Intro

Characteristics of Databases

- data persistence(**数据持久性**)
- convenience in accessing data(**数据访问便利性**)
- data integrity（**数据完整性**）
- concurrency control for multiple user(**多用户并发控制**)
- failure recovery（**故障恢复**）
- security control（**安全控制**）

- Type & Variable
- Class & Object
- Schema & Instance

Three level: $physical \rightarrow logical \rightarrow view$

- Table & Column & Row
- Relation & Attribute & Tuple

Non-procedural query languages such as SQL

ACID

- **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Relation Model

Formally, given sets $D_1, D_2, \ldots D_n$ a relation r is a subset of $D_1 \times D_2 \times \ldots \times D_n$.Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$

Key

- K is a superkey(超键) of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
- Superkey K is a candidate key(候选键) if K is minimal (adequate)
- One of the candidate keys is selected to be the primary key（主键）
- Foreign key（外键）constraint: value in one relation must have already appeared in another

Operation

- select σ
- project $\prod$
- union $\cup$
- set difference $-$
- Cartesian product $\times$
- rename ρ

Additional operation:

- Set intersection $\cap$
- Natural join

Natural Join Example

- Relations r, s:

r

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

- r ⋈s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Natural Join and Theta Join

- Find the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Comp. Sci."}}(instructor \bowtie teaches \bowtie course))$
- Natural join is associative
  - $(instructor \bowtie teaches) \bowtie course$ is equivalent to $instructor \bowtie (teaches \bowtie course)$
- Natural join is commutative
  - $instructor \bowtie teaches$ is equivalent to $teaches \bowtie instructor$
- The **theta join** operation $r \bowtie_\theta s$ is defined as
  - $r \bowtie_\theta s = \sigma_\theta(r \times s)$

- Assignment
  $w \leftarrow \prod(\sigma(\dots))$

- Outer join

Outer Join – Example

- Join

  instructor ⋈ *teaches*

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201

- Left Outer Join

  instructor ⟕ *teaches*

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	*null*

Outer Join using Joins

- Outer join can be expressed using basic operations
  - r ⟕ s can be written as

    $(r \bowtie s) \cup (r - \Pi_R(r \bowtie s) \times \{(null, \ldots, null)\}$
  - r ⟖ s can be written as

    $(r \bowtie s) \cup \{(null, \ldots, null)\} \times (s - \Pi_S(r \bowtie s)$
  - r ⟗ s can be written as

    $(r \bowtie s) \cup$

    $(r - \Pi_R(r \bowtie s) \times \{(null, \ldots, null)\} \cup$

    $\{(null, \ldots, null)\} \times (s - \Pi_S(r \bowtie s)$

- Division Operator (整除)

Division Operator

- Given relations r(R) and s(S), such that $S \subset R$, $r \div s$ is the largest relation t(R-S) such that

  $(t \times s) \subseteq r$
- E.g. let $r(ID, course\_id) = \Pi_{ID,\ course\_id}(takes)$ and

  $s(course\_id) = \Pi_{course\_id}(\sigma_{dept\_name=\text{“Biology”}}(course))$

  then $r \div s$ gives us students who have taken all courses in the Biology department
- Can write $r \div s$ as

  $temp1 \leftarrow \Pi_{R-S}(r)$

  $temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$

  $result = temp1 - temp2$
  - The result to the right of the ← is assigned to the relation variable on the left of the ←.
  - May use variable in subsequent expressions.

Extended Relational-Algebra-Operations

Generalized Projection:

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

- E is any relational-algebra expression
- Each of $F_1, F_2, \ldots, F_n$ are are arithmetic expressions involving constants and attributes in the schema of E.
- Given relation *instructor(ID, name, dept_name,* salary) where salary is annual salary, get the same information but with monthly salary

$$\Pi_{ID,\ name,\ dept\_name,\ salary/12}\ (instructor)$$

Aggregate operation:

Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- **Aggregate operation** in relational algebra

$$G_1, G_2 \ldots, G_n\ \mathcal{G}\ F_1(A_1), F_2(A_2 \ldots, F_n(A_n)}(E)$$

  E is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

- Note: Some books/articles use γ instead of $\mathcal{G}$ (Calligraphic G)

Aggregate Operation – Example

- Find the average salary in each department

$$dept\_name\ \mathcal{G}\ \mathbf{avg}(salary)\ (instructor)$$

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

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

dept_name could be omitted, then it calculates the average of the whole.

To easily rename the attribute:

$$dept\_name\ \mathcal{G}\ \textbf{avg}(salary)\ \textbf{as}\ avg\_sal\ (instructor)$$

SQL instructions

- *char()*: fixed length, unused cells will be filled with space
- *varchar()*: variable length, unused cells will be abandoned

Except in MySQL:

```
select A.id
from A left natural join B on (A.id = B.id)
where B.id = NULL;
```

Intersect in MySQL:

```
select A.id
from A natural join B;
```

table

Create:

create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
(integrity-constraint$_1$),
...,
(integrity-constraint$_k$))

create table *instructor* (
ID **char**(5),
name **varchar**(20) **not null,**
dept_name **varchar**(20),
salary **numeric**(8,2),
primary key (*ID*),
foreign key (*dept_name*) **references** *department*)

on delete cascade no action|set null |restrict |set default
on update cascade no action|set null |restrict |set default

> Declare primary keys and DBMS will check uniqueness automatically
>
> Declare foreign keys and DBMS will check consistency automatically

```
create table section (
     course_id varchar (8),
     sec_id varchar (8),
     semester varchar (6),
     year numeric (4,0),
     building varchar (15),
     room_number varchar (7),
     time slot id varchar (4),
     primary key (course_id, sec_id, semester, year),
     constraint building_con unique (building),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

- `constraint building_con unique (building)` is renaming the unique constraint
- remove a not null constraint: `alter table stu modify name varchar(10)`
- remove a unique constraint: `alter table section drop index building_con`
- remove a primary key constraint: `alter table stu drop primary key`
- remove a foreign key constraint: `alter table stu drop foreign key dorm`

Insert:

`insert into table_name VALUES (.., .., ..);`

> It is important that we evaluate the **select** statement fully before we carry out any insertions. If we carry out some insertions even as the **select** statement is being evaluated, a request such as:
>
> ```
> insert into student
>     select *
>     from student
> ```
>
> might insert an infinite number of tuples, if the primary key constraint on *student* were absent. Without the primary key constraint, the request would insert the first tuple in *student* again, creating a second copy of the tuple. Since this second copy is part of *student* now, the **select** statement may find it, and a third copy would be inserted into *student*. The **select** statement may then find this third copy and insert a fourth copy, and so on, forever. Evaluating the **select** statement completely before performing insertions avoids such problems. Thus, the above **insert** statement would simply duplicate every tuple in the *student* relation, if the relation did not have a primary key constraint.

Drop:

Deletes the table and its contents(nothing left)

Delete:

Deletes all contents of table, but retains table

Alter:

```
alter table student add resume varchar(256);
```

```
alter table student drop resume;
```

Select:

```
select *
```

```
select all ...
```

```
select distinct ...
```

- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Natural Join (Cont.)

Beware of unrelated attributes with same name which get equated incorrectly

List the names of instructors along with the titles of courses that they teach

course(course_id,title, dept_name,credits)
teaches(ID, course_id,sec_id,semester, year)
instructor(ID,name, dept_name,salary)

- Incorrect version (makes *course.dept_name* = *instructor.dept_name*)
  - **select** *name*, *title*
    from *instructor* **natural join** *teaches* **natural join** *course*;
- Correct version
  - **select** *name*, *title*
    from *instructor* **natural join** *teaches*, *course*
    where *teaches.course_id* = *course.course_id*;
- Another correct version
  - **select** *name*, *title*
    from (*instructor* **natural join** *teaches*) **join** *course* **using**(*course_id*);

> The names may be the same but the meanings are different
>
> Consider: first **from**, then **where**, and then **select**
>
> Natural join: Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once. Notice also the order in which the attributes are listed: first the attributes **common** to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

string

- percent (%). The % character matches any **substring**.

- underscore (_). The _ character matches any **character**

  Find the names of all instructors whose name includes the substring “dar”.

  select *name*
  from *instructor*
  where *name* **like** '%dar%'

  Match the string “100 %”

  like '100 \%' **escape** '\'

Support:

- concatenation (using “||”)
- converting from upper to lower case (and vice versa)
- finding string length, extracting substrings, etc.

order

- List in alphabetic order the names of all instructors

```
select distinct name
from    instructor
order by name desc (asc)
```

- Can sort on multiple attributes

```
order by  dept_name, name
```

- List names of instructors whose salary is among top 3.

```
select  name
from    instructor
order by salary desc
limit 3;    //  limit 0,3
```

group

> **where** goes ahead of **group by**
>
> **group by** goes ahead of **having**

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg(salary)
from instructor
group by dept_name
having avg (salary) > 42000;
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

```
select dept_name, count (*) as cnt
from instructor
where  salary >=100000
group by dept_name
having  count (*) > 10
order by cnt;
```

find the student information who has at least one roommate

```
select *
from stu
where room in(select room
from stu
group by room
having count(sid)>1);
```

some & all

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

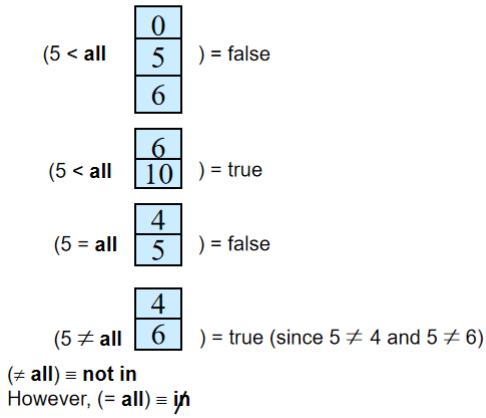
(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≡ **not in**

- F <comp> **all** $r \Leftrightarrow \forall\, t \in r$ (F <comp> t)




```
select name
from instructor
where salary > some (select salary
              from instructor
              where dept_name = 'Biology');
```

case & set

```
update instructor
set salary = case
        when salary <= 100000 then salary * 1.05
        else salary * 1.03
        end
```

variable

Globally:

```
set @a=1;
select @a; # correct
select var; # error
select 1+1;
select salary into @sa
from employee
where eid = 1001;
```

In a procedure:

```
delimiter $$
create procedure p1()
begin
    declare var varchar(10); # NULL in default
    select var, @a; # correct
end;
$$
delimiter ; # switch the ending symbol
call p1();
drop procedure p1;
```

divide & exists & unique

For a table `drivers`:

```
+-------------+------------+
| driver_name | vehicle_id |
+-------------+------------+
| yan         |          1 |
| wei         |          1 |
| li          |          1 |
| wei         |          2 |
| wei          |          3 |
| li          |        2 |
+-------------+------------+
```

For a table `vehicles`:

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
+----+
```

To find the drivers who can drive all kinds of vehicles:

```
SELECT DISTINCT D1.driver_name
    FROM drivers AS D1
WHERE NOT EXISTS
            (SELECT * FROM vehicles AS V1
                WHERE NOT EXISTS
                (
                    SELECT * FROM drivers AS D2
                    WHERE D1.driver_name = D2.driver_name
                        AND D2.vehicle_id = V1.id
                )
            );
```

> See from the outmost nest:

> for a driver in $D1$, create a set containing the id of vehicles that he/she can't drive. If the set is empty, then the driver is the one.
>
> See from the inner nest:
>
> for a vehicle in $V1$, create a set containing the driver information if the driver can man it. If the set is empty, then the driver can't man the vehicle (only one check is significant here)

- `exists` checks for empty set
- `unique` checks for duplicate tuples (times≤ 1)

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id
              and R.year = 2009)
     and exists (select R.course_id
              from section as R
              where T.course_id= R.course_id
              and R.year = 2009);
```

is the same as

```
select T.course_id
from course as T
where unique (select R.course_id, R.year, y.semester
              from section as R
              where T.course_id= R.course_id );
```

with

> The **with** clause provides a way of defining a temporary relation whose definition is available only to the (immediately following) query in which the **with** clause occurs.

find those departments with the maximum budget:

```
with max_budget(value) as # name a relation temporarily
    (select max(budget)
    from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

assertion & triggers

- describe the constraint between two tables/relations
- hardly no DBMS has achieved aseertion for its complexity

```
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'))
```

Trigger: **ECA**

- Event
    - before / after update / delete / insert
- Condition
- Action

> Be careful to use triggers

```
create trigger account_trigger after update on account(balance)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.balance - orow.balance > =200000 or orow.balance -nrow.balance >=50000
begin
    insert into account_log values (nrow.account-number, nrow.balance-
orow.balance , current_time() )
end
```

```
create trigger credits_earned after update on takes(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;
```

Statement level triggers:

```
create trigger score_trigger after update on register(score)
referencing new table as ntable
for each statement
 when some( select average(score)
         from ntable
         group by cno) < 60
begin
    rollback
end
```

```
create trigger yxl before update on person(spouse)
referencing new table as ntbl
for each statement
when exists (select *
        from ntbl as A, ntbl as B
        where A.spouse = B.id and A.gender = B.gender)
begin
    rollback
end
```

> Notice the `before`

enable & disable:

```
disable trigger trigger_name;
alter trigger trigger_name disable;
enable trigger trigger_name;
alter trigger trigger_name enable;
```

view

- a kind of projection
- no table is generalized for a view(except for a materialized view, which requires maintaining in time)
- to insert by a view, all the attributes of the original table should be filled
- Most SQL implementations allow updates only on simple views
    - The **from** clause has only one database relation.
    - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
    - Any attribute not listed in the **select** clause can be set to null
    - The query does not have a **group** by or **having** clause.
    - No **where** clause
- **materialize views**
    - the view is kept up-to-date
    - the result is stored

Recursive views, a base query and a **recursive query**:

```
with recursive c_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
    where course_id = "CS-347"
    union
    select prereq.prereq_id, c_prereq.course_id
    from prereq, c_prereq
    where prereq.course_id = c_prereq.prereq_id)
select *
from c_prereq
```

> The result should be monotonic

type & domain

- both are like `typedef` in C
- Domains can have constraints, such as **not null**, specified on them, and can have default values defined for variables of the domain type, whereas user-defined types cannot have constraints or default values specified on them.
- Domains are not strongly typed. As a result, values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.

index

- speed up indexing
- require management
- additional space (B+ tree)
- a kind of physical schemas

transaction

A **transaction** consists of a sequence of query and/or update statements. The SQLstandard specifies that a transaction begins implicitly when an SQL statement is executed. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the fifirst statement of the transaction was executed.

- Transaction example :
- **SET AUTOCOMMIT=0**

UPDATE account **SET** balance=balance-100 **WHERE** ano='1001';
UPDATE account **SET**balance=balance+100 **WHERE** ano='1002';
COMMIT;

UPDATE account **SET** balance=balance -200 **WHERE** ano='1003';
UPDATE account **SET** balance=balance+200 **WHERE** ano='1004';
COMMIT;

UPDATE account **SET** balance=balance+balance*2.5%;
COMMIT;

> It means that the two update instruction should both be executed or none be executed
>
> `AUTOCOMMINT=1` then the system commits after each single instruction is executed
>
> Combine relevant instructions as one atom and make use of atomicity

authorization

privilege list:

- **Select** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data
- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations
- **all privileges**

user list:

- a user-id
- **public**, which allows all valid users the privilege granted
- role

transfer:

```
grant select on department to Amit with grant option;
revoke select on department from Amit, Satoshi cascade;
revoke select on department from Amit, Satoshi restrict;
```

grant

```
grant update (budget) on department to U1,U2
```

- 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)
- Concerning foreign keys, the privilege of reference needs to be granted, like

  ```
  grant reference (dept_name) on department to Mariano;
  ```

revoke

```
revoke select on branch from U1, U2, U3
```

- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation
- All privileges that depend on the privilege being revoked are also revoked

role

```
create role instructor;
grant instructor to Amit;
```

- Privileges can be granted to roles
- Roles can be granted to users, as well as to other roles

rank*

Those sharing the same GPA would get the same rank:

```
select ID, rank() over(order by (GPA) desc) as s_rank
from student_grade
order by s_rank;

select ID, (1+(select count(*)
          from student_grade B
          where B.GPA > A.GPA)) as s_rank
from student_grade A
order by s_rank;

select ID, dept_name, rank() over(partition by dept_name order by (GPA) desc) as
dept_rank
from dept_grade
order by dept_name, dept_rank;
```

> When ranking (possibly with partitioning) occurs along with a **group by** clause, the **group by** clause is applied first, and partitioning and ranking are done on the results of the group by.

window*

Automatically calculate every period:

```
select year, avg(num_credits) over (order by year rows 3 preceding) as
avg_total_credits
from tot_credits;
# last two and current

select year, avg(num_credits) over (order by year rows unbounded preceding) as
avg_total_credits
from tot_credits;
# all the former and current

select year, avg(num_credits) over (order by year rows between 3 preceding and 2
following) as avg_total_credits
from tot_credits;

select year, avg(num_credits) over (order by year range between year-4 and year)
as avg_total_credits
from tot_credits;
# totally 5 years

select year, avg(num_credits) over (partition by dept_name
                        order by year rows between 3 preceding and current row)
as avg_total_credits
from tot_credits_dept;
```

Advanced SQL

- API (application-program interface) for a program to interact with a database server
- ODBC (Open Database Connectivity) works with C, C++, C#

- JDBC (Java Database Connectivity) works with Java
- Embedded SQL in C
- SQLJ - embedded SQL in Java
- JPA(Java Persistence API) - OR mapping of Java

Prepared Statement

```
PreparedStatement pStmt = conn.prepareStatement(
                          "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");     pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");   pStmt.setInt(4, 125000);
pStmt.executeUpdate();
```

- Safer
- Always use prepared statements when taking an input from the user and adding it to a query

Metadata

- finding all the relations in the database
- finding the names and types of columns of a query result or a relation in the database.

Embedded SQL

- the Java embedding uses: `# SQL {...} ;`
- the C embedding uses: `EXEC SQL  <embedded SQL statement > ;`
    - Mark the start point and end point of Embedded SQL
    - Communication between database and programming language: **SQLCA**(communication) and **SQLDA**(data)
    - Address the mismatching issue between SQL and host lanugage
        - Handle result (set) with cursor
        - Mapping of basic data types

```
main( )
{  EXEC SQL INCLUDE SQLCA; //声明段开始
   EXEC SQL BEGIN DECLARE SECTION;
      char account_no [11];    //host variables(宿主变量)声明
      char branch_name [16];
      int  balance;
   EXEC SQL END DECLARE SECTION;//声明段结束
   EXEC SQL CONNECT  TO  bank_db  USER Adam Using Eve;
   scanf (“%s  %s  %d”, account_no, branch_name, &balance);
   EXEC SQL insert into account
               values (:account_no, :branch_name, :balance);
   If (SQLCA.sqlcode ! = 0)    printf ( “Error!\n”);
   else       printf (“Success!\n”);
}
```

```
// mind the :
```

Signal `NULL`:

```
main( )
{  EXEC SQL INCLUDE SQLCA; //声明段开始
   EXEC SQL BEGIN DECLARE SECTION;
      char account_no [11];    //host variables(宿主变量)声明
      int  balance;
      int mask;    //mask指示变量
   EXEC SQL END DECLARE SECTION;//声明段结束
   EXEC SQL CONNECT  TO  bank_db  USER Adam Using Eve;
   scanf (“%s  %d”, account_no, balance);
   EXEC SQL  select balance into :balance:mask          // 指示变量=0  正常
                       from account          //指示变量<0  NULL
                       where account_number = :account_no; //指示变量 >0  截断
   If (SQLCA sqlcode ! = 0)    printf ( “Error!\n”);
   else       if  (mask ==0)  printf (“balance= %d \n”, balance);
                   else printf(“balance is NULL!\n”)
}
```

Embedded SQL with cursor(select multiple records):

```
 main( )
{  EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      char customer_name[21];
      char account_no [11];
      int  balance;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL CONNECT  TO  bank_db  USER Adam Using Eve;

   // create a cursor
   EXEC SQL DECLARE account_cursor CURSOR for
              select account_number, balance
              from depositor natural join account
              where depositor.customer_name = : customer_name;
    scanf (“%s”, customer_name);
    EXEC SQL open account_cursor;   // begin execution
    for (; ;)
    {   EXEC SQL fetch account_cursor into :account_no, :balance;
     if (SQLCA.sqlcode!=0)  // the table is end
         break;
     printf( “%s  %d \ n”, account_no, balance);
    }
    EXEC SQL close account_cursor;
}
```

Update the content with the cursor:

```
    scanf (“%s”, customer_name);
    EXEC SQL open account_cursor;
    for (; ;)
    {    EXEC SQL fetch account_cursor into :account_no, :balance;
     if (SQLCA.sqlcode!=0)
         break;
     if (balance <1000)
         EXEC SQL update account set balance=balance*1.05
         where current of account_cursor;   // update with a cursor
     else
         EXEC SQL update account set balance=balance*1.06
         where current of account_cursor;

    }
    EXEC SQL close account_cursor;
```

Dynamic SQL:

```
char *  sqlprog = “update account
    set balance = balance * 1.05
    where account_number = ?”
    EXEC SQL prepare dynprog  from :sqlprog;
char account [10] = “A-101”;
EXEC SQL execute dynprog using :account;
```

- create a template of SQL queris
- `?` is place holder that can be replaced
- built at run time, otherwise static embeded SQL is built when compling

EXECUTE IMMEDIATE:

- execute the instruction immediately
- all the host variables have been put into the string

```
EXEC SQL  begin declare section;
    char statement[256];
EXEC SQL end declare section;
char table_name[32], condition[128];
gets (table_name);
gets( condition);
// put in the variables
if (strlen (condition)>0)
         sprintf (statement, “delete from %s where %s”, table_name, condition);
else
      sprintf (statement, “delete from%S”, table_name);
// execute
EXEC SQL EXECUTE  IMMEDIATE  :statement;
```

PREPARE:

- the instructions prepared are complied at first
- the variables are transferred with `USING`

```
// update %s  set %s=?  where %s=?
// variables: statement, new_value, search_val;
//             table_name, update_column, search_column;
gets (table_name); gets(update_column); gets (search_column);
sprintf (statement, "update %s  set %s=? where %s=?",
    table_name, update_column, search_column)
EXEC SQL PREPARE mystmt FROM :statement;    // compiled at first
//check sq1 state
for (; ;){
    scanf ( "%d  %d", &new_value, &search_value);
    EXEC SQL EXECUTE mystmt USING  : new_value,  : search_value;
        //check sq1 state
}
```

Procedural Constructs

function:

```
create function instructors_of (dept_name char(20) )
returns table ( ID varchar(5),
            name varchar(20),
            dept_name varchar(20),
            salary numeric(8,2))
         # declare the type to return
return table
    (select ID, name, dept_name, salary
    from instructor
    where instructor.dept_name = instructors_of.dept_name)
          # declare the content to return
```

```
delimiter $$
create function dept_count (dept_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count (* ) into d_count
    from instructor
    where instructor.dept_name = dept_name;
    return d_count;
end;
delimiter ;
```

procedure:

```
create procedure dept_count_proc (in dept_name varchar(20), out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

for(on all rows):

```
declare n integer default 0;
for r as
     select budget from department
     where dept_name = 'Music'
do
     set n = n - r.budget
end for
```

while and repeat:

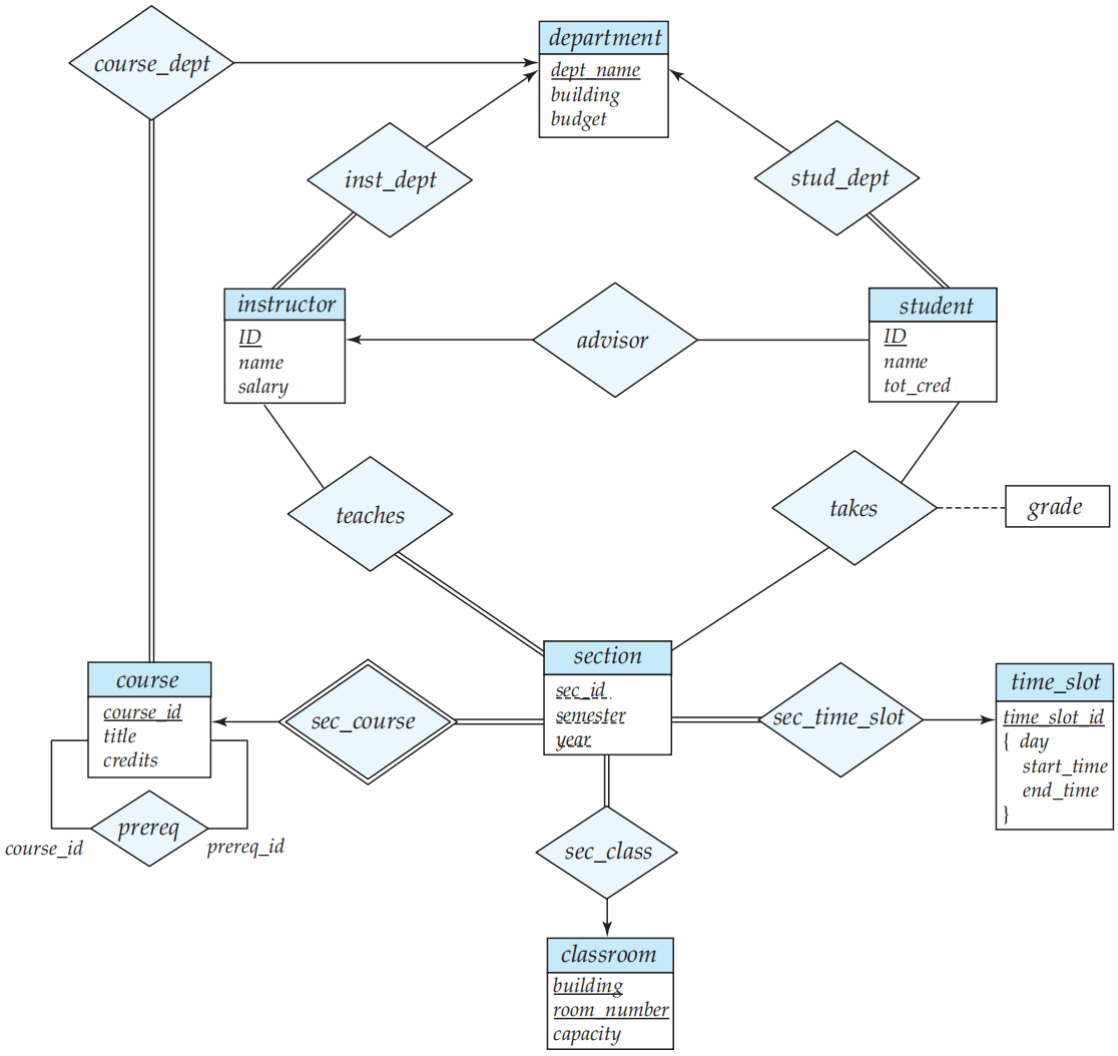
```
declare n integer default 0;
while n < 10 do
 set n = n + 1
end while

repeat
 set n = n - 1
until n = 0
end repeat
```

Entity-Relationship Model

- Entity: a "thing" or "object" in the enterprise that is distinguishable from other objects
    - An entity set that does not have sufficient attributes to form a primary key is termed a **weak entity set**
        - discriminator, partial key
        - The existence of a weak entity set depends on the existence of a **identifying entity set**
        - It must relate to the identifying entity set via a **total, one-to-many** relationship set from the identifying to the weak entity set
        - **Identifying relationship** depicted using a double diamond
    - An entity set that has a primary key is termed a **strong entity set**.
- Relationship: an association among several entities

E-R Diagram

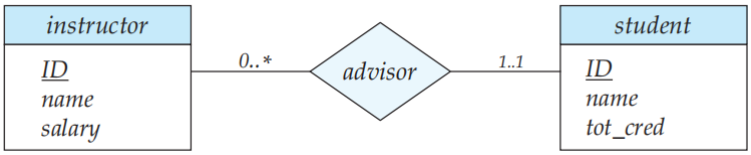


- **Merely a conceptual diagram**
- **Rectangles divided into two parts** represent entity sets. The first part, which in this textbook is shaded blue, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.
    - The discriminator of a weak entity is underlined with a dashed, rather than a solid line.
- **Diamonds** represent relationship sets.
    - The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond
    - The primary key for R is consists of the union of the primary keys of entity sets $E1, E2, . . En$
    - If the relationship set R has attributes $a1, a2, . . , am$ associated with it, then the primary key of R also includes the attributes $a1, a2, . . , am$
- **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.
- **Lines** link entity sets to relationship sets.
    - **Directed / Undircted lines** indicate the **Mapping Cardinality Constraints**, e.g. one-to-one
    - Each occurrence of an entity set plays a **role** in the relationship
    - The number of the entity sets involved is called **the degree of a relationship set**

- When the degree is larger than 2, to avoid confusion we **outlaw more than one arrow**
  - one-to-many relationship between an *instructor* and a *student*
    - an *instructor* is associated with several (including 0) *students* via *advisor*
    - a *student* is associated with at most one *instructor* via advisor,

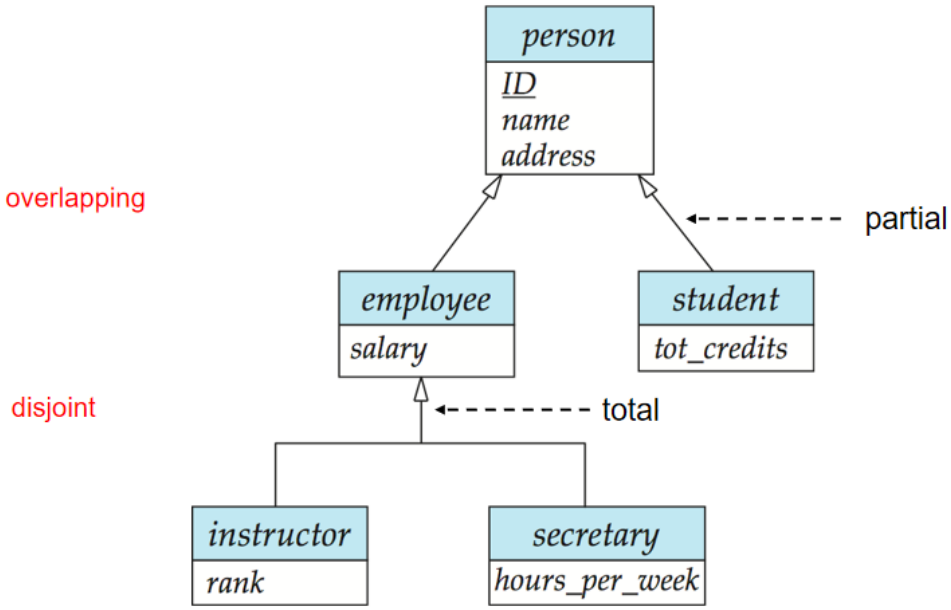
  - In a many-to-one relationship between an *instructor* and a *student*,
    - an *instructor* is associated with at most one *student* via *advisor*,
    - and a *student* is associated with several (including 0) *instructors* via *advisor*

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where *l* is the minimum and *h* the maximum cardinality. A minimum value of 1 indicates total participation of the entity set in the relationship set; that is, each entity in the entity set occurs in at least one relationship in that relationship set. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value ∗ indicates no limit.



- **Dashed lines** link attributes of a relationship set to the relationship set.
- **Double lines** indicate total participation of an entity in a relationship set.

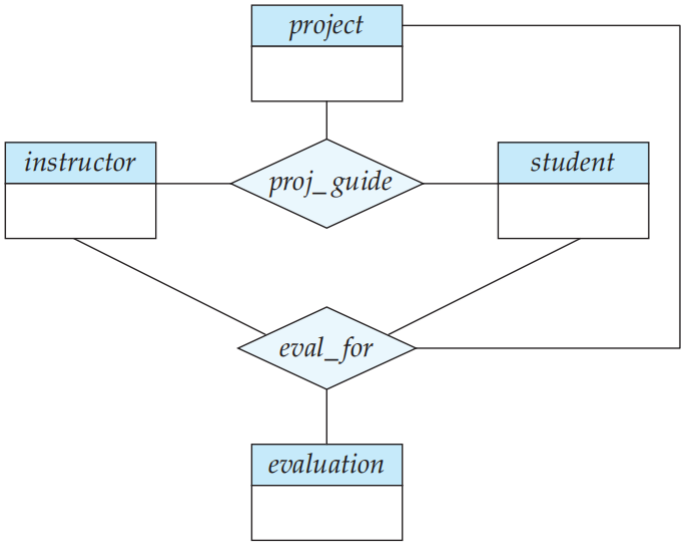
Specialization (Or inversely generalization):



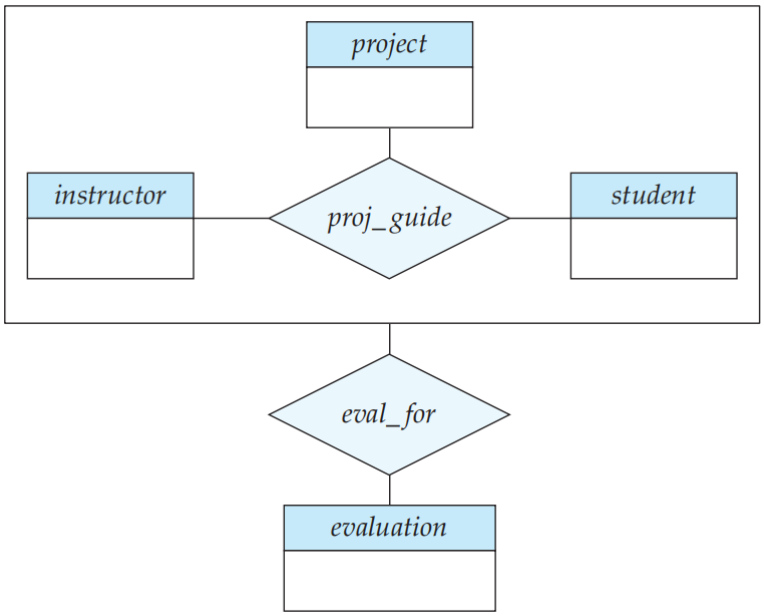
- Constrasints
  - condition-defined / user-defined
    - condition-defined: an entity is arranged to a lower-level set by some conditions **automatically**
    - user-defined: an entity is arranged **mannually**
  - total / partial
    - total generalization or specialization: each higher-level entity must belong to a lower-level entity set
    - partial generalization or specialization: some higher-level entities may not belong to any lower-level entity set
  - disjoint / overlapping
- Schema implementation
schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary
schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary
schema	attributes
person	*ID, name, street, city, **person_type**, tot_cred, salary)*

Aggregation & Reduction to Relational Schemas

Aggregation:



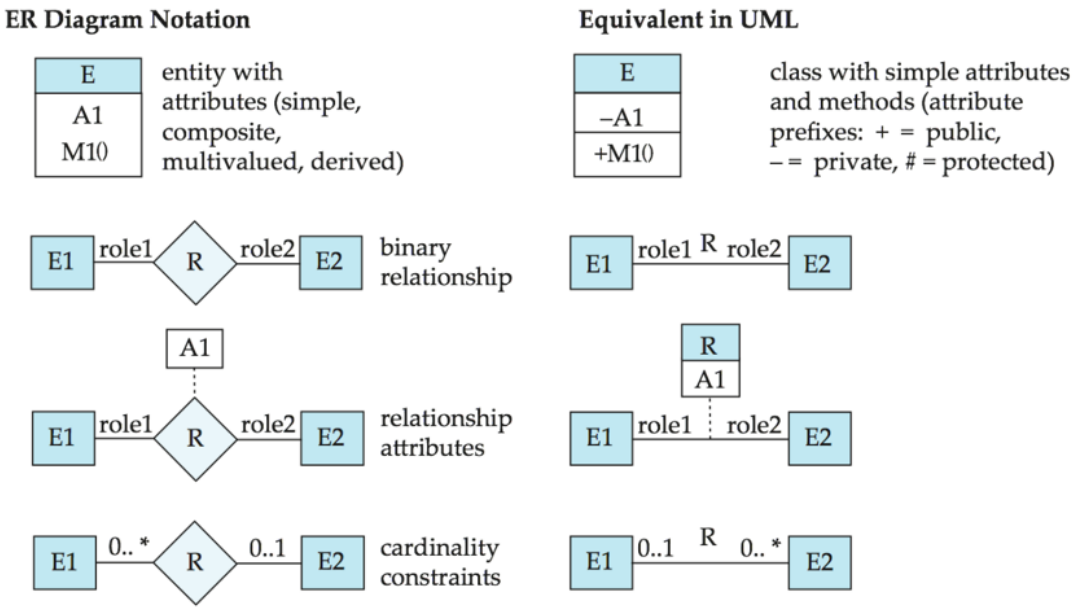
This is an E-R diagram with redundant information. Make *proj_guide* **be treated as a new entity set**:



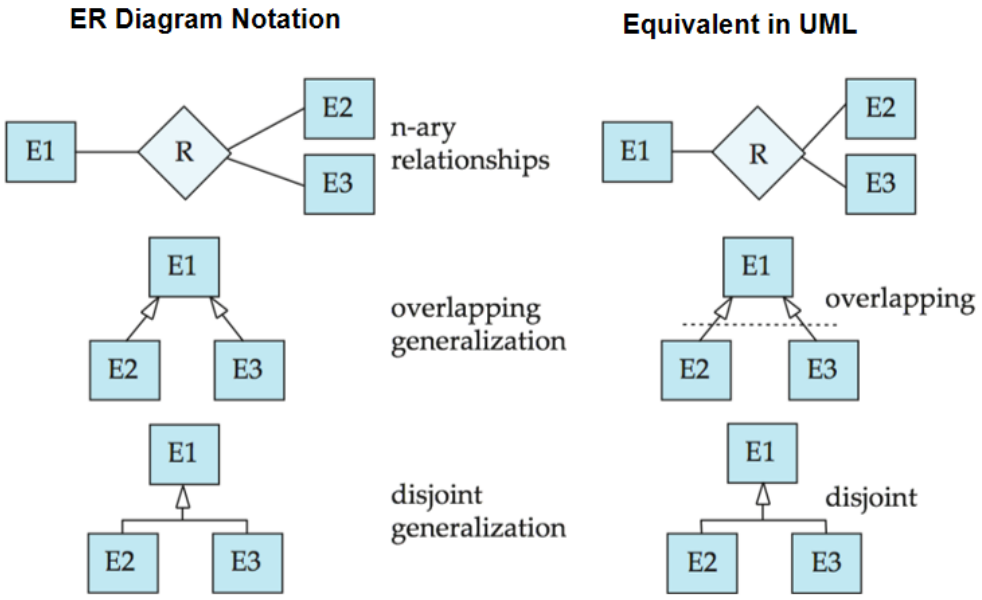
Reduction to Relational Schemas:

- A one-to-many or a many-to-one relationship can be implemented by simply adding one attribute to the many- side
- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the participating entity sets, and any descriptive attributes of the relationship set.
    - Considering storage, storing small tables is more preferable
    - Even we can store each column respectively
- Multivalued attribute:
    - Multiple tuples, and mind the constitution of the primary key
    - Use a separator and store in one tuple, suitable for a one-to-many relationship
    - Use a new table and a relationship, allowing a more flexible relationship
- Reduce non-binary relationships

UML



*Note reversal of position in cardinality constraint depiction



*Generalization can use merged or separate arrows independent of disjoint/overlapping

Relational Databse Design

When to split up a single schema:

- if a non-key attribute determines other several attributes, which can be a candidate key in a new schema
    - otherwise, the decomposition may be a lossy decomposition

    - when all the dependencies are functional:
        - For the case of R = (R1, R2), we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

        - A decomposition of R into R1 and R2 is lossless join if and only if at least one of the following dependencies holds:
            - $R_1 \cap R_2 \rightarrow R_1$
            - $R_1 \cap R_2 \rightarrow R_2$
    - lossless: perfect reconstruction can be done
- some attributes are needed together to represent an entity
- redundancy

1 judge lossless decomposition

A domain is **atomic** if elements of the domain are considered to be indivisible units

- Not atomic: *name* consists of *firstname*, *middlename* and *lastname*
- integers, (not encoded) strings are assumed to be atomic

Requirement

Performance:

- **Throughput**: the number of queries or updates (often referred to as *transactions*) that can be processed on average per unit of time.
- **Response time**: the amount of time a *single* transaction takes from start to finish in either the average case or the worst case
    - Indices can be created for possible `join` to improve response time, for a `join` in a `where` clause takes much time

Dataflow & Workflow:

> The term *workflow* refers to the combination of data and tasks involved in processes like those of the preceding examples. Workflows interact with the database system as they move among users and users perform their tasks on the workflow. In addition to the data on which workflows operate, the database may store data about the workflow itself, including the tasks making up a workflow and how they are to be routed among users. Workflows thus specify a series of queries and updates to the database that may be taken into account a part of the database design process.

Other issues:

- Prepare for potential changes in the future

> A good design should account not only for current policies, but should also avoid or minimize changes due to changes that are anticipated, or have a reasonable chance of happening

- Multiple databases may need to interact
- Interact extensively with experts in the application domain to understand the data requirements of the application.

Functional Dependency

Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

> *dept* can decide *budget*, but we can't find *dept* simply by *budget*, and this is a **functional dependency**
>
> **key** is a special case of **functional dependency**
> K is a superkey for relation schema R **if and only if** $K \rightarrow R$
> K is a candidate key for R if and only if
> - $K \rightarrow R$, and
> - **for no** $\alpha \subset K$, $\alpha \rightarrow R$

trivial: *ID, name*$\rightarrow$*ID*

Closure

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$,
  - then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by *F* is the **closure** of *F*.
- We denote the closure of F by F+.
- F^+ is a superset of *F*.
- F={A→B, B→C} , what is F^+?
  F^+ ={A→B, B→C, A→C, AB→B, AB→C, … }

We can find F+, the closure of F, by repeatedly applying Armstrong's Axioms:

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity,自反率)**
- if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation，增补率)**
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity，传递率)**

These rules are

- **Sound（正确有效的）** (generate only functional dependencies that actually hold), and
- **Complete（完备的）** (generate all functional dependencies that hold).

- Additional rules:
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds
  - **(union，合并)**
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition，分解)**
  - If $\alpha \rightarrow \beta$ holds and $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds **(pseudotransitivity，伪传递)**
- The above rules can be inferred from Armstrong's axioms.

> Augmentation for proof: $\alpha \rightarrow \alpha\beta \rightarrow \beta\gamma$

Closure of Attribute Sets

A denotation:

Given a set of attributes ***a***, define the closure of ***a*** under F (denoted by ***a+***) as the set of attributes that are functionally determined by ***a*** under F

R(A,B,C,D) *F={A→B,B→C, B→D}*

A^+=*ABCD*

B^+=*BCD*

C^+=*C*

Algorithm to compute α^+, the closure of α under *F*

```
result := α;
while (changes to result) do
    for each β → γ in F do
      begin
        if β ⊆ result then result := result ∪ γ
      end
```

Usage:

- Testing for superkey
- Testing functional dependencies
- Computing closure of F

Computing closure of F:
R(A,B,C), F={A→B, B→C}
computing:
A^+=ABC, B^+=BC, C^+=C
$(AB)^+$=ABC, $(AC)^+$=ABC, $(BC)^+$=BC, $(ABC)^+$=ABC
F⇔{A→ABC, B→BC, C→C, AB→ABC, AC→ABC, BC→BC, ABC→ABC}
⇔{A→A, A→B,A→C, A→AB,A→AC,A→BC,A→ABC
B→B,B→C, B→BC,
C→C,
AB→A, AB→B,AB→C, AB→AB,AB→AC,AB→BC,AB→ABC,
AC→A, AC→B,AC→C, AC→AB,AC→AC,AC→BC,AC→ABC,
BC→B, BC→C, BC→BC,
ABC→A, ABC→B,ABC→C, ABC→AB, ABC→AC, ABC→BC, ABC→ABC}
=F^+

Canonical cover

Canonical cover (simplification):

Sets of functional dependencies may have redundant dependencies that can be inferred from the others

- For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C, A \rightarrow C\}$

Parts of a functional dependency may be redundant

- E.g.: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$
  can be simplified to
  $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

  .A→CD=> $A \rightarrow D$

  .A→B,B→C => $A \rightarrow C$, A→D =>$A \rightarrow CD$

- E.g.: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to
  $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

  .A→B,B→C => A→C=> A→AC, AC→D =>A→D

  .A→D => $AC \rightarrow CD$ => AC→D

Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

> minimal set representing a functional dependency

A canonical cover for F is a set of dependencies Fc such that

- F logically implies all dependencies in F_c, and
- F_c logically implies all dependencies in F, and
- No functional dependency in F_c contains an extraneous attribute, and
- Each left side of functional dependency in F_c is unique.

To compute a canonical cover for F:
repeat
Use the union rule to replace any dependencies in F
$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
Find a functional dependency $\alpha \rightarrow \beta$ with an
extraneous attribute either in α or in β
If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until F does not change

Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

> Extraneous: remove an attribute, and the original dependency can be obtained by F

NF & Decomposition

> 第一范式: 列不可再分
>
> 第二范式: 非主键属性全部依赖于主键属性
>
> 第三范式: 非主键属性之间无依赖关系
>
> BCNF：主键列之间，不存在依赖
>
> 第四范式: 多值依赖中，主键属性之间无依赖关系

[1 all kinds of normal forms]

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Decomposition:

Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose *R* into:

- $(\alpha \cup \beta)$
- $(R - \beta)$

In our example,

- α = ***dept_name***
- β = ***building, budget***

and *inst_dept* is replaced by

- $(\alpha \cup \beta)$ = ***(dept_name, building, budget)***
- $(R - \beta)$ = ***(ID, name, salary, dept_name)***

> Notice the dependency here is **non-trivial**, which means $\alpha \cap \beta = \phi$ here

Algorithm:

```
    result := {R };
    done := false;
    compute F +;  //  not necessary
    while (not done) do
        if (there is a schema Ri in result that is not in BCNF)
        then begin
            let (alpha to beta) be a nontrivial functional dependency that
                        holds on Ri such that (alpha to Ri) is not in F +,   //
alpha is not key of Ri
                    and alpha cap beta = NULL;
               result := (result - Ri ) cup (Ri - beta) cup (alpha, beta );
            end
        else done := true;
```

> Remove the dependency violating the definition

> The attributes of candidate keys with no dependencies among themselves will be left in a schema. That is, the primary key should be kept in a schema for completeness.

The decomposition guarantees **dependency preservation**. If it is not, then checking updates for violation of functional dependencies may **require computing joins**, which is expensive. Example:

$R = (A, B, C)$
$F = \{A \rightarrow B$
$\quad B \rightarrow C\}$
Key = $\{A\}$

R is not in BCNF

Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$

- R_1 and R_2 in BCNF
- Lossless-join decomposition
- Dependency preserving

$$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

> The dependencies used before union are directly given

> It may not be possible to preserve dependencies while decomposing a relation into relations that are in BCNF

Since there is no guarantee that dependency perservation and BCNF can be both maintained always, we sometimes reduce BCNF to 3NF:

- Allows some redundancy
- But functional dependencies can be checked on individual relations without computing a join.
- There is **always** a lossless-join, dependency-preserving decomposition into 3NF.

A relation schema R is in third normal form (3NF) if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

  (**NOTE**: each attribute may be in a different candidate key)

If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).

Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

> Relaxation: an attribute belonging to a candidate key can depend on other attributes belonging to a candidate key
>
> each attribute A in $\beta - \alpha$ may be contained in a *different* candidate key

Algorithm:

```
1   Let Fc be a canonical cover for F;
2   i := 0;
3   for each  functional dependency (alpha to beta) in Fc do
4       begin
5           i := i  + 1;      Ri  := alpha, beta
6      end

7   if none of the schemas Rj, 1 <= j <= i contains a candidate key for R
8   then begin
9           i := i  + 1;
10          Ri := any candidate key for R;
11     end

12  /* Optionally, remove redundant relations */
13  repeat
14      if any schema Rj is contained in another schema Rk
15       then /* delete Rj  */
16           begin
17              Rj = Ri ;   i=i-1;
18  until no more Rj s can be deleted
19  return (R1, R2, ..., Ri)
```

> Create a schema for each dependency and a schema for the candidate key
> Finally remove redundancy

Multivalued Dependency (MVD)

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_3[\alpha] = t_4[\alpha] = t_1[\alpha] = t_2[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

> example:

Tabular representation of $\alpha \twoheadrightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

> the left 2 columns are the same but the rightmost column is not

ID	*child_name*	*phone*
99999 99999 99999 99999	David David William Willian	512-555-1234 512-555-4321 512-555-1234 512-555-4321

inst_info

In our example:

$$ID \twoheadrightarrow child\_name$$
$$ID \twoheadrightarrow phone\_number$$

The above formal definition is supposed to formalize the notion that given a particular value of Y (ID) it has associated with it a set of values of Z (child_name) and a set of values of W (phone_number), and these two sets are in some sense independent of each other.

Note:

- If $Y \rightarrow Z$ then $Y \twoheadrightarrow Z$
- Indeed we have (in above notation) $Z_1 = Z_2$

> every functional dependency is also a multivalued dependency

Fourth Normal Form

A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

- $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
- α is a superkey for schema R

If a relation is in 4NF it is in BCNF

The closure *D+* of *D* is the set of all functional and multivalued dependencies logically implied by *D*

Algotirhm:

result: = {*R*};
done := false;
compute D^+;
Let D_i denote the restriction of D^+ to R_i

while (**not** *done*)
  if (there is a schema $\mathbf{R}_i$ in *result* that is not in 4NF) **then**
    begin
      let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
      result := (*result* - R_i) $\cup$ (R_i - β) $\cup$ (α, β);
    end
  else *done*:= true;

Note: each R_i is in 4NF, and decomposition is lossless-join

Issues of Redundancy

Use non-normalized schema for performance

- faster lookup
- extra space and extra execution time for updates
- extra coding work for programmer and possibility of error in extra code

For analysis and further analysis:

- *earnings (company_id, year, amount)*
- *earnings_2012(company_id, earnings)*
  earnings_2013(company_id, earnings)
  ...
- *company_year (company_id, earnings_2012, earnings_2013, earnings_2014)*

Storage and File Structure

Storage hierarchy:

- primary storage: Fastest media but volatile

- secondary storage: next level in hierarchy, non-volatile, moderately fast access time
- tertiary storage: lowest level in hierarchy, non-volatile, slow access time

Magnetic disk:

- Read-write head
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular tracks（磁道）
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into sectors（扇区）.
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- Cylinder *i* consists of *i*-th track of all the platters
- Disk controller(磁盘控制器) – interfaces between the computer system and the disk drive hardware.
  accepts high-level commands to read or write a sector
  - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
  - Computes and attaches checksums to each sector to verify that data is read back correctly
    If data is corrupted, **with very high probability stored checksum won't match recomputed checksum**
  - Ensures successful writing by reading back sector after writing it

Performance Measures

- Access time(访问时间)
  - Seek time（寻道时间）
  - Rotational latency （旋转等待时间）
- Data-transfer rate（数据传输率）
- I/O operations per second (IOPS)
- Mean time to failure (MTTF，平均故障时间)

Optimization:

- Block oriented access
  - Smaller blocks: more transfers from disk
  - Larger blocks: more space wasted due to partially filled blocks
  - Typical block sizes today range from 4 to 16 kilobytes
- Buffering
- Read-ahead
- Disk-arm-scheduling

    - elevator algorithm
- File organization
    - related information is stored in nearby location
    - fragment / defragment
    - same cylinder > nearby sectors > nearby tracks
- Use nonvolatile write buffers before writing to disks
- Log disk
- Journaling file systems reorders writes to disk to improve performance

File Organization, Record Organization and Storage Access

The database is stored as a collection of files.

Each file is a sequence of records.

A record is a sequence of fields.

> A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks. A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files
>
> Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer.

Large files: store separately and use pointers

Fixed-length

Insertion:

- 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, which may cause waste of space

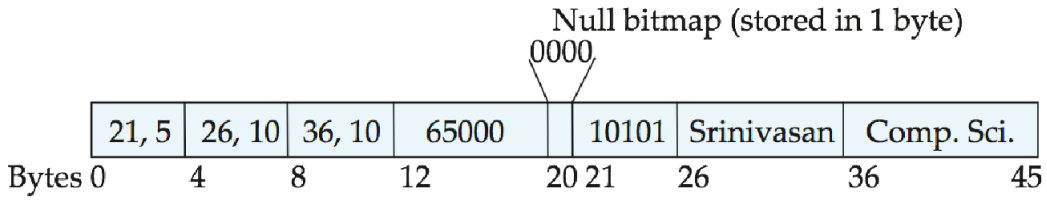
Deletion:

- move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
- move record n to i
- do not move records, but link all free records on a free list

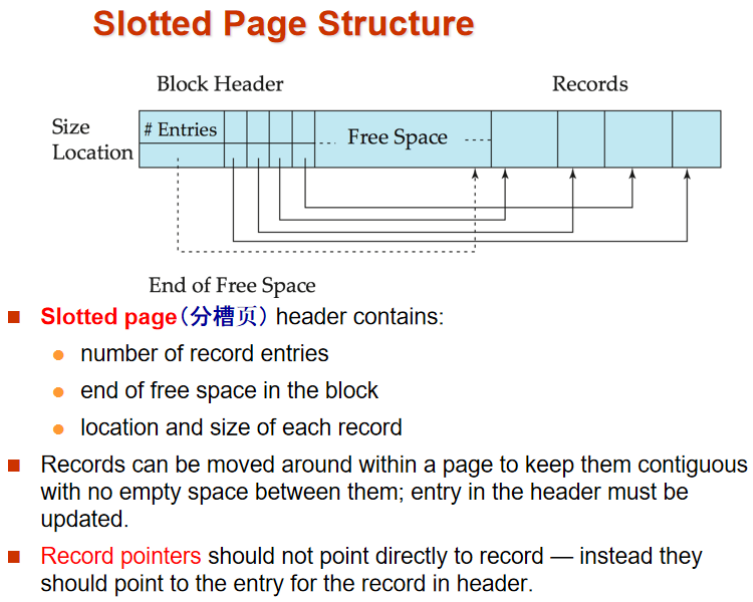
Variable-Length

For a record:

- Variables are stored in order
- *(21, 5)* means the first attribute start from byte 25 and the length is 5 bytes
- fixed-length attribute is directly stored
- *Null bitmap* is used to indicate whether an attribute is *NULL*. None of the 4 attributes is *NULL* here, so the bitmap is 0000



For a block, slotted-page structure:



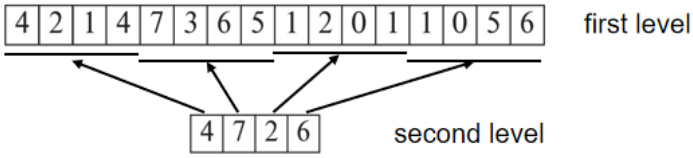
- If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- lazy-deletion and consolidate the structure in certain cases
- use indirect index rather than use address directly
- The slotted-page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header

Organization of Records in Files

Possible ways of organizing records in files:

- heap file organization
    - randomly placing
    - 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
- In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free
  - 

- Can have second-level free-space map
- In example above, each entry stores maximum from 4 entries of first-level free-space map

- 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
  - Implement with a **pointer chain**
  - Need to reorganize the file from time to time to restore sequential order
- hashing file organization
- multitable clustering file organization
  - multiple tables
    - Store several relations in one file using a multitable clustering file organization
    - *department*

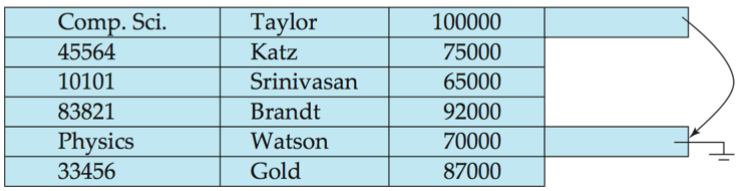
dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

    - *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
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

    - Can add pointer chains to link records of a particular relation
  - 

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

  - results in variable size records
- B+ tree file organization

> Generally, a separate file is used to store the records of each relation. However, in a **multitable clustering file organization**, records of several different relations are stored in the same file; further, related records of the different relations are stored on the same block, so that one I/O operation fetches related records from all the relations

Metadata

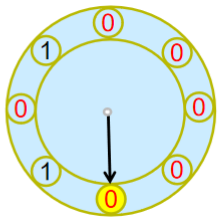
The Data dictionary (also called system catalog) stores **metadata**; that is, data about data

- Information about relations
- User and accounting information, including passwords
- Statistical and descriptive data
- Physical file organization information
- Information about indices

Storage Access & Buffer Manager

- 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
- To fully make use of main memory
- Buffer managers also support **forced output** of blocks for the purpose of recovery
- Replacement:
    - LRU (Least Recently Used)
        - Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
        - LRU can be a bad strategy for certain access patterns involving repeated scans of data
        - An approximation: **Clock**
            - When replacement necessary

```
do for each block in cycle {
        if (reference_bit ==1)
                set reference_bit=0;
        else if (reference_bit ==0)
                choose this block for replacement;
} until a page is chosen;
```



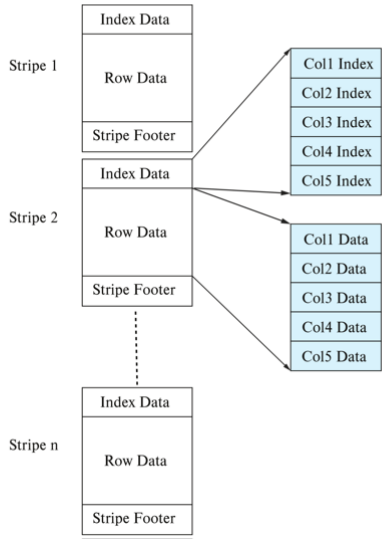
    - LFU (Least Frequently Used)
    - Pinned block — memory block that is not allowed to be written back to disk
    - Toss-immediate strategy — frees the space occupied by a block as soon as the final tuple of that block has been processed
    - Most recently used (MRU) strategy — system must pin the 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

Column-Oriented Storage

- Also known as columnar representation
- Store each attribute of a relation separately
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
    - Called hybrid row/column stores

Columnar File Representation

- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:

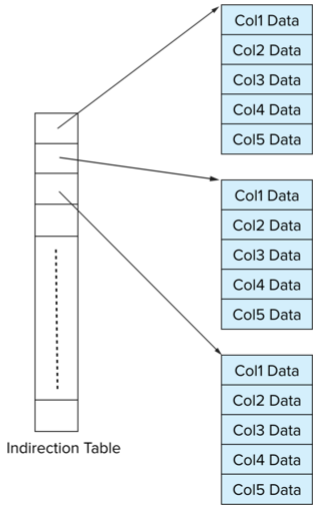


Storage Organization in Main-Memory Databases

Can store records directly in memory without a buffer manager

Column-oriented storage can be used in-memory for decision support applications

- Compression reduces memory requirement



Indexing and Hashing

Concepts:

- Indexing mechanisms used **to speed up access** to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in **sorted order**
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index evaluation:

- Access types supported efficiently. E.g.,
  - Point query: records with a specified value in the attribute
  - Range query: or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

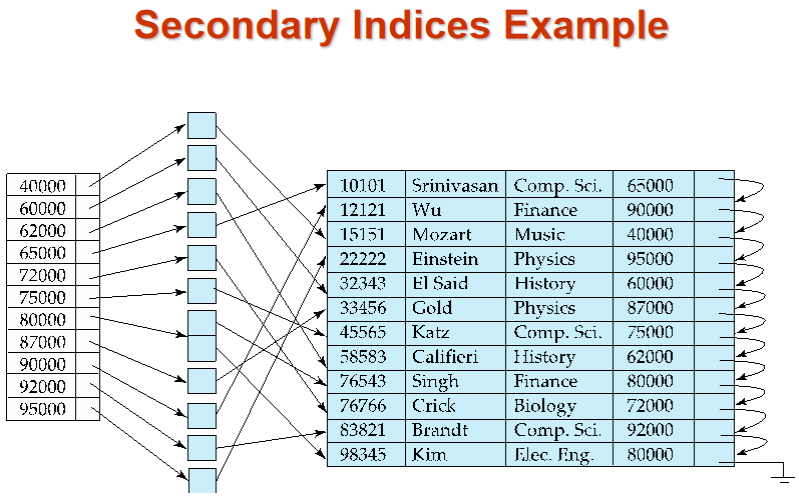
Ordered Indices

- Performance degrades when the scale is large
- Primary index（主索引）: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called clustering index（聚集索引）
  - The search key of a primary index is usually but not necessarily the primary key.

Primary index（主索引）

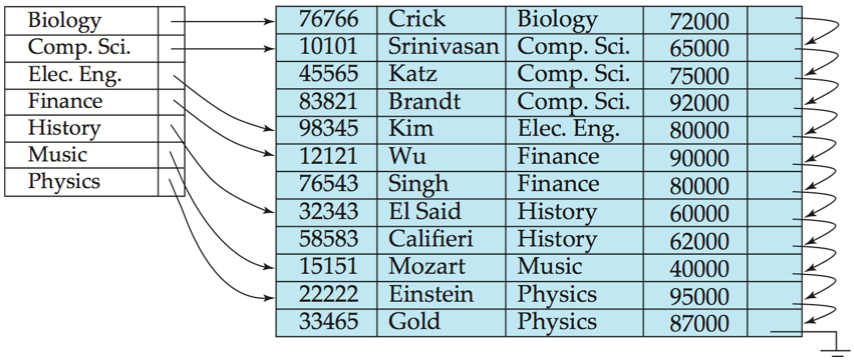
10101	10101	Srinivasan	Comp. Sci.	65000	
12121	12121	Wu	Finance	90000	
15151	15151	Mozart	Music	40000	
22222	22222	Einstein	Physics	95000	
32343	32343	El Said	History	60000	
33456	33456	Gold	Physics	87000	
45565	45565	Katz	Comp. Sci.	75000	
58583	58583	Califieri	History	62000	
76543	76543	Singh	Finance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

- Secondary index（辅助索引）: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index.
- Index-sequential file（索引顺序文件）: ordered sequential file with a primary index

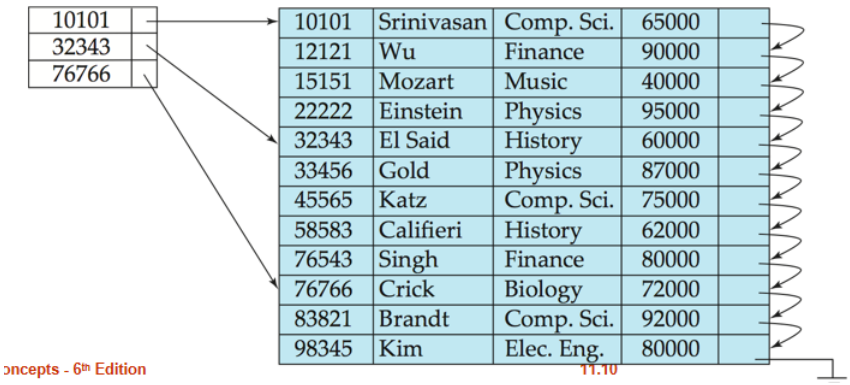


- Dense index(稠密索引) — Index record appears for every search-key value in the file.

  - Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



- Sparse Index（稀疏索引）: contains index records for only some search-key values.
  - Applicable when records are **sequentially ordered on search-key**
  - Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block



- Multilevel Index is somehow similar to B+-Tree Index

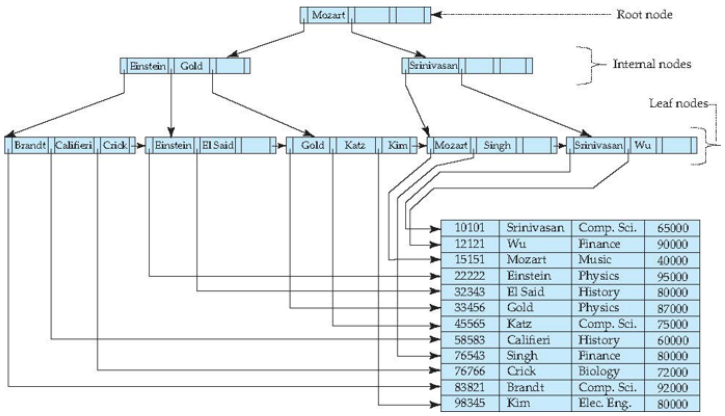
B+-Tree Index & Organization

Index

Advantages of B+-trees outweigh disadvantages and B+-trees are used extensively

A B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Inner node(not a root or a leaf): between $\lceil n/2 \rceil$ and n children.
- Leaf node: between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
  - If the root is not a leaf: at least 2 children.
  - If the root is a leaf : between 0 and $(n-1)$values.



Estimation:

B+- tree : height and size estimation

- *person*(*pid* **char**(18), *name* **char**(8), *age* **smallint**,
- address **char(40),** **primary key** (*pid*));
- *Block size : 4K*
- *1,000,000 persons*
- *Records per block* = 4096/(18+8+2+40) =60.235 → 60

  blocks for storing 1M persons= ⌈ 1000000/60 ⌉=16667

  B+ tree n(fan-out) = (4096-4)/(18+4) +1 = 187

  ⌈n/2⌉= 94 , inner pointers : 94~187

  ⌈n-1/2⌉=93, leaf values : 93~186

  2 levels: min=2*93 =186 max= 187*186= 34,782

  3 levels: min=2*94*93 =17484 max= 187*187*186=6,504,234

  4 levels: min=2*94*94*93=1,643,496 max= 187*187*187*186= 1,216,291,758

2.641029	<=	B+ level	<=	3.8883		
3	<=	B+ level	<=	3		
		half full leaf nodes				
		leaf level nodes		10752.69	10752	nodes
		second level nodes			114	nodes
		root			1	node
		total nodes			10867	
		full leaf nodes				
		leaf level nodes		5376.344	5377	nodes
		second level nodes			29	nodes
		root			1	node
		total nodes			5407	

$4096 - 4$: for a pointer pointing to the next block

$18 + 4$: *pid* and a pointer

fan-out: the number of pointers within a node

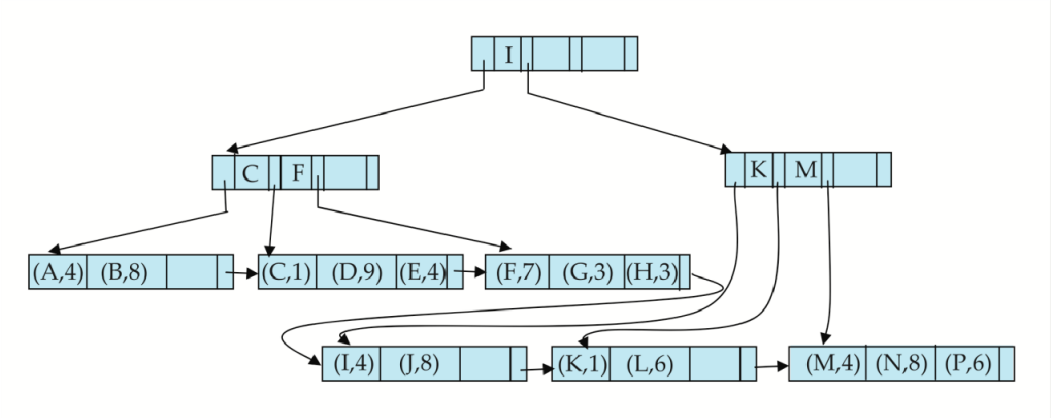
[为什么生产环境中B+树的高度总是3-4层]

Non-Unique Search Key:

- If a search key a_i is not unique, create instead an index on a composite key (a_i, A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- Extra storage overhead for keys
- Simpler code for insertion/deletion
- More I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation
- Widely used

File organization:

- The leaf nodes in a B+-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index



- Good space utilization is important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges

Other Issues

Record relocation and secondary indices

- If a record moves, all secondary indices that store record pointers have to be updated
- Node splits in B+-tree file organizations become very expensive
- *Solution: use primary-index search key instead of record pointer in secondary index*
    - Extra traversal of primary index to locate record
        - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique
    - Example: to find a record by *name*, an index is used to find *pid* by *name* and an index is used to find actual record by *pid*

Bulk Loading and Bottom-Up Building

Efficient alternative 1: Insert in sorted order

- sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
- insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes **half full**

Efficient alternative 2: Bottom-up B+-tree construction

- As before sort entries
- And then create tree layer-by-layer, starting with leaf level
- Implemented as part of bulk-load utility by most database systems

Multiple keys:

- **Composite search keys** are search keys containing more than one attribute
    - E.g. (*dept_name*, *salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
    - $a_1 < b_1$, or
    - $a_1 = b_1$ and $a_2 < b_2$

On disk: B+-tree

On flash: Write-optimized tree structures (i.e., LSM-tree)

In cache: Skip list

Indexing Strings

- Variable fanout
- Use space utilization as criterion for splitting, not number of pointers
- Prefix compression
    - Key values at internal nodes can be prefixes of full key
    - Keys in leaf node can be compressed by sharing common prefixes

Query Processing

- Parsing and translation
    - translate the query into its internal form. This is then translated into relational algebra.
    - Parser checks syntax, verifies relations
- Optimization
    - Amongst all equivalent evaluation plans choose the one with lowest cost.
- Evaluation
    - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
    - disk accesses, CPU, or even network communication
- Typically disk access is the predominant cost, and is also relatively easy to estimate
- For simplicity we just use the number of block transfers from disk and the number of seeks as the cost measures
  tT – time to transfer one block
  tS – time for one seek
  Cost for b block transfers plus S seeks: b * tT + S * tS
- tS and tT depend on where data is stored; with 4 KB blocks:
  High end magnetic disk: tS = 4 msec and tT =0.1 msec, tS / tT ≈ 40
  SSD: tS = 20-90 microsec and tT = 2-10 microsec for 4KB , tS / tT ≈ 10
- We ignore CPU costs for simplicity
    - Real systems do take CPU cost into account
- We ignore cost to writing output to disk in our cost formulae

Selection

Time cost for linear selection:

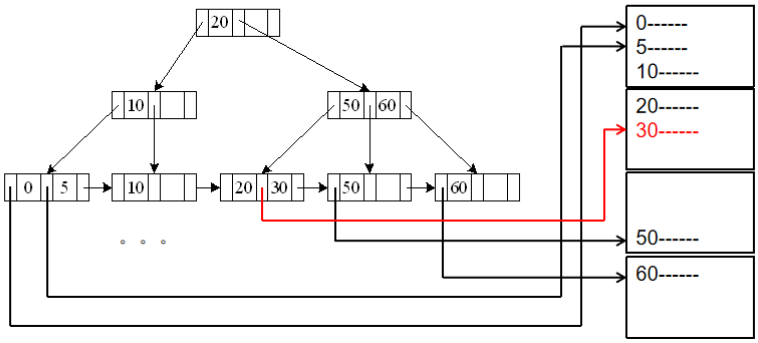
- **File scan**
- Algorithm **A1 (linear search).** Scan each file block and test all records to see whether they satisfy the selection condition.
  - **worst cost = $b_r * t_T + t_S$**

    b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record

    average cost = $(b_r/2)\ t_T + t_S$
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

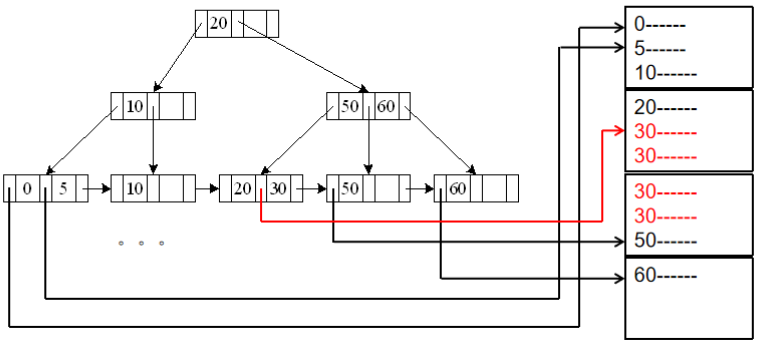
Time cost for (primary/secondary B+-)index selection:

- **Index scan –** search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2 (primary B+-tree index , equality on key).** Retrieve a single record that satisfies the corresponding equality condition
  - ***Cost* = $(h_i + 1) * (t_T + t_S)$**



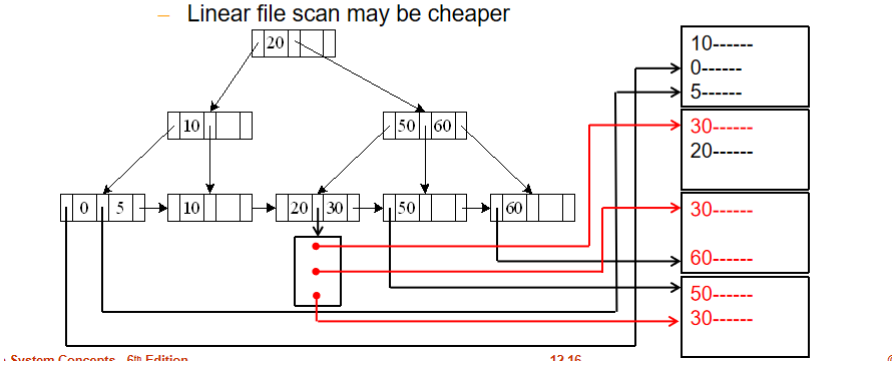
Time cost for (primary B+-)index selection on a nonkey:

- **A3 (primary B+-tree index, equality on nonkey)** Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - ***Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$**



With comparisons:

- **A6 (secondary B+-tree index, comparison).**
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper



Complex selections:

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A7 (conjunctive selection using one index).**
  - Select a combination of θ_i and algorithms A1 through A6 that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- **A8 (conjunctive selection using composite index).**
  - Use appropriate composite (multiple-key) index if available.
- **A9 (conjunctive selection by intersection of identifiers).**
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

External Sort-Merge

1. Create sorted runs(归并段).
   Let i be 0 initially.
   Repeatedly do the following till the end of the relation:
   (a) Read M blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run Ri; increment i.

   Let the final value of i be N

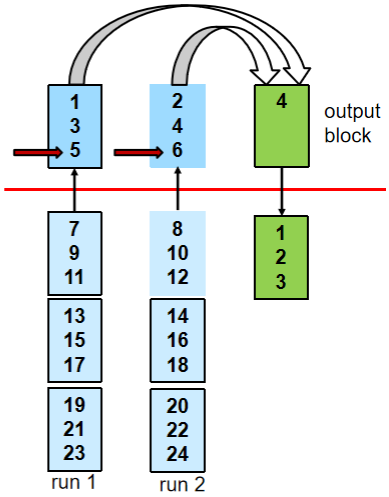
2. Merge the runs (N-way merge)

2. **Merge the runs (N-way merge).**

- **IF N < M, single merge pass is required**

  (如果归并段少于可用内存页)

  1. Use *N* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
  2. **repeat**
     1. Select the first record (in sort order) among all buffer pages
     2. Write the record to the output buffer. If the output buffer is full write it to disk.
     3. Delete the record from its input buffer page.
        If the buffer page becomes empty **then**
        read the next block (if any) of the run into the buffer.
  3. **until** all input buffer pages are empty:



2-way merge here and M=3

3.

- **If *N* ≥ *M*, several merge *passes* are required.**
  - In each **pass**, contiguous groups of ***M* - 1** runs are merged.
  - A pass reduces the number of runs by a factor of ***M* -1**, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

If M is larger, then the times of I/O per block can be reduced

External Sort and 外部排序

Join

Several different algorithms to implement joins

- Nested-loop join
  - process by tuple
- Block nested-loop join
  - process by block
  - reduce I/O for the outer loop
- Indexed nested-loop join
  - if some attribute of one relation is the index of the another
- Merge-join
  - sort both relations on their join attribute
- Hash-join
  - applicable for equi-join and natural join
  - each partition should fit in the memory
  - a hash function h is used to **partition tuples of both relations with common JoinAttrs**

- other methods are used to join each pair of partitions
- - **Recursive partitioning** required if number of partitions *n* is greater than number of pages *M* of memory.
  - Instead of partitioning n ways, use M – 1 partitions for s
  - Further partition the M – 1 partitions using a different hash function
  - Use same partitioning method on r
  - A relation does not need recursive partitioning if $M > n_h + 1$, or equivalently $M > (b_s/M) + 1$, which simplifies (approximately) to $M > \sqrt{b_s}$.
  - For example, consider a memory size of 12M bytes, divided into 4K bytes blocks; it would contain a total of 3K (3072) blocks. We can use a memory of this size to partition relations of size up to 3K ∗ 3K blocks, which is 36G bytes.
  - Similarly, a relation of size 1 gigabyte requires just over √256K blocks, or 2 megabytes, to avoid recursive partitioning.
- hash-table overflow
  - fudge factor
  - overflow resolution
  - overflow avoidance
- HASH-JOIN

Other

- Duplicate Elimination
  - sort
- Projection
  - projection on each tuple
  - remove duplicate ones
- Set Operations
  - sort both relations
  - scan once through the sorted relations
- Outer join
  - Strategy 1:
    - join
    - add further tuples
  - Strategy 2:
    - modify the join algorithm
- Aggregation
  - sort / hash
  - apply aggregation operations

Evaluation of Expressions

Alternatives for evaluating an entire expression tree

- Materialization(物化): generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk. Repeat.

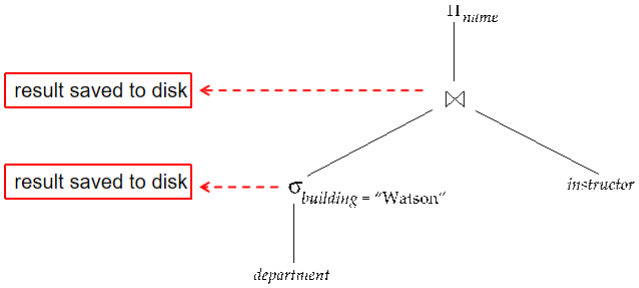
- Pipelining(流水线): pass on tuples to parent operations even as an operation is being executed

Materialization:

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building="\text{Watson}"}(department)$$

then compute the store its join with *instructor,* and finally compute the projection on *name*.



- costly if writing and reading are frequent
- always applicable
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
- temporary files are used for intermediate results

Pipelining:

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building="\text{Watson}"}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass **tuples** directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

- no temporary files are used for intermediate results, and a tuple is passed once generated
- Implementation of demand-driven pipelining: Each operation is implemented as an iterator implementing the following operations
  - open()
    E.g. file scan: initialize file scan; state: pointer to beginning of file
    E.g. merge join: sort relations;; state: pointers to beginning of sorted relations
  - next()
    E.g. for file scan: Output next tuple, and advance and store file pointer
    E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
  - close()
- Implementation of producer-driven pipelining: A buffer is used to hold tuples being passed
  - each operation executes the input from the input buffer and remove the input
  - each operation put the output in the output buffer
- Some algorithms are not able to output results even as they get input tuples
  - **blocking operations**: those cannot output while processing
  - otherwise, the operations are called **pipelined**
  - **Double-pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples

Query Optimization

Steps in cost-based query optimization

- Generate logically equivalent expressions using equivalence rules
- Annotate resultant expressions to get alternative query plans
- Choose the cheapest plan based on estimated cost

Estimation of plan cost based on:

- Statistical information about relations. Examples:
  number of tuples, number of distinct values for an attribute

- Statistics estimation for intermediate results （Cardinality Estimation）
  to compute cost of complex expressions
- Cost formulae for algorithms, computed using statistics

Generating equivalent expressions

- Perform projections as early as possible
- Perform selections as early as possible
- Perform more restrictive operations first
- Special cases

Cost Estimation

Statistical Information for Cost Estimation:

- n_r: number of tuples in a relation r.
- b_r: number of blocks containing tuples of r.
- l_r: size of a tuple of r.
- f_r: blocking factor of r — i.e., the number of tuples of r that fit into one block.
- $V(A, r)$: number of distinct values that appear in r for attribute A; same as the size of $\Pi_A(r)$.
- If tuples of r are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

- Histograms
- Probability
- Bound Estimation
- Machine Learning

Choice of Evaluation Plans

- Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm
- Practical query optimizers incorporate elements of the following two broad approaches:
  - Search all the plans and choose the best plan in a cost-based fashion.
    - An interesting sort order is a particular sort order of tuples that could be useful for a later operation
    - Sort order may also be useful for order by and for grouping
    - Must find the best join order for each subset, for each interesting sort order
    - Physical equivalence rules: majorly depend on adequate space
  - Uses heuristics to choose a plan.

Join Order Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way
        of accessing S  /* Using selections on S and indices on S */
    else for each non-empty subset S1 of S such that S1 ≠ S
        P1= findbestplan(S1)
        P2= findbestplan(S - S1)
        A = best algorithm for joining results of P1 and P2
        cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                join results of P1 and P2 using A"
    return bestplan[S]
```

* Some modifications to allow indexed nested loops joins on relations that have selections (see book)

> still expensive

Heuristic optimization

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.
- Many optimizers considers only left-deep join orders.
    - Reduces optimization complexity and generates plans amenable to pipelined evaluation
- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
    - Frequently used approach
        - heuristic rewriting of nested block structure and aggregation
        - followed by cost-based join-order optimization for each block
    - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
    - Optimization cost budget to stop optimization early (if cost of plan is less than cost of optimization)
    - Plan caching to reuse previously computed plan if query is resubmitted
- Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Additional Optimization Techniques

Optimizing Nested Subqueries

- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- A temporary relation is created

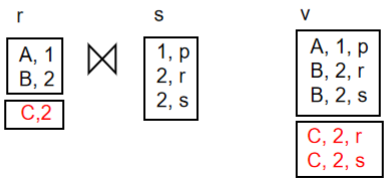
  In general, SQL queries of the form below can be rewritten as shown

  - Rewrite: **select** …
    from L_1
    where P_1 **and exists** (**select** *
    from L_2
    where P_2)
  - To: **create table** t_1 as
    select distinct V
    from L_2
    where P_2^1

    select …
    from L_1, t_1
    where P_1 **and** P_2^2
    - P_2^1 contains predicates in P_2 that do not involve any correlation variables
    - P_2^2 reintroduces predicates involving correlation variables, with relations renamed appropriately
    - V contains all attributes used in predicates with correlation variables
- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.

Materialized Views

- Materializing the view would be very useful if certain information is required frequently
- The task of keeping a materialized view up-to-date with the underlying data is known as materialized view maintenance
- incremental view maintenance: **Changes to database relations are used to compute changes to the materialized view, which is then updated**
  - The changes (inserts and deletes) to a relation or expressions are referred to as its differential（差分）
  - - Consider the materialized view $v = r \bowtie s$ and an update to r
    - Let r^{old} and r^{new} denote the old and new states of relation r
    - Consider the case of an insert to r:
      - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
      - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
      - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
    - Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
    - Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

  - For selection and projection:

Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more difficult operation
  - $R = (A,B)$, and r(R) = { (a,2), (a,3)}
  - $\Pi_A(r)$ has a single tuple (a).
  - If we delete the tuple (a,2) from r, we should not delete the tuple (a) from $\Pi_A(r)$, but if we then delete (a,3) as well, we should delete the tuple
- For each tuple in a projection $\Pi_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\Pi_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\Pi_A(r)$
    - if the count becomes 0, we delete the tuple from $\Pi_A(r)$

- For aggregation operations:
  - update
  - maintain indirect information
- Issues
  - **Materialized view selection**:
    "What is the best set of views to materialize?".
  - **Index selection**:
    "what is the best set of indices to create"
    - closely related, to materialized view selection
      - but simpler
  - Materialized view selection and index selection based on typical system **workload** (queries and updates)
    - Typical goal: minimize time to execute workload , subject to constraints on space and time taken for some critical queries/updates
    - One of the steps in database tuning
      - more on tuning in later chapters
  - Commercial database systems provide tools (called "tuning assistants" or "wizards") to help the database administrator choose what indices and materialized views to create

Transaction

- ACID
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Concurrent

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - increased processor and disk utilization, leading to better transaction throughput
  - reduced average response time for transactions: short transactions need not wait behind long ones.
- Anomalies in Concurrent Executions

- Lost Update（丢失修改）

T1	T2
Read A(100)	
	Read A(100)
A = A - 1	
	A = A - 1
Write A(99)	
	Write A(99)

- Dirty Read（读脏数据）

T1	T2
Read A(100)	
A = A - 1	
Write A(99)	
	Read A(99)
	A = A - 1
rollback	
	Write A(98)
	commit

- Unrepeatable Read （不可重复读）

T1	T2
Read A(100)	
	Raed A(100)
	A = A - 1
	Write A(99)
Read A(99)	

- Phantom Problem（幽灵问题 ）

T1	T2

T1	T2
select * from student where age=18(100 records with age=18)	
	insert into student(id, gender, age) values('008','M', 18)
select * from student where age=18(*101* records with age=18 !!!)	

Serializability

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  - conflict serializability (冲突可串行化)
    - same order of the conflict operation pairs
  - view serializability（视图可串行化）
    - same data content of the corresponding read operations
    - A schedule S is view serializable if it is view equivalent to a serial schedule
    - **Every conflict serializable schedule is also view serializable**
    - Every view serializable schedule that is not conflict serializable has **blind writes** (no reads before)
    - Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
      1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
      2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
      3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

    Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

    1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
    2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
    3. I_i = **write**(Q), I_j = **read**(Q). They conflict
    4. I_i = **write**(Q), I_j = **write**(Q). They conflict

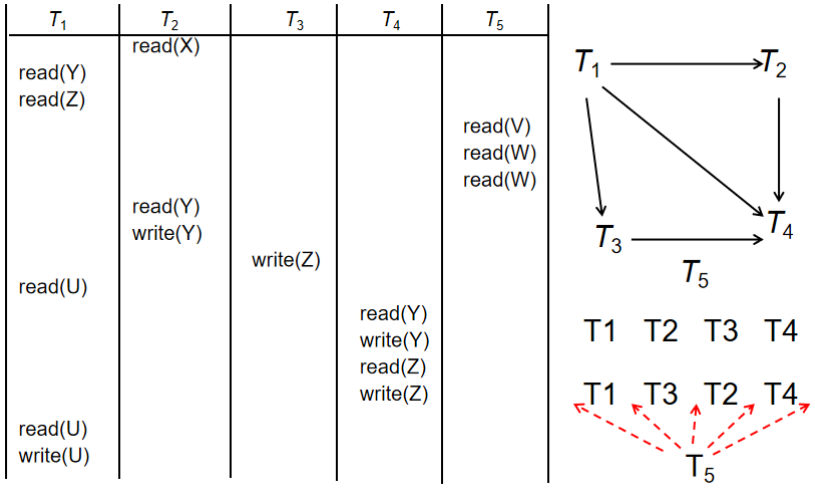
    Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.

    - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Test:

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph（前驱图）** — a directed graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

- A schedule is conflict serializable if and only if its precedence graph is acyclic
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph



> A feasible solution: 5 -> 1 -> 3 -> 2 -> 4

- Recoverable Schedules
  - The view reading the dirty data should not commit before the view accessing the data commits
- Cascading Rollbacks
  - a single transaction failure leads to a series of transaction rollbacks
  - Can lead to the undoing of a significant amount of work
- Cascadeless schedules
  - Every cascadeless schedule **is also recoverable**
  - **Reading dirty data is forbidded**
- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- Different transaction isolation levels may be adopted for performance
- Concurrency Control Protocols
  - Lock-Based
  - Timestamp-Based
  - Validation-Based

Concurrency Control

Lock-Based

- exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
- shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.
- compatibility:

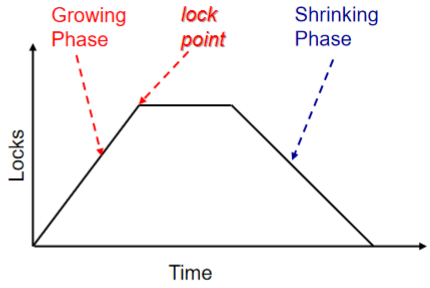
	S	X
S	true	false
X	false	false

  - any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- Pitfalls:
  - deadlock
  - starvation
- Lock is acquired automatically

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks

lock-S(A);
read *(A)*;
lock-S(B); ← ***lock point***
read *(B)*;
unlock*(A)*;
unlock(B);
display*(A+B)*



> Two-Phase Locking Protocol assures serializability but not recoverability
>
> Two-phase locking does not ensure freedom from deadlocks

Extension:

- Strict two-phase locking（严格两阶段封锁）: a transaction must **hold all its exclusive locks** till it commits/aborts.
  - Ensures recoverability and avoids cascading roll-backs
- Rigorous two-phase locking（强两阶段封锁）: a transaction must **hold all locks** till commit/abort.
  - Transactions can be serialized in the order in which they commit.
- Lock Conversions:
  - First Phase:
    - can acquire a lock-S or lock-X on a data item

        - can convert a lock-S to a lock-X (lock-upgrade)
    - Second Phase:
        - can release a lock-S or lock-X
        - can convert a lock-X to a lock-S (lock-downgrade)

Deadlock

- Deadlock prevention protocols ensure that the system will never enter into a deadlock state. Some prevention strategies :
    - Require that each transaction locks all its data items before it begins execution (predeclaration).
    - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
- Timeout-Based Schemes:
    - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
    - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.
- When deadlock is detected :
    - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
    - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
- More effective to roll back transaction only as far as necessary to break deadlock.
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

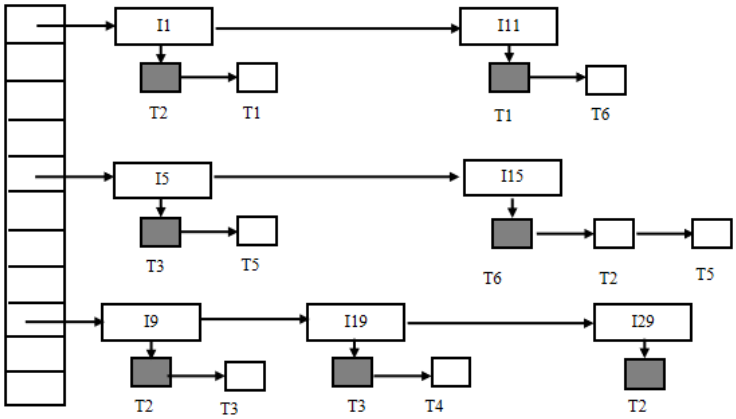
Example:

Following figure shows an example of a lock table. There are 6 transactions(T1-T6) and 6 data items(I1, I11, I5, I15, I9, I19, I29). Granted locks are filled (black) rectangles, while waiting requests are empty rectangles.

a) Which transaction are involved in deadlock?

b) To break the deadlock, which transaction should be rolled back?

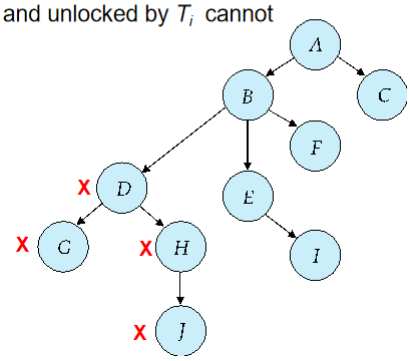
c) What kind data structure can be helpful for a transaction to efficiently release all locks its holds when it commits?



Tree Protocol & Multi-Granularity

- a simple kind of graph protocol
- The tree protocol ensures conflict serializability as well as freedom from deadlock.

1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



Advantages

- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
- protocol is deadlock-free
  - no rollbacks are required

Disadvantages

- Protocol does not guarantee recoverability or cascade freedom
  - Need to introduce commit dependencies to ensure recoverability
- Transactions may have to lock more data items than needed.
  - increased locking overhead, and additional waiting time
  - potential decrease in concurrency

Multiple granularity:

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity（细粒度）** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity（粗粒度）** (higher in tree): low locking overhead, low concurrency

Extra locks for multiple granularity:

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared (IS)***: indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive (IX):*** indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive (SIX):*** the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check ***all*** descendent nodes.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

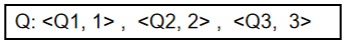
- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by **T_i in S or IS** mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by **T_i in X, SIX, or IX** mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Intention: may do something in the future but not now

Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, Q—must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

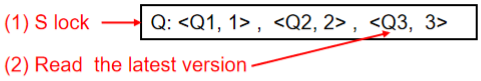
Multiversion

- Multiversion schemes keep old versions of data item to increase concurrency.
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- Read only transactions never have to wait as an appropriate version is returned immediately for every read operation..

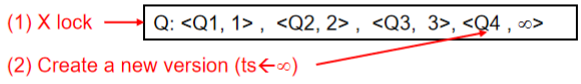


Update transaction:

- When an update transaction wants to read a data item:
  - it obtains a S lock on it, and reads the latest version.

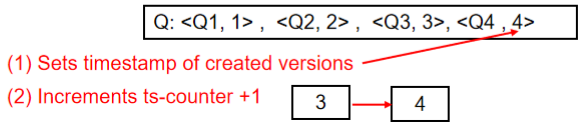


- When an update transaction wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.



(2) Create a new version (ts←∞)

- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1
  - Unlock all locks it holds

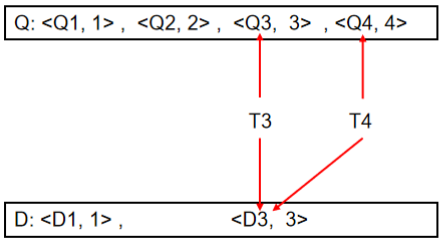


(1) Sets timestamp of created versions

(2) Increments ts-counter +1

Read-only transaction:

- When a read-only transaction T_i issues a read(Q), the value returned is the contents of the version whose timestamp is the largest timestamp less than or equal to TS(T_i)



- Only serializable schedules are produced.

Obsolete versions should be garbage collected

Recovery System

Failure:

- transaction failure
- system crash
- disk failure

Log-Based

- To ensure atomicity despite failures, we first output information describing the modifications(**log**) to stable storage without modifying the database itself.
- A **log** is kept on **stable storage(稳定存储器).**
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a **"start"** log record:
  $<T_i$ start>
- *Before* T_i executes **write**(X), writing **"update "** log record
  $<T_i, X, V_1, V_2>$
  where V_1 is the value of X before the write (the **old value**),
  V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, writing **"commit"** log record:
  $<T_i$ commit>
- When T_i complete rollback, writing **"abort"** log record:
  $<T_i$ abort>

- Write-Ahead Logging or WAL(先写日志原则)
- used with strict 2PL locking
- A transaction is said to have committed when its **commit log record is output to stable storage**
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

- **Undo** of a log record **$<T_i, X, V_1, V_2>$** writes the **old** value V_1 to X
- **Redo** of a log record **$<T_i, X, V_1, V_2>$** writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - each time a data item X is restored to its old value V a special log record **$<T_i, X, V>$** is written out – **compensation log(补偿日志)**
    - when undo of a transaction is complete, a log record **$<T_i$ abort>** is written out.
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Recover from the log:

- When recovering after failure:
  - Transaction T_i needs to be **undone** if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be **redone** if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>

- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**（重复历史）
    - Seems wasteful, but simplifies recovery greatly

> Problem: redoing/undoing the log can be time-consuming

Add **checkpoints**:

Streamline recovery procedure by periodically performing **checkpointing**

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.

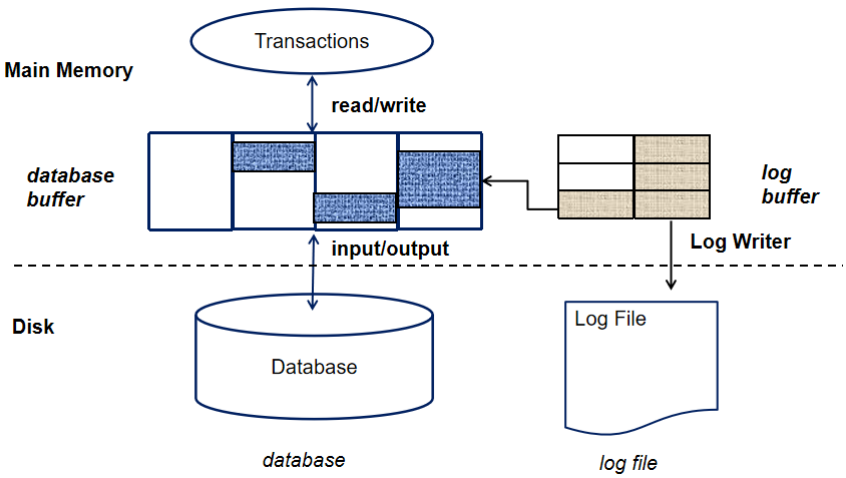
- ***All updates are stopped while doing checkpointing!!!***

- Only transactions that are in *L* or started after the checkpoint need to be redone or undone
- Some earlier part of the log may be needed for undo operations

Summary:

> - Logging (during normal operation)
> - Transaction rollback (during normal operation)
> - Recovery from failure
>   - Redo phase
>   - Undo phase

Log Buffer & Database Buffer



- Log record buffering: log records are buffered in main memory, instead of of being output directly to stable storage.
    - Log records are output to stable storage when a block of log records in the buffer is full, or a log force operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Group commit : several log records can be output using a single output operation, reducing the I/O cost.
- Rules:
    - Log records are output to stable storage in the order in which they are created.
    - Transaction T_i enters the commit state only when the log record <T_i ***commit***> has been output to stable storage
    - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
- To output a block to disk
    - First acquire an exclusive latch on the block; Ensures no update can be in progress on the block
    - Then perform a log flush
    - Then output the block to disk
    - Finally release the latch on the block

> To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing

- **Fuzzy checkpointing** is done as follows:
    1. Temporarily stop all updates by transactions
    2. Write a <**checkpoint** *L*> log record and force log to stable storage
    3. Note list *M* of modified buffer blocks
    4. Now permit transactions to proceed with their actions
    5. Output to disk all modified buffer blocks in list *M*
        - blocks should not be updated while being output
        - Follow WAL: all log records pertaining to a block must be output before the block is output
    6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

- When recovering using a fuzzy checkpoint, start scan from the checkpoint record pointed to by last_checkpoint
    - Log records before last_checkpoint have their updates reflected in database on disk, and need not be redone.
    - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

> Failure with Loss of Nonvolatile Storage

- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    1. Output all log records currently residing in main memory onto stable storage.
    2. Output all buffer blocks onto the disk.
    3. Copy the contents of the database to stable storage.
    4. Output a record <**dump**> to log on stable storage.

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump;
  - known as fuzzy dump or online dump
  - Similar to fuzzy checkpointing

Recovery with Early Lock Release and Logical Undo Operations

- Some operations may be hard to recover using a physical undoing, like an insertion causing splitting

Logical Undo Logging

- Operations like B+-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B+-tree.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called **logical undo logging**, in contrast to **physical undo logging**
    - Operations are called **logical operations**
  - Other examples:
    - delete of tuple, to undo insert of tuple
      - allows early lock release on space allocation information
    - subtract amount deposited, to undo deposit
      - allows early lock release on bank balance

Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo

- Logical redo is very complicated since database state on disk may not be “operation consistent” when recovery starts
- Physical redo logging does not conflict with early lock release

Operation logging is done as follows:

1. When operation starts, log $<T_i, O_j,$ **operation-begin**$>$. Here O_j is a unique identifier of the operation instance.
2. While operation is executing, normal log records with physical redo and physical undo information are logged.
3. When operation completes, $<T_i, O_j,$ **operation-end**, $U>$ is logged, where U contains information needed to perform a logical undo information.

- If crash/rollback occurs before operation completes:
  - the operation-end log record is not found, and
  - the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes:
  - the operation-end log record is found, and in this case
  - logical undo is performed using U; the physical undo information for the operation is ignored.

Rollback of transaction T_i is done as follows:

- Scan the log backwards
  1. If a log record $<T_i, X, V_1, V_2>$ is found, perform the undo and log $<T_i, X, V_1>$.
  2. If a $<T_i, O_j,$ **operation-end**, $U>$ record is found
     - Rollback the operation logically using the undo information U.
       - Updates performed during roll back are logged just like during normal operation execution.
       - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record
         $<T_i, O_j,$ **operation-abort**$>$.
     - Skip all preceding log records for T_i until the record $<T_i, O_j$ **operation-begin**$>$ is found

- Transaction rollback, scanning the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a $<T_i, O_j,$ **operation-abort**$>$ record is found:
     - skip all preceding log records for T_i until the record $<T_i, O_j,$ **operation-begin**$>$ is found.
  5. Stop the scan when the record $<T_i,$ start$>$ is found
  6. Add a $<T_i,$ **abort**$>$ record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

ARIES

- ARIES is a state-of-the-art recovery method
  - Uses log sequence number (LSN) to identify log records
  - Physiological redo
  - Dirty page table to avoid unnecessary redos during recovery
  - Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time
- data structures
  - log sequence number (LSN)

- Page LSN
- Log records of several different types
- Dirty page table

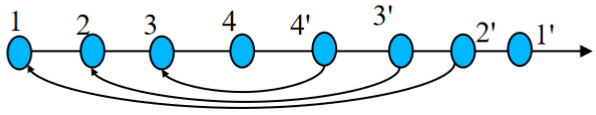
Log Record:

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo

  - LSN in log record may be implicit
- Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in earlier recovery algorithm
  - Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone
    - Required to avoid repeated undo of already undone actions

LSN	TransID	UndoNextLSN	RedoInfo



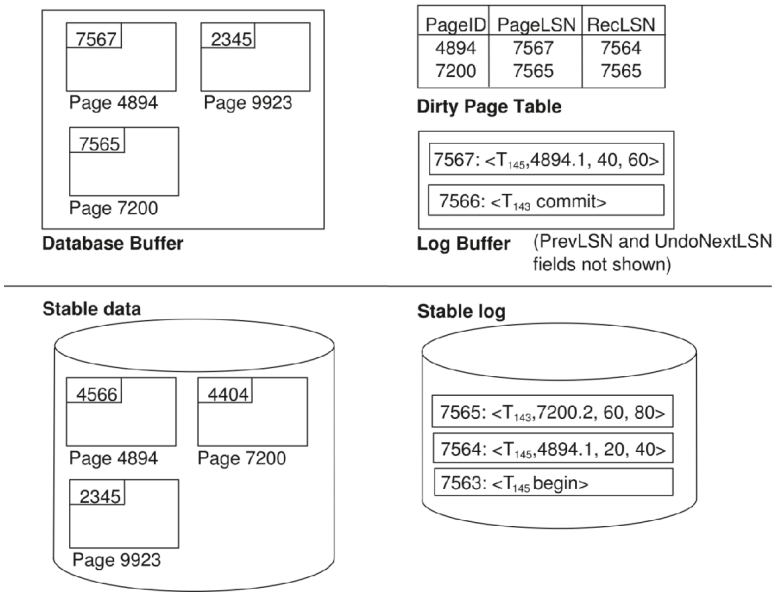
Dirty Page Table:

DirtyPageTable

- List of pages in the buffer that have been updated
- Contains, for each such page
  - **PageLSN** of the page
  - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
    - Set to current end of log when a page is inserted into dirty page table (just before being updated)
    - Recorded in checkpoints, helps to minimize redo work

Checkpoint Log:

- **Checkpoint log record**
  - Contains:
    - DirtyPageTable and list of active transactions
    - For each active transaction, LastLSN, the LSN of the last log record written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time
  - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently



ARIES Recovery Algorithm (3 passes):

- Analysis pass
- Redo pass
- Undo pass

Object-Based

- Advanced Database Applications need Complex Data Types
- Extend the relational data model by including object orientation and constructs to deal with added data types.

Extensions to SQL to support complex types include:

- Collection types
  - Nested relations are an example of collection types
- Structured types
  - Nested record structures like composite attributes
- Inheritance
- Object orientation
  - Including object identifiers and references
- Large object types
  - BLOB, CLOB

> final and not final indicate whether subtypes can be created

```
create type CustomerType as (
    name Name,
    address Address,
    dateOfBirth date,
    method ageOnDate (onDate date)
    returns interval year
```

```
)  not final

create type Student
under Person
(degree     varchar(20),
 department  varchar(20))

create type Book as
     (title                 varchar(20),
      author-array   varchar(20) array [10],
      pub-date         date,
      publisher        Publisher,
      keyword-set   varchar(20) multiset )
create table books of Book
```

Methods（方法）

- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)
  returns interval year
  for *CustomerType*
  begin
  return *onDate* - **self**.*dateOfBirth*;
  end

- We can now find the age of each customer:

  select *name.lastname*, *ageOnDate* (**current_date**)
  from *customer*

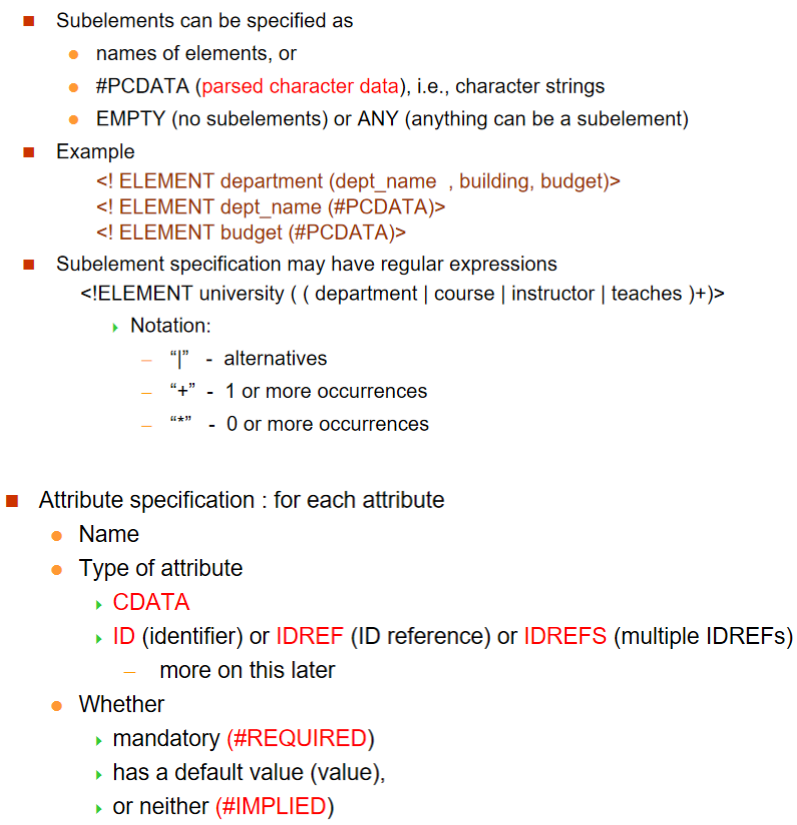
XML

- XML: **Extensible Markup Language**
- Users can add new tags, and separately specify how the tag should be handled for display
- Much of the use of XML has been in **data exchange applications**
  - specify new tags
  - create nested tag structures
- Tags make data (relatively) **self-documenting**
- for computers
- XML type specification languages to specify the syntax
  - DTD (Document Type Descriptors)
  - XML Schema
- Specifying a unique string (namespace) as an element name avoids confusion: use unique-name:element-name
- **Every document must have a single top-level element**
- Suggestion:

- use attributes for identifiers of elements
- use subelements for contents

Document Type Definition (DTD)

- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes)>



- An element can have at most one attribute of type ID
  - The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

```
<!DOCTYPE university-3 [
        <!ELEMENT university ( (department|course|instructor)+)>
        <!ELEMENT department ( building, budget )>
        <!ATTLIST department  dept_name ID #REQUIRED >
        <!ELEMENT course (title, credits )>
        <!ATTLIST course
```

```
            course_id ID #REQUIRED
            dept_name IDREF #REQUIRED
            instructors IDREFS #IMPLIED >
        <!ELEMENT instructor ( name, salary )>
        <!ATTLIST instructor
            IID ID #REQUIRED
            dept_name IDREF #REQUIRED
            salary CDATA #IMPLIED>
]>
```

Disadvantage:

- No typing of text elements and attributes
- Difficult to specify unordered sets of subelements
- IDs and IDREFs are untyped

XML Schema

- Typing of values
- User-defined, complex types
- Many more features, including: uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
- Standard XML querying/translation languages
    - XPath: Simple language consisting of path expressions
    - XQuery: An XML query language with a rich set of features

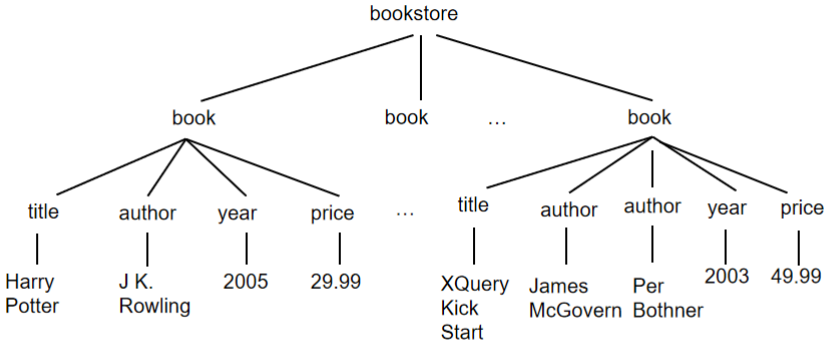
Restriction on simple types

Restriction	Description
enumeration	可接受值的一个列表(枚举)
fractionDigits	最大的小数位数。
length	字符或者列表项目的精确数目。
maxExclusive	数值的上限。所允许的值必须小于此值。
maxInclusive	数值的上限。所允许的值必须小于或等于此值。
maxLength	字符或者列表项目的最大数目。
minExclusive	数值的下限。所允许的值必需大于此值。
minInclusive	数值的下限。所允许的值必需大于或等于此值。
minLength	字符或者列表项目的最小数目。
pattern	可接受的字符串的精确序列。
totalDigits	所允许的阿拉伯数字的精确位数。
whiteSpace	空白字符（换行、回车、空格以及制表符）的处理方式。

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
     <xs:complexType>
          <xs:sequence>
```

```
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
….
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Tree model of XML documents



/bookstore/book

/bookstore/book[1]/title

/bookstore/book[price>30]/title

> XPath expressiona are on the bottom

XPath:

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by "/"
  - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g. /university-3/instructor/name evaluated on the university-3 data returns
  <name>Srinivasan</name>
  <name>Brandt</name>
- E.g. /university-3/instructor/name/text( )
  returns the same names, but without the enclosing tags

- Selection predicates may follow any step in a path, in [ ]
- Attributes are accessed using "@"
- XPath provides several functions
- Boolean connectives and and or and function not() can be used in predicates

- IDREFs can be referenced using function id()
- Operator "|" used to implement union
- "//" can be used to skip multiple levels of nodes
- doc(name) returns the root of a named document

XQuery:

- XQuery is a general purpose query language for XML data
- XQuery syntax (FLWOR) : `for … let … where … order by …return …`

FLWOR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWOR expression in XQuery
  - find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

  - Items in the **return** clause are XML text unless enclosed in {}, in which case they are evaluated
- Let clause not really needed in this query, and selection can be done In XPath. Query can be written as:

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

- Alternative notation for constructing elements:

```
return element course_id { $x/@course_id }
```

XSLT:

- A stylesheet stores formatting options for a document, usually separately from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The XML Stylesheet Language (XSL) was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language

Application Program Interface

- There are two standard application program interfaces to XML data:
  - **SAX** (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
  - **DOM** (Document Object Model)
    - **XML** data is parsed into a tree representation
    - Variety of functions provided for traversing the DOM tree
    - E.g.: Java DOM API provides Node class with methods
      getParentNode( ), getFirstChild( ), getNextSibling( )
      getAttribute( ), getData( ) (for text node)
      getElementsByTagName( ), …
    - Also provides functions for updating DOM tree

- Publishing: process of converting relational data to an XML format
- Shredding: process of converting an XML document into a set of tuples to be inserted into one or more relations