

# 数据库原理

---

## **Chp 3** **Introduction to SQL**

王朝坤  
清华大学软件学院  
2025年/春

# 本节课的内容

内容	核心知识点	对应章节
DBMS理论	关系数据模型（以及关系演算）、 <b>SQL语言</b>	CHP. 1、2、 <b>3</b>
DBMS设计	存贮、索引、查询、优化、事务、并发、恢复	CHP. 12、13、14、15、16、17、18、19
DBMS实现	大作业	小班辅导
DBMS应用	实体-联系图、关系范式、DDL、JDBC	CHP. 4、5、6、7

# Main Contents

---

- **Complex SQL**
  - Set Operations
  - Aggregation
  - **Nested Sub-queries**
- Modification of the Database

# With 子句

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as  
    (select max(budget)  
     from department)  
select department.dept_name  
from department, max_budget  
where department.budget = max_budget.value;
```

# Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as  
    (select dept_name, sum(salary)  
     from instructor  
     group by dept_name),  
dept_total_avg(value) as  
    (select avg(value)  
     from dept_total)  
select dept_name  
from dept_total, dept_total_avg  
where dept_total.value > dept_total_avg.value;
```

# Find ALL the supervisors (direct and indirect) of “Bob”

```
create table emp_super (person char(15), super char(15));
```

```
insert into emp_super values('Bob', 'Alice');  
insert into emp_super values('Mary', 'Susan');  
insert into emp_super values('Alice', 'David');  
insert into emp_super values('David', 'Mary');
```

```
select es1.person, es2.super as supsup  
from emp_super as es1, emp_super as es2  
where es1.person = 'Bob' and es1.super = es2.person;
```

person	supsup
Bob	David

# With Recursive Clause \*

Relation *emp\_super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

Since the SQL:1999 version the SQL standard supports a limited form of recursion, using the **with recursive** clause, where a view (or temporary view) is expressed in terms of itself. (见 课本 5.4.2)

■ Find ALL the supervisors (direct and indirect) of “Bob”


# Find ALL the supervisors (direct and indirect) of “Bob”

```
create table emp_super (person char(15), super char(15));
```

```
insert into emp_super values('Bob', 'Alice');  
insert into emp_super values('Mary', 'Susan');  
insert into emp_super values('Alice', 'David');  
insert into emp_super values('David', 'Mary');
```

s	p
Alice	Bob
David	Bob
Mary	Bob
Susan	Bob

```
select es1.person, es2.super as supsup  
from emp_super as es1, emp_super as es2  
where es1.person = 'Bob' and es1.super = es2.person;
```

```
with recursive TMP(S, P) as (  
  select SUPER, PERSON from EMP_SUPER where PERSON = 'Bob'  初始值  
  union  
  select SUPER, P from EMP_SUPER, TMP where  
    EMP_SUPER.PERSON = TMP.S  
) select * FROM TMP
```

请在PG上完成测试，并思考语句何时结束？



# 有限与无限

```
with recursive Integers(num) as (  
  values(1)  
  union  
  select num+1 from Integers  
) select * from integers
```

## Question:

何时结束？ 如果希望得到1到100个自然数，如何修改？

---

# Subqueries in the Select Clause

# Select子句中：结果为标量的子查询

- **Scalar subquery** is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,  
       (select count(*)  
        from instructor  
        where department.dept_name = instructor.dept_name)  
       as num_instructors  
from department
```

这个语句还有更简单的写法吗？  
有何区别

```
select dept_name, count(ID) from  
instructor group by dept_name;
```

- Runtime error if subquery returns more than one result tuple

# Main Contents

---

- Complex SQL
  - Set Operations
  - Aggregation
  - Nested Sub-queries
- **Modification of the Database**
- Join Expressions

# 删除-Deletion

- Delete all instructors

**delete from** *instructor*

- Delete all instructors from the Finance department

**delete from** *instructor*

**where** *dept\_name*= 'Finance';

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the **Watson** building.

**delete from** *instructor*

**where** *dept\_name* in (**select** *dept\_name*  
**from** *department*  
**where** *building* = 'Watson');

# Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

**delete from instructor  
where salary < (select avg (salary)  
from instructor);**

- **Problem:**  
**as we delete tuples from deposit, the average salary changes**
- **Solution used in SQL:**
  1. **First, compute avg (salary) and find all tuples to delete**
  2. **Next, delete all tuples found above (without recomputing avg or retesting the tuples)**

# 插入-Insertion

- Add a new tuple to *course*

**insert into course**

**values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);**

<i>course</i>
<u><i>course_id</i></u>
<i>title</i>
<i>dept_name</i>
<i>credits</i>

- or equivalently

**insert into course (course\_id, title, dept\_name, credits)**

**values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);**

- Add a new tuple to *student* with *tot\_creds* set to null

**insert into student**

**values ('3003', 'Green', 'Finance', null);**

<i>student</i>
<u><i>ID</i></u>
<i>name</i>
<i>dept_name</i>
<i>tot_cred</i>

# Insert Into **Select** ...

- Add all instructors to the *student* relation with tot\_creds set to 0

```
insert into student
  select ID, name, dept_name, 0
  from instructor
```

- The **select from where** statement is evaluated fully **before** any of its results are inserted into the relation.

**Otherwise**, queries like

```
insert into table1 select * from table1
```

would cause problem



# 更新-Updates

- Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others by a 5%
  - Write two **update** statements:  

```
update instructor
  set salary = salary * 1.03
  where salary > 100000;
update instructor
  set salary = salary * 1.05
  where salary <= 100000;
```
  - The order is important?
  - Can be done better using the **case** statement (next slide)

# Case Expression for Conditional Updates

- Same query as before but with case expression

*update instructor*

*set salary = case*

*when salary <= 100000*

*then salary \* 1.05*

*else salary \* 1.03*

*end*

# Updates with Scalar Subqueries

- Recompute and update `tot_creds` value for all students

```
update student S
  set tot_cred = (select sum(credits)
                  from takes, course
                 where takes.course_id = course.course_id and
                       S.ID= takes.ID and
                       takes.grade <> 'F' and
                       takes.grade is not null);
```

- Sets `tot_creds` to null for students who have not taken any course

id	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	7
12345	Shankar	Comp. Sci.	14
19991	Brandt	History	3
23121	Chavez	Finance	3
44553	Peltier	Physics	4
45678	Levy	Physics	7
54321	Williams	Comp. Sci.	8
55739	Sanchez	Music	3
70557	Snow	Physics	(Null)
76543	Brown	Comp. Sci.	7

# Updates with Scalar Subqueries(c1)

- Update statement would set the *tot\_cred* attribute value to 0 instead of null

**update** student S

**set** tot\_cred = (**select case**

**when** sum(credits) **is not null then** sum(credits)

**else** 0

**end**

**from** takes, course

**where** takes.course\_id = course.course\_id **and**

S.ID= takes.ID **and**

takes.grade <> 'F' **and**

takes.grade **is not null**);

# Updates 会失败吗

- Create table myStudent (id int primary key, name varchar(20));
- Insert into myStudent values( 1, 'Bob');
- Insert into myStudent values( 2, 'Alice');
- Update myStudent set id = 1 where name = 'Alice';
- ?

错误: 重复键违反唯一约束"mystudent\_pkey"  
DETAIL: 键值"(id)=(1)" 已经存在

# Content

---

- Complex SQL
  - Set Operations
  - Aggregation
  - Nested Sub-queries
- Modification of the Database
- **Join Expressions**

# Schema of a Bank Database

---

1. **Branch** (branch-name, branch-city, assets)
2. **Customer** (id, customer-name, customer-street, customer-city)
3. **Account** (account-number, branch-name, balance)
4. **Loan** (loan-number, branch-name, amount)
5. **Depositor** (id, account-number)
6. **Borrower** (id, loan-number)

# Schema of a University Database

1. *classroom*(building, room\_number, capacity)
2. *department*(dept\_name, building, budget)
3. *course*(course\_id, title, dept\_name, credits)
4. ***instructor***(**i\_ID**, name, dept\_name, salary)
5. *section*(course\_id, sec\_id, semester, year, building, room number, time\_slot\_id)
6. *teaches*(i\_ID, course\_id, sec\_id, semester, year)
7. *student*(s\_ID, name, dept\_name, tot\_cred)
8. *takes*(s\_ID, course\_id, sec\_id, semester, year, grade)
9. *advisor*(s\_ID, i\_ID)
10. *Time\_slot*(time\_slot\_id, day, start\_time, end\_time)
11. *prereq*(course\_id, prereq\_id)



# From 子句中的连接操作

- 主要考虑以下两个方面：
  1. *A join operation is a Cartesian product which requires that tuples in the two relations **match under some condition**.*
  2. *It also specifies **the attributes that are present in the result of the join***
- These additional operations are typically used as subquery expressions(子查询) in the **from** clause

# 连接类型与条件

- Join type (连接类型)
  - defines how tuples in each relation **that do not match any tuple in the other relation** based on the join condition are treated
- Join condition(连接条件)
  - defines which tuples in the two relations match, and what attributes are present in the result of the join

Join Types	
	inner join
left	} outer join
right	
full	

Join Conditions
<b>natural</b> on <predicate> using (A1, A2, ..., An)

# 内连接-自然连接的扩展

## ● natural Join 的等价形式

*loan* **natural** *inner* **join** *borrower*

*loan* *inner* **join** *borrower* **using**(*loan-number*)

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

## ● Inner join 的一般形式 inner theta join

*loan* *inner* **join** *borrower* **on**  
*loan.loan-number* = *borrower.loan-number*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>	<i>loan-number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

# 自然连接及其悬浮元组

已知：关系 *loan* 和 *borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
<b>L-260</b>	<b>Perryridge</b>	<b>1700</b>

<i>customer-name</i>	<i>loan-number</i>
Jones	L-170
Smith	L-230
<b>Hayes</b>	<b>L-155</b>

Dangling Tuple

● *loan* **natural join** *Borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

# 内连接-自然连接的扩展

## ● Inner join 的一般形式 inner theta join

```
select count(distinct i1.id)
from instructor as i1 inner join instructor as i2 on
i1.salary > i2.salary;
```

——拿非最低工资的教师人数

如果 *select count(distinct id) from instructor* 和

*select count(distinct salary) from instructor*

的返回值都是50，那么上面语句的返回结果是多少？

29 如果 **count(distinct salary)** 是20，上面语句最少是多少？

# 外连接

- An extension of the inner join operation that avoids **loss of information**.
- Computes the inner join **and then** adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

# 外连接操作举例

## ■ Relation *course*

	<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
悬浮元组 →	BIO-301	Genetics	Biology	4
	CS-190	Game Design	Comp. Sci.	4
	CS-315	Robotics	Comp. Sci.	3

## Relation *prereq*

	<i>course_id</i>	<i>prereq_id</i>
悬浮元组 →	BIO-301	BIO-101
	CS-190	CS-101
	CS-347	CS-101

Observe that

prereq information is missing for CS-315 and  
course information is missing for CS-437

# Result of *student* natural left outer join *takes*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

**Figure 4.1** The *student* relation.

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	<i>null</i>

**Figure 4.2** The *takes* relation.



# Result of *student* natural left outer join *takes*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
70557	Snow	Physics	0	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	<i>null</i>

**Figure 4.4** Result of *student* natural left outer join *takes*.

# Result of full outer join example

```
select *
from (select *
      from student
      where dept_name = 'Comp. Sci.')
natural full outer join
(select *
 from takes
 where semester = 'Spring' and year = 2017);
```

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	null

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	Zhang	Comp. Sci.	102	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
76543	Brown	Comp. Sci.	58	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
76653	<i>null</i>	<i>null</i>	<i>null</i>	ECE-181	1	Spring	2017	C

## Result of full outer join example

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	<i>null</i>

# 外连接

- left join 等价于 left outer join

```
select count(distinct i1.id)
from instructor as i1 left outer join instructor as i2
on i1.salary > i2.salary;
```

——拿非最低工资的教师人数

如果 *select count(distinct id) from instructor* 和

*select count(distinct salary) from instructor*

的返回值都是50，那么上面语句的返回结果是多少？

如果 *count(distinct salary)* 是20，上面语句最少是多少？

# NL2SQL(Nature Language to SQL)

## ScienceBenchmark: A Complex Real-World Benchmark for Evaluating Natural Language to SQL Systems

Yi Zhang

Zurich University of Applied Sciences  
Switzerland

Jan Deriu

Zurich University of Applied Sciences  
Switzerland

George

Katsogiannis-Meimarakis  
Athena Research Center  
Greece

Catherine Kosten

Zurich University of Applied Sciences  
Switzerland

Georgia Koutrika

Athena Research Center  
Greece

Kurt Stockinger

Zurich University of Applied Sciences  
Switzerland

### ABSTRACT

Natural Language to SQL systems (NL-to-SQL) have recently shown improved accuracy (exceeding 80%) for natural language to SQL query translation due to the emergence of transformer-based language models, and the popularity of the Spider benchmark. However, Spider mainly contains simple databases with few tables, columns, and entries, which do not reflect a realistic setting. Moreover, complex real-world databases with domain-specific content have little to no training data available in the form of NL/SQL-pairs leading to poor performance of existing NL-to-SQL systems.

In this paper, we introduce *ScienceBenchmark*, a new complex NL-to-SQL benchmark for three real-world, highly domain-specific databases. For this new benchmark, SQL experts and domain experts created high-quality NL/SQL-pairs for each domain. To garner more data, we extended the small amount of human-generated data with synthetic data generated using GPT-3. We show that our benchmark is highly challenging, as the top performing systems on

### 1 INTRODUCTION

Enabling users to query structured data using natural language is considered the key to data democratization. Natural Language Interfaces for Databases (or NL-to-SQL systems) emerged in the 1970s [1, 4]. Early systems relied on the database schema to build a SQL query from a natural language (NL) query (e.g., SODA [5], Precis [40]) or focused on understanding the structure of the natural language query to map it to SQL (e.g., ATHENA [37], NaLIR [24]). As early as 1995, the lack of benchmarks was apparent: “No standard benchmarks have yet been developed [...], any appraisal of the current state of the field must be impressionistic” [4]. This situation changed recently, when the first large-scale benchmarks, WikiSQL [53] and Spider [51], emerged. These allowed for training and evaluating *neural machine translation (NMT) approaches* (e.g., [44, 48, 53]). These approaches formulate the NL-to-SQL problem as a language translation problem, and train neural networks with large amounts of NL/SQL-pairs.

ScienceBenchmark: A Complex Real-World Benchmark for Evaluating Natural Language to SQL Systems. Yi Zhang etc.



# Window Function Expression

## Window Function Expression: Let the Self-join Enter

Radim Bača

VSb - Technical University of Ostrava  
Ostrava, Czech Republic  
radim.baca@vsb.cz

### ABSTRACT

Window function expressions (WFEs) became part of the SQL:2003 standard, and since then, they have often been implemented in database systems (DBS). They are especially essential to OLAP DBSs, and people use them daily. Even though WFEs are a heavily used part of the SQL language, the amount of research done on their optimization in the last two decades is not significant.

WFE does not extend the expressive power of the SQL language, but it makes writing SQL queries easier and more transparent. DBSs always compile SQL queries with WFE using a sequence of partition-sort-compute operators, which we call a linear strategy. Plans resulting from the linear strategy are robust and, in many cases, efficient.

This article introduces an alternative strategy using a self-join, which is not considered in the current DBSs. We call it the self-join strategy, and it is based on an SQL query transformation where the result query uses a self-join query plan to compute WFE. One output of this work is a tool that can automatically perform such SQL query transformations.

We created a microbenchmark showing that the self-join strategy is more effective than the linear strategy in many cases. We also performed a cost-based experiment to evaluate the query opti-

implement WFEs; however, fetching first N records with ties is not supported.

A growing interest in the WFE is also observed on websites like StackOverflow. To gain some insight into this trend, we analyzed SQL queries posted on StackOverflow<sup>1</sup> and counted the percentage of queries with a WFE for each year. Figure 1 shows the results, and we can observe that the percentage of such queries has steadily increased since 2008.

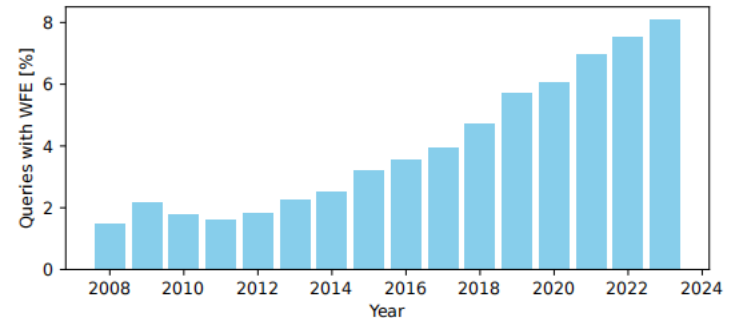


Figure 1: Percentage of SQL queries on StackOverflow that contain a WFE for each year.

# 数据库原理

---

## **Chp 12-13** **Physical Storage Systems** **and Data File Structures**

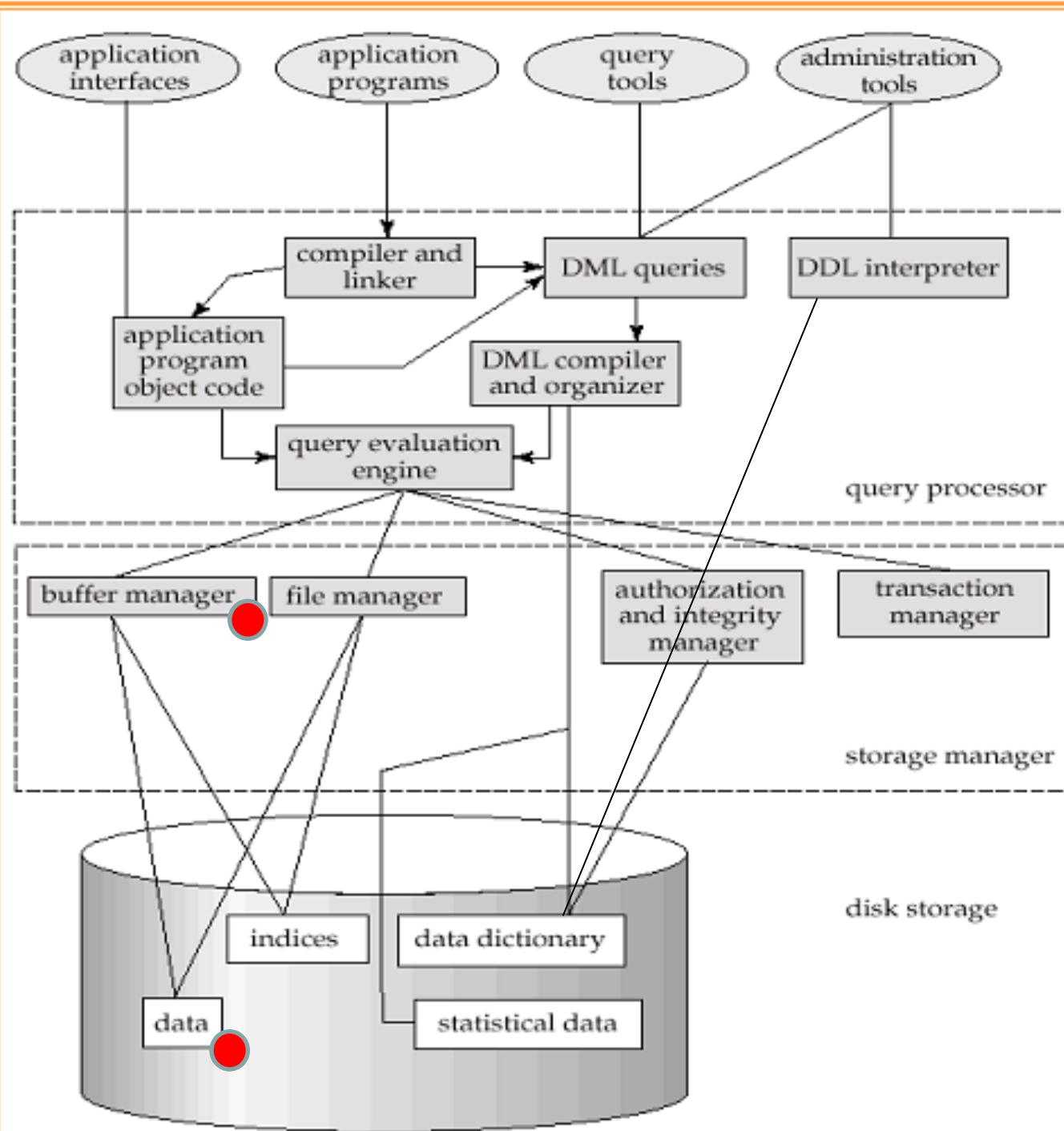
王朝坤  
清华大学软件学院  
2025年/春

# 本节课的内容

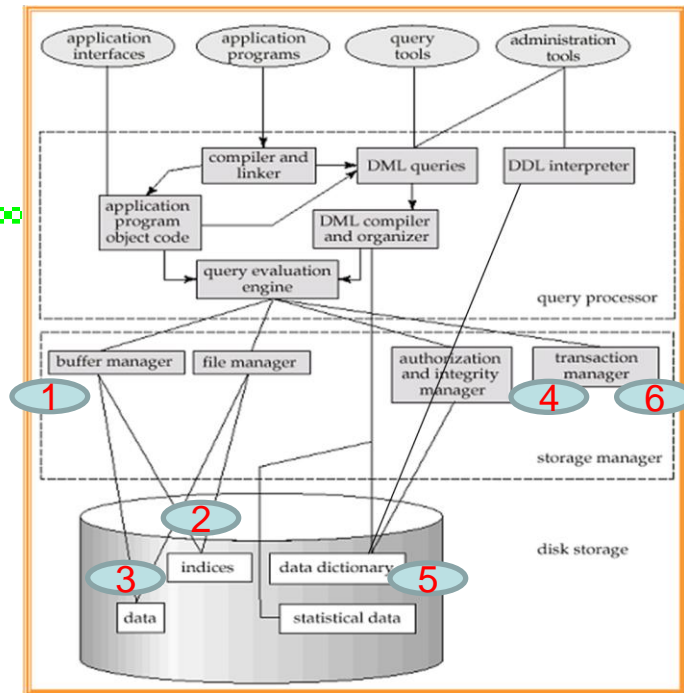
内容	核心知识点	对应章节
DBMS理论	关系数据模型（以及关系演算）、SQL语言	CHP. 1、2、3
DBMS设计	<b>存贮</b> 、索引、查询、优化、事务、并发、恢复	CHP. <b>12</b> 、 <b>13</b> 、14、15、16、17、18、19
DBMS实现	大作业	小班辅导
DBMS应用	实体-联系图、关系范式、DDL、JDBC	CHP. 4、5、6、7



# DBMS Architecture



# DBMS Architecture

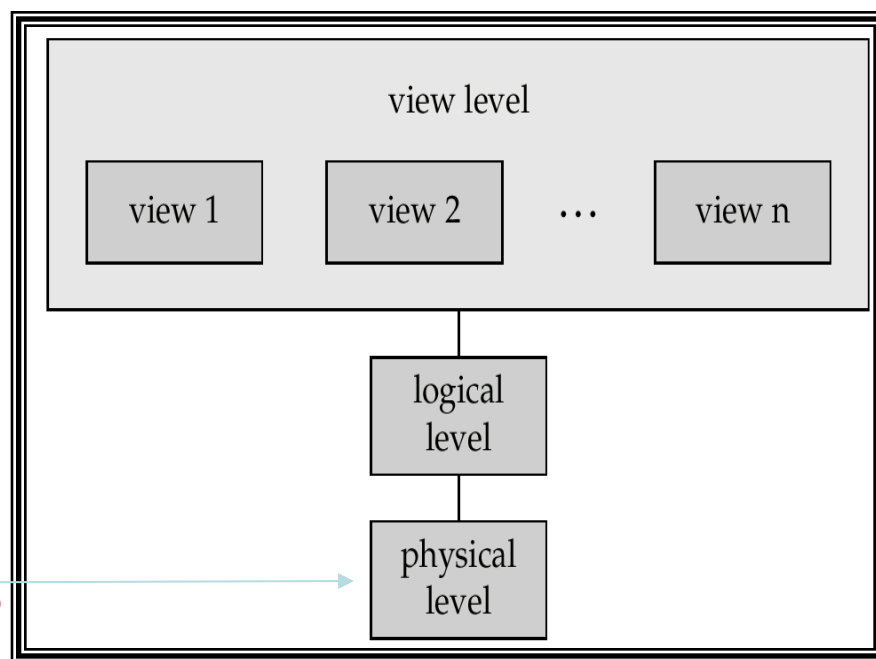
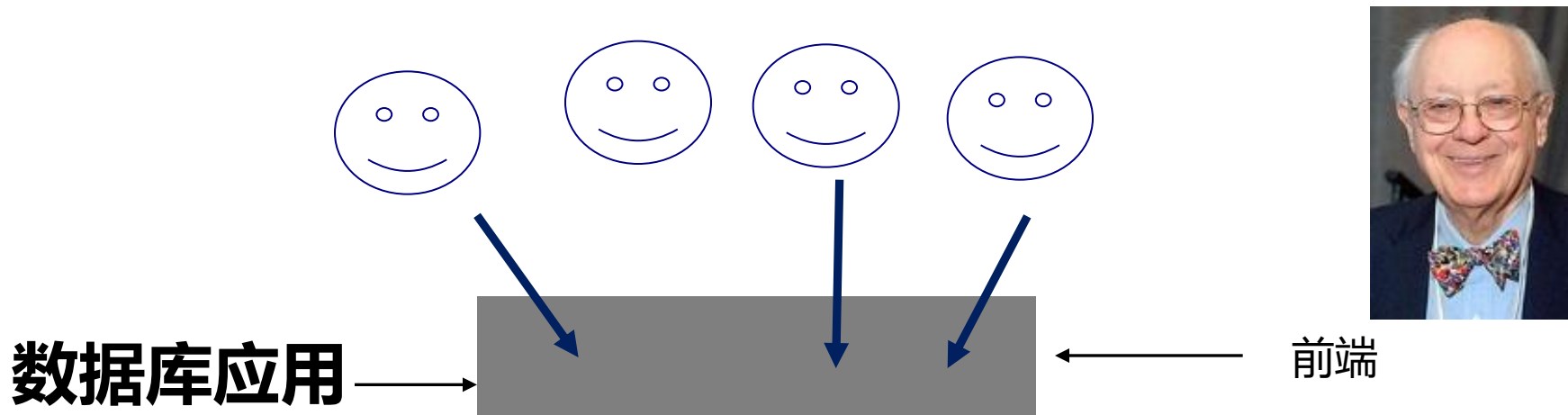


此电脑 > 新加卷 (E:) > TDB2024 > TDB > src > server > storage\_engine

排序 查看

名称	修改日期	类型	大小
1 buffer ✓	2024/3/17 15:14	文件夹	
2 index	2024/3/17 15:14	文件夹	
3 recorder ✓	2024/3/17 15:14	文件夹	
4 recover	2024/3/17 15:14	文件夹	
5 schema	2024/3/17 15:14	文件夹	
6 transaction	2024/3/17 15:14	文件夹	

# 三层抽象-图灵奖得主Bachman发明



应用模式

逻辑模式

物理模式

关系表如何  
43 存储到硬盘上?

# Contents

---

## **1.Overview of Physical Storage Media**

## 2.Magnetic Disks

## 3.Storage Access

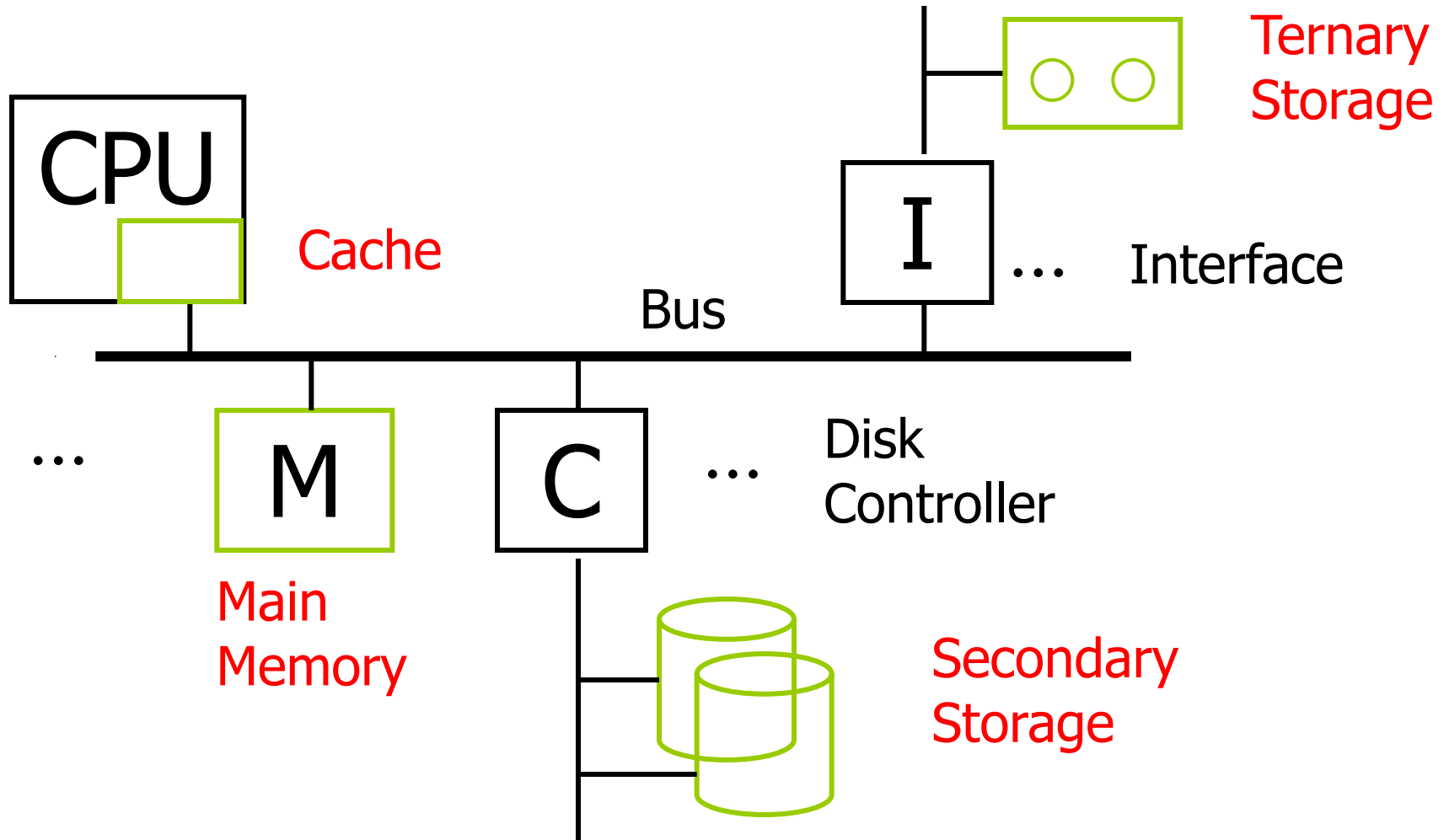
## 4.File Organization

## 5.Organization of Records in Files

## 6.Data-Dictionary Storage

## 7.Distributed File Systems

# 典型计算机结构



# 物理存储评价指标与分类

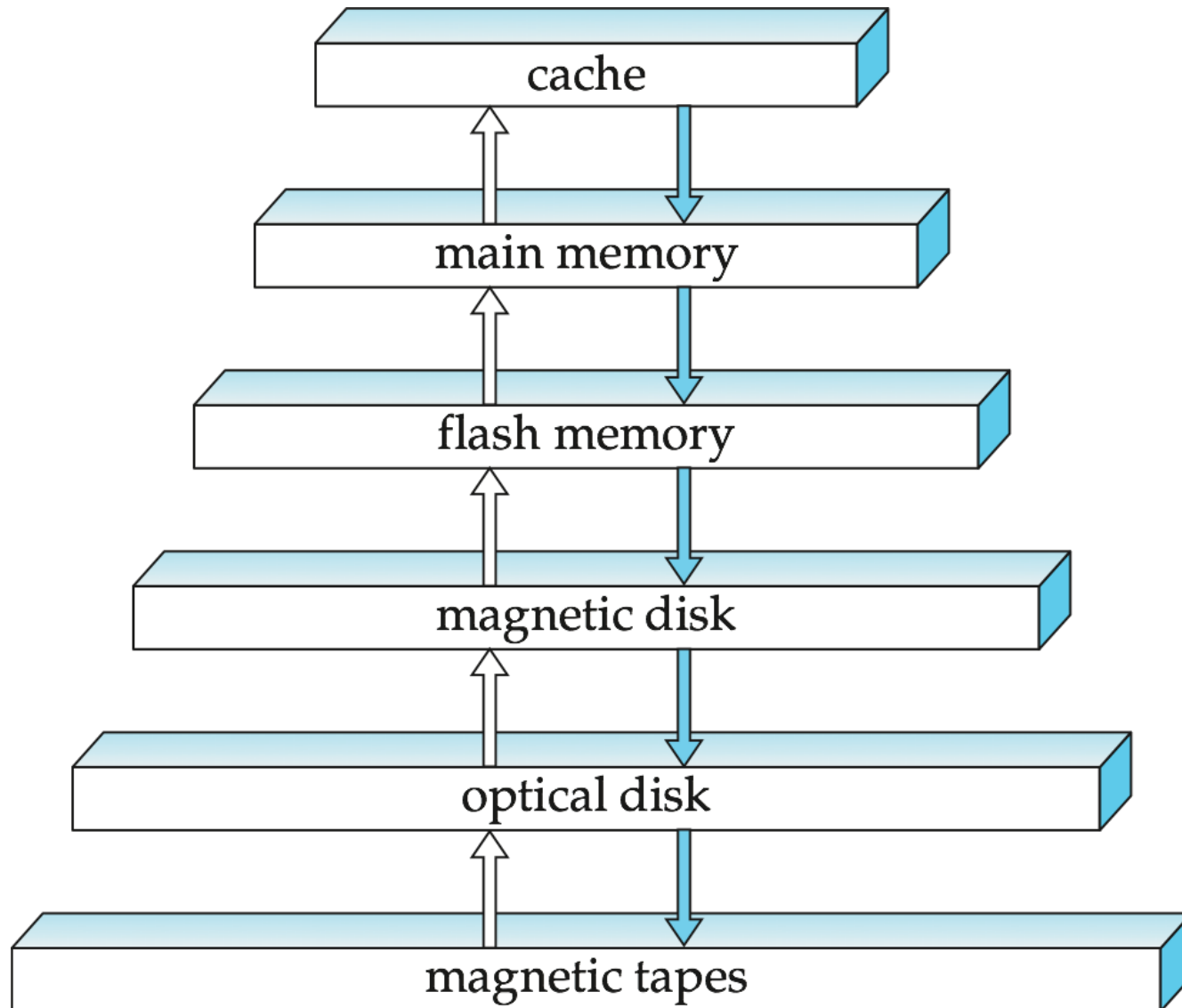
## □ Metrics

1. **Speed** with which data can be accessed
2. **Cost** per unit of data
3. **Reliability**
  - data loss on power failure or system crash
  - physical failure of the storage device

## □ Can differentiate storage into:

- **volatile storage:**
  - loses contents when power is switched off
- **non-volatile storage:**
  - Contents persist even when power is switched off.
  - Includes secondary and tertiary storage, as well as batter-backed up main-memory.

# 存储的层次结构



# 存储可分三个主要层次

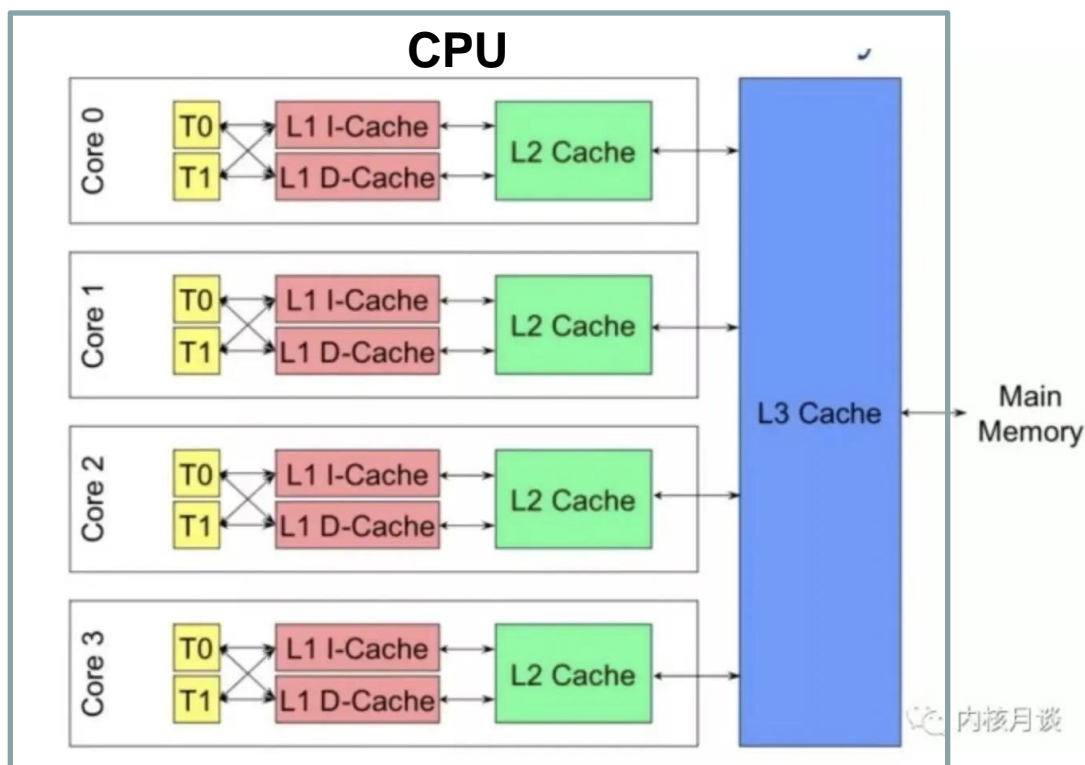
---

- **primary storage**
  - Fastest media but volatile (cache, main memory)
- **secondary storage**
  - next level in hierarchy, non-volatile, moderately fast access time(E.g. flashmemory, magnetic disks)
  - also called **on-line storage**
- **tertiary storage**
  - lowest level in hierarchy, non-volatile, slow access time(E.g. magnetic tape, optical storage)
  - also called **off-line storage**



# 片上缓存 - Cache

- fastest and most costly form of storage;
- volatile;
- managed by the computer system hardware.



在拥有L3Cache的CPU中，  
只有5%的数据从主存调用

# 主存 - Main Memory

---

- fast access (10s of nanoseconds, **ns** =  $10^{-9}$  seconds)
- generally too small to store the entire database
  - capacities of up to hundreds of Gigabytes widely used currently
  - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

# 闪存 - Flash

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
  - Can support only a limited number (10K – 1M) of write/erase cycles.
  - **Erasing of memory has to be done to an entire bank of memory**
- Reads are roughly as fast as main memory, But writes are slow (few microseconds), erase is slower
- **Widely used in embedded devices such as digital cameras, phones, and USB keys**

# 硬盘 – Hard disk

- Data is stored on spinning disk, and read/written magnetically
  - Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
- **direct-access**
  - possible to read data on disk in any order, unlike magnetic tape
- **Capacities range up to roughly 36 TB**
  - Much larger capacity and cost/byte than main memory/flash memory
  - Survives power failures and system crashes
  - disk failure can destroy data, but is rare



# 光盘 - Optical disc

---

- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Blu-ray disks: 27 GB to 54 GB
- Reads and writes are slower than with magnetic disk
- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

# 磁帶 – Magnetic Tape

---

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
  - **sequential-access** – much slower than disk
- tape can be removed from drive  $\Rightarrow$  storage costs much cheaper than disk, but drives are expensive
- **Oracle** StorageTek SL8500 released in 2013 is the world's most scalable tape library, accommodating growth **up to 1.2 EB** native **Oracle StorageTek SL8500 Modular Library System.**

# Contents

---

1. Overview of Physical Storage Media

**2. Magnetic Disks**

3. Storage Access

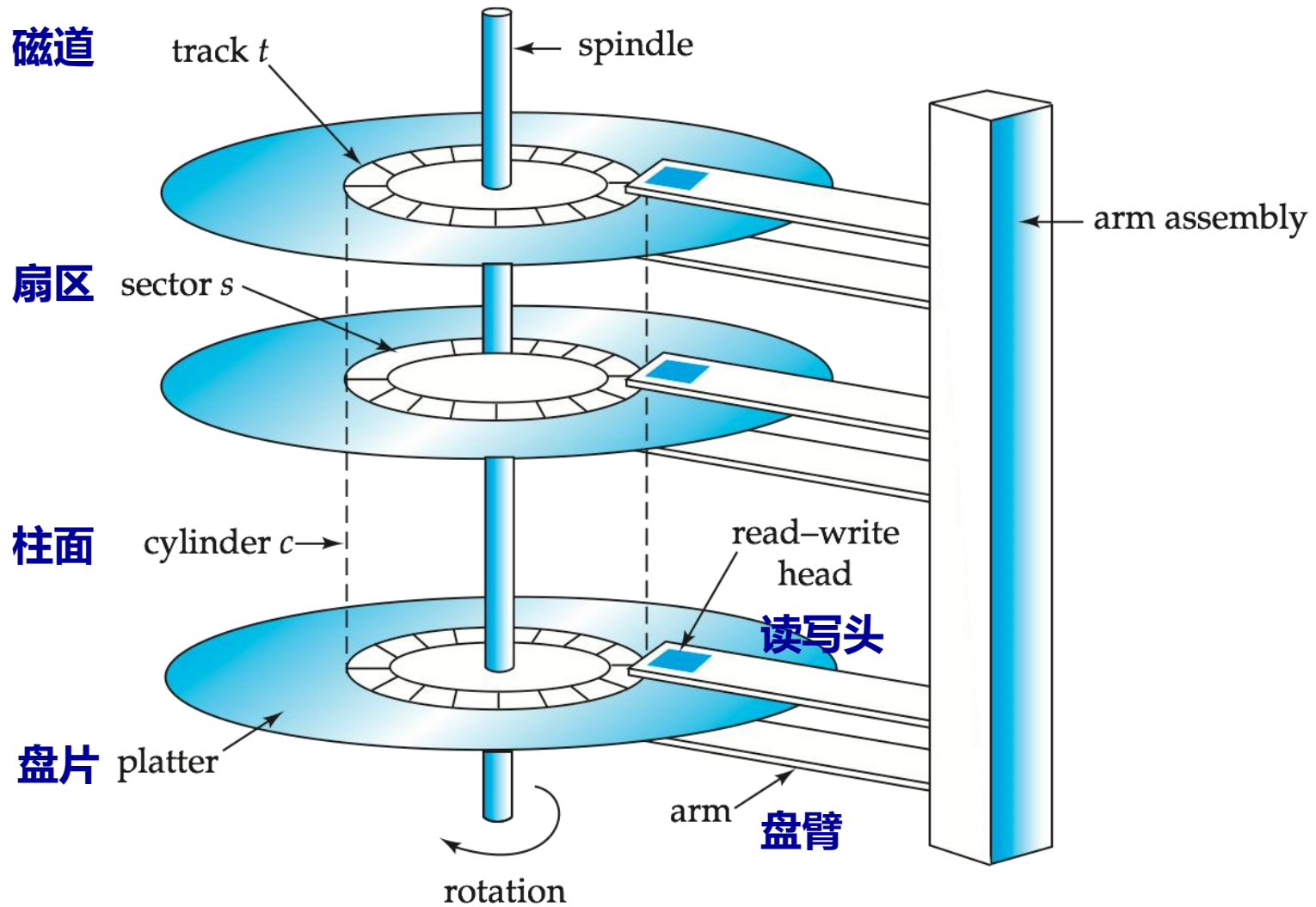
4. File Organization

5. Organization of Records in Files

6. Data-Dictionary Storage

7. Distributed File Systems

# 硬盘机构





# 访问准备时间

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is  $1/2$  the worst case seek time.
      - Would be  $1/3$  if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - Average latency is  $1/2$  of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)

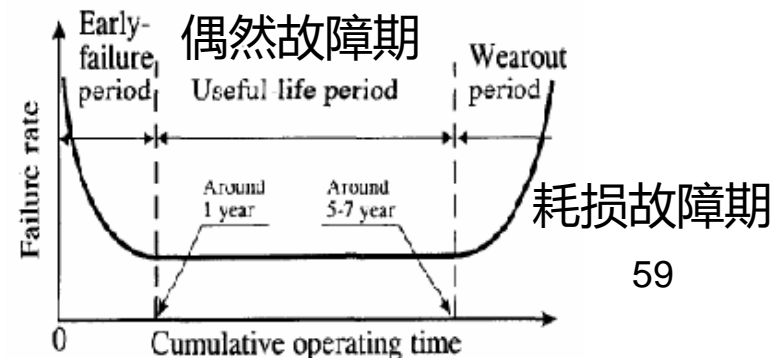
# 数据传输时间

- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 100 MB per second max rate, lower for inner tracks (角速度相同, 内道的线速度小, 外道的线速度大)
- Data-transfer-rate that **disk-controller** can handle is also important. (硬盘接口决定着硬盘与计算机之间的连接速度)
  - SATA: 150 MB/s, SATA-II 300 MB/s
  - Ultra 320 SCSI: 320 MB/s
  - Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s

# 平均失效间隔

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as disk ages

早期故障期



# 盘块与盘臂调度优化

- **Block** – a contiguous sequence of sectors from a single track
  - data is transferred between disk and main memory in blocks
  - sizes range from 512 bytes to several kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
    - Typical block sizes today range from 4K to 16K
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
  - **elevator algorithm 即 电梯算法**

# 文件碎片及其优化

- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
  - Store related information on the same or nearby cylinders.
  - Files may get **fragmented** over time
  - Some systems have utilities to defragment the file system, in order to speed up file access

# Contents

---

1. Overview of Physical Storage Media

2. Magnetic Disks

**3. Storage Access**

4. File Organization

5. Organization of Records in Files

6. Data-Dictionary Storage

7. Distributed File Systems

# 硬盘访问策略 - 缓存及其管理器

- A database file is partitioned into fixed-length storage units called **blocks**.
- Blocks are units of both storage allocation and data transfer.
- Database system seeks **to minimize the number of block transfers** between the disk and memory.
  - **Buffer** – portion of main memory available to store copies of disk blocks.
  - **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

# 缓存管理器

- Programs call on the **buffer manager** when they need a block from disk.
  1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
  2. If the block is not in the buffer, the buffer manager
    1. Allocates space in the buffer for the block
      1. Replacing some other block, if required, to make space for the new block.
      2. Replaced block written back to disk **only if** it was modified.
    2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



# 读写过程中需要加锁PIN/UNPIN

- While the block is being read, if a concurrent process evicts the block and replaces it with a different block, the reader that was reading the contents of the old block will see incorrect data;
- If the block was being written when it was evicted, the writer would end up damaging the contents of the replacement block
- Require reading processes to execute a **pin operation** before accessing data, and an **unpin operation** after completing access

# 策略1：LRU-最近最少使用

- Most operating systems replace the block **least recently used** (LRU strategy)
  - 更新/淘汰最近最少使用的块
- Idea behind LRU – use past pattern of block references as a predictor of future references

# LRU策略的局限

- LRU **can be a bad strategy** for certain access patterns involving repeated scans of data
  - For example: when computing the join of 2 relations  $r$  and  $s$  by a nested loops

```
for each tuple  $tr$  of  $r$  do /*  $r$ 是外层关系 */  
  for each tuple  $ts$  of  $s$  do /*  $s$ 是内层关系 */  
    if the tuples  $tr$  and  $ts$  match ...
```
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

# 立即销毁与最近最多使用

- **策略2: Toss-immediate strategy** – frees the space occupied by a block as soon as the final tuple of that block has been processed
  - **对于外层关系来说**，扫描过的块意味着本次操作后不会再用了
- **策略3: Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
  - **对于内层关系来说**，扫描过的块将来可能还会使用，但是接下来要扫描的块是内层关系中上一轮最后扫描（LRU）的块，这时如果一定要淘汰缓冲的，刚刚扫描过的块是合适的（MRU），因此MRU和LRU刚好相反

# 进一步的讨论

- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - 基于预测的策略
  - E.g., the data dictionary is frequently accessed.
  - Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support **forced output** of blocks for the purpose of recovery (more in Chapter 19)

# Contents

---

1. Overview of Physical Storage Media

2. Magnetic Disks

3. Storage Access

**4. File Organization**

5. Organization of Records in Files

6. Data-Dictionary Storage

7. Distributed File Systems

# 1) 定长记录方式

- 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
    - Modification: do not allow records to cross block boundaries

- Deletion of record  $i$ :  
alternatives:
  - move records  $i + 1, \dots, n$  to  $i, \dots, n - 1$
  - move record  $n$  to  $i$
  - do not move records, but link all free records on a *free list*

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

# 删除 “Record 3” 并整理

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



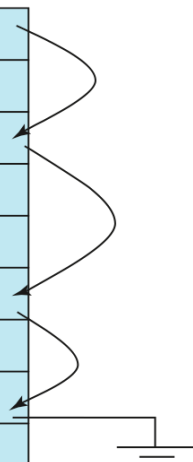
# 删除 “Record 3” 后，原位回填

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

# 用自由链表回收定长记录

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records)

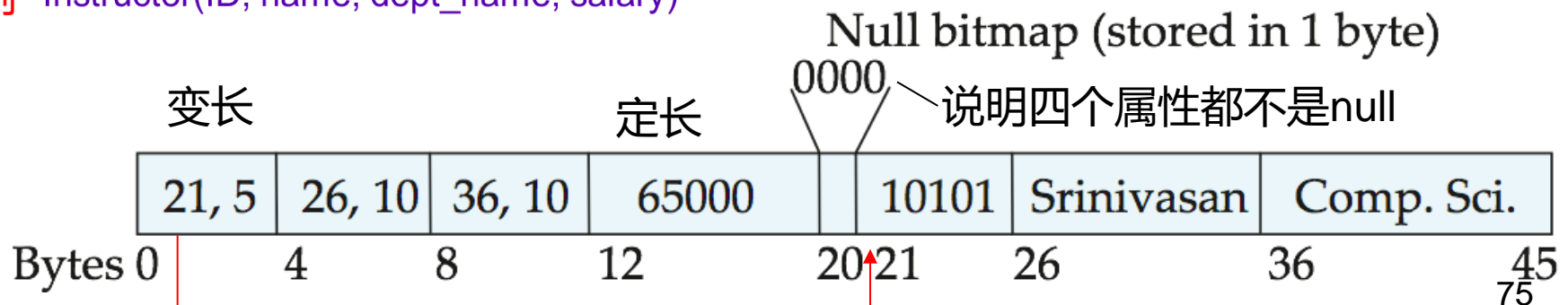
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology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



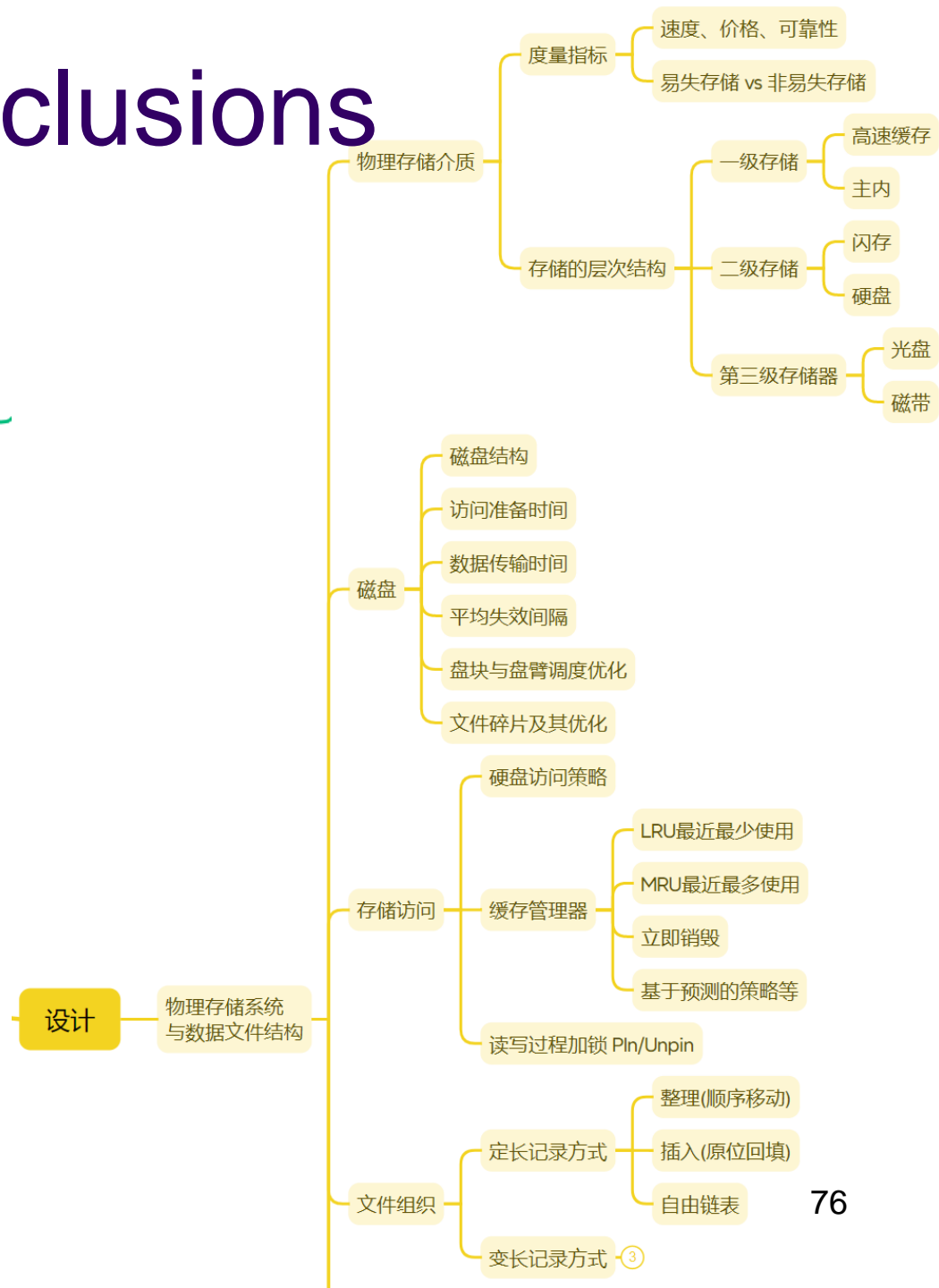
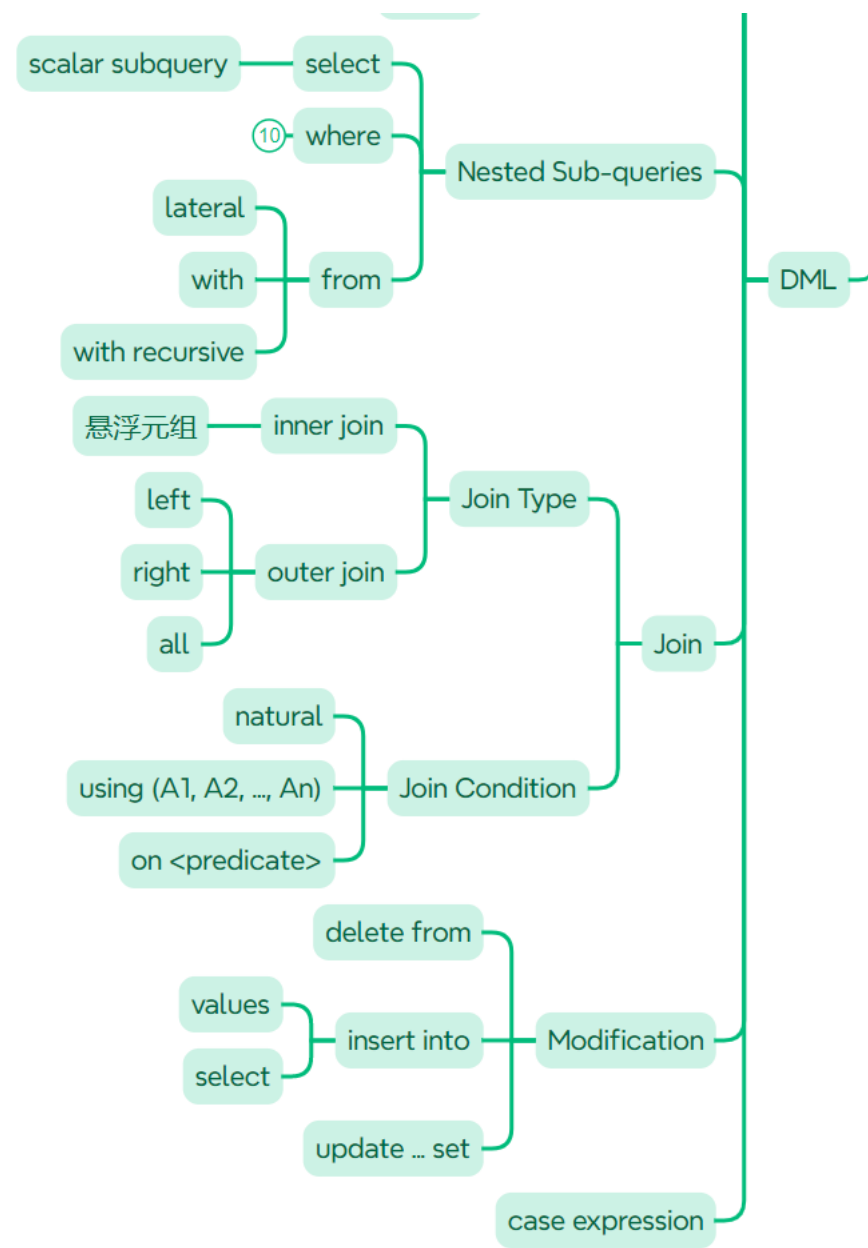
## 2) 变长记录方式

- Variable-length records arise in database systems in several ways:
  - ① Storage of multiple record types in a file.
  - ② Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - ③ Record types that allow repeating fields (used in some older data models).
- I. Attributes are stored in order
- II. Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- III. Null values represented by null-value bitmap

举例 Instructor(ID, name, dept\_name, salary)



# Conclusions



---

谢谢!