Principles of Database Systems (CS307)

Lecture 12: Trigger; Data Storage Structures

Ran Cheng

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult, Dr. Yuxin Ma and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

Trigger

Trigger - Actions When Changing Tables

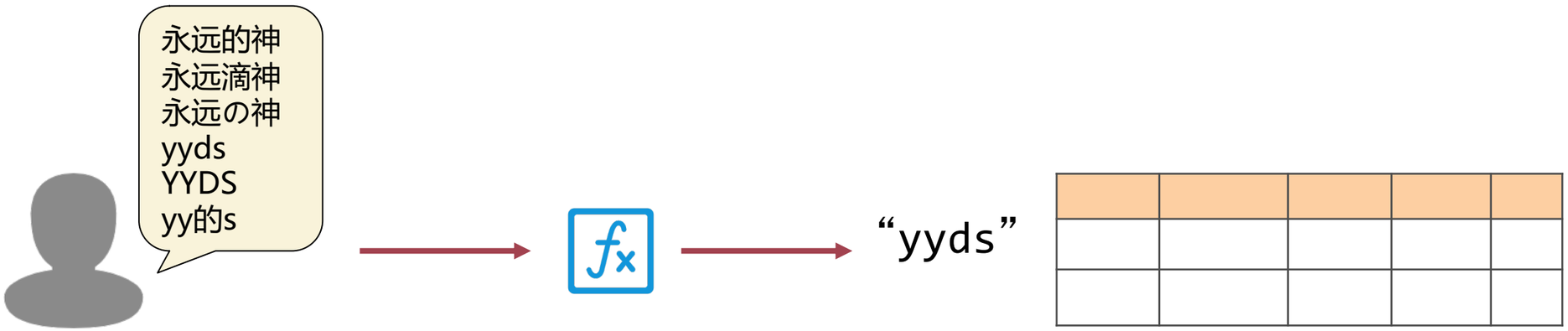
> A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed.
>
> -- Chapter 39, PostgreSQL Documentation
>
> A trigger is a statement that the system executes automatically as a side effect of a modification to the database.
>
> -- Chapter 5.3, Database System Concepts, 7th

- We can attach “actions” to a table
  - They will be executed automatically whenever the data in the table changes
- Purpose of using triggers
  - Validate data
  - Checking complex rules
  - Manage data redundancy

Purpose of Using Triggers

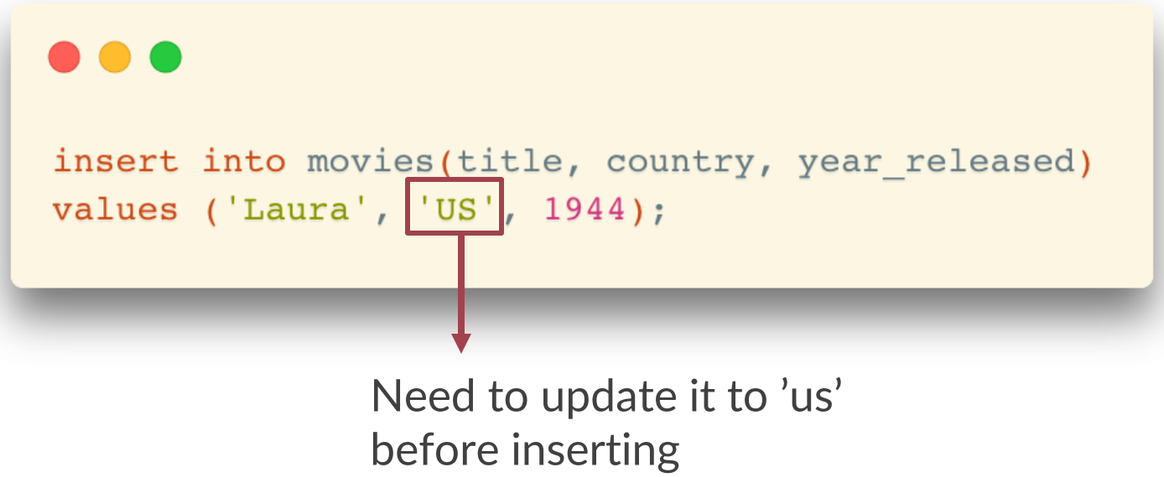
- Validate data
  - Some data are badly processed in programs before sending to the database
  - We need to validate such data before inserting them into the database

  - “On-the-fly” modification
    - Change the input directly when the input arrives



Purpose of Using Triggers

- Validate data
  - Example: insert a row in the movies table
    - In the JDBC program, an `insert` request is written like the following:



  - Although,
    - Such validation or transformation should be better handled by the application programs

Purpose of Using Triggers

- Check complex rules
  - Sometimes, the business rules are so complex that it cannot be checked via declarative constraints

Purpose of Using Triggers

- Manage data redundancy
  - Some redundancy issues may not be avoided by simply adding constraints

  - For example: We inserted the same movie but in different languages

```sql
-- US
insert into movies(title, country, year_released)
values ('The Matrix', 'us', 1999);

-- China (Mainland)
insert into movies(title, country, year_released)
values ('黑客帝国', 'us', 1999);

-- Hongkong
insert into movies(title, country, year_released
values ('22世紀殺人網絡', 'us', 1999);
```

It satisfies the constraint of uniqueness on (`title, country, year_released`)

- ... but they represent the same movie

Trigger Activation

- Two key points:
  - When to fire a trigger?
  - What (command) fires a trigger?

Trigger Activation

- When to fire a trigger?
  - In general: “During the change of data”
    - … but we need a detailed discussion

--- Note: “During the change” means `select` queries won’t fire a trigger.

Trigger Activation: When

- Example: Insert a set of rows with “`insert into select`”
  - One statement, multiple rows

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais;
```

Trigger Activation: When

- Example: Insert a set of rows with "`insert into select`"
  - One statement, multiple rows

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais;
```

  - Option 1: Fire a trigger only once for the statement
    - Before the first row is inserted, or after the last row is inserted
  - Option 2: Fire a trigger for each row
    - Before or after the row is inserted

Trigger Activation: When

- Different options between DBMS products

- Before statement
  - Before each row
  - After each row
- After statement

- Before statement
  - Before each row
  - After each row
- After statement

- Before statement
  - Before each row
  - After each row
- After statement

Trigger Activation: What

- What (command) fires a trigger?
  - `insert`
  - `update`
  - `delete`

Example of Triggers

- A (Toy) Example
  - For the following `people_1` table, count the number of movies when updating a person and save the result in the `num_movies` column

```sql
-- auto-generated definition
create table people_1
(
    peopleid   integer,
    first_name varchar(30),
    surname    varchar(30),
    born       integer,
    died       integer,
    gender     bpchar,
    num_movies integer
);
```

	peopleid	first_name	surname	born	died	gender	num_movies
1	13	Hiam	Abbass	1960	<null>	F	<null>
2	559	Aleksandr	Askoldov	1932	<null>	M	<null>
3	572	John	Astin	1930	<null>	M	<null>
4	585	Essence	Atkins	1972	<null>	F	<null>
5	598	Antonella	Attili	1963	<null>	F	<null>
6	611	Stéphane	Audran	1932	<null>	F	<null>
7	624	William	Austin	1884	1975	M	<null>
8	637	Tex	Avery	1908	1980	M	<null>
9	650	Dan	Aykroyd	1952	<null>	M	<null>
10	520	Zackary	Arthur	2006	<null>	M	<null>
11	533	Oscar	Asche	1871	1936	M	<null>
12	546	Elizabeth	Ashley	1939	<null>	F	<null>

Example of Triggers

- Create a trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

Example of Triggers

- Create a trigger

Name of the trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

Example of Triggers

- Create a trigger

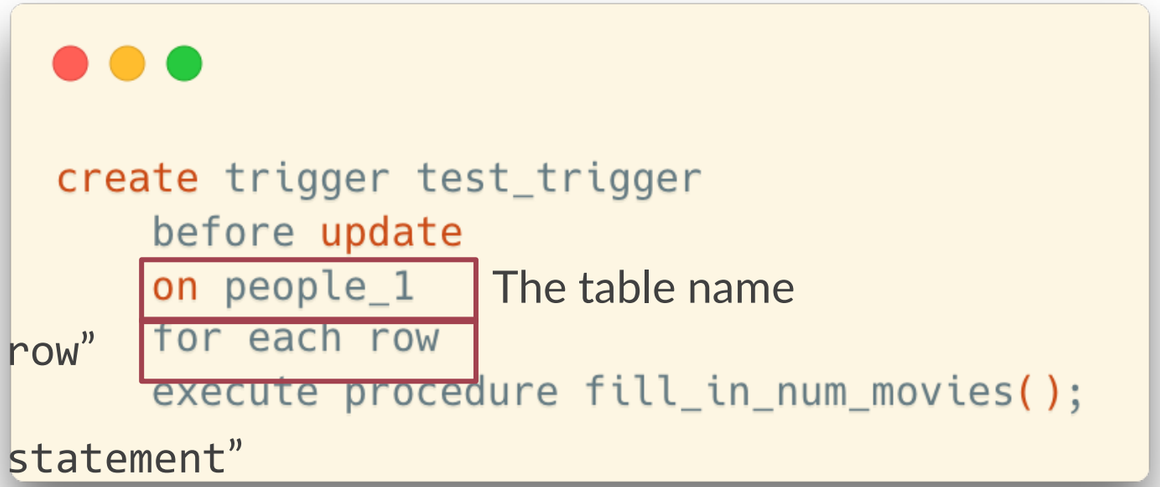
{ BEFORE | AFTER | INSTEAD OF } { event [OR ...] }

- Specify when the trigger will be executed
  - `before` | `after`
- … and on what operations the trigger will be executed
  - `insert [or update [or delete]]`

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

Example of Triggers

- Create a trigger



Example of Triggers

- Create a trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

The actual procedure for the trigger

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

"trigger" is the return type

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

”new” and “old” are two internal variables that represents the row before and after the changes

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Remember to return the result which will be used in the `update` statement

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well
    - Remember to create the procedure before creating the trigger

- Run test updates

```
-- create the procedure fill_in_num_movies() first

-- then, create the trigger

-- finally, we can run some test update statements
update people_1 set num_movies = 0 where people_1.peopleid <= 100;
```

Before and After Triggers

- Differences between before and after triggers
  - “Before” and “after” the operation is done (`insert`, `update`, `delete`)
  - If we want to update the incoming values in an `update` statement, the “before trigger” should be used since the incoming values have not been written to the table yet

Before and After Triggers

- Typical usage scenarios for trigger settings
  - Modify input on the fly
    - before insert / update
    - for each row
  - Check complex rules
    - before insert / update / delete
    - for each row
  - Manage data redundancy
    - after insert / update / delete
    - for each row

Example: Auditing

- One good example of managing some data redundancy is keeping an audit trail
  - It won't do anything for people who steal data
    - (remember that `select` cannot fire a trigger – although with the big products you can trace all queries)
  - … but it may be useful for checking people who modify data that they aren't supposed to modify

Example: Auditing

- Trace the insertions and updates to employees in a company

```
create table company(
   id int primary key     not null,
   name           text    not null,
   age            int     not null,
   address        char(50),
   salary         real
);


create table audit(
   emp_id int not null,
   change_type char(1) not null,
   change_date text not null
);
```

Example: Auditing

- Trace the insertions and updates to employees in a company

```
create trigger audit_trigger
    after insert or update
    on company
    for each row
execute procedure auditlogfunc();


create or replace function auditlogfunc() returns trigger as
$example_table$
begin
    insert into audit(emp_id, change_type, change_date)
    values (new.id,
            case
                when tg_op = 'UPDATE' then 'U'
                when tg_op = 'INSERT' then 'I'
                else 'X'
            end,
            current_timestamp);
    return new;
end ;
$example_table$ language plpgsql;
```

Example: Auditing

- Trace the insertions and updates to employees in a company

```
insert into company (id, name, age, address, salary)
values (2, 'Mike', 35, 'Arizona', 30000.00);
```

company

	id	name	age	address	salary
1	2	Mike	35	Arizona	30000

audit

	emp_id	change_type	change_date
1	2	I	2022-04-25 18:37:35.515151+00

Data Storage Structures

File Organization

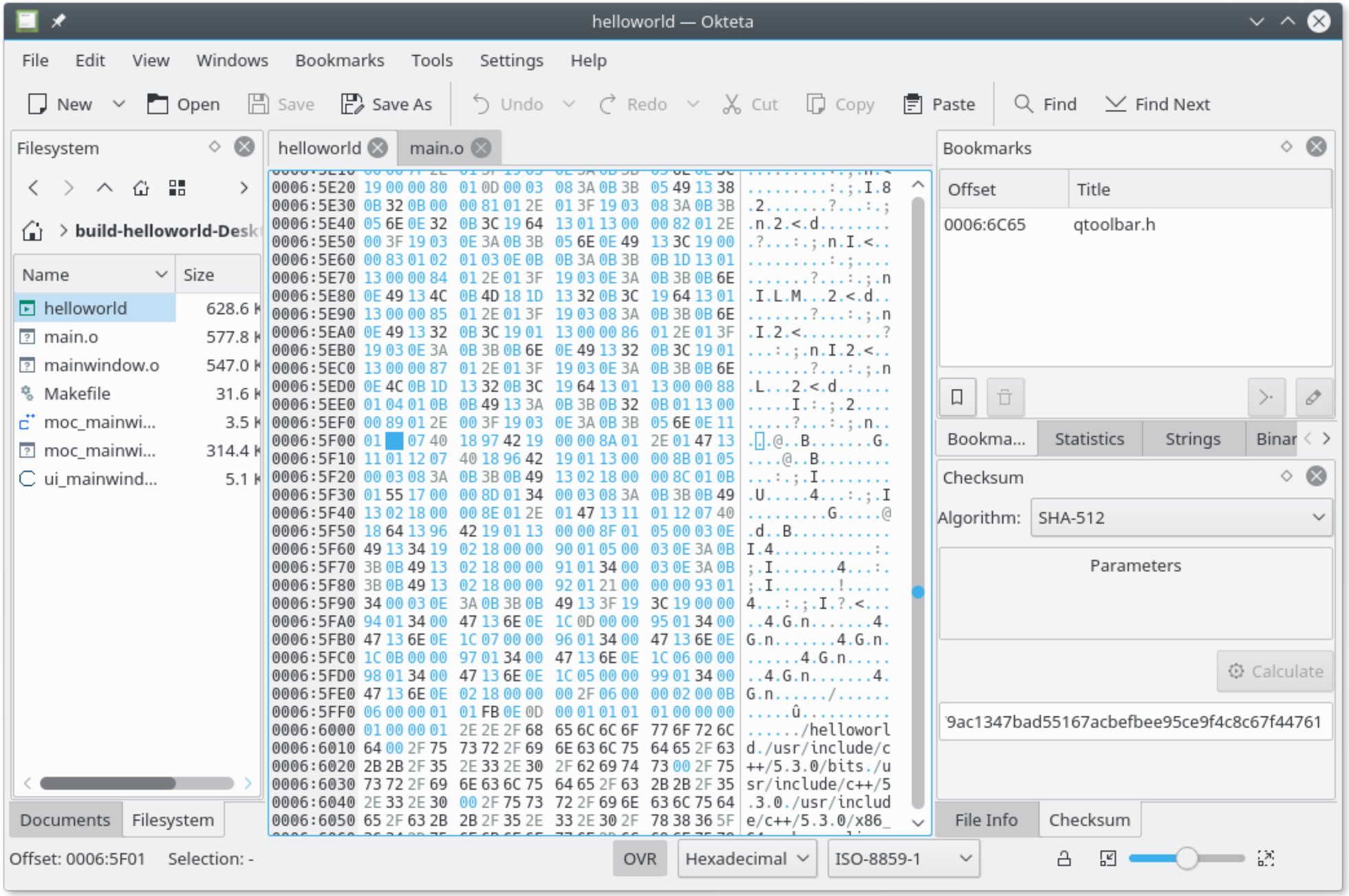
- The database is stored as a collection of files
  - Each file is a sequence of records
  - A record is a sequence of fields.
- One approach
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations

  * This case is easiest to implement; will consider variable length records later

- We assume that records are smaller than a disk block

File Organization

- Bitmap of a file



https://apps.kde.org/okteta/

File Organization

- Goals: Time and Space
  - Support CURD (Create, Read, Update and Delete) operations as fast as possible
  - Save storage space as much as possible

  - Maintain data integrity

Fixed-Length Records

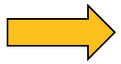
- Simple approach:
  - Store record i starting from byte $n*(i-1)$, where n is the size of each record
  - Record access is simple but records may cross blocks

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Fixed-Length Records

- Deletion of record i
  - Way #1: move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

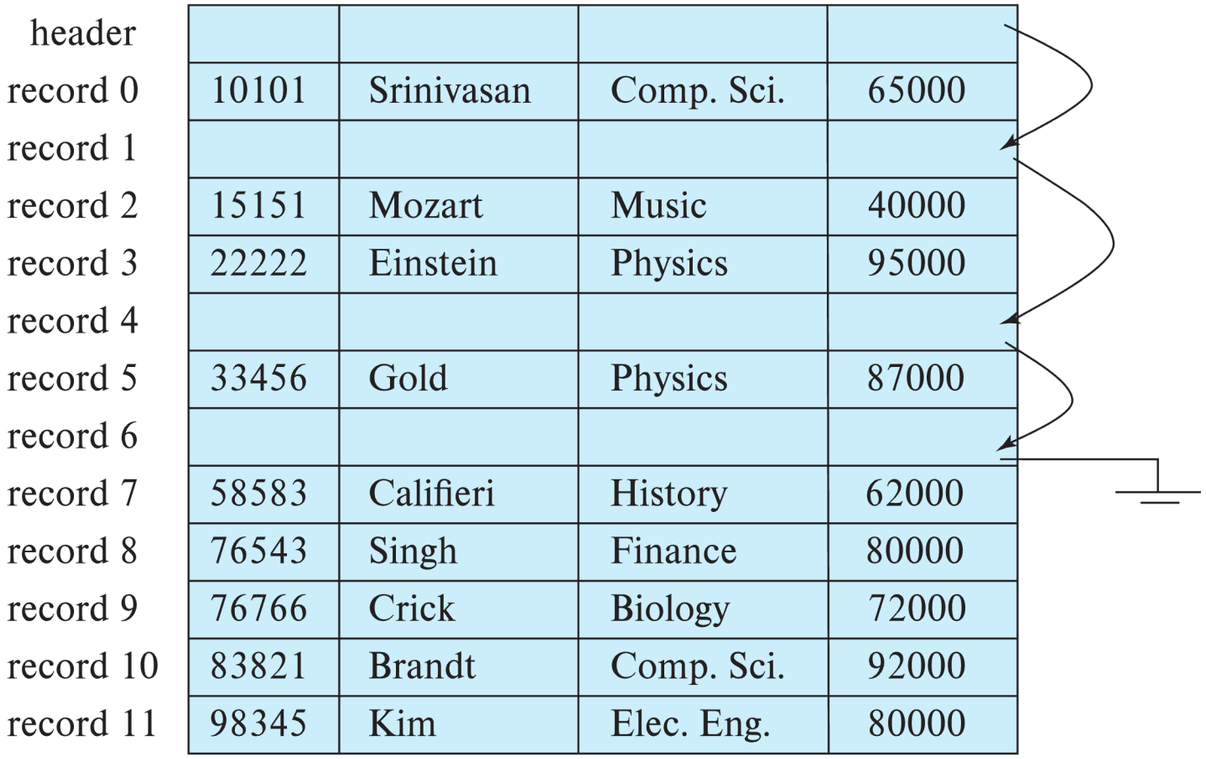
Fixed-Length Records

- Deletion of record i
  - Way #2: move record n to i
    - Record 3 is removed and replaced by record 11

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Fixed-Length Records

- Deletion of record i
  - Way #3: Do not move records, but link all free records on a *free list*



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

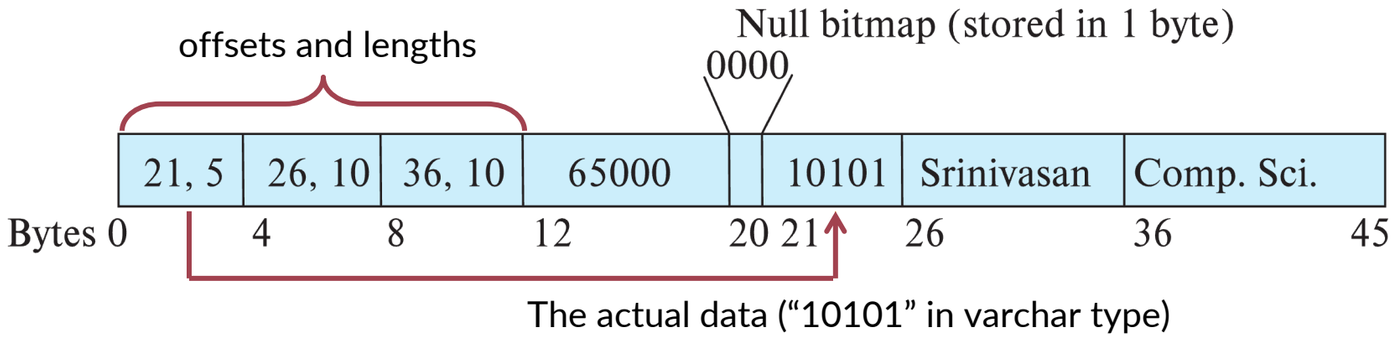
Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (`varchar`)

- Problem with variable-length records
  - How can we retrieve the data in an easy way without wasting too much space
    - `varchar(1000)`: do we really need to allocate 1000 bytes for this field, even if most of the actual data items only costs less than 10 bytes?

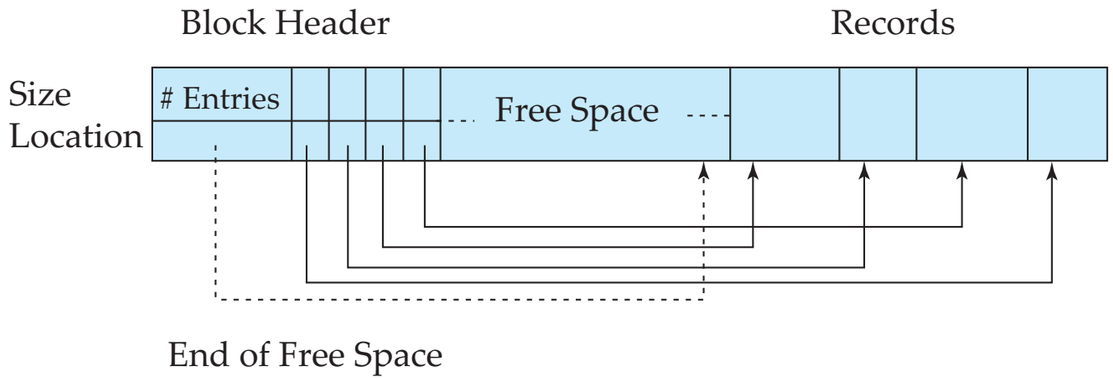
Variable-Length Records

- Attributes are stored in order
  - Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
  - Null values represented by null-value bitmap



Variable-Length Records

- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
  - Records can be moved around within a page to keep them contiguous (连续的) with no empty space between them; entry in the header must be updated.
- Pointers should NOT point directly to record — instead they should point to the entry for the record in header.



Storing Large Objects

- E.g., blob/clob types
- Records must be smaller than pages
- Alternatives:
  - Store as files in file systems
  - Store as files managed by database
  - Break into pieces and store in multiple tuples in separate relation
    - PostgreSQL TOAST

TOAST是**“The Oversized-Attribute Storage Technique”（超尺寸属性存储技术）**的缩写，主要用于存储一个大字段的值。

BLOB和CLOB都是大字段类型，BLOB是按二进制来存储的，而CLOB是可以直接存储文字的。

通常像图片、文件、音乐等信息就用BLOB字段来存储，先将文件转为二进制再存储进去。

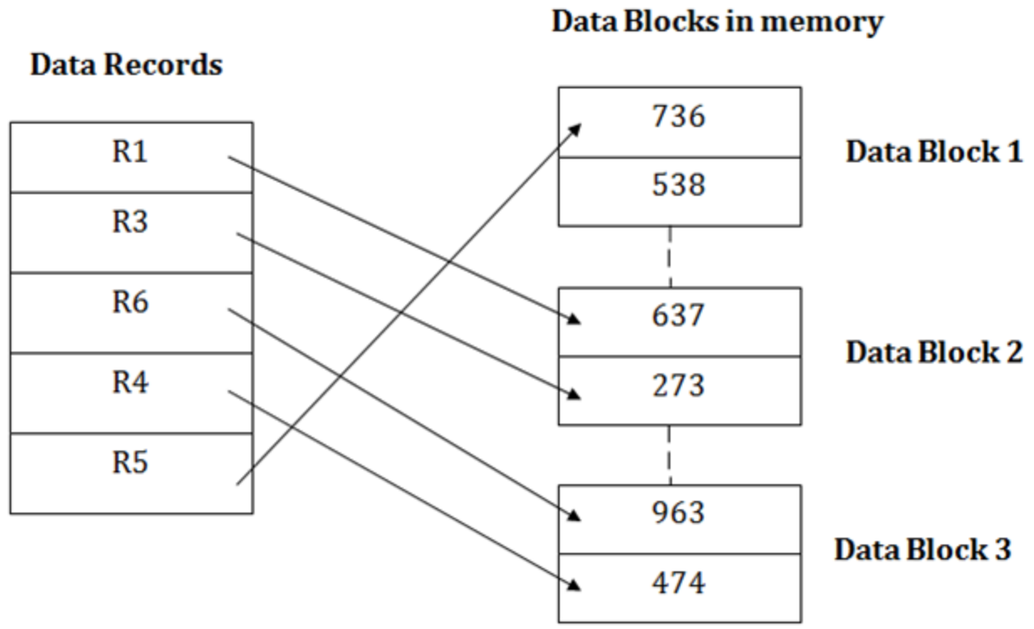
而像文章或者是较长的文字，就用CLOB存储，这样对以后的查询更新存储等操作都提供很大的方便。

Organization of Records in Files

- Heap – record can be placed anywhere in the file where there is space
- Sequential – store records in sequential order, based on the value of the search key of each record
- Multitable clustering file organization
  - Records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- B+-tree file organization
  - Ordered storage even with inserts/deletes
- Hashing – a hash function computed on search key; the result specifies in which block of the file the record should be placed

Heap File Organization

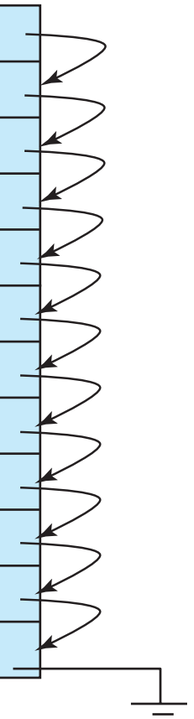
- Records can be placed anywhere in the file where there is free space
  - Records usually do not move once allocated
- Important to be able to efficiently find free space within file
- Free-space map
  - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)



Sequential File Organization

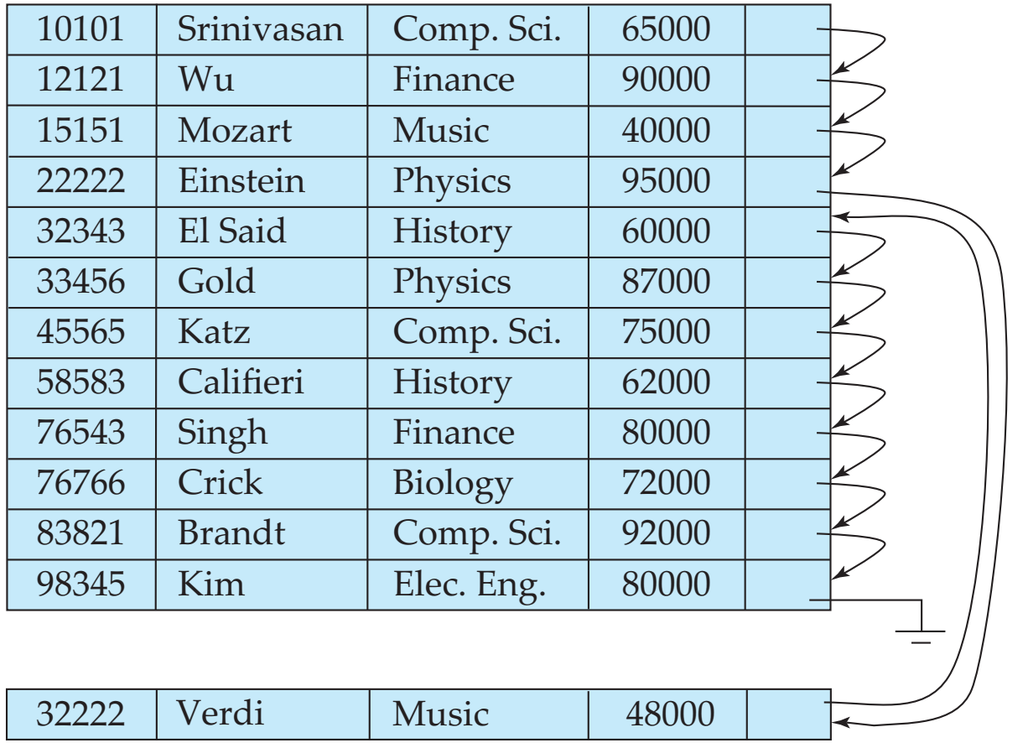
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



Sequential File Organization

- Deletion – Use pointer chains
- Insertion – Locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



Multitable Clustering File Organization

- Store several relations in one file using a multitable clustering file organization
  - Good for queries involving:
    - *department* ⋈ *instructor*
    - or, one single department and its instructors
  - Bad for queries involving only *department*
  - Results in variable size records
  - Can add pointer chains to link records of a particular relation

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

Multitable clustering of department and instructor

Comp. Sci.	Taylor	100000	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000

Partitioning

- Table partitioning: Records in a relation can be partitioned into smaller relations that are stored separately
  - E.g., `transaction` relation may be partitioned into `transaction_2018`, `transaction_2019`, etc.
- Queries written on transaction must access records in all partitions
  - Unless query has a selection such as year=2019, in which case only one partition in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., transaction partition for current year on SSD, for older years on magnetic disk