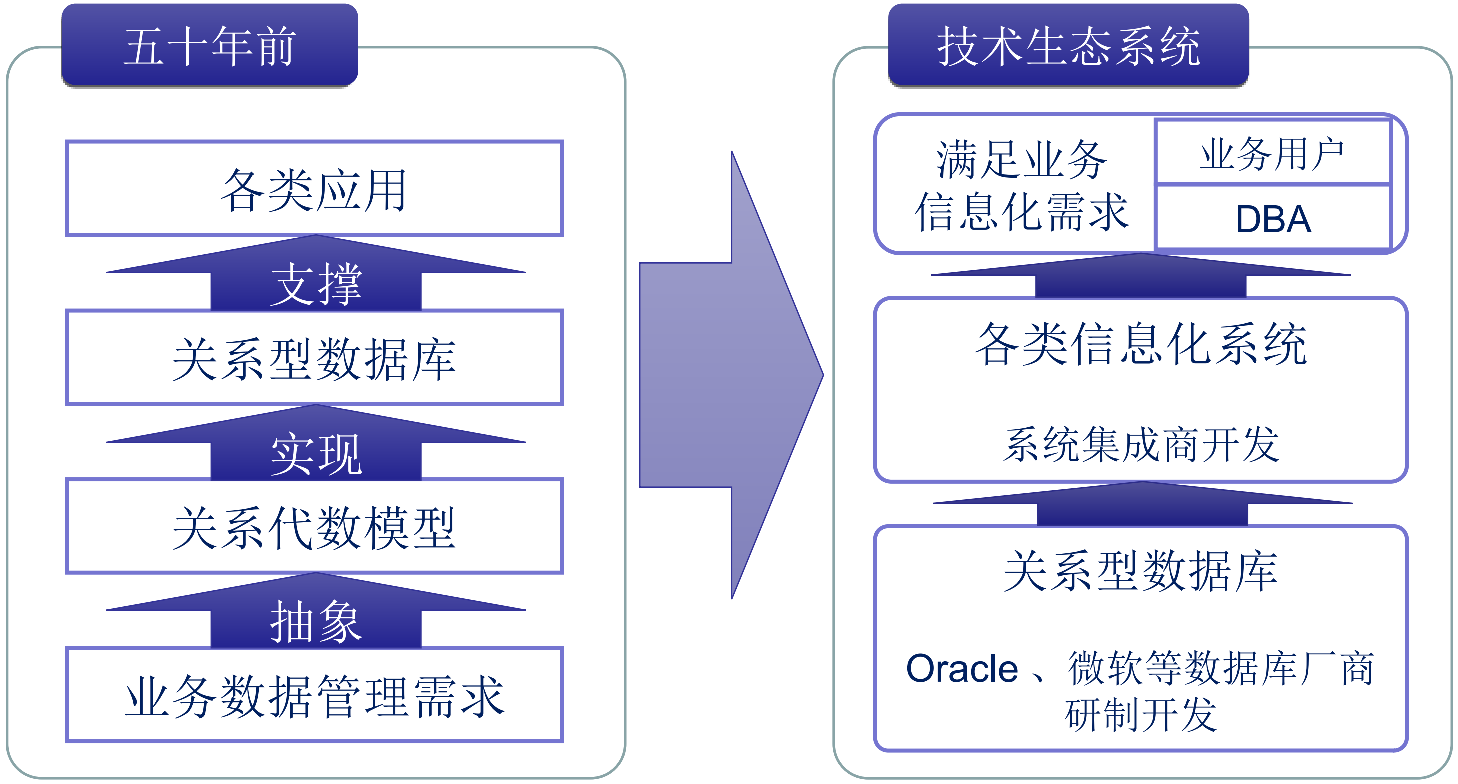
Database System

Review

Gang Chen(陈刚)
College of Computer Science
Zhejiang University
Spring & Summer 2023
18667106736 / cg@zju.edu.cn

数据库的由来


五十年前
各类应用
支撑
关系型数据库
实现
关系代数模型
抽象
业务数据管理需求
技术生态系统
满足业务信息化需求
业务用户
DBA
各类信息化系统
系统集成商开发
关系型数据库
Oracle 、微软等数据库厂商研制开发

Characteristics of DBMS

- Efficiency and scalability in data access.
- Reduced application development time.
- Data independence (including physical data independence and logical data independence).
- Data integrity and security.
- Concurrent access and robustness (i.e., recovery).

Drawbacks of File-Processing System

- Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation — multiple files and multiple formats
  - Difficult to retrieve, difficult to share
- Integrity problems
  - Integrity constraints (e.g. account balance > 0) become part of program code
  - Hard to add new constraints or change existing ones

Drawbacks of File-Processing System (Cont.)

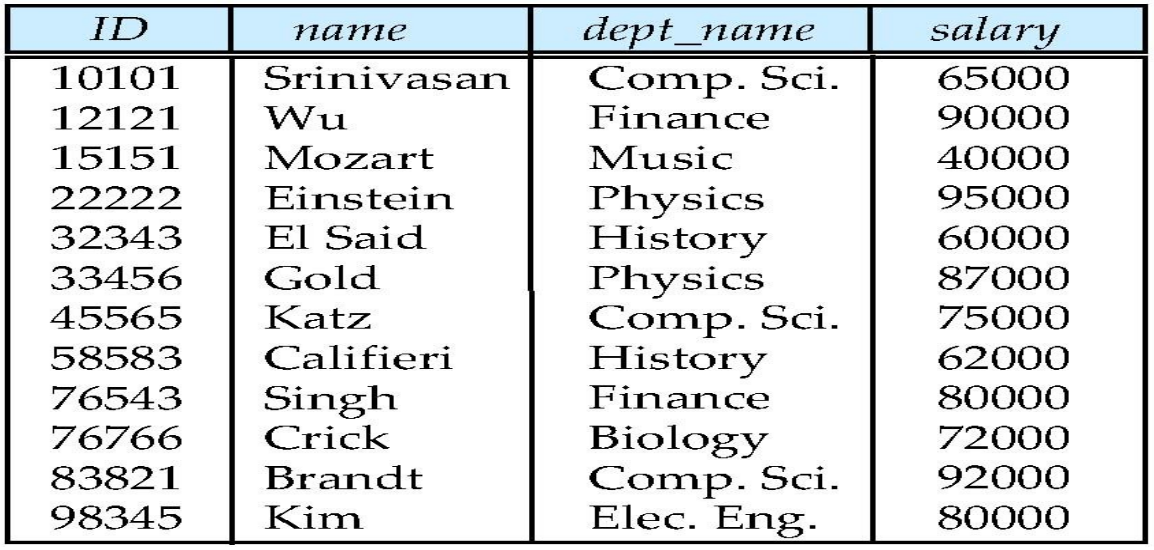
- No atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Difficult to concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
  - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems (i.e., Right person uses right data)
- Database systems offer solutions to all the above problems!

What is relational model

- The relational model is very simple and elegant.
- A relational database is a collection of one or more relations, which are based on the relational model.
- A relation is a table with rows and columns.
- The major advantages of the relational model are its straightforward data representation and the ease with which even complex queries can be expressed.
- Owing to the great language SQL, the most widely used language for creating, manipulating, and querying relational database.

Example of a Relation

A relation for **instructor**



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

- Cf.: relationship and relation
  - A relationship is an association among several entities.
  - A relation is the mathematical concept, referred to as a table.

Entity set and relationship set ←→ real world

Relation --- table,   tuple --- row ←→ machine world

Database

- A database consists of multiple relations
- Information about an enterprise (e.g., university) is broken up into parts
  - Instructor
  - Student
  - Advisor
- Bad design:

  univ (*instructor -ID, name, dept_name, salary, student_Id*, ..)

  results in
  - Repetition of information (e.g., two students have the same instructor)
  - The need for null values (e.g., represent an student with no advisor)
- Normalization theory (Chapter 7) deals with how to design "good" relational schemas

Key (码 / 键)

- Let $K \subseteq R$
- *K* is a superkey (超码) of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r*(*R*)
  - E.g., both {*ID*} and {*ID*, *name*} are superkeys of the relation ***instructor***.
- *K* is a candidate key (候选码) if *K* is minimal superkey.
  - E.g., {*ID*} is a candidate key for the relation ***instructor***, since it is a superkey and no any subset.
- *K* is a primary key (主码), if *K* is a candidate key and is defined by user explicitly.
  - Primary key is usually marked by underline.

Foreign Key (外键 / 外码)

- Assume there exists relations *r* and *s*: *r*(<u>*A*</u>, *B*, *C*), *s*(<u>*B*</u>, *D*), we can say that attribute *B* in relation *r* is foreign key referencing *s*, and *r* is a referencing relation (参照关系), and *s* is a referenced relation (被参照关系).
  - E.g.1: 学生 (<u>学号</u> , 姓名 , 性别 , 专业号 , 年龄) --- 参照关系

    专业 (<u>专业号</u> , 专业名称) --- 被参照关系 （ 目标关系)

    其中属性专业号称为关系学生的外码 .
  - E.g.2: Account(<u>account-number</u>, branch-name, balance) --- referenced relation

    depositor (<u>customer-name</u>, account-number) --- referencing relation

参照关系中外码的值必须在被参照关系中实际存在，或为 null 。

- Primary key and foreign key are integrated constraints.

Fundamental Relational-Algebra Operations

- Six basic operators
  - Select 选择
  - Project 投影
  - Union 并
  - set difference 差（集合差）
  - Cartesian product 笛卡儿积
  - Rename 改名（重命名）

- The operators take one or two relations as inputs, and return a new relation as a result.

Additional Relational-Algebra Operations

- Four basic operators

Set intersection	交
Natural join	自然连接
Division	除
Assignment	赋值

- We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

Extended Relational-Algebra Operations

- Extended relational-algebra operators
  - Generalized Projection
  - Aggregate Functions

SQL Query

Consider the following relation schemas and then answer the subsequent problems. Note that the key attributes in the relation schemas are underlined.

***Student*(<u>*Sid*</u>, *Name*, *Age*)**
***Project*(<u>*ProjectName*</u>, <u>*Sid*</u>, *Score*)**

(1) Find the names of students who are in the project with project name ‘MiniSQL’.

$\Pi_{name}((student) \bowtie (\sigma_{projectName=\text{‘MiniSQL’}}(project)))$

(2) Find the names of students who are the youngest.

Method 1: $\Pi_{name}(student) - \Pi_{name}(\sigma_{student.age>st2.age}(student \times (\rho_{st2}(student))))$

Method 2: $\text{Temp} \leftarrow g_{\text{min}(Age)}(student);$

$\Pi_{name}(\sigma_{age=minage}(student \times (\rho_{\text{T}(minage)}(\text{Temp}))))$

Note: Other operations such as **Update**, **Group by**, **Avg** should be noticed.

SELECT

- The format of SELECT statement:

  SELECT <[DISTINCT] c_1, c_2, …>

  FROM <r_1, …>

  [WHERE <*condition*>]

  [GROUP BY <c_1, c_2, …> [HAVING <$cond_2$>]]

  [ORDER BY <c_1 [DESC] [, c_2 [DESC|ASC], …]>]

- The execution order of SELECT:

  From → where → group (aggregate) → having → select → distinct → order by

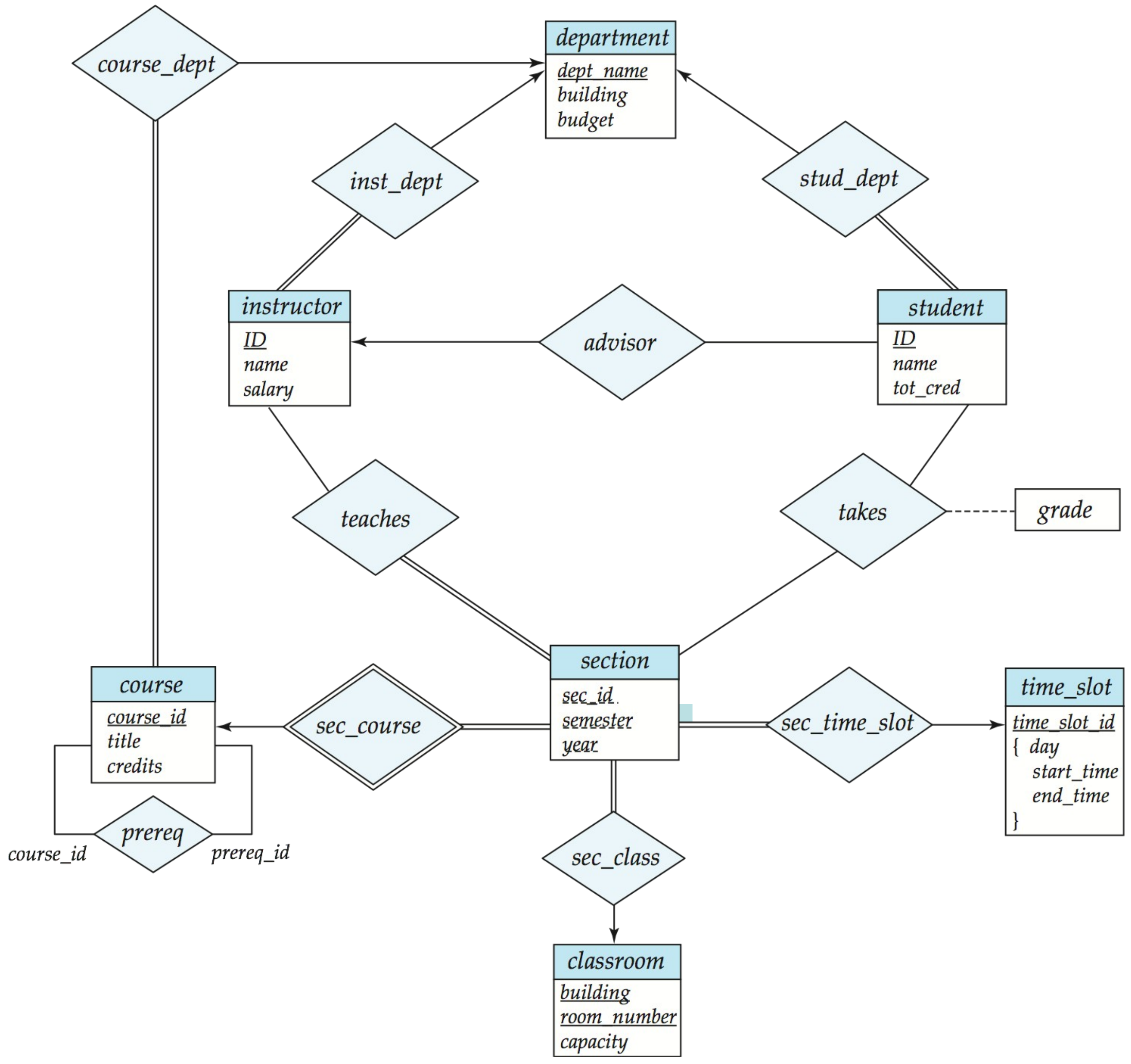
  - Note that predicates in the having clause are applied after the formation of groups, whereas predicates in the where clause are applied before forming groups.

- Find the names of all branches located in city Brooklyn where the average account balance is more than $1,200.

```
SELECT A.branch_name, avg(balance)
FROM account A, branch B
WHERE A.branch_name = B.branch_name and
          branch_city ='Brooklyn'
GROUP BY A.branch_name
HAVING avg(balance) > 1200
```

branch(*branch-name*, *branch-city*, *assets*)
account(*account-number*, *branch-name*, *balance*)

E-R Diagram for a University Enterprise



Representing Entity Sets as Tables

- A strong entity set => to a table with the same attributes.

- *customer*(*customer-id*, *cust-name*, *cust-street*, *cust-city*);
- *branch*(*branch-name*, *branch-city*, *assets*);
- *account*(*account-number*, *balance*);
- *loan*(*loan-number*, *amount*);
- *employee*(*employee-id*, *employee-name*, *phone*, *start-date*);

Entity Set with Composite Attributes

- Composite attributes are flattened out by creating a separate attribute for each component attribute.
  - E.g., given entity set *customer* with composite attribute *name* with component attributes *first-name* and *last-name*. The table corresponding to the entity set has two attributes:

    name.first-name and *name.last-name*

    or first-name and last-name

customer(*<u>customer-id</u>*, *first-name*, *last-name*, *cust-street*, *cust-city*)

Entity Set with Multivalued Attributes

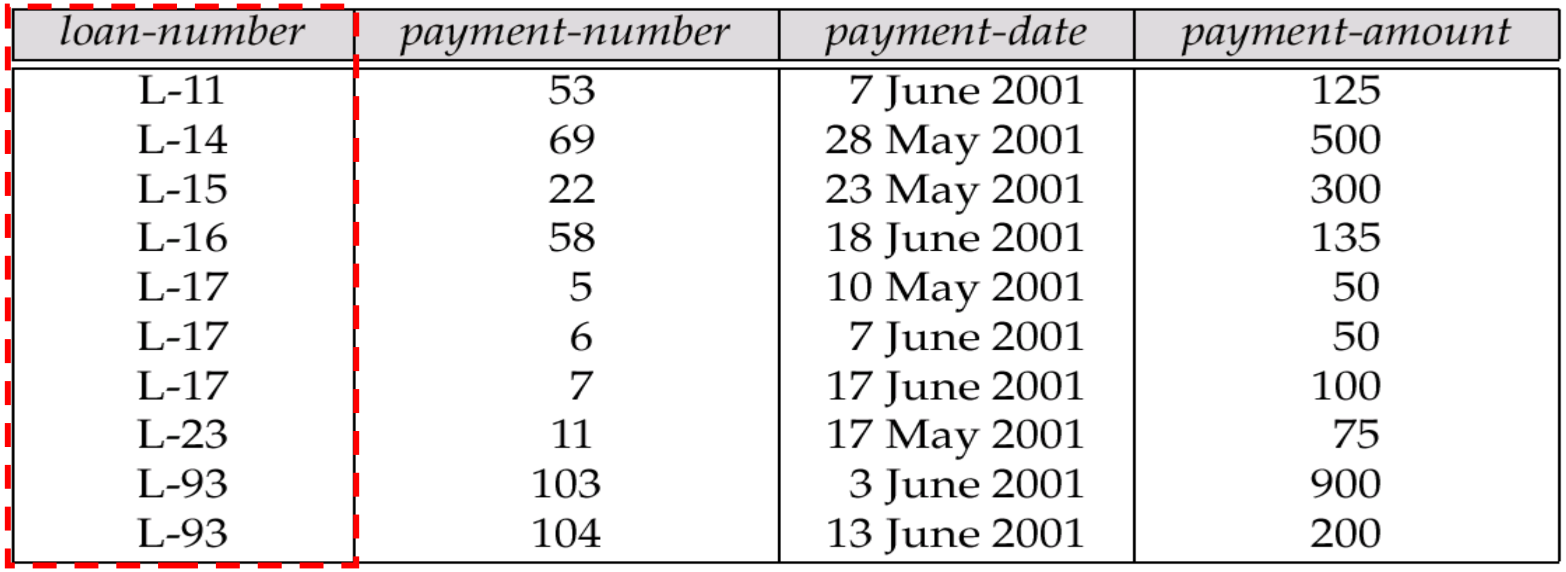
- A multivalued attribute *M* of an entity *E* is represented by a separate table *EM*:
  - E.g., multivalued attribute *dependent-names of employee* is represented by a table
  - *employee*(*emp-id*, *ename*, *sex*, *age*, *dependent-names*)

  - *employee*(*emp-id, ename, sex, age*),
  - *employee-dependent-names*(*emp-id*, *dependent-name*)
  - Each value of the multivalued attribute maps to a separate row of the table *EM*
    - E.g., an employee entity with primary key *John* and dependents *Johnson* and *Johndoter* maps to two rows:

      (*John*, *Johnson*) and (*John*, *Johndoter*) in relation *employee-dependent-names*.

Representing Weak Entity Sets

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set.



loan-number	*payment-number*	*payment-date*	*payment-amount*
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

payment(loan-number, payment-number, payment-date, payment-amount)

Representing Relationship Sets as Tables

- A relationship set is represented as a table with columns for the primary keys of the two participating entity sets, (which are foreign keys here) and any descriptive attributes of the relationship set itself.

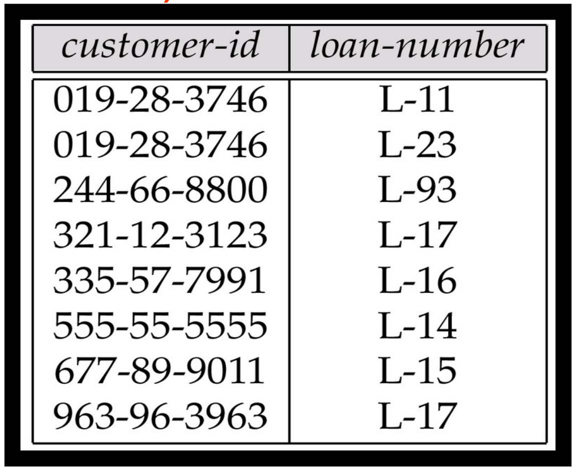
- (a) tables for many to many relationship sets:

  borrower(<u>*customer-id, loan-number*</u>);

  depositor(<u>*customer-id, account-number*</u>, *access-date*)

The two attributes corresponding to the two entity sets become the primary key of the table.

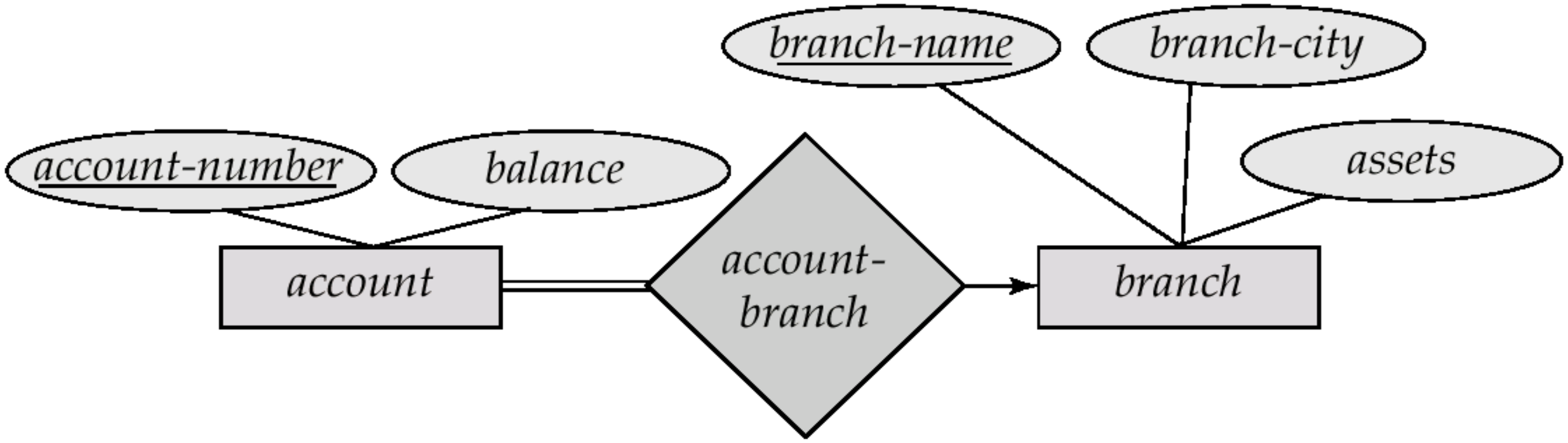
borrower



customer-id	*loan-number*
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

Redundancy of Tables

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the "many" side, containing the primary key of the one side (对 1:*n* 联系，可将“联系”所对应的表，合并到对应“多”端实体的表中).



account(*<u>account-number</u>*, *balance*); *branch*(*<u>branch-name</u>*, *branch-city*, *assets*);

account-branch(*<u>account-number</u>*, *branch-name*)

account(*<u>account-number</u>*, *branch-name*, *balance*)

Redundancy of Tables (Cont.)

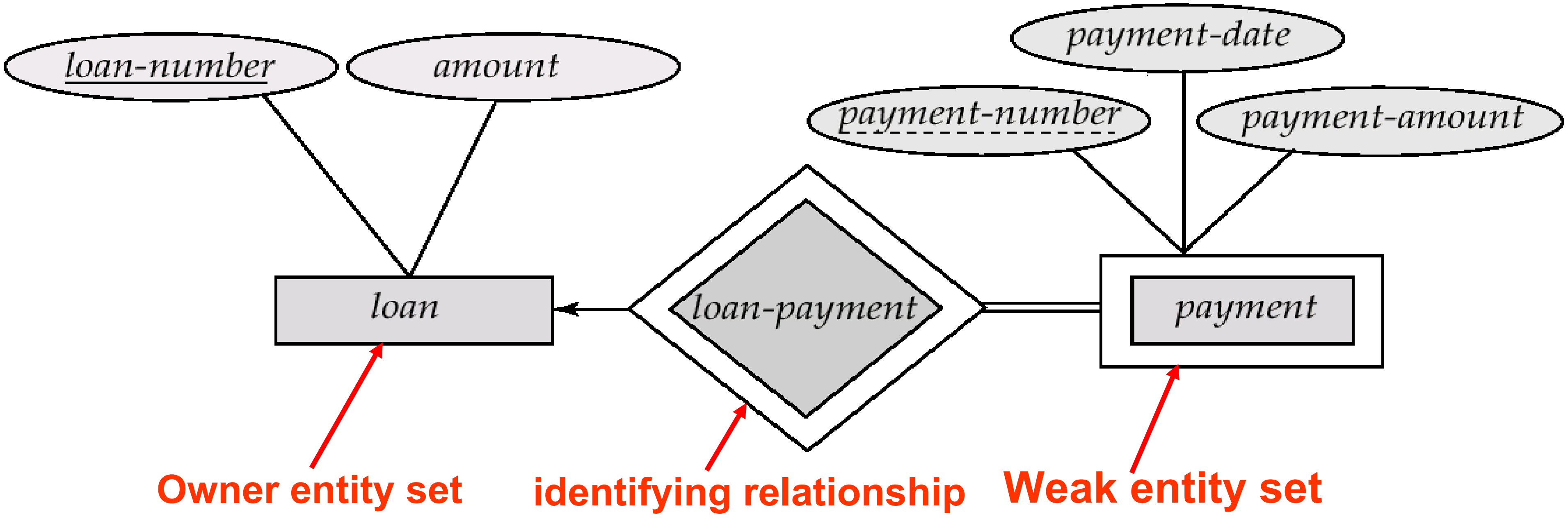
- If participation is partial on the many side, replacing a table by an extra attribute in the relation corresponding to the "many" side could result in null values.
  - E.g., *cust-banker*(*customer-id*, *employee-id*, *type*); 但有的 customer 没有 banker, 则合并之后得:

    Customer(*customer-id*, *cust-name*, *cust-street*, *cust-city*, *banker-id*, *type*) ，导致 *Customer* 中有些元组的 *banker-id* 、 type 为 null) 。

- For one-to-one relationship sets, either side can be chosen to act as the "many" side
  - That is, extra attribute can be added to either of the tables corresponding to the two entity sets.

Redundancy of Tables (Cont.)

- (b) The table corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant (联系弱实体集及其标识性实体集的联系集对应的表是冗余的，即对应 identifying relationship 的表是多余的。).
  - E.g., The payment table already contains the information that would appear in the loan-payment table (i.e., the columns loan-number and payment-number).



ER-model

The information includes:

matches (identified by match_id, with attributes location, time)

teams (identified by name, with attribute city)

players (identified by name, with attributes age and several phone numbers),

individual player statistics (such as score, shooting, foul, etc.) for each match

the score of each team for the match

Please answer the following questions:

1)Draw an *E-R diagram* for this database model with primary key underlined

2) Transform the E-R diagram into a number of *relational database schemas*, with the primary key underlined.

ER-model

Match(<u>match_id</u>, location, time)

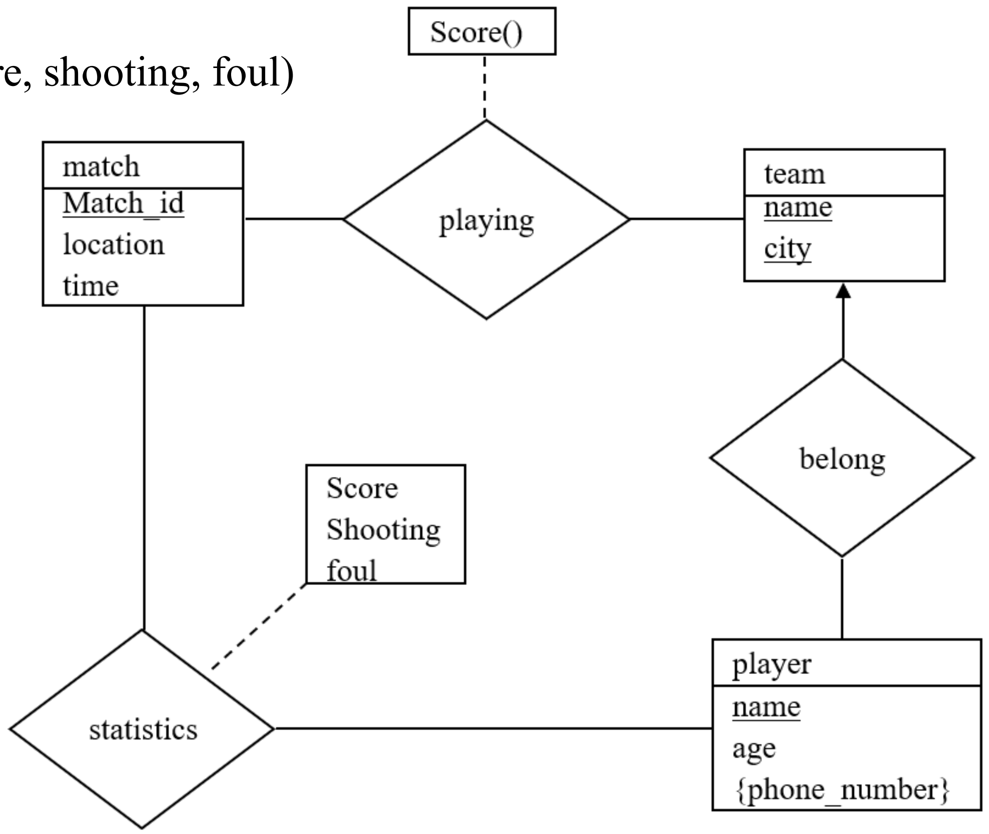
Statistics(match_id, player_name, score, shooting, foul)

Team(<u>name</u>, city)

Playing(<u>match_id, name</u>, score)

Player(<u>name</u>, age,team_name)

Phone(<u>player_name, phone_number</u>)



Boyce-Codd Normal Form

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*

Example schema *not* in BCNF:

instr_dept (*ID, name, salary, dept_name, building, budget*)

because *dept_name* $\rightarrow$ *building, budget*
holds on *instr_dept,* but *dept_name* is not a superkey

Decomposing a Schema into BCNF

- Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF. We decompose *R* into:
  - (αU β)
  - (*R* - (β - α))
- In our example,
  - α = *dept_name*
  - β = *building, budget*

  and *inst_dept* is replaced by
  - (αU β) = (*dept_name, building, budget*)
  - (*R* - (β - α)) = (*ID, name, salary, dept_name*)

BCNF Decomposition Algorithm

```
result := {R};
   done := false;
   compute $F^+$;
   while (not done) do
      if (there is a schema $R_i$ in result that is not in BCNF)
         then begin
            let $\alpha \rightarrow \beta$ be a nontrivial functional
            dependency that holds on $R_i$ such
            that $\alpha \rightarrow R_i$ is not in $F^+$, and $\alpha \cap \beta = \emptyset$;
            result := (result – $R_i$) ∪ $(\alpha, \beta)$ ∪ $(R_i – \beta)$;
                                $R_{i1}$      $R_{i2}$
              end
         else done := true;
```

It means there is nontrivial FD $\alpha \rightarrow \beta$ in R_i, and α is not a key.

将 R_i 分解为二个子模式：$R_{i1} = (\alpha, \beta)$ 和 $R_{i2} = (R_i – \beta)$, α 是 R_{i1} R_{i2} 的共同属

Note: Finally, every sub-schema is in BCNF, and the decomposition is lossless-join.

Example of BCNF Decomposition

- *class* (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)
- Functional dependencies:
  - *course_id*→ *title*, *dept_name*, *credits*
  - *building*, *room_number*→*capacity*
  - *course_id*, *sec_id*, *semester*, *year*→*building*, *room_number*, *time_slot_id*
- A candidate key {*course_id*, *sec_id*, *semester*, *year*}.
- BCNF Decomposition:
  - *course_id*→ *title*, *dept_name*, *credits* holds
    - but *course_id* is not a superkey.
  - We replace *class* by:
    - *course*(*course_id*, *title*, *dept_name*, *credits*)
    - *class-1* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number, capacity*, *time_slot_id*)

BCNF Decomposition (Cont.)

- *course* is in BCNF
  - How do we know this?
- *building*, *room_number*→*capacity* holds on *class-1*
  - but {*building*, *room_number*} is not a superkey for *class-1*.
  - We replace *class-1* by:
    - *classroom* (*building*, *room_number*, *capacity*)
    - *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
- *classroom* and *section* are in BCNF.

BCNF and Dependency Preservation

- Therefore, we cannot always satisfy all three design goals:
  - Lossless join
  - BCNF
  - Dependency preservation
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

Desirable properties of decomposition

- All attributes of an original schema (R) must appear in the decomposition (R_1, R_2): $R = R_1 \cup R_2$

- Lossless-join decomposition.

  For all possible relations r on schema R

  - $r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$
  - A decomposition of R into R_1 and R_2 is lossless-join if and only if at least one of the following dependencies are held in F^+ :
    - $\{R_1 \cap R_2\} \rightarrow R_1$
    - $\{R_1 \cap R_2\} \rightarrow R_2$

无损连接分解的条件： 分解后的二个子模式的共同属性必须是 R_1 或 R_2 的码（适用于一分为二的分解）。

Desirable properties of decomposition (Cont.)

- Dependency preservation （依赖保持）
  - To check updates (to ensure not violate any FD) efficiently, allow updates validation in sub-relations R_i respectively, without executing the join of them.
  - Restriction of F to R_i is: $F_i \subseteq F^+$, F_i includes only attributes of R_i
  - $(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$, where F_i be the set of dependencies in F^+ that include only attributes in R_i.

- No redundancy: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form, i.e., BCNF or 3NF.

$R = (A, B, C)$, $F = \{A \rightarrow B, B \rightarrow C\}$,
$R_1 = (A, B)$, $R_2 = (A, C)$

Third Normal Form （ 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.

Canonical Cover (正则覆盖)

- DBMS should always check to ensure not violate any Functional Dependency (FD) in *F*.
  - If the *F* is too big, the check is costly. Thus, we need simplify the set of FDs.

- Intuitively, a canonical cover of *F*, denoted by F_c, is a "minimal" set of FDs equivalent to *F*.
  - Having no redundant FDs and no redundant parts of FDs, i.e., no functional dependency in *Fc* contains an extraneous attribute.
  - Each left side is unique.
  - E.g., $\alpha_1 \rightarrow \beta_1$, $\alpha_1 \rightarrow \beta_2$, $\Rightarrow \alpha_1 \rightarrow \beta_1\beta_2$

$$\boxed{F_c \underset{\textbf{Imply}}{\overset{\textbf{Logically}}{==========}} F}$$

3NF Decomposition Algorithm

```
Let $F_c$ be a canonical cover for $F$;
$i$ := 0;
for each functional dependency $\alpha \rightarrow \beta$ in $F_c$ do
    {if none of the schemas $R_j$, $1 \leq j \leq i$ contains $\alpha\ \beta$
        then begin
            $i := i + 1$;
            ~~$R_i := (\alpha\ \beta)$~~
        end}
if none of the schemas $R_j$, $1 \leq j \leq i$ contains a candidate key for $R$ then
begin
    $i := i + 1$;
    $R_i$ := any candidate key for $R$;
end
return ($R_1$, $R_2$, ..., $R_i$)
```

将 F_c 中的每个 $\alpha \rightarrow \beta$ 分解为子模式 $R_i := (\alpha, \beta)$，从而保证 dependency-preserving.

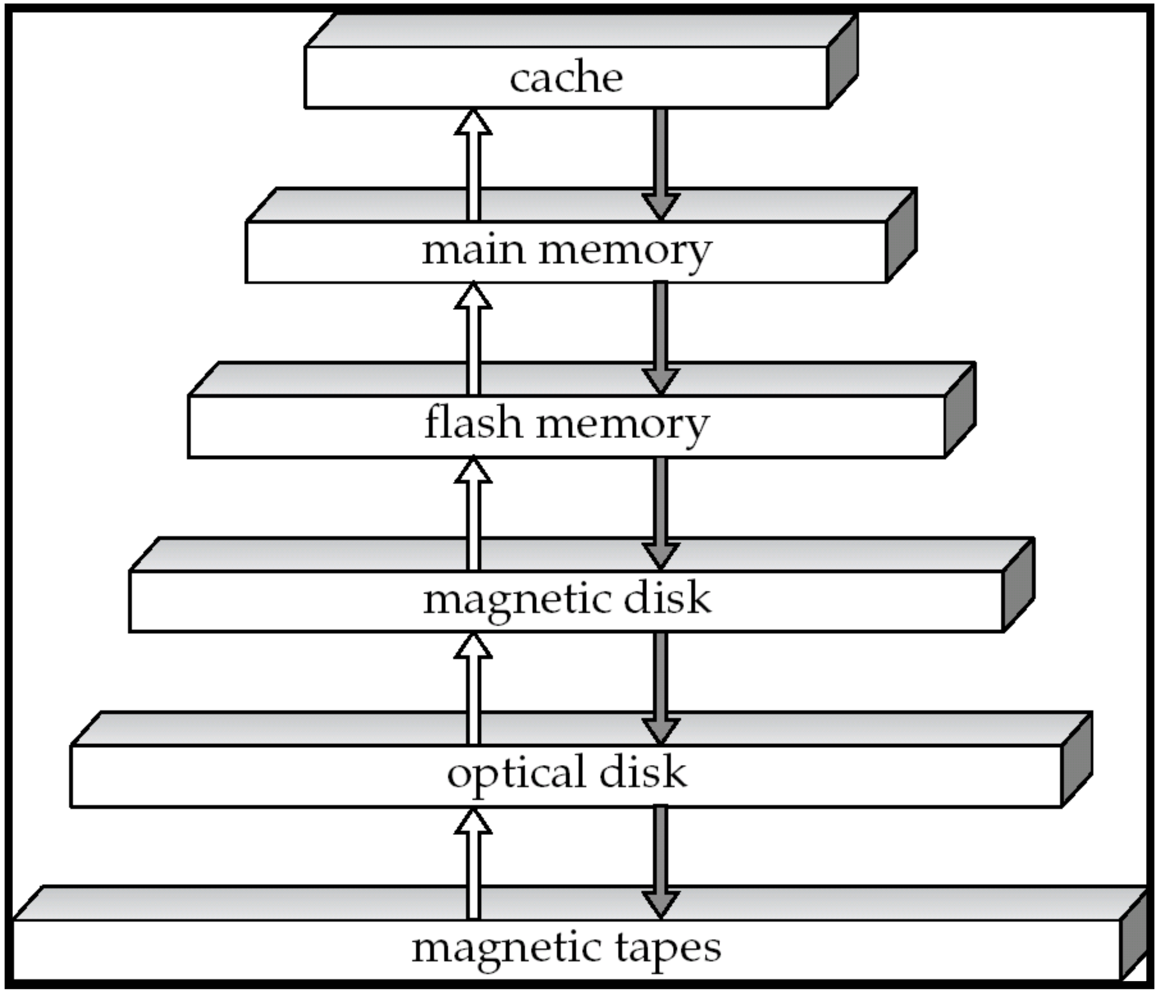
保证至少在一个 R_i 中存在 R 的候选码，从而保证 lossless-join.

讨论：对于多于二个子模式 R_i ($i > 2$) 的分解，判别是否无损连接的方法，其他教材中是用一张 i 行 n 列的表来表示．如果各子模式中函数依赖的相关性使得 R 中所有的属性都涉及，则是无损连接分解．而根据候选码的含义，候选码必与所有属性相关．从而二者本质上一致．

Problems

- Data Storage
- High Performance (Index, buffer manager)
- SQL query processing/ Optimization
- Concurrent Control
- Reliability

Storage Hierarchy



Storage Access

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

Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- **B$^+$-tree file organization**
  - Ordered storage even with inserts/deletes
  - More on this in Chapter 14
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
  - More on this in Chapter 14

Index

- 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

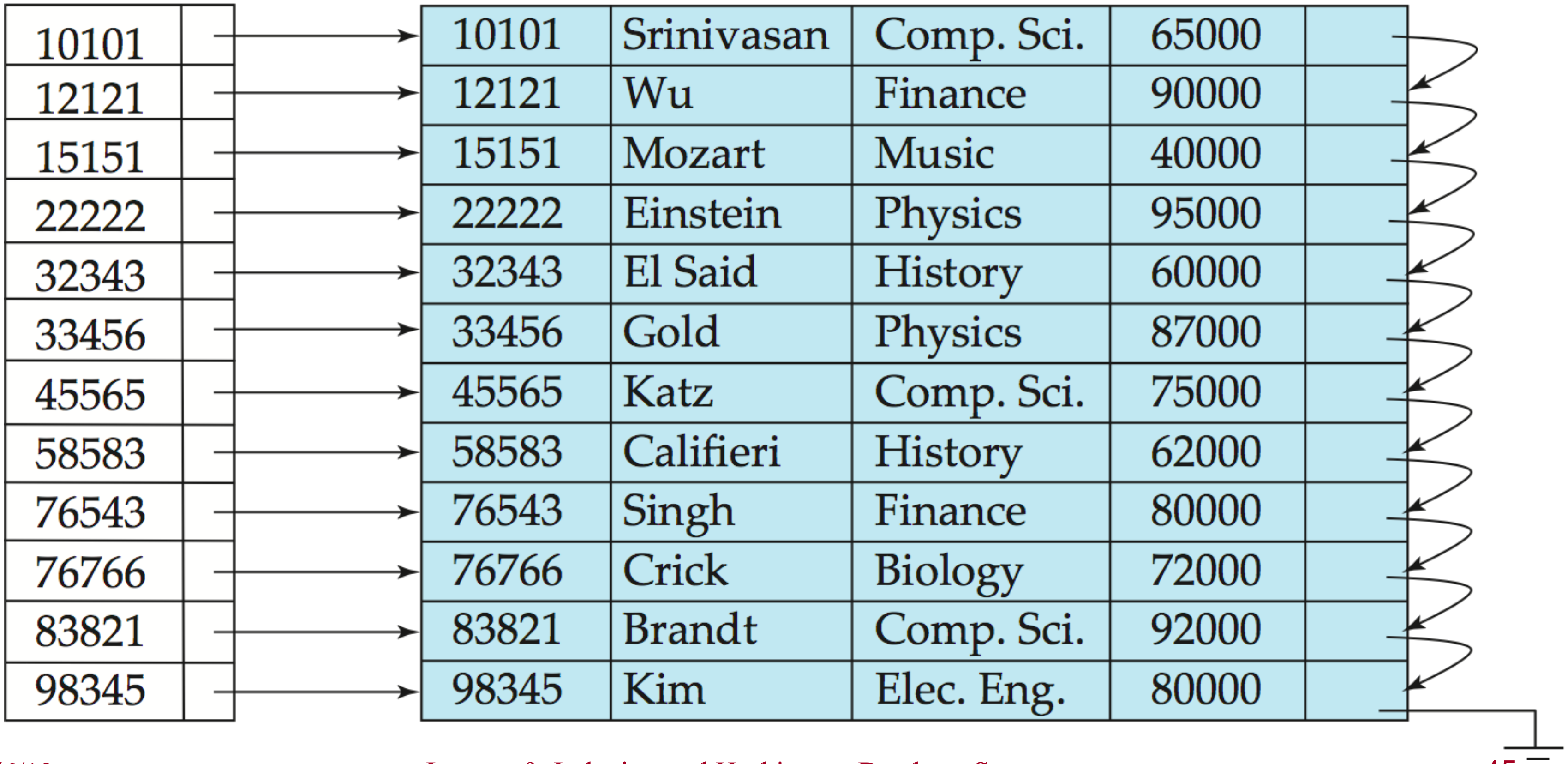
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.

(1) Dense Index Files (稠密索引文件)

- Dense index — Index entry appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation

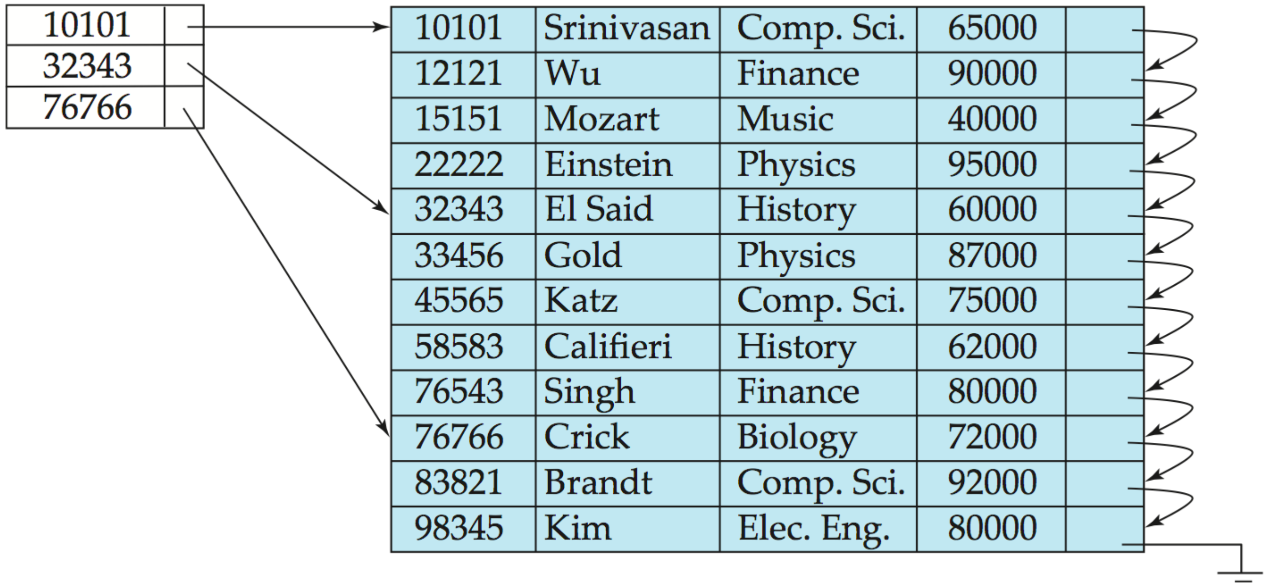


10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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

(2) Sparse Index Files

- Sparse Index: contains index entries for only some search-key values. (Usually, one block of data gives an index entry, a block contains a number of ordered data records)
  - Applicable only when data file records are sequentially ordered on search-key
- To locate a record with search-key value *K* , (搜索方法)
  - Step1: Find index record with largest search-key value < *K*
  - Step2: Search file sequentially starting at the record to which the index entry points



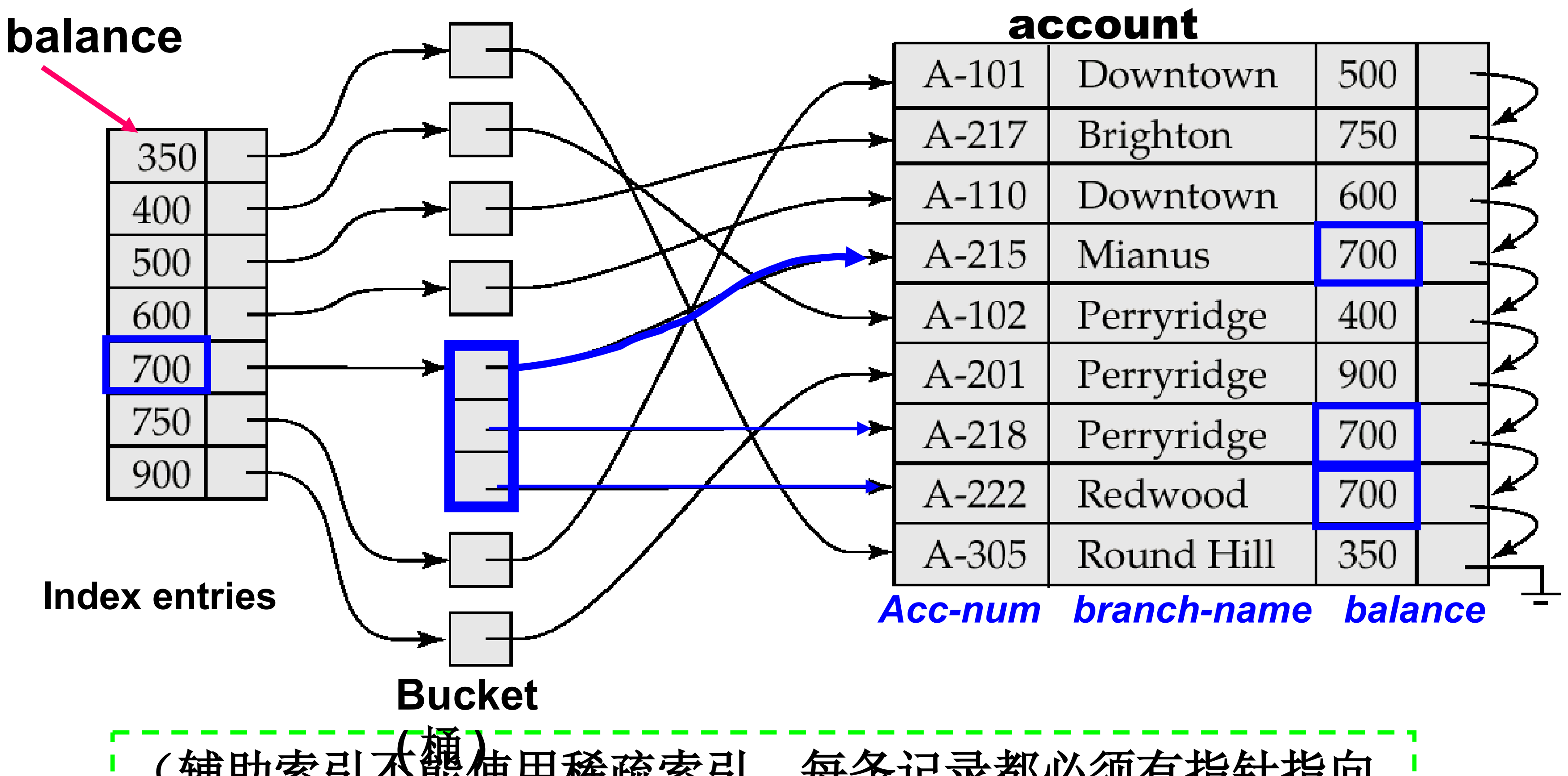
10101	
32343	
76766	

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

(3) Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field *which is not the search-key of the primary index* satisfy some condition. （实际应用中常有多种属性作为查询条件）
  - Example : In the *account* database stored sequentially by account number, we may want to find all accounts with a specified balance or range of balances. *(see next slide)*
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that one particular search-key value.

Secondary Index on *balance* field of *account*



（辅助索引不能使用稀疏索引，每条记录都必须有指针指向。但 **search-key** 常存在重复项 **---** 如 **700**， 而 **index entry** 不能有重复，否则查找算法复杂化，为此，使用 **bucket** 结构。）

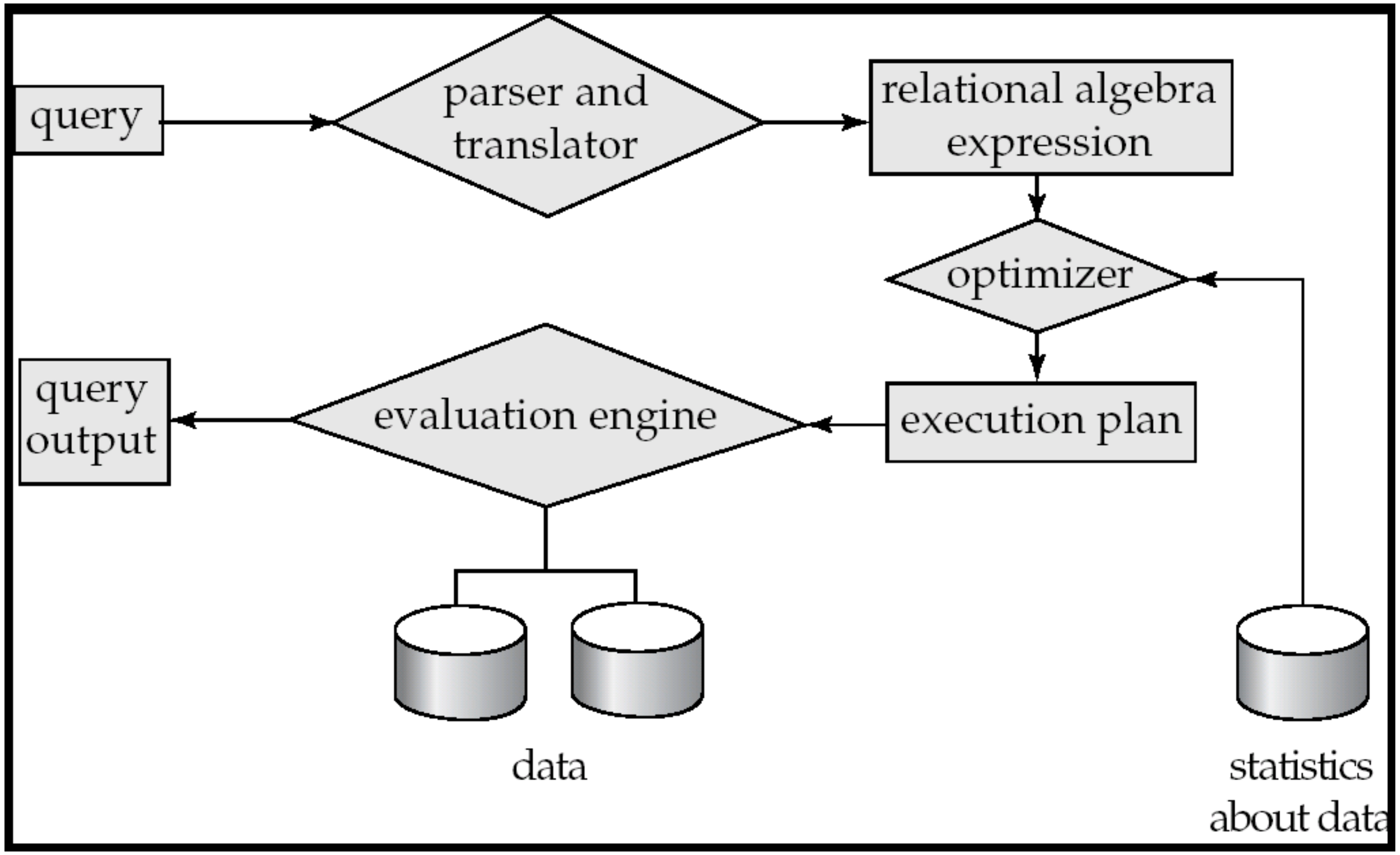
B^+-Tree Index Files

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

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Measures of Query Cost

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

    $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

```
for each tuple $t_r$ in $r$ do begin
    for each tuple $t_s$ in $s$ do begin
        test pair $(t_r, t_s)$ to see if they satisfy the join condition $\theta$
        if they do, add $t_r \cdot t_s$ to the result.
    end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

  $n_r * b_s + b_r$ block transfers, plus

  $n_r + b_r$ seeks

- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 ∗ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
**for each** block $B_r$ **of** $r$ **do begin**
    **for each** block $B_s$ **of** $s$ **do begin**
        **for each** tuple $t_r$ **in** $B_r$ **do begin**
            **for each** tuple $t_s$ **in** $B_s$ **do begin**
                Check if $(t_r,t_s)$ satisfy the join condition
                if they do, add $t_r \bullet t_s$ to the result.
            **end**
        **end**
    **end**
**end**
```

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use *M* — 2 disk blocks as blocking unit for outer relations, where *M* = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)

Indexed Nested-Loop Join

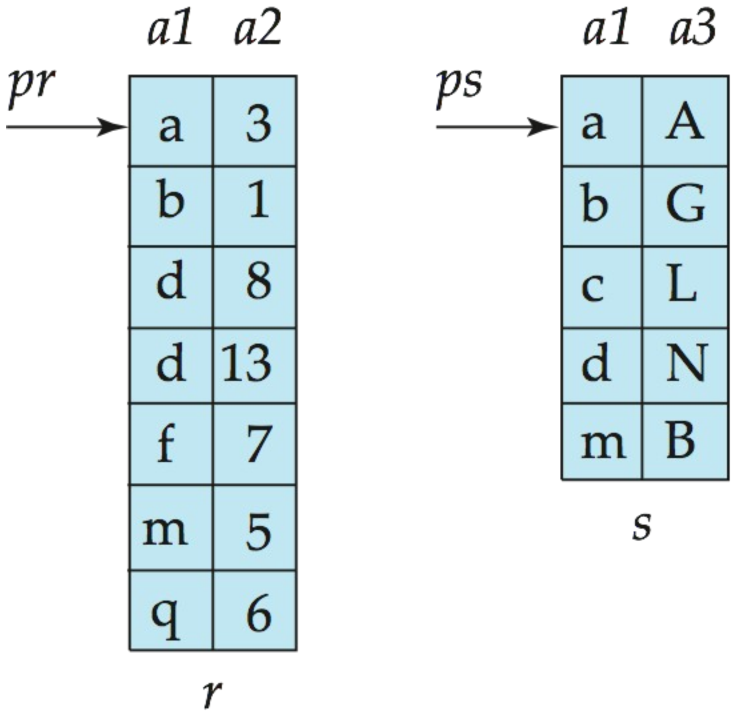
- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r (t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *student* ⋈ *takes,* with *student* as the outer relation.
- Let *takes* have a primary B$^+$-tree index on the attribute *ID,* which contains 20 entries in each index node.
- Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *student* has 5000 tuples
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
   1. Join step is similar to the merge stage of the sort-merge algorithm.
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
   3. Detailed algorithm in book



a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

a1	*a3*
a	A
b	G
c	L
d	N
m	B

Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

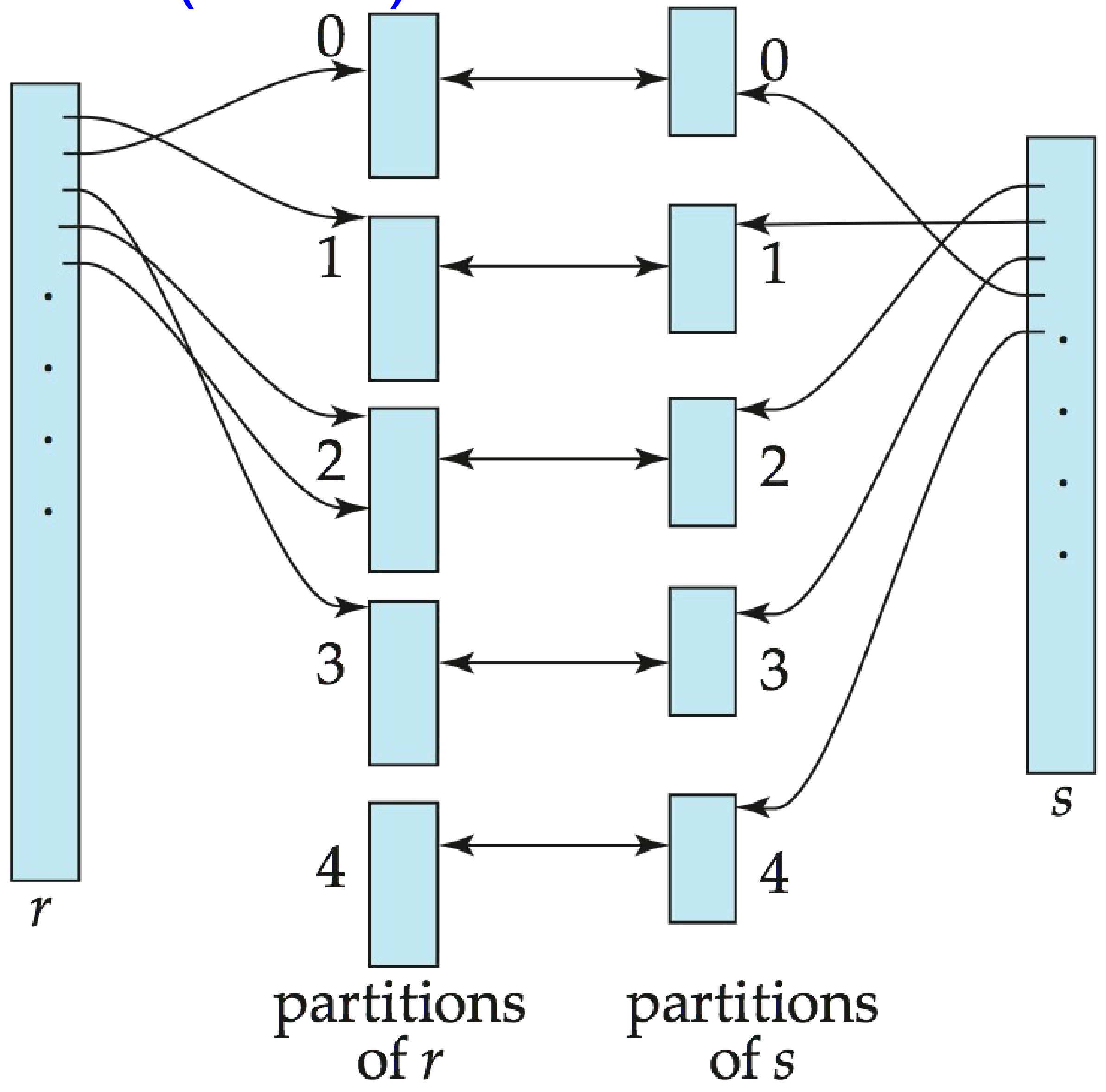
  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
  - + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B$^+$-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B$^+$-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, . . ., r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r [JoinAttrs])$.
  - $r_0,, r_1. . ., r_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s [JoinAttrs])$.

- *Note:* In book, r_i is denoted as $H_{ri,}$ s_i is denoted as H_{si} and n is denoted as $n_{h.}$

Hash-Join (Cont.)



Hash-Join (Cont.)

- *r* tuples in r_i need only to be compared with *s* tuples in s_i

  Need not be compared with *s* tuples in any other partition, since:

  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h.*
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation *s* is called the **build input** and *r* is called the **probe input**.

Hash-Join algorithm (Cont.)

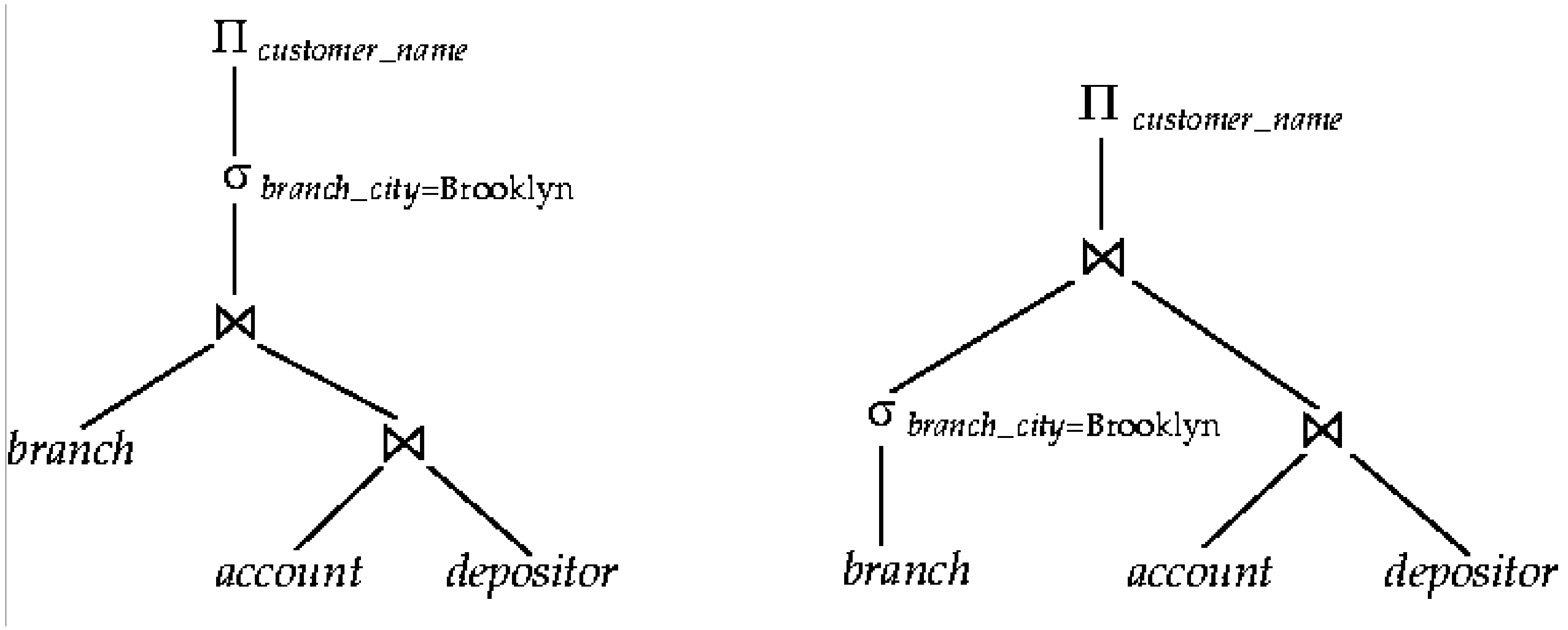
- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
  - The probe relation partitions r_i need not fit in memory
- **Recursive partitioning** required 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 > nh + 1, or equivalently M > (bs/M)+1, which simplifies (approximately) to M >√ bs
  - .

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers +
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2\, n_h$ seeks
- If recursive partitioning required:
  - number of passes required for partitioning build relation s is $\lceil log_{M-1}(b_s) - 1 \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    $2(b_r + b_s)\lceil log_{M-1}(b_s) - 1 \rceil + b_r + b_s$ block transfers +
    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)\lceil log_{M-1}(b_s) - 1 \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

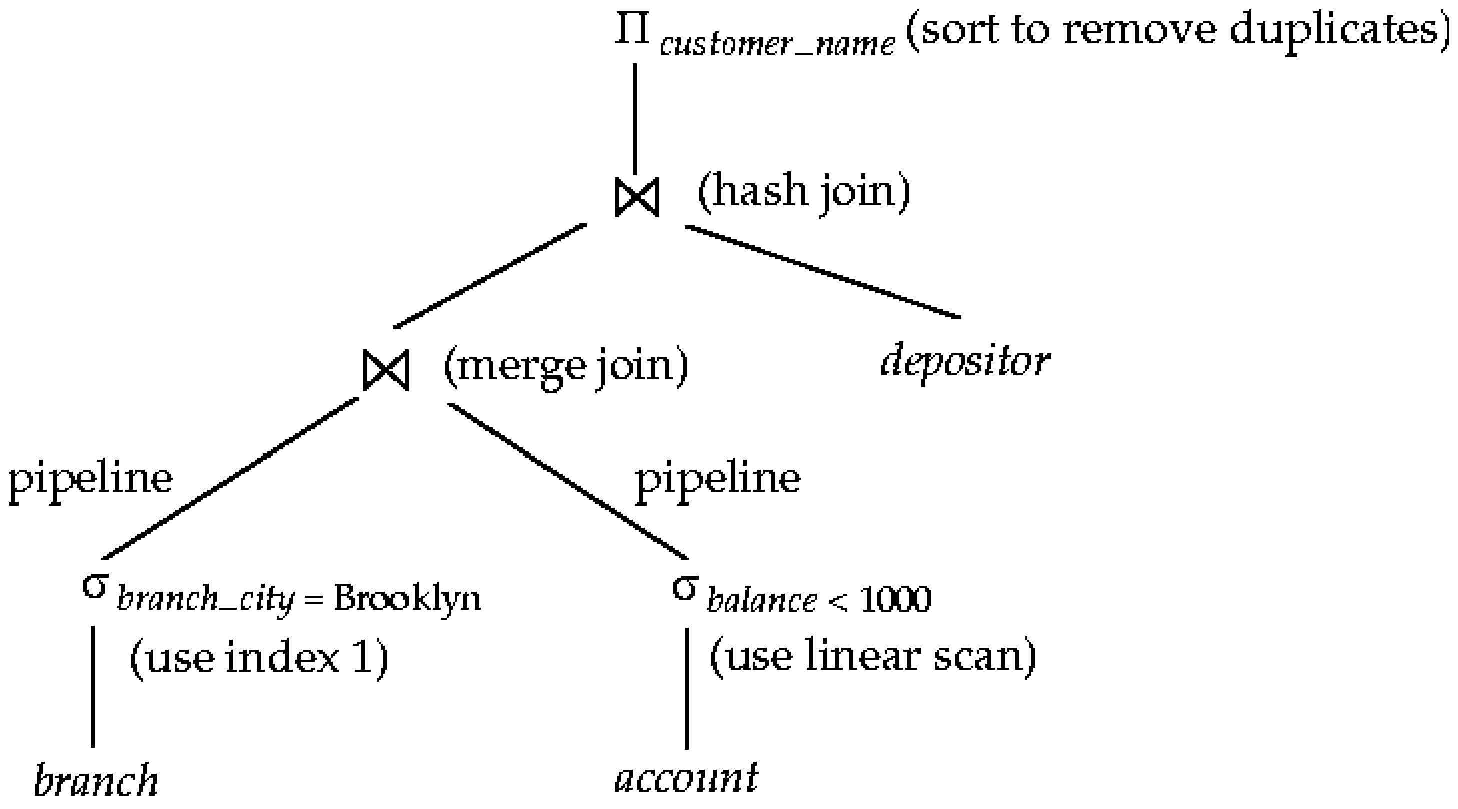
Relational Expression

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



Evaluation Plan

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.


$\Pi_{customer\_name}$ (sort to remove duplicates)
$\bowtie$ (hash join)
$\bowtie$ (merge join)
depositor
pipeline
pipeline
$\sigma_{branch\_city = \text{Brooklyn}}$
(use index 1)
$\sigma_{balance < 1000}$
(use linear scan)
branch
account

Transformation Example: Pushing Selections

- Query: find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using rule 7a.
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\text{“Music”}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course)$
- When we compute

  $(\sigma_{dept\_name = \text{"Music"}}(instructor \bowtie teaches))$

  we obtain a relation whose schema is:
  (*ID*, *name*, *dept_name*, *salary*, *course_id*, *sec_id*, *semester*, *year*)
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

  $$\Pi_{name,\ title}(\Pi_{name,\ course\_id}(\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches)) \bowtie \Pi_{course\_id,\ title}(course)$$
- Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- The ordering of join is important for reducing the size of temporary

  results.
- For all relations $r_1, r_2,$ and r_3

  $$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

  (Join Associativity)
- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

  $$(r_1 \bowtie r_2) \bowtie r_3$$

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course)$$

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}(course)$ first, and then join the result with

$$\sigma_{dept\_name=\text{“Music”}}(instructor)$$

  but the result of the first join is likely to be a large relation.
- Only a small fraction of the university's instructors are likely to be from the Music department
  - It is better to compute

$$\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches$$

    first.

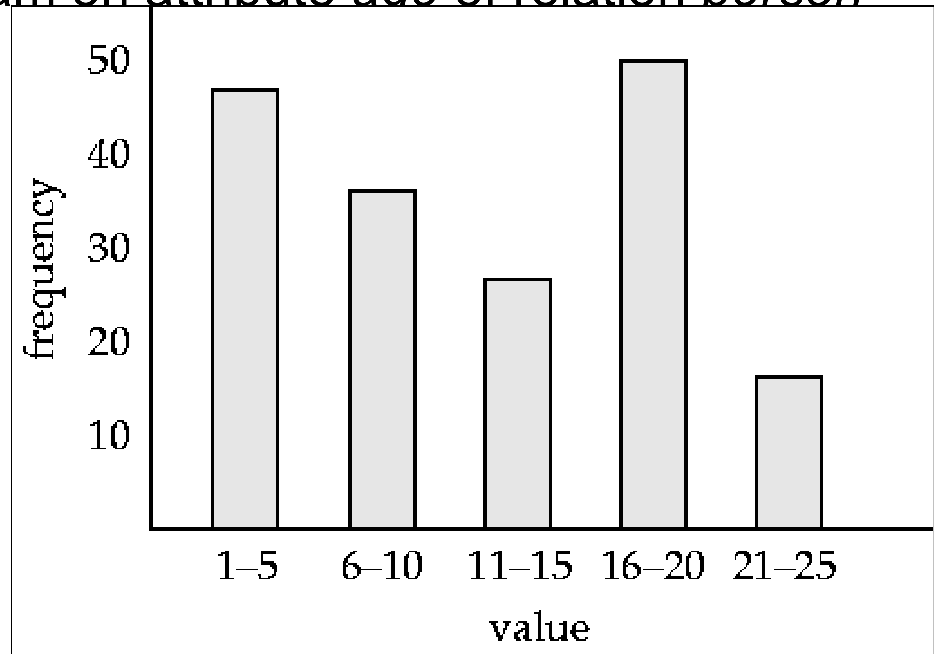
Statistical Information for Cost Estimation

- 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 $\prod_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

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

Histograms

- Histogram on attribute *age* of relation *person*



- Equi-width histograms
- Equi-depth histograms

Transaction Concept

- For a DBMS, two main issues must be dealt with:
  - Concurrent executions of multiple users or multi-programs.
  - Failures of various kinds, such as hardware failures and system crashes.

Transaction Concept (cont.)

- How to keep the database correctness, consistency, and integrity when it concurrently executes?

  Transaction --- Proposed by Jim Gray

- A transaction is a unit of program execution that accesses and possibly updates various data items.
  - Usually a transaction is of a number of SQL statements, ended with commit（提交） or rollback（回滚） statement.
  - A transaction must see a consistent database.

- During transaction execution the database may be inconsistent, but when the transaction is committed, the database must be consistent.

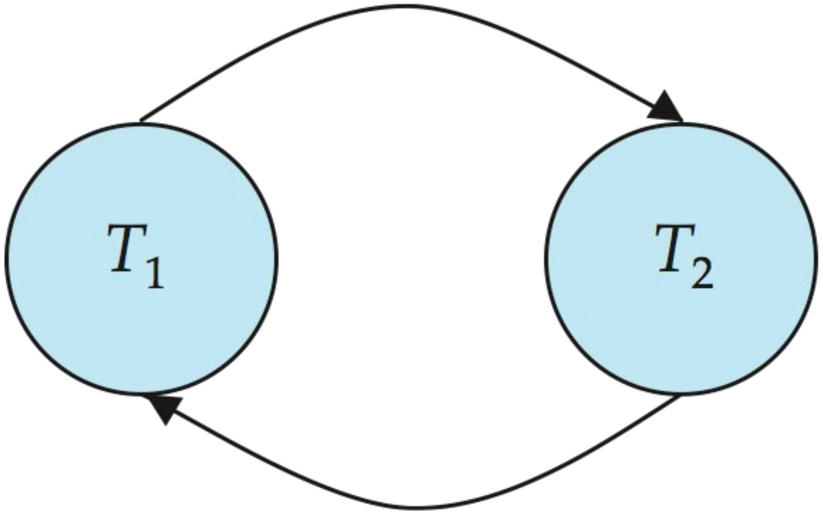
ACID Properties

A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data, the database system must ensure:

- 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.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a direct 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.
- Example 1



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

- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol (Cont.)

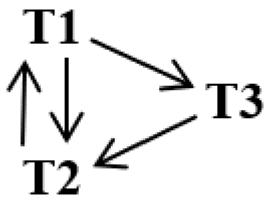
- Two-phase locking does not ensure freedom from deadlocks.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.
- Rigorous two-phase locking is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Concurrency Control

For the following schedule, please draw the precedence graph, and explain whether it is conflict serializable.

T1	T2	T3
Read(A)		
Read(B)		
		Read(A)
		Write(A)
	Read(B)	
	Read(A)	
Write(B)		
	Write(A)	

Answer: The schedule is not serializable, because there are cycles in the graph.



Note: Every schedule generated by 2PL is serializable. So, this schedule cannot be generated by two–phase locking protocol.

Exercise

Following is a schedule for transactions T1,T2,T3, and T4:

T1	T2	T3	T4
		write B	
			write C
			write A
write C			
	write C		
		write A	
write B			
write A			

a) Draw the precedence graph of the schedule
b) If the schedule is conflict serializable?
c) Can the schedule be generated by 2PL protocol?

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

How to handing?

- Deadlock prevention
- Deadlock detection and deadlock recovery

Deadlock prevention

- Deadlock prevention protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - 1) Require that each transaction locks all its data items before it begins execution (predeclaration) – conservative 2PL. (Either all or none are locked)
    - Disadvantages: bad concurrency, hard to predict
  - 2) Impose partial ordering of all data items and require that a transaction can lock data items only in the order (graph-based protocol). ---- therefore never form a cycle.

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

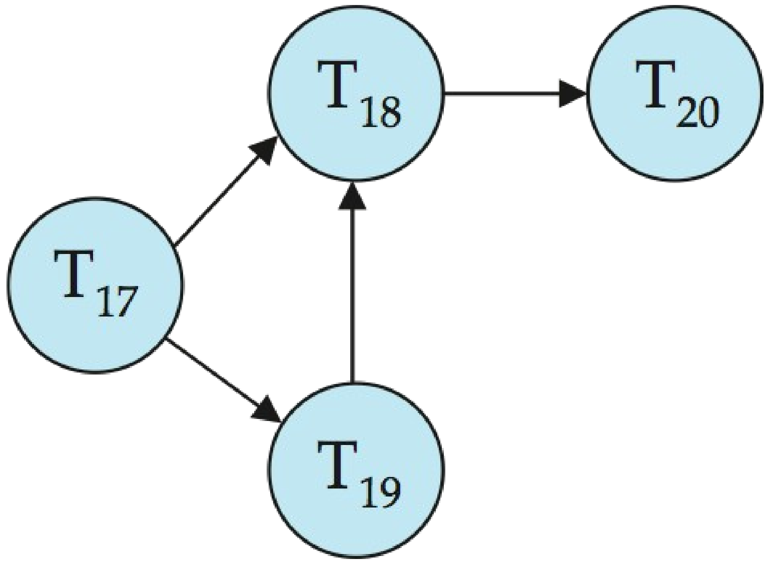
- Wait-die scheme — non-preemptive
  - Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring needed data item

- Wound-wait scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - May be fewer rollbacks than *wait-die* scheme

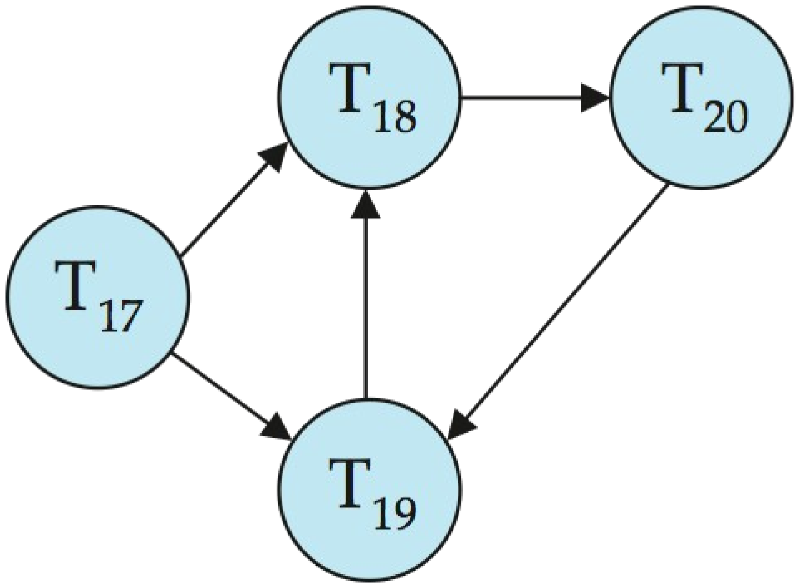
Deadlock Detection

- Deadlocks can be described as a wait-for graph, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- 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.
    - Partial rollback: 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

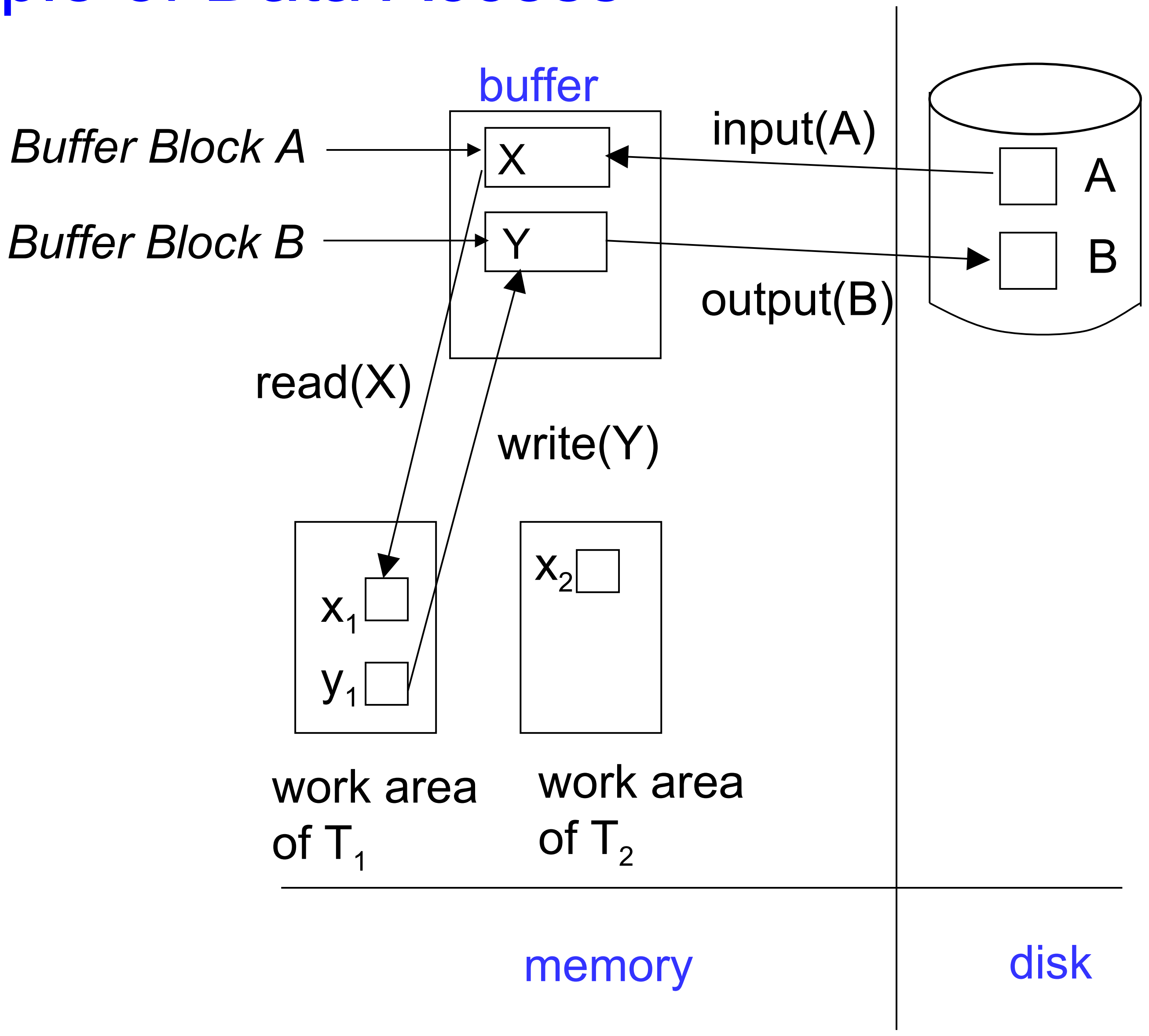
Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition: overflow, bad input, data not found,…
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Storage Structure

- **Volatile storage**:
  - Does not survive system crashes
  - Examples: main memory, cache memory
- **Nonvolatile storage**:
  - Survives system crashes
  - Examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - But may still fail, losing data
- **Stable storage**:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

Example of Data Access



Log-Based Recovery

- A **log** is kept on stable storage.
  - 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
  $<T_i$ **start**$>$ log record
- *Before* T_i executes **write**(X), a log record
  $<T_i, X, V_1, V_2>$
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record $<T_i$ **commit**$>$ is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate Database Modification Example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950 B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  1. Processing the entire log is time-consuming if the system has run for a long time
  2. We might unnecessarily redo transactions which have already output their updates to the database.

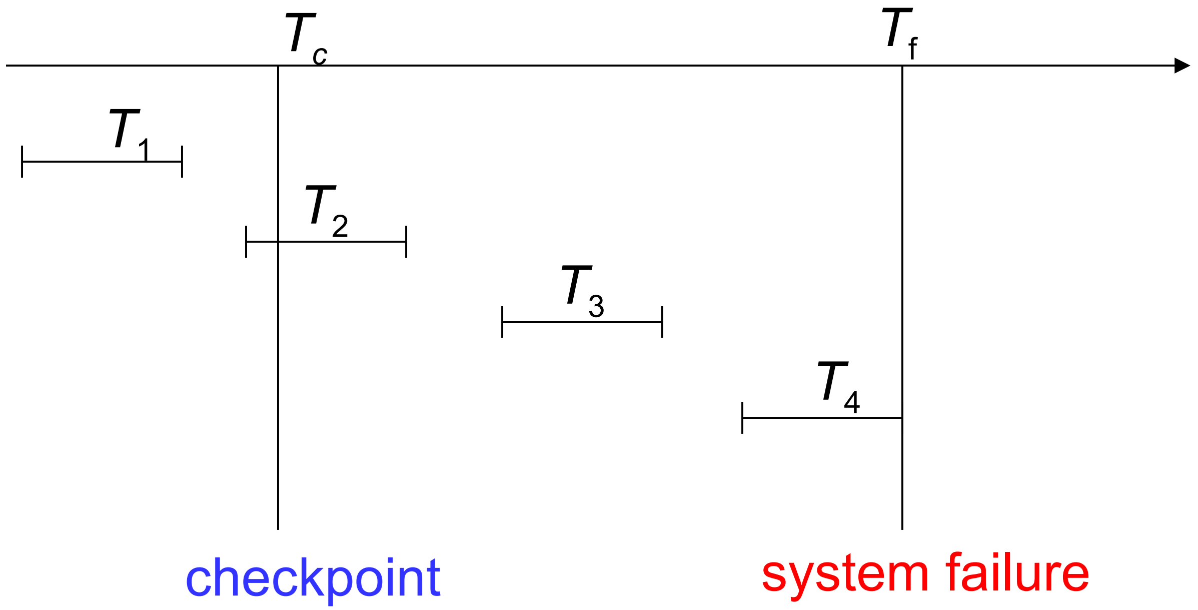
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
  - All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**, i.e., updated blocks need not be written to disk when transaction commits
  - **Force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**, i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     1. Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

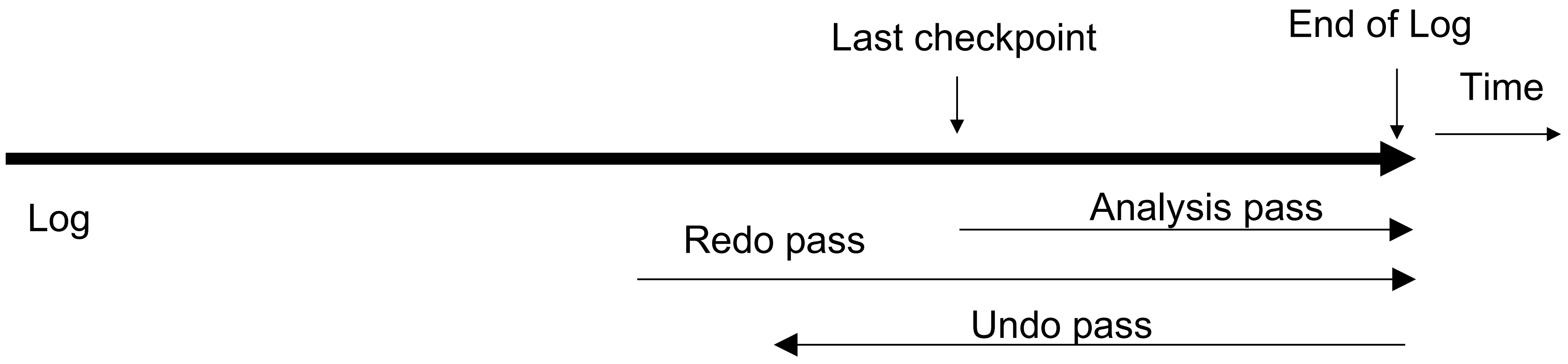
ARIES Recovery Algorithm

ARIES recovery involves three passes

- **Analysis pass**: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- **Redo pass**:
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- **Undo pass**:
  - Rolls back all incomplete transactions
    - Transactions whose abort was complete earlier are not undone
      - Key idea: no need to undo these transactions: earlier undo actions were logged, and are redone as required

Aries Recovery: 3 Passes

- Analysis, redo and undo passes
- Analysis determines where redo should start
- Undo has to go back till start of earliest incomplete transaction



ARIES Recovery: Analysis

Analysis pass

- Starts from last complete checkpoint log record
  - Reads DirtyPageTable from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
    - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record

- Scans forward from checkpoint
- .. Cont. on next page …

ARIES Recovery: Analysis (Cont.)

Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in undo-list, adds transaction to undo-list
  - Whenever an update log record is found
    - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
  - If transaction end log record found, delete transaction from undo-list
  - Keeps track of last log record for each transaction in undo-list
    - May be needed for later undo
- At end of analysis pass:
  - RedoLSN determines where to start redo pass
  - RecLSN for each page in DirtyPageTable used to minimize redo work
  - All transactions in undo-list need to be rolled back

ARIES Redo Pass

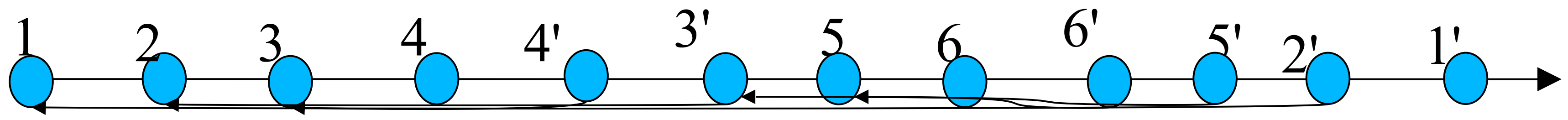
Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:

- Scans forward from RedoLSN. Whenever an update log record is found:
  1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record
  2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

NOTE: if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk!

ARIES Undo Actions

- When an undo is performed for an update log record
  - Generate a CLR containing the undo action performed (actions performed during undo are logged physicaly or physiologically).
    - CLR for record *n* noted as *n*' in figure below
  - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
    - Arrows indicate UndoNextLSN value
- ARIES supports partial rollback
  - Used e.g. to handle deadlocks by rolling back just enough to release reqd. locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback



ARIES: Undo Pass

Undo pass:

- Performs backward scan on log undoing all transaction in undo-list
  - Backward scan optimized by skipping unneeded log records as follows:
    - Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
    - At each step pick largest of these LSNs to undo, skip back to it and undo it
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
      - For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record
        - All intervening records are skipped since they would have been undone already
- Undos performed as described earlier

Recovery System

The system crashes just after the record 1014.

1) Which log record is the start point of Redo Pass?
2) Which log record is the end point of Undo Pass?
3) After Analysis Pass, what is the undo list?
4) After recovery, what are the value of "102.1" and "102.2"?
5) What additional log records are appended to log?

```
1001: <T1, begin>
1002: <T1, 101.1, 11, 21>
1003: <T2, begin>
1004: <T2, 102.1, 52, 62>
1005: <T2, commit>
1006: <T3 begin>
1007: <T3, 102.2, 73, 83>
1008: checkpoint
```

Active Transaction Table →

Tx	LastLSN
T1	1002
T3	1007

Dirty Page Table →

PageID	PageLSN	RecLSN
101	1002	1002
102	1007	1004

```
1009: <T1, 101.2, 31, 41>
1010: <T4 begin>
1011: <T3, 102.2, 73>
1012: <T3, abort>
1013: <T4, 102.1, 62, 64>
1014: <T1, commit>
```

Recovery System

The system crashes just after the record 1014.

1) Which log record is the start point of Redo Pass?
2) Which log record is the end point of Undo Pass?
3) After Analysis Pass, what is the undo list?
4) After recovery, what are the value of "102.1" and "102.2"?
5) What additional log records are appended to log?

1) 1002 (The smallest value in RecLSN of Dirty Page Tale)

2) 1010 (Due to T4 is undone)

3) T4,1013

4) "102.1" = 62, "102.2" = 73

5) 1015: <T4, 102.1, 62>

1016::<T4, abort> (Including all the undone steps)

1001: <T1, begin>
1002: <T1, 101.1, 11, 21>
1003: <T2, begin>
1004: <T2, 102.1, 52, 62>
1005: <T2, commit>
1006: <T3 begin>
1007: <T3, 102.2, 73, 83>
1008: checkpoint

Tx	LastLSN
T1	1002
T3	1007

PageID	PageLSN	RecLSN
101	1002	1002
102	1007	1004

1009: <T1, 101.2, 31, 41>
1010: <T4 begin>
1011: <T3, 102.2, 73>
1012: <T3, abort>
1013: <T4, 102.1, 62, 64>
1014: <T1, commit>

Q & A

Your questions and suggestions are expected for me.

Thanks a lot!

Questions?