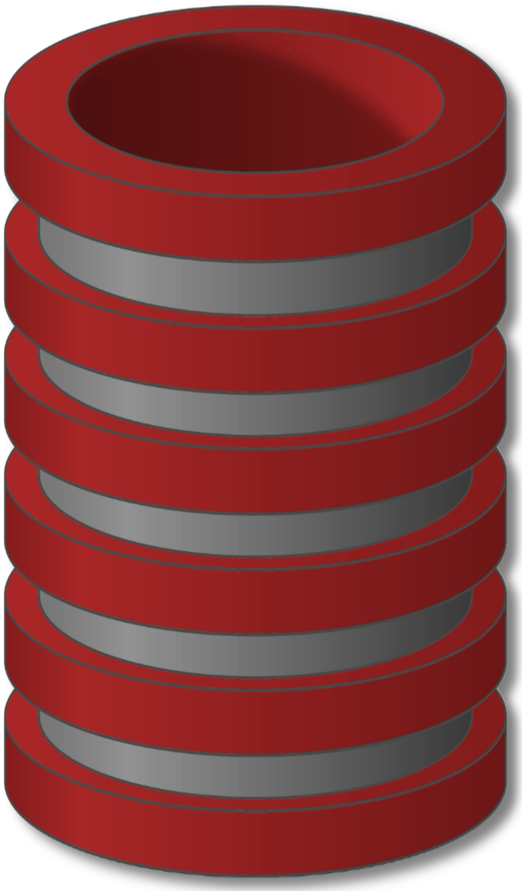
Advanced SQL

Advanced SQL

- Data Types
- Integrity Constraints
- Trigger
- Authorization
- Index



Data Types

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Adding or subtracting an interval to a date or time gives back a date or time, respectively

Build-in Data Types in SQL (Cont.)

- Can extract values of individual fields from date/time/timestamp
  - **extract** ($field$ **from** d)
    - $field$: can be one of year, month, day, hour, minute, or second.
    - d: date or time value
  - Example: SELECT EXTRACT(YEAR FROM ‘2019-09-22')

- Can cast string types to date/time/timestamp
  - Example: **cast** <string-valued-expression> **as date**
  - Example: **cast** <string-valued-expression> **as time**

User-Defined Types

- **create type** creates user-defined type in SQL

  create type *Dollars* **as numeric (12,2) final**

  - **final:** no meaning here, required by the SQL:1999

  create table department
  (dept_name **varchar** (20),
  building **varchar** (15),
  budget *Dollars*);

- **create domain** construct in SQL-92 creates similar user-defined domain types

  create domain *person_name* **char**(20) **not null**

User-Defined Types

- User-defined types and domains are similar.
- Differences between types and domains
  - Domains can have constraints, such as **not null** specified on them, and can have default values defined for variables of the domain type
  - Domains are not strongly typed. Values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible
  - Example: **create domain** *Dollars* **numeric**(12, 2)
    create domain *Pounds* **numeric**(12, 2)

Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraint.
  - They test values inserted in the database, and test queries to ensure that the comparisons make sense.

- We cannot assign or compare a value of **type** Dollars to a value of **type** Pounds.
  - However, we can convert type as below
    (**cast** *r.A* **as** *Pounds*)
    (Should also multiply by the dollar-to-pound conversion-rate)

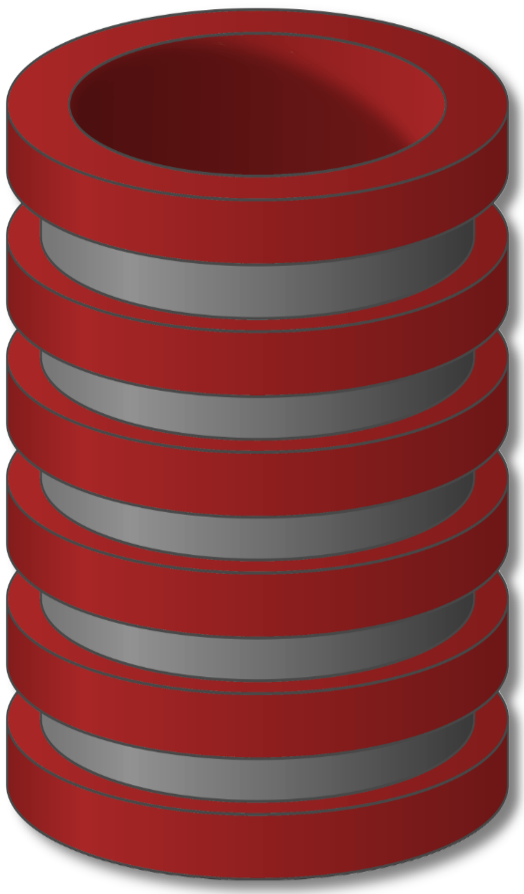
Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object
    - object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object
    - object is a large collection of character data

      book_review **clob**(10KB)

      image **blob**(10MB)

      movie **blob**(2GB)
  - When a query returns a large object, a pointer is returned rather than the large object itself.



Integrity Constraints

Integrity Constraints

- **Integrity Constraints (ICs):** conditions that must be true for any instance of the database; e.g., data type.
  - An instructor name cannot be null
  - A salary of a bank employee must be at least $4.00 an hour
  - No two instructors can have the same instructor ID
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- ICs are based upon the semantics of the application that is being described in the database relations.
- A **legal** instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
  - Avoids data entry errors, too!

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (*P*), where *P* is a predicate

Not Null Constraint

- Declare *branch_name* for *branch* is **not null**

  branch_name **char**(15) **not null**

- Declare the domain *Dollars* to be **not null**

  create domain *Dollars* **numeric**(12,2) **not null**

The Unique Constraint

- **unique** (A_1, A_2, …, A_m)
- The unique specification states that the attributes

  A_1, A_2, …, A_m
  form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).

Primary Key Constraints

- A set of attribute is a **key** for a relation if :
  1. No two tuples can have same values in all these attributes, and
  2. This is not true for any subset of the key.

- Part 2 false? A **superkey**
- If there's more than one key for a relation (i.e., **candidate keys**), one is chosen as the **primary key**.

Students(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real).

- *sid* is a key. (What about *name*? What about *login*?)
- {*sid, gpa*} is a superkey.

Primary & Candidate Keys in SQL

- Primary key specified as the PRIMARY KEY
  - Primary key cannot take a null value
- Candidate keys specified using UNIQUE

```
CREATE TABLE Students
        (sid  CHAR(20),
         name  CHAR(20),
         login   CHAR(10),
         age   INTEGER,
         gpa  REAL,
         PRIMARY KEY (sid),
         UNIQUE (login) )
```

- What's the result of executing the following statements?
  - INSERT INTO Students VALUES ('00001', 'Bob', 'bob@comp', 18, 3.2)
  - INSERT INTO Students VALUES ('00001', 'Tom', 'tom@comp', 18, 3.2)
  - INSERT INTO Students VALUES ('00002', 'Bob', 'bob@comp', 18, 3.2)

Checking arbitrary constraints

- We want to make sure that every tuple in STUDENTS has a positive gpa at all times.
- CREATE TABLE STUDENTS
  (…,
    PRIMARY KEY(…),
    CHECK (*gpa* > 0))
- As a result, the database will reject an insertion or an update, if the resulting tuple has a 0 or negative gpa.
- We can write more complex conditions in CHECK (see next).

Check – Example 2

- We already have a table CLUB, recording the GPA of all members in the dancing club.
- We want to create a table
  PANEL(*stu-id*, *major*)
  where each tuple corresponds to a member in the panel of the club.
- We require that every panel member should have a GPA at least 1.7.
- We can ensure this with CHECK.

CLUB

stu-id	*gpa*
1	3
2	1.8
3	1.7
4	1.2
5	1.2

PANEL

stu-id	*major*
1	EE
3	CS

Check – Example 2

- We require that every panel member should have a GPA at least 1.7.

- CREATE TABLE PANEL
  (*stu-id* INTEGER,
  major CHAR(20),
  PRIMARY KEY (*stu-id*),
  CHECK (*stu-id* IN
  (SELECT *stu-id* FROM CLUB
  WHERE *gpa* >= 1.7)))

CLUB

stu-id	*gpa*
1	3
2	1.8
3	1.7
4	1.2
5	1.2

PANEL

stu-id	*major*
1	EE
3	CS

- The database will check the condition whenever
  - there is an insertion/update on PANEL

- *Note: Oracle does not allow CHECK conditions to use subqueries.*

The check clause

- **check** (P)

  where P is a predicate
- The **check** clause in SQL-92 permits domains to be restricted:
  - Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.

    create domain *hourly_wage* **numeric(5,2)**
    constraint *value_test* **check**(*value* > = 4.00)
  - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
  - The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

Referential Integrity

- Example: If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".
- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
  - The primary key clause lists attributes that comprise the primary key.
  - The unique key clause lists attributes that comprise a candidate key.
  - The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key.
    - By default, a foreign key references the primary key attributes of the referenced table.

Foreign Key

- **Foreign key**: Set of attributes in one relation that is used to `refer' to a tuple in another relation (can be itself).
  - Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - E.g., *sid* is a foreign key referring to Students

Students(<u>sid</u>: string, name: string, login: string, age: integer, gpa: real)
Enrolled(sid: string, cid: string, grade: string)

Foreign Key – Why do we need it?

- Consider Students and Enrolled:

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

- Assume that we want to insert a tuple (‘50000’, ‘CS160’, ‘A’) into Enrolled.
- Before we do so, we may want to make sure there exists a student in Students with *sid* = ‘50000’.

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

- Foreign key is used to achieve this
  - If every *sid* in Enrolled exists in Students, referential integrity is achieved.

Foreign Keys in SQL

- Only students listed in the *Students* relation should be allowed to enroll for courses.
- But some tuples in *Students* may not be referenced.

```
CREATE TABLE Enrolled
  (sid CHAR(20),  cid CHAR(20),  grade CHAR(2),
   PRIMARY KEY  (sid, cid),
   FOREIGN KEY (sid) REFERENCES Students )
```

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Foreign Keys in SQL

- Attribute names can be different.
  - Specify the attribute name explicitly in the definition.

Students(sid: string, name: string, login: string, age: integer, gpa: real)
Enrolled2(stuid: string, cid: string, grade: string)

```
CREATE TABLE Enrolled2
    (stuid CHAR(20),  cid CHAR(20),  grade CHAR(2),
     PRIMARY KEY  (stuid, cid),
     FOREIGN KEY (stuid) REFERENCES Students (sid) )
```

Foreign Keys in SQL

- Foreign keys can refer to the same relation. E.g.,

Students2(<u>sid</u>:string, name:string, login:string, age:integer, gpa:real, partner:string)

sid	name	login	age	gpa	partner
53666	John	john@cs	18	2.7	53668
53668	Smith	smith@cs	18	3.8	53666
53650	Smith	smith@ee	19	3.3	NULL

Foreign Keys in SQL

- Foreign keys can refer to the same relation. E.g.,

Students2(<u>sid</u>:string, name:string, login:string, age:integer, gpa:real, partner:string)

```
CREATE TABLE Students2
    (sid CHAR(20), name CHAR(20), login CHAR(10),
     age INTEGER, gpa REAL, partner CHAR(20),
     PRIMARY KEY (sid),
     FOREIGN KEY (partner) REFERENCES Students2 (sid) )
```

- If a student has no partner, this field can be NULL (a special keyword in SQL denoting *`unknown'* or *`inapplicable'*).
- NULL is allowed in non-primary keys, including foreign keys.

Enforcing Referential Integrity

- Consider *Students* and *Enrolled* in the example; *sid* in *Enrolled* is a foreign key that references *Students*.
- What should be done if an *Enrolled* tuple with a non-existent student id is inserted? **Reject it!**
- What should be done if a Students tuple is deleted?
  1. Disallow deletion of a Students tuple that is referred to.
  2. Also delete all *Enrolled* tuples that refer to it.
  3. Set *sid* in Enrolled tuples that refer to it to a *default* *sid*.
  4. Another possible option: Set *sid* in Enrolled tuples that refer to it to a special value *NULL*.
     - In this example, cannot do it here because *sid* is part of primary key of Enrolled.

     → *Null is allowed in a foreign key field but not in a primary key field!*
- Similar if primary key of *Students* tuple is updated.

Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is NO ACTION (*delete/update is rejected*)
  - CASCADE (also delete all tuples that refer to deleted tuple)
  - SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

```
CREATE TABLE Enrolled
  (sid CHAR(20) DEFAULT '53688',
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid, cid),
   FOREIGN KEY (sid)
     REFERENCES Students
          ON DELETE CASCADE
          ON UPDATE SET DEFAULT )
```

Referential Integrity – Example

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

- What happens if the table Students is updated as follows:
  - Delete the tuple with sid = 53666
  - Insert a tuple with sid = 53600?
  - Update the tuple with sid=53650 → 53700?

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions.
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- SQL does not provide

  for all X, $P(X)$

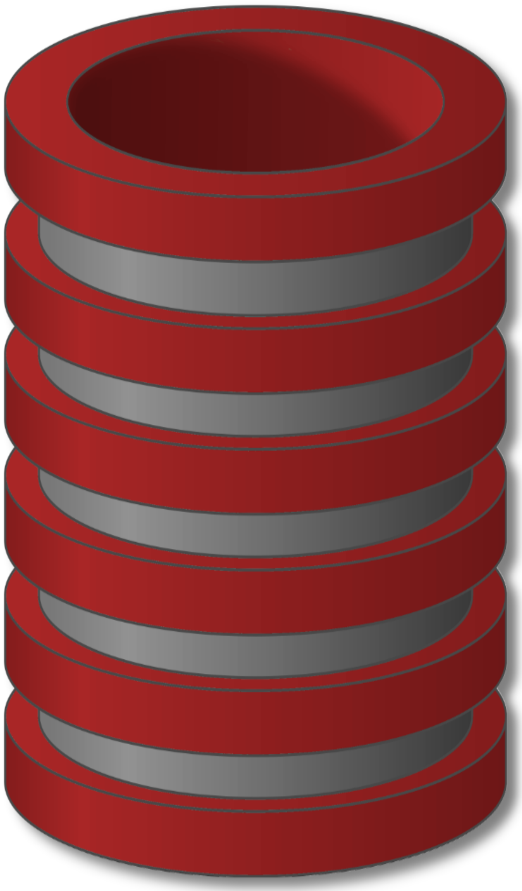
  above is achieved by using

  not exists X such that not $P(X)$

Assertion Example

- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.

```
create assertion credits_earned_constraint check
    (not exists (select ID
            from student
            where tot_cred <> (select sum(credits)
                    from takes natural join course
                    where student.ID= takes.ID
                    and grade is not null and grade<> 'F' ))
```



Triggers

Overview

- We have learned to write the following constraints.
  - Primary/candidate key (unique)
  - Foreign key
  - CHECK
  - assertion

- Next, we will discuss another powerful mechanism for writing constraints.

- Trigger.
- A trigger can be regarded as a procedure automatically executed by the database, whenever a certain table is modified.

Triggers

- Two requirements of designing a trigger:
  - Specify when a trigger is to be executed.
    - An event: causes the trigger to be checked
    - A condition: must be satisfied for trigger execution to proceed
  - Specify the actions to be taken when the trigger executes.

Triggers

- Referential integrity on the *time_slot_id* attribute of the *section* relation

create trigger timeslot_check1 **after insert on** section
referencing new row as nrow
for each row — iterate over each inserted row
when (nrow.time_slot_id **not in** (
    select time_slot_id
    from time_slot)) — *time_slot_id* not present in *time_slot.*
begin
    rollback — any transaction that violates the referential integrity constraint gets rolled back
end;

Triggers

- Referential integrity on the *time_slot_id* attribute of the *section* relation

create trigger timeslot_check2 **after delete on** time_slot
referencing old row as orow
for each row
when (orow.time_slot_id **not in (**
select time_slot_id
from time_slot**)**
and orow.time_slot_id **in (**
select time_slot_id
from section))
begin
rollback
end;

last tuple for *time_slot_id* deleted from *time_slot*

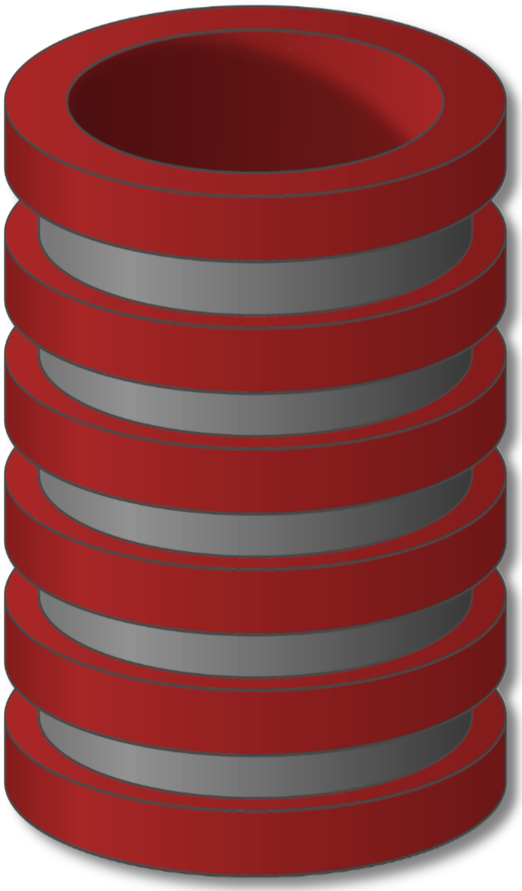
and *time_slot_id* still referenced from *section*

Triggers

- Handle grade corrections that change a successful completion grade from a fail grade

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
begin atomic
        update student
        set tot_cred= tot_cred+
                (select credits
                from course
                where course.course id= nrow.course id)
        where student.id = nrow.id;
end;
```

Complete a previously failed course



Authorization

Authorization Specification in SQL

- Authorizations on data include:
  - Authorization to read data.
  - Authorization to insert new data.
  - Authorization to update data.
  - Authorization to delete data.
- Each of these types of authorizations is called a privilege.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name>

  to <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

    grant select on *branch* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples.
- **update**: the ability to update using the SQL update statement.
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges.

Revoking Authorization in SQL

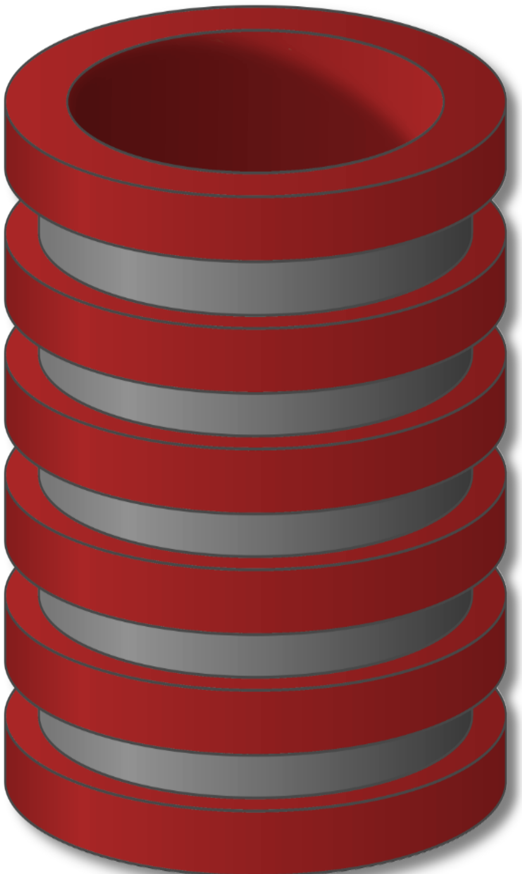
revoke <privilege list>

on <relation name or view name> **from** <user list>

- **revoke select on** *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the **revoke** statement
- All privileges that depend on the privilege being revoked are also revoked.

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *staff*
- Suppose that a staff member issues
  - **select** *
    from *geo_instructor*;



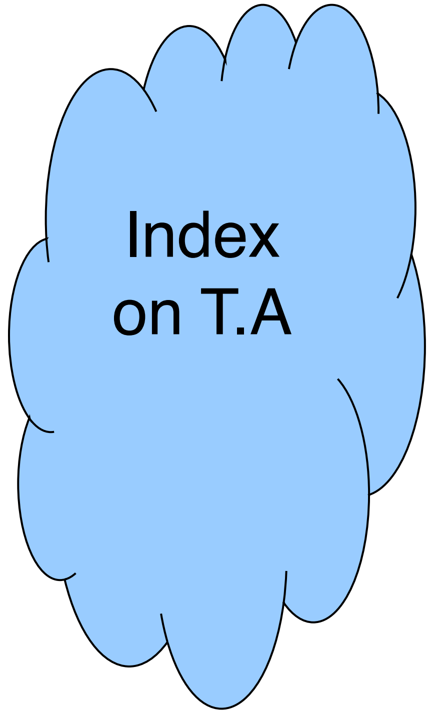
Indexes

Indexes

- Primary mechanism to get improved performance on a database
- Persistent data structure, stored in database
- Many interesting implementation issues

  But we are focusing on user/application perspective

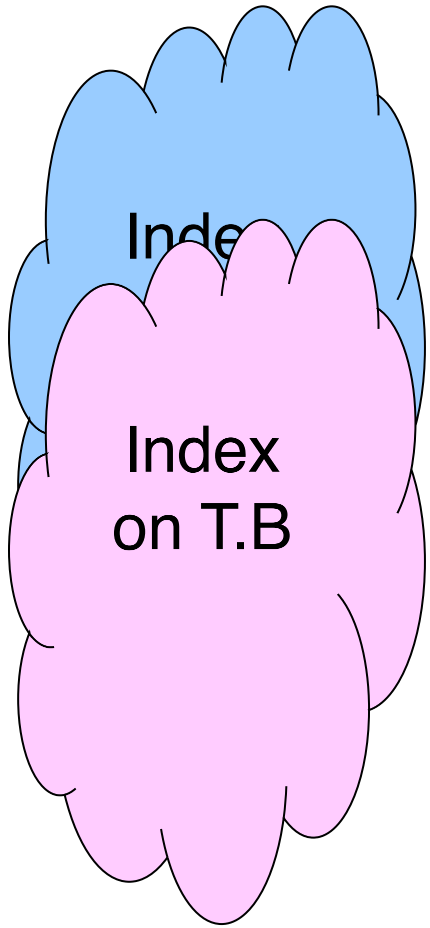
Functionality



T

	A	B	C
1	cat	2	...
2	dog	5	...
3	cow	1	...
4	dog	9	...
5	cat	2	...
6	cat	8	...
7	cow	6	...
	...	...	...

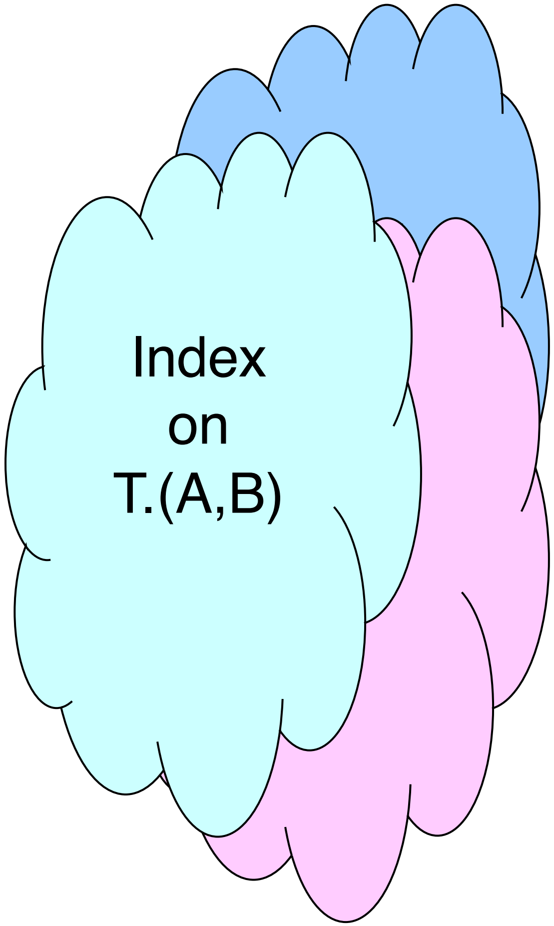
Functionality



T

	A	B	C
1	cat	2	...
2	dog	5	...
3	cow	1	...
4	dog	9	...
5	cat	2	...
6	cat	8	...
7	cow	6	...
	...	...	...

Functionality



T

	A	B	C
1	cat	2	...
2	dog	5	...
3	cow	1	...
4	dog	9	...
5	cat	2	...
6	cat	8	...
7	cow	6	...
	...	...	...

Utility

- Index = difference between full table scans and immediate location of tuples
  - ✳ Orders of magnitude performance difference
- Underlying data structures
  - – Balanced trees (B trees, B+ trees)
  - – Hash tables

```
Select sName
From Student
Where sID = 18942
```

- Many DBMS's build indexes automatically on **PRIMARY KEY** (and sometimes **UNIQUE**) attributes

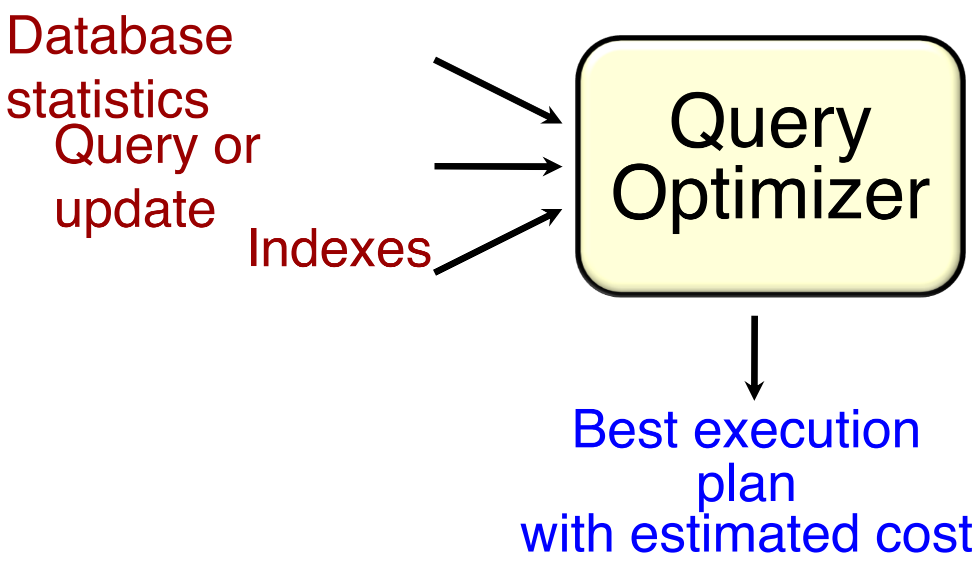
Picking which indices to create

Benefit of an index depends on:

- Size of table (and possibly layout)
- Data distributions
- Query vs. update load

"Physical design advisors"

Input: database (statistics) and workload
Output: recommended indexes



SQL Syntax

```
Create Index IndexName on T(A)

Create Index IndexName on T(A1,A2,…,An)

Create Unique Index IndexName on T(A)

Drop Index IndexName
```