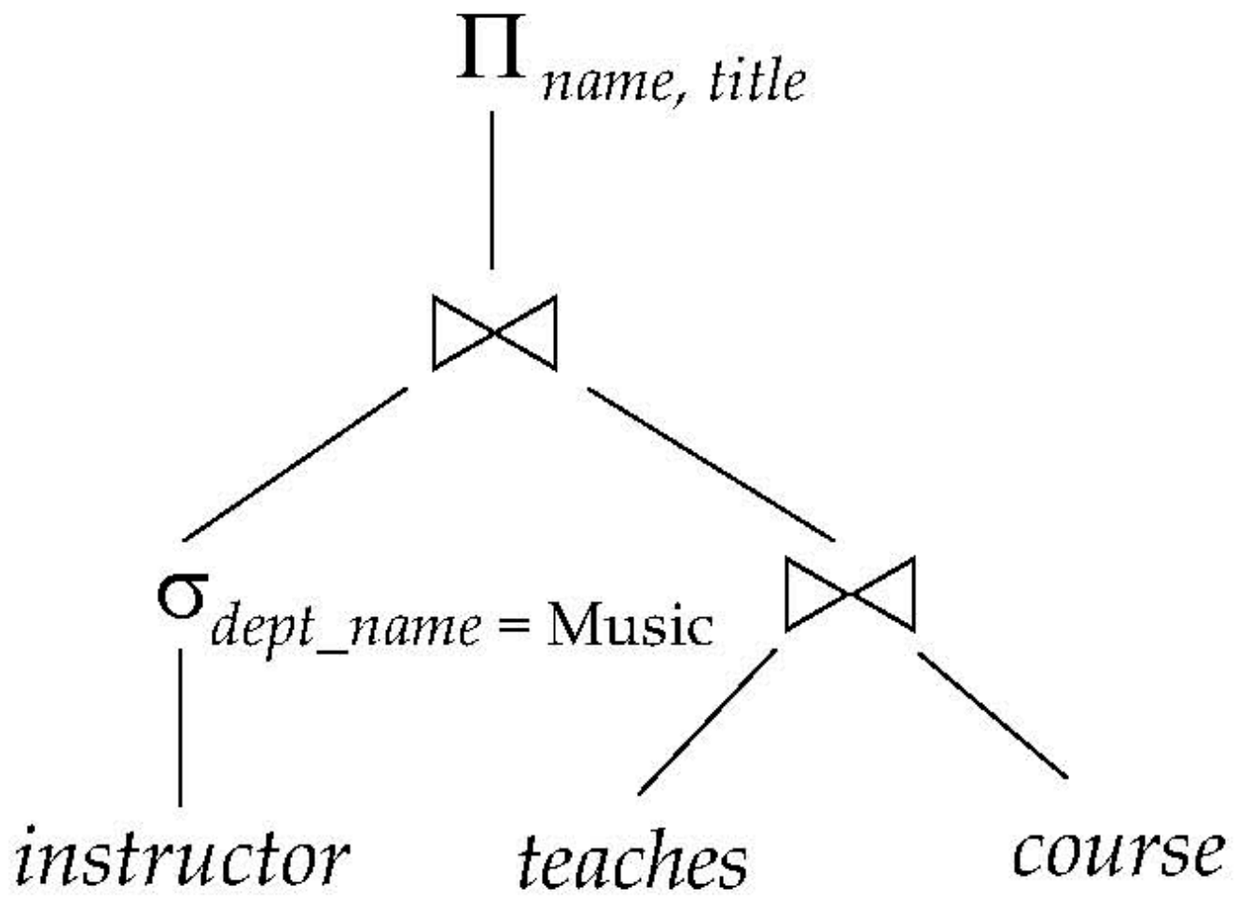
Query Optimization and Relational Integrity (Keys)

Is it possible to express a query in multiple ways?

Example:





Evaluation Plans

- There is not in general an unique way for evaluating a query.
- What is the difference between the different ways?
- What if a choice is particularly “expensive”?
- Need for an **evaluation plan**

How to evaluate the “cost” of a query?

- Statistical informations:
  - How many fields?
  - How many records?
- Statistical informations of the intermediate results.
- Computational cost of algorithms.

Relational Equivalences (from previous lectures)

$\sigma_{c_1 \wedge c_2 \wedge \ldots} = \sigma_{c_1}(\sigma_{c_2}(\sigma_{c_3} \ldots)))$ cascading select

$\sigma_{c1}(\sigma_{c2}(T)) = \sigma_{c2}(\sigma_{c1}(T))$ select commutativity

$\pi_{f1}(\pi_{f2}(\pi_{f3} \ldots)) = \pi_{f1}(T)$ only the last projection matters

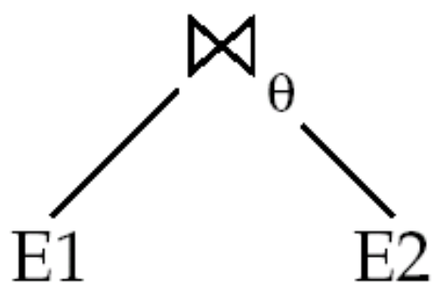
$T1 \bowtie_\theta T2 = T2 \bowtie_\theta T1$ theta-join commutativity

$(T1 \bowtie T2) \bowtie T3 = T1 \bowtie (T2 \bowtie T3)$ natural-join associativity

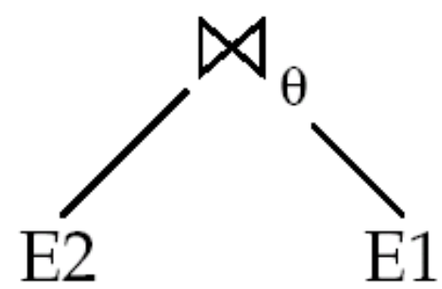
$(T1 \bowtie_{c_1} T2) \bowtie_{c_2 \wedge c_3} T3 = T1 \bowtie_{c_1 \wedge c_3} (T2 \bowtie_{c_2} T3)$ theta-join associativity

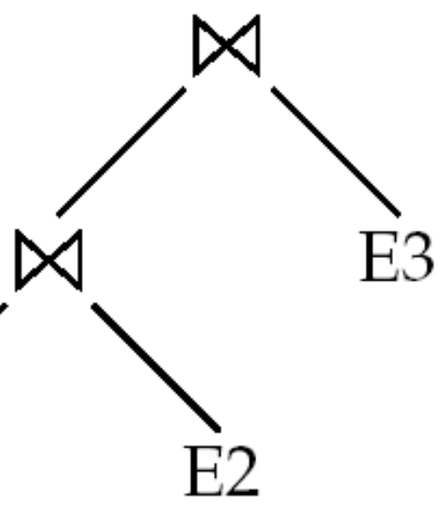
Union and Intersection are commutative and associative

Graphical Representations Examples

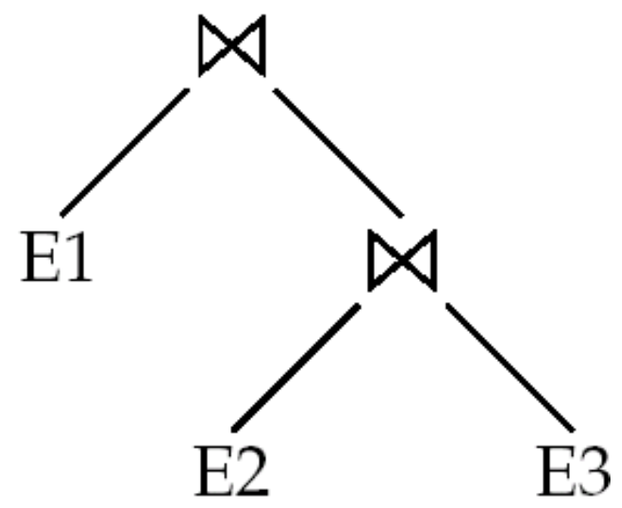


$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$





$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$



Query Optimizer

- Databases have query optimizers built in.

- Algorithmically complex: for complicated queries the possible equivalences are many.

- The number of equivalences must be “weighted” by the statistics of the tables.

- Non-trivial problem: still subject of active research!

What can we do as programmers?

Writing optimal queries to begin with is always a good practice:

- Less optimization needed
- Faster execution

Some general rules:

1) Avoid the use of "*" if you know the fields you want
2) Minimize the amount of subqueries
3) > (or<) is in general better than "!="
4) Avoid mathematical calculations if you can do them in advance
5) When WHERE-in strings: use LIKE instead of SUBSTR()=''
6) Use BETWEEN in stead of AND with "> / <".
7) Try to write a single query instead of JOIN-ing results from multiple queries.
8) Use LIMIT if needed
9) Use indices for speeding up SELECT/WHERE

Primary Keys

A **primary key** is a column or a set of columns that uniquely identifies each row in the table. A primary key must satisfy the following rules:

- Must contain unique values. If the primary key consists of multiple columns, the combination of values in these columns must be unique.

- A primary key column cannot contain NULL values. It means that you have to declare the primary key column with the NOT NULL attribute. If you don't, MySQL will force the primary key column as NOT NULL implicitly.
  A table has only one primary key.

NOTE:
MySQL works faster with integers, so the data type of the primary key column should be the integer e.g.,
INT, BIGINT, TINYINT, SMALLINT.
Make sure that the type can contain all your records.

NOTE: Identity columns vs Primary Keys: An identity column may be a primary key, but a primary key is not an identity column: it is a set of columns that you define that determine what makes the data in your table unique. It defines your data and it may be an identity column, it may be a varchar column or a datetime column or an integer column, or it may be a combination of multiple columns (-> **Composite Primary Keys**).

Primary Keys

```
CREATE TABLE users(
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(40),
    password VARCHAR(255),
    email VARCHAR(255)
);
```

Or:

```
CREATE TABLE users(
    user_id INT AUTO_INCREMENT,
    username VARCHAR(50),
    password VARCHAR(255),
    email VARCHAR(255)
    PRIMARY KEY(user_id,…)
);
```

The second syntax is mandatory when more than one field composes the primary key.

Addition after creation

```
ALTER TABLE table_name
ADD PRIMARY KEY(primary_key_column);
```

Foreign Keys

A **foreign key** is a field in a table that matches another field of another table.
A foreign key places constraints on data in the related tables, which enables MySQL to maintain **referential integrity**.

A table may have more than one foreign key, and each foreign key in the child table may refer to a different parent table.

Sometimes, the child and parent tables are the same. The foreign key refers back to the primary key of the table

Foreign Keys Creation in MySQL

```
CONSTRAINT constraint_name
FOREIGN KEY foreign_key_name (columns)
REFERENCES parent_table(columns)
ON DELETE action
ON UPDATE action
```

The `CONSTRAINT` clause defines the constraint name for the foreign key constraint. If not declared, MySQL will generate a name automatically.

The `FOREIGN KEY` clause specifies the columns in the child table that refers to primary key columns in the parent table. You can put a foreign key name after FOREIGN KEY clause or leave it to let MySQL create a name for you. MySQL automatically creates an index with the foreign_key_name name.

The `REFERENCES` clause specifies the parent table and its columns to which the columns in the child table refer. The number of columns in the child table and parent table specified in the FOREIGN KEY and REFERENCES must be the same.

ON… : Specifies what happens to the child table's rows when parent's rows are deleted or updated.

Full Example (MySQL)

```
CREATE DATABASE IF NOT EXISTS test;

USE test;

CREATE TABLE student(
    st_id int not null auto_increment primary key,
    st_name varchar(255) not null,
    st_description text
) ENGINE=InnoDB;
```

Storage Engine
(more on this later)

```
CREATE TABLE courses(
    c_id int not null auto_increment primary key,
    c_name varchar(355) not null,
    c_price decimal,
    st_id int not null,
    FOREIGN KEY fk_cat(st_id)
    REFERENCES categories(st_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT
)ENGINE=InnoDB;
```

Add a Foreign Key

It is also possible to add a foreign key after the creation of a table:

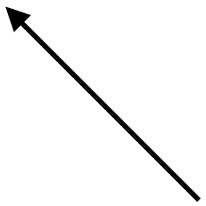
```
ALTER table_name
ADD CONSTRAINT constraint_name
FOREIGN KEY foreign_key_name(columns)
REFERENCES parent_table(columns)
ON DELETE action
ON UPDATE action;
```

Or to remove one:

```
ALTER TABLE table_name
DROP FOREIGN KEY constraint_name;
```

Other Useful Commands

```
SHOW CREATE TABLE table_name;
```



Prints out the creation code.

Disables (or enables) the action of foreign keys:

```
SET foreign_key_checks = 0 (or 1);
```