



数据库管理系统及其实现

东南大学计算机科学与工程系 徐立臻
南京，2013.3



Course Goal and its Preliminary Courses

The Preliminary Courses are:

- Data Structure
- Database Principles
- Database Design and Application

The students should already have the basic concepts about database system, such as data model, data schema, SQL, DBMS, transaction, database design, etc.

Now we will introduce the implementation techniques of Database Management Systems.

The goal is to **build the foundation of further research in database field** and to **use database system better** through the study of this course.



主要内容

数据库管理系统(DBMS)的体系结构及其实现。主要内容包括：DBMS体系结构、用户接口、语法分析、查询处理、目录管理、并发控制、恢复机制、物理存储管理等。不管什么样的数据库系统，其DBMS一般都含有以上几个部分，只不过具体的实现方法及考虑问题的侧重点有所不同。本课程的主要内容就是介绍DBMS核心所涉及的基本概念、基本原理及其实现方法。



课程重点

由于关系模型是主流数据模型，而分布式数据库管理系统在并发控制、恢复等方面包容了集中式数据库管理系统的所有内容，所以本课程将以关系型分布式数据库管理系统为主线，介绍数据库管理系统中各部分的实现。并适当补充一些其它类型数据库系统的内容。



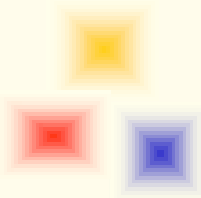
课程沿革

- 数据库系统原理（~1984）— 关系模型理论、一些查询优化算法。
- 分布式数据库系统（~1994）— 全面、系统地介绍DDBMS的各项实现技术及算法。
- 数据库管理系统及其实现（1995~）— 不限于DDBMS，更全面地介绍DBMS的实现技术，因为DDBMS只是数据库技术发展中的一个分支。将随着技术的发展不断调整、添加新的内容。



主要参考书

- 1) Stefano Ceri, “Distributed Databases”
- 2) 王能斌, “数据库系统教程（上下册）”, 电子工业出版社, 2008
- 3) Raghu Ramakrishnan, Johannes Gehrke, “Database Management Systems”, 3rd Edition, McGraw-Hill Companies, 2002
- 4) Hector Garcia-Molina, Jeffrey.D.Ullman, “Database Systems: the Complete Book”
- 5) S.Bing Yao et al, “Query Optimization in DDBS”
- 6) Courseware:
<http://cse.seu.edu.cn/people/lzxu/resource>



目录

1. 概述

数据库系统的发展、分类、及主要研究内容；分布式数据库系统

2. DBMS体系结构

DBMS的组成及进程结构；分布式数据库系统的体系结构

3. 数据库访问管理

物理文件组织、索引及存取原语

4. 数据分布

数据的分割及分布、分布式数据库设计、联邦式数据库设计、并行数据库设计、数据目录及其分布



目录

5. 查询优化

基本问题；查询优化技术；分布式数据库系统的查询处理；其它类型DBMS 的查询处理

6. 恢复机制


基本问题；更新策略及恢复技术；分布式数据库系统的恢复机制

7. 并发控制

基本问题；并发控制技术；分布式数据库系统的并发控制；其它类型DBMS 的并发控制




1. 概述



1.1 数据库技术的发展历史及分类

(1) 从数据模型的发展来看

- 无管理(60年代之前): 科学计算
- 文件系统: 简单的数据管理
- 数据管理需求不断增长, 数据库管理系统应运而生。
 - 1964, 美通用电气公司开发出第一个DBMS: IDS, 网状
 - 1969, IBM推出第一个商品化DBMS, 层次
 - 1970, IBM研究员E.F.Codd提出关系模型
 - 其它数据模型: 面向对象、演绎、XML等



(2) 从DBMS体系结构的发展来看

- 集中式数据库系统
- 并行数据库系统
- 分布式数据库系统（含联邦式数据库系统）
- 移动数据库系统

(3) 从基于数据库的应用系统的发展来看

- 集中式结构：主机+哑终端
- 分布式结构
- Client/Server结构
- 三层/多层结构
- 移动计算
- 网格计算（数据网格）、云计算



(4) 从应用领域的拓展来看

- OLTP
- 工程数据库
- 演绎数据库
- 多媒体数据库
- 时态数据库
- 空间数据库
- 数据仓库、OLAP及数据挖掘等
- XML数据库
- 大数据, NoSQL, NewSQL



1.2 分布式数据库系统

What is DDB?

A DDB is a collection of correlated data which are spread across a network and managed by a software called DDBMS.

两种：

- (1)物理上分布，逻辑上统一(一般的DDB)
- (2)物理上分布，逻辑上也分布(FDBS)

本课程以第一种DDBS为主。



DDBS特征:

- Distribution
- Correlation
- DDBMS




DDBS优点:

- 地方自治
- 可用性好(由于支持多副本)
- 灵活性好
- 系统代价较低
- 效率较高(大多数访问就地处理, 减少通信)
- 并行处理能力

DDBS弱点:

- DB集成较难
- 太复杂(系统本身及使用, 如DDB设计)



DDBS中的主要问题:

与集中式DBMS相比，DDBS的不同主要体现在以下几个方面:

- 查询处理(目标不同)
- 并发控制(要考虑全局)
- 恢复机制(要考虑复杂的故障组合)

作为DDB，还多一个问题:

- 数据分布



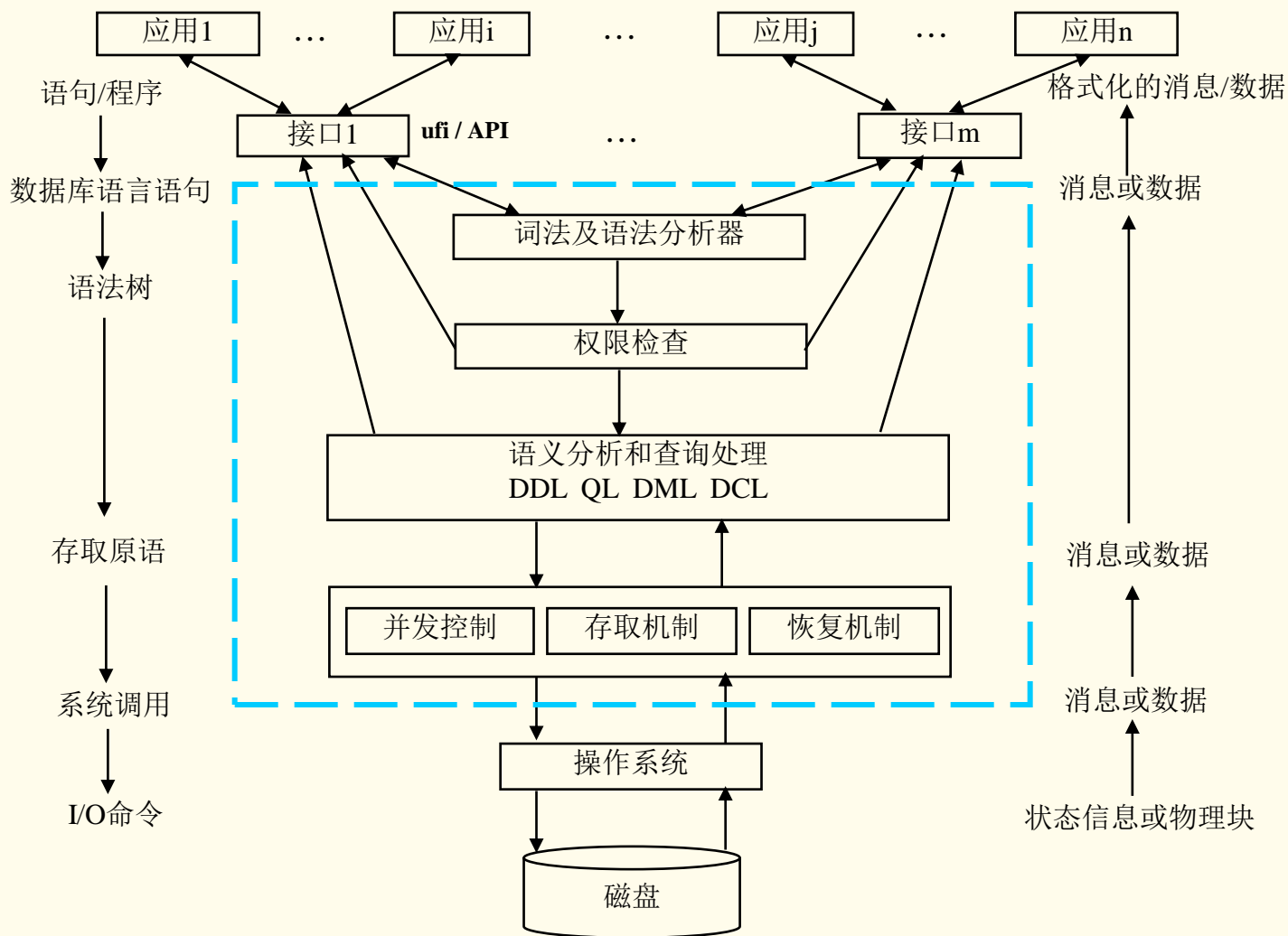
2. DBMS体系结构



主要内容

- DBMS核心组成
- DBMS进程结构
- DDBMS核心组成
- DDBMS进程结构

2.1 DBMS核心的构成



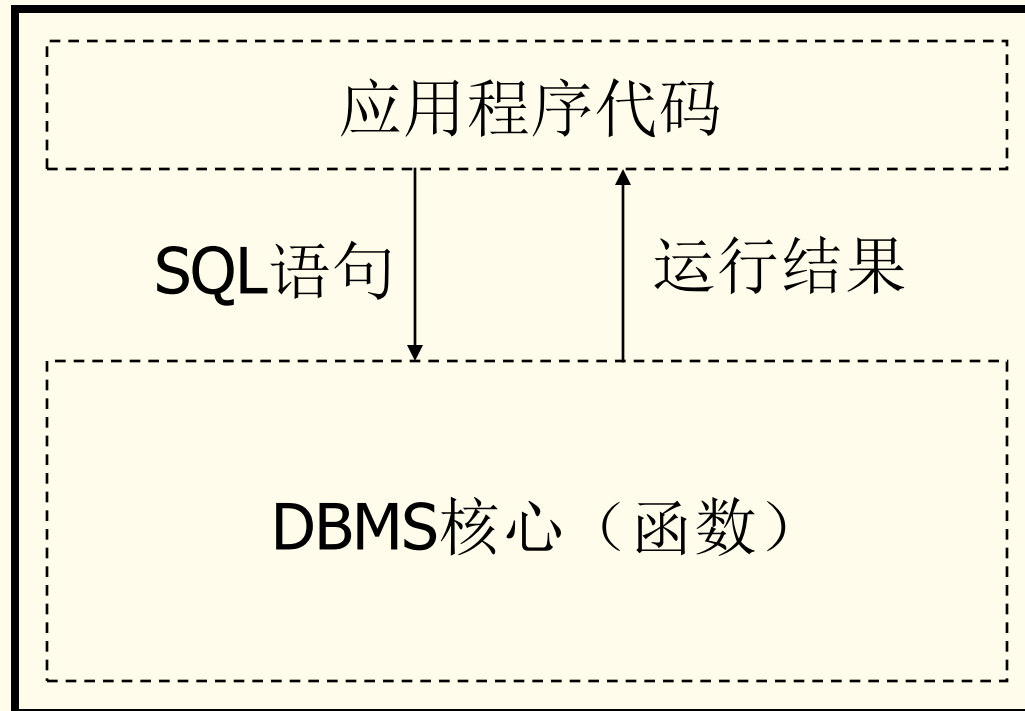


2.2 DBMS进程结构

- 单进程结构
- 多进程结构
- 多线程结构
- 进程/线程间通信协议

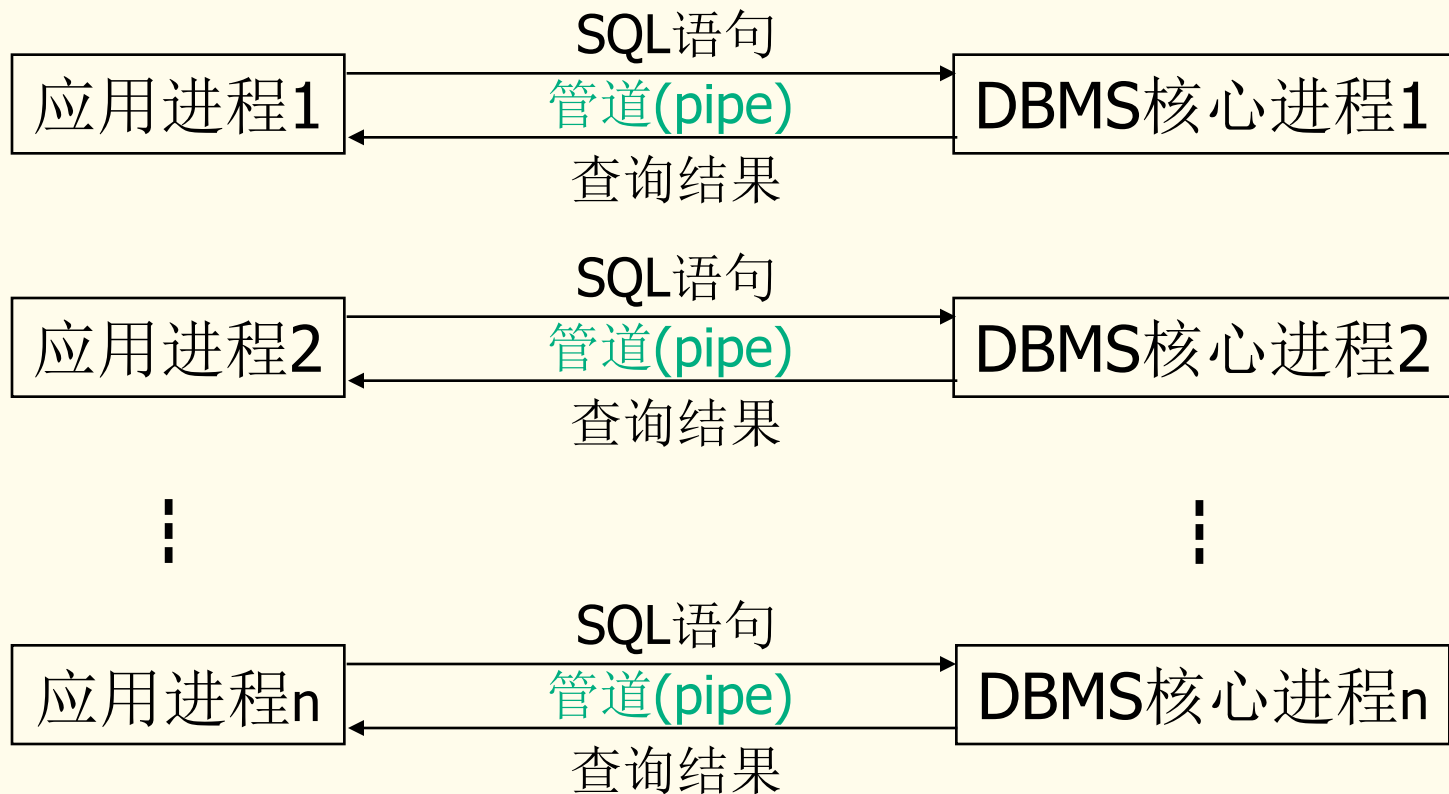
单进程结构

- 应用程序与DBMS核心编译为一个EXE，单进程运行。



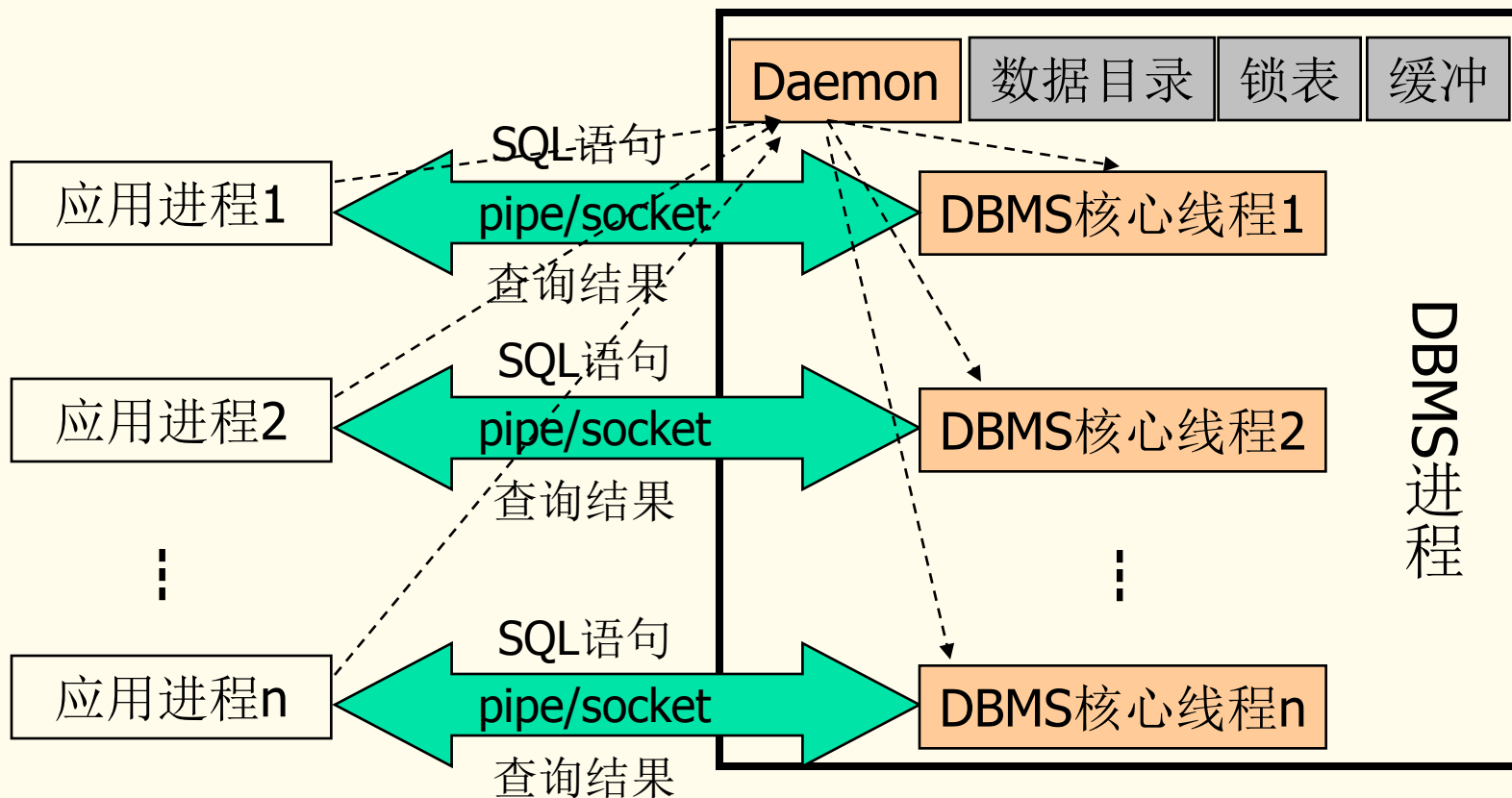
多进程结构

- 一个应用进程对应一个DBMS核心进程



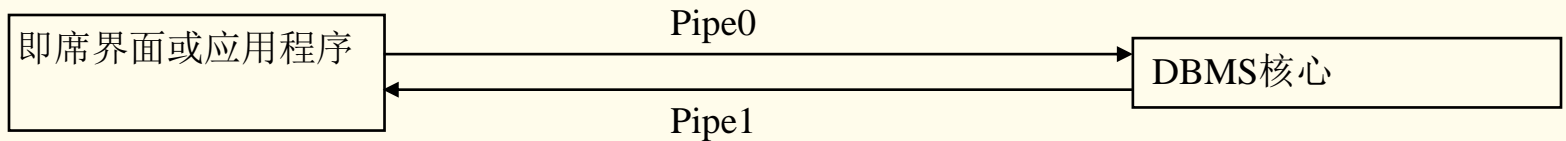
多线程结构

- 只有一个DBMS进程，每个应用进程对应一个DBMS核心线程



进程/线程间通信协议

- 应用程序通过DBMS提供的API或嵌入式SQL访问数据库，根据通信协议进行同步控制：



Pipe0: 发送SQL语句、内部命令；

Pipe1: 返回结果。结果格式：

State	TupNum	AttNum	AttName	AttType	AttLen	TmpFileName
-------	--------	--------	---------	---------	--------	--------	-------------

一个属性的定义

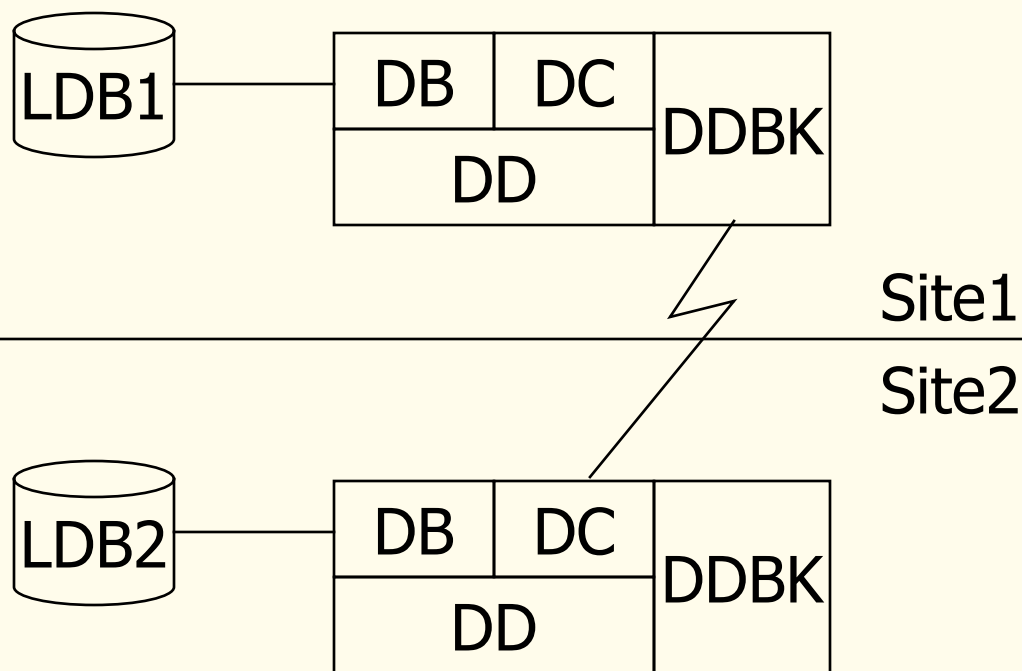
其它属性的定义



进程/线程间通信协议

- State: 0 — 出错, 1 — 插、删、改操作成功, 2 — 查询成功, 需进一步处理结果。
- TupNum: 结果中元组数。
- AttNum: 结果中属性个数。
- AttName: 属性名。
- AttType: 属性类型。
- AttLen: 属性长度。
- TmpFileName: 存放结果数据的临时文件名, 其中的数据要用上述字典信息来解释。

2.3 DDBMS核心的构成



DB: 数据库管理

DC: 通信控制

DD: 数据字典管理

DDBBK: 核心。负责语法分析、分布事务管理、并发、恢复以及**全局查询优化**等。



全局查询优化例

R1

Site1

R2

Site2

```
Select *  
From R1,R2  
Where R1.a = R2.b;
```

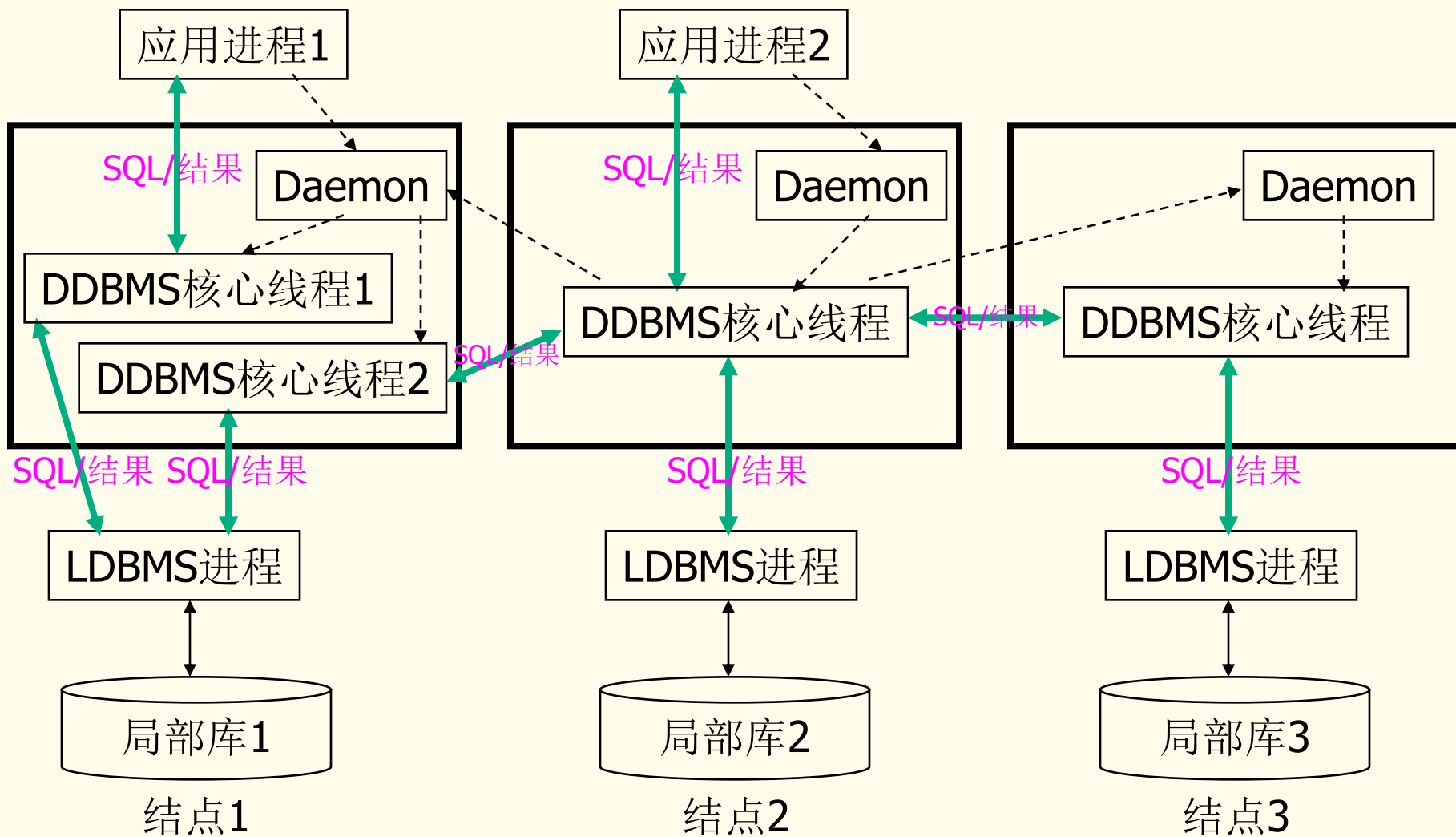
全局查询优化根据代价估算得出一个执行计划表。比如：

(1)将R2 \rightarrow Site1, R'

(2)在Site1上执行：

```
Select *  
From R1,R'  
Where R1.a = R'.b;
```

2.4 DDBMS进程结构





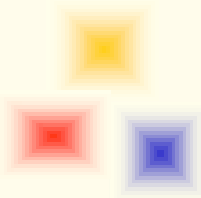
3. 数据库访问管理



主要内容

对数据库的操作最终要落实到对文件的操作，文件结构及其所提供的存取路径直接影响数据访问的速度，一种文件结构不可能对所有数据访问都有效的。

- 访问类型
- 文件组织
- 索引技术
- 存取原语



访问类型

- 查询文件的全部或相当多的记录(>15%)
- 查询某一特定记录
- 查询某些记录(<15%)
- 范围查询
- 记录的更新



文件组织

- 堆文件：按插入先后次序存放，顺序搜索，最基本、通用的文件组织形式
- 直接文件：按某属性值散列映射成记录地址
- 索引文件：索引+堆文件/簇集
- 动态散列：p111
- 栅格文件结构：p114（多属性查询）
- Raw Disk（注意区别文件的逻辑块和物理块，利用Raw Disk可直接控制物理块）



索引技术

- B+树 (✓✓)
- 簇集索引 (✓)
- 倒排文件
- 动态散列
- 栅格结构文件和分段散列
- 位图索引 (数据仓库)
- 其它类型的索引

位图索引—索引本身就是数据

Date	Store	State	Class	Sales
3/1/96	32	NY	A	6
3/1/96	36	MA	A	9
3/1/96	38	NY	B	5
3/1/96	41	CT	A	11
3/1/96	43	NY	A	9
3/1/96	46	RI	B	3
3/1/96	47	CT	B	7
3/1/96	49	NY	A	12

Binary Index for Sales

8bit	4bit	2bit	1bit
0	1	1	0
1	0	0	1
0	1	0	1
1	0	1	1
1	0	0	1
0	0	1	1
0	1	1	1
1	1	0	0

Binary Index for State

AK	AR	CA	CO	CT	MA	NY	RI	...
0	0	0	0	0	0	1	0	
0	0	0	0	0	1	0	0	
0	0	0	0	0	0	1	0	
0	0	0	0	1	0	0	0	
0	0	0	0	0	0	1	0	
0	0	0	0	0	0	0	1	
0	0	0	0	1	0	0	0	
0	0	0	0	0	0	1	0	

for Class

A	B	C
1	0	0
1	0	0
0	1	0
1	0	0
1	0	0
0	1	0
0	1	0
1	0	0

- 总销售量=? ($4*8+4*4+4*2+6*1=62$)
- NY的A类店有多少? (3)
- NY的A类店销售量=? ($2*8+2*4+1*2+1*1=27$)
- CT有多少家店? (2)
- 处理连接(查NY的A类店销售产品清单)



存取原语

- `int dbopendb(char * dbname)`
功 能：打开一个数据库。
- `int dbclosedb(unsigned dbid)`
功 能：关闭一个数据库。
- `int dbTableInfo(unsigned rid, TableInfo * tinfo)`
功 能：取得rid对应的表的信息。
- `int dbopen(char * tname, int mode, int flag)`
功 能：以flag打开一个名为tname的表并为其分配rid。
- `int dbclose(unsigned rid)`
功 能：关闭以rid为标识的表，释放rid。
- `int dbrename(oldname, newname)`
功 能：重命名表。



存取原语

- `int dbcreateattr (unsigned rid, sstree * attrlist)`

功 能：在rid关系中创建一些属性。

- `int dbupdateattrbyidx(unsigned rid, int nth, sstree attrinfo)`

功 能：更新关系rid中第nth个属性说明。

- `int dbupdateattrbyname(unsigned rid, char * attrname, sstree attrinfo)`

功 能：更新关系rid中名为attrname的属性说明。

- `int dbinsert(unsigned rid, char * tuple, int length, int flag)`

功 能：将长为length的元组插入标识为rid的关系中。



存取原语

- `int dbdelete(unsigned rid, long offset, int flag)`

功 能：删除rid对应的表中偏移为offset的元组。

- `int dbupdate(unsigned rid, long offset, char * newtuple, int flag)`

功 能：对偏移为offset的元组用newtuple更新。

- `int dbgetrecord(unsigned rid, int nth, char* buf)`

功 能：取出rid表中的第n个元组放入buf中。

- `int dbopenidx(unsigned rid, indexattrstruct * attrarray, int flag)`

功 能：打开表rid的索引文件并为其分配一个iid。



存取原语

- `int dbcloseidx(unsigned iid)`
功 能：关闭标识为*iid*的索引。
- `int dbfetch(unsigned rid, char * buf, long offset)`
功 能：从表*rid*中取出标识为*offset*的元组到*buf*中。
- `int dbfetchtid(unsigned iid, void * pvalue, long*offsetbuf, flag)`
功 能：在标识为*iid*的索引文件中取出索引属性值与*pvalue*有*flag*关系的元组的标识(实际上是元组在文件中的偏移量)放入*offsetbuf*中。
- `int dbpack(unsigned rid)`
功 能：对关系进行压缩，对有删除标志的记录进行物理删除并重整文件。



4 数据分布



4.1 数据分布策略

- (1) Centralized:即分布式系统，但数据还是集中存放的。最简单，但无DDB优点。
- (2) Partitioned:数据是分布的，但数据间不能重复(无复本)。
- (3) Replicated:A complete copy of DB at each node. Good for retrieval-intensive system.
- (4) Hybrid (前几种混合): An arbitrary fraction of DB at various nodes. 最灵活、最复杂的分布方法。



四种分布方式比较

1

2

3

4

flexibility

complexity

Advantage of DDBS

Problems with DDBS



4.2 数据分布的单位

(1) 以关系(或文件)为单位(即不划分)

(2) 以裂片(fragments)为单位

- 水平分割(Horizontal fragmentation): tuple partition
- 垂直分割(Vertical fragmentation): attribute partition
- 混合分割(Mixed fragmentation): both



分割准则:

- (1) Completeness: 每个元组或属性都要在 fragmentation 中有它的映象。
- (2) Reconstruction: 要能由 fragments \rightarrow global relation。
- (3) Disjointness: 对水平分割而言。



分割方法

(1) Horizontal Fragmentation

由带谓词的选择操作来定义，用并操作重构。

SELECT *		R \rightarrow n个裂片(用 P_1, P_2, \dots, P_n)
FROM R		满足: $P_i \wedge P_j = \text{false} \quad (i \neq j)$
WHERE P ;		$P_1 \vee P_2 \vee \dots \vee P_n = \text{true}$

Derived Fragmentation: 不是据本关系属性的特征来划分，而是据另一关系的划分来做。



导出分割例子

TEACHER(TNAME,DEPT)

COURSE(CNAME,TNAME)

设TEACHER已按DEPT水平分割， COURSE中无DEPT属性，但可按TNAME所在系对课程按系划分，此即由TEACHER导出的分割。

Semi join: $R \bowtie S = \Pi_R (R \bowtie S)$

\therefore TEACHER9=SELECT * FROM TEACHER
WHERE DEPT='9th';

COURSE9 = COURSE \bowtie TEACHER9



(2) Vertical Fragmentation

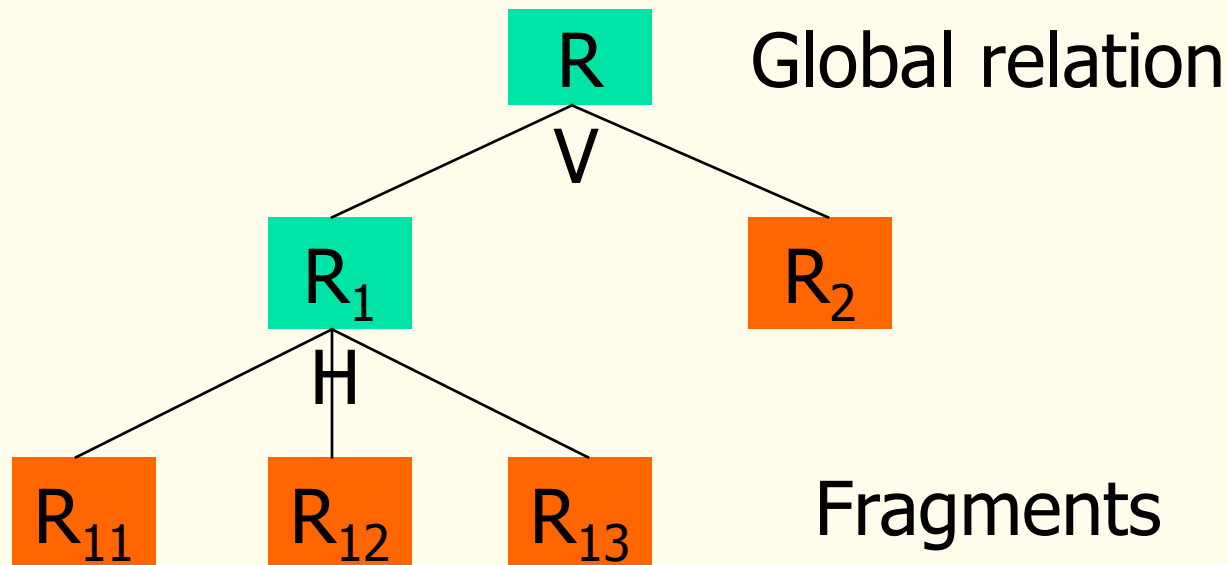
由投影操作定义，用连接操作重构。注意：

- **Completeness:**每个属性至少在某个裂片中出现。
- **Reconstruction:**分割时要满足无损连接条件。
 - a. 在每个裂片中都包含一个原关系的主键。
 - b. 在每个裂片中都包含一个系统产生的元组标识符(TID).



(3) Mixed Fragmentation

Apply fragmentation operations recursively.
Can be showed with a fragmentation tree:





4.3 不同级别的透明性

采用information hiding的方法简化复杂问题

■ Level 1: Fragmentation Transparency

用户只看到global relation，而不知它是如何划分的，当然也就不需要了解这些裂片的分布情况了，在这种情况下，用户感觉不到数据的分布，就好像他在使用一个集中式数据库一样。



- Level 2: Location Transparency

用户要了解关系被分成多少 fragments，但不需知道它们的存放位置。

- Level 3: Local Mapping Transparency

用户需了解 fragments 及其存放位置，但各结点上的局部库使用何种 DBMS、何种 DML，用户不需了解。

- Level 4: No Transparency



4.4 数据分布带来的问题

1) Multicopy consistency

2) Distribution consistency

主要是由于更新操作引起的元组存放位置的改变。
解决：

(1) Redistribution

After Update:Select-Move-Insert-Delete

(2) Piggybacking

更新后立即检查，如不符随响应信息返回。



3) Translation of Global Queries to Fragment Queries and Selection of Physical Copies.

4) Design of Database Fragments and Allocation of Fragments.

上述1)~3)应由DDBMS解决；
而4)是分布式数据库设计的问题。



4.5 分布式数据库设计

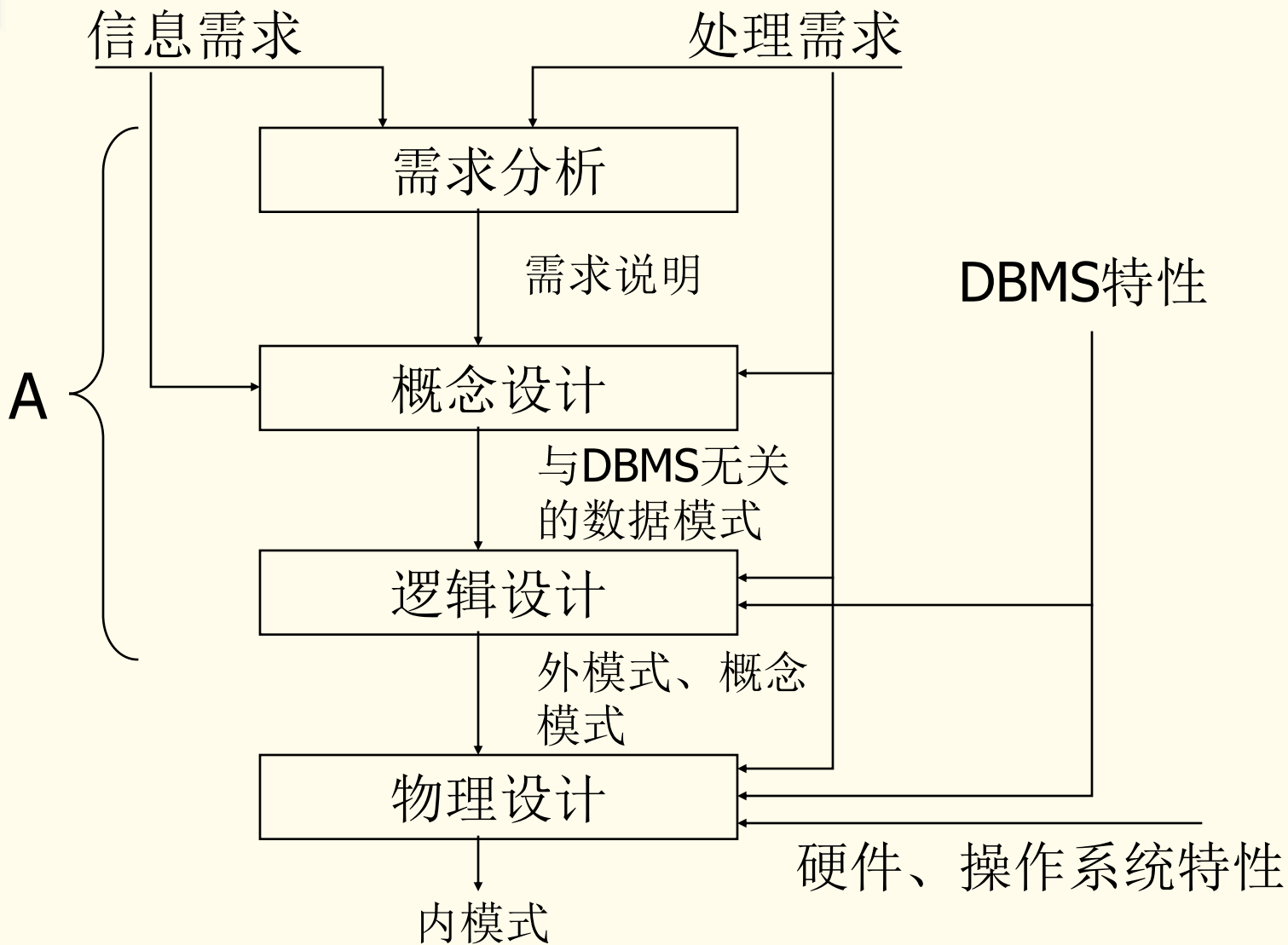
1) 分布式数据库

- 数据分割的设计
- 裂片分布方案的设计

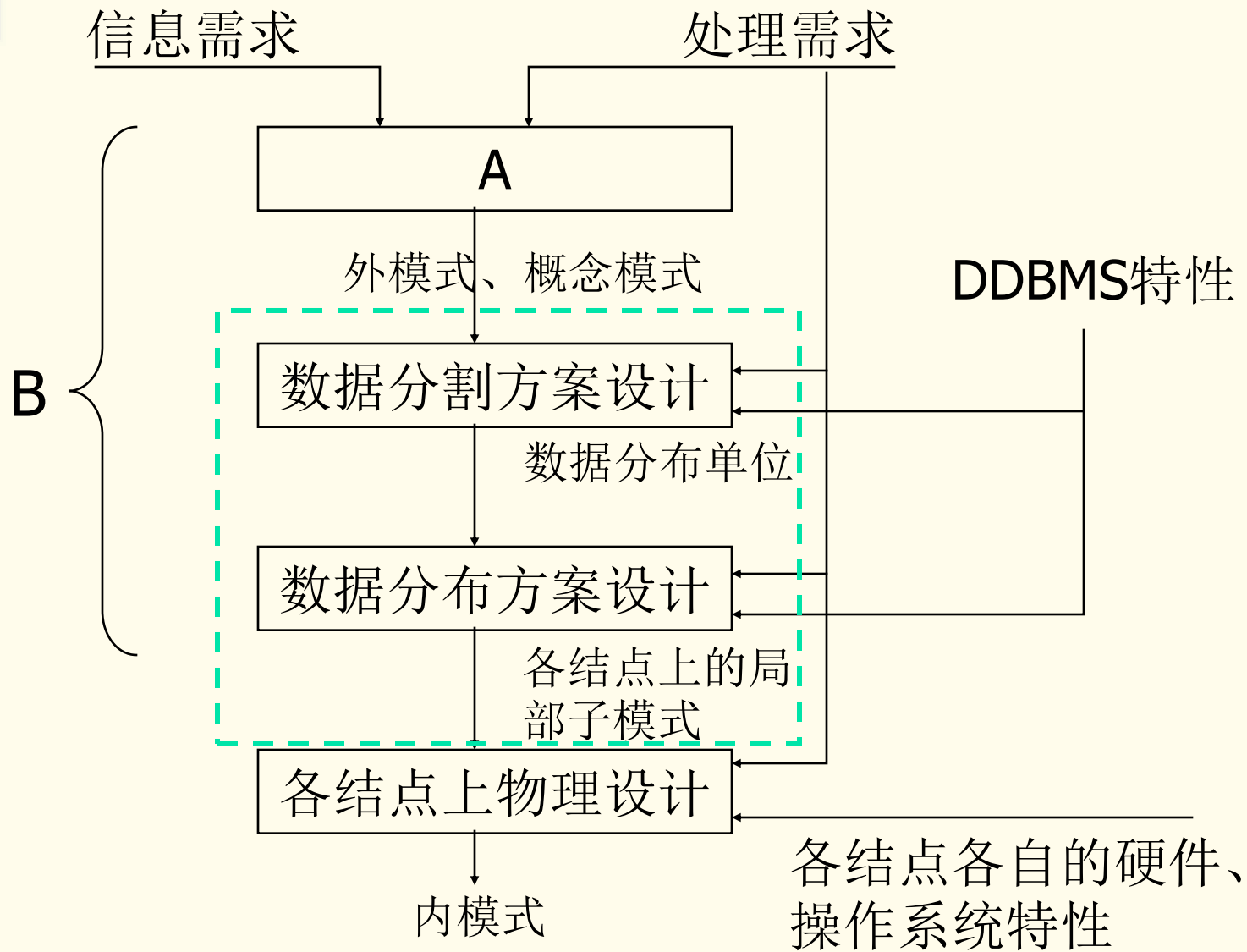
根据需求，对每个应用，提出基本问题：

- 该应用的发生结点
- 该应用发生的频度
- 要访问的数据对象

集中式数据库设计流程图



分布式数据库设计流程图





(1) 数据分割的设计

DDB中，裂片并非分得越细越好，完全根据应用的需要。例如下面两个应用：

应用1： `SELECT GRADE FROM STUDENT
WHERE DEPT=c AND AGE>20;`

应用2： `SELECT AVG(GRADE)
FROM STUDENT
WHERE SEX='Male';`

是否应将STUDENT按系作水平分割？



一般的规则:

①选择一些重要的、经常发生的典型应用。

②分析这些应用的访问局部性。

■对水平分割:

③用合适的谓词将全局关系划分成裂片以适应不同结点上的局部需求。如有矛盾，以更重要的应用的需求为准。

④分析应用的连接操作以决定是否需要作导出分割。



- 对垂直分割:

- ③分析属性亲密度。结合考虑:

- ✓节省空间和I/O代价

- ✓安全性。某些属性不让某些用户看到。

(2)数据分布方案的设计

通过代价估算以决定各分布单位的合适的存放位置(结点)。王书p252。



2) 并行数据库

- 什么是并行数据库系统？
- 无共享(SN)结构
- 垂直并行和水平并行
- 查询操作可分解为若干执行步骤，这些步骤间的并行执行就是垂直并行。
- 对扫描类子任务，如被扫描关系事先分割成若干裂片存放在SN结构并行机的不同硬盘上，则可以按并行扫描的方式进行。这就是水平并行。



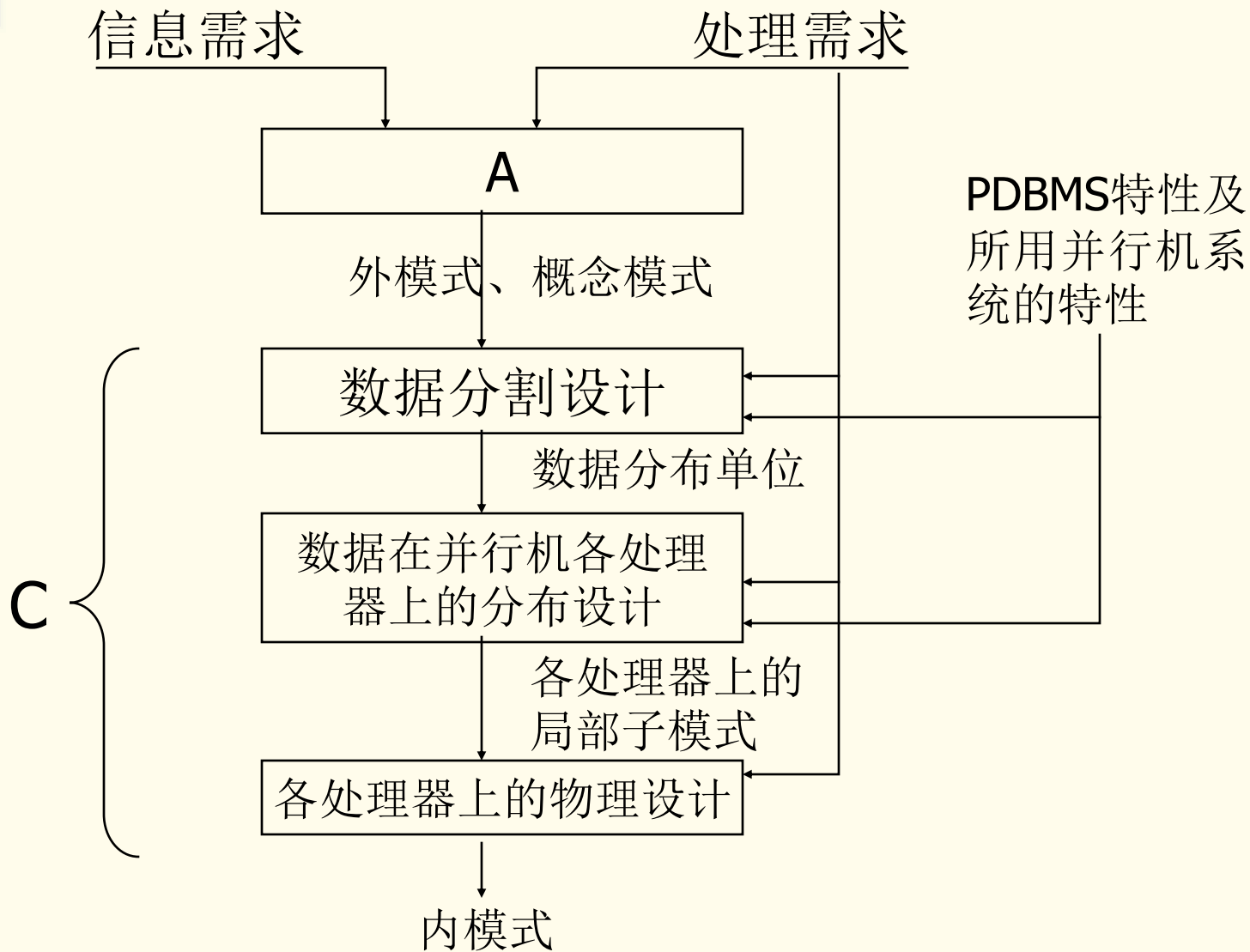
例子:

```
SELECT *  
FROM R,S  
WHERE R.a=S.a AND  
R.b>20 AND  
S.c<10;
```

$R' = \sigma_{b>20}(R)$ } 垂直
 $S' = \sigma_{c<10}(S)$ } 并行
 $R' \bowtie S'$ } 水平
并行

水平并行的前提是R、S事先被分割存放在不同硬盘上。这也是PDB设计时要解决的主要问题。

并行数据库设计流程图





PDB中数据分割方式

(1)任意方式

将某关系R以任意方式分割成裂片，分别存放在不同的处理器上。比如可以将R等分成几段、以hash方式散列成n段等。

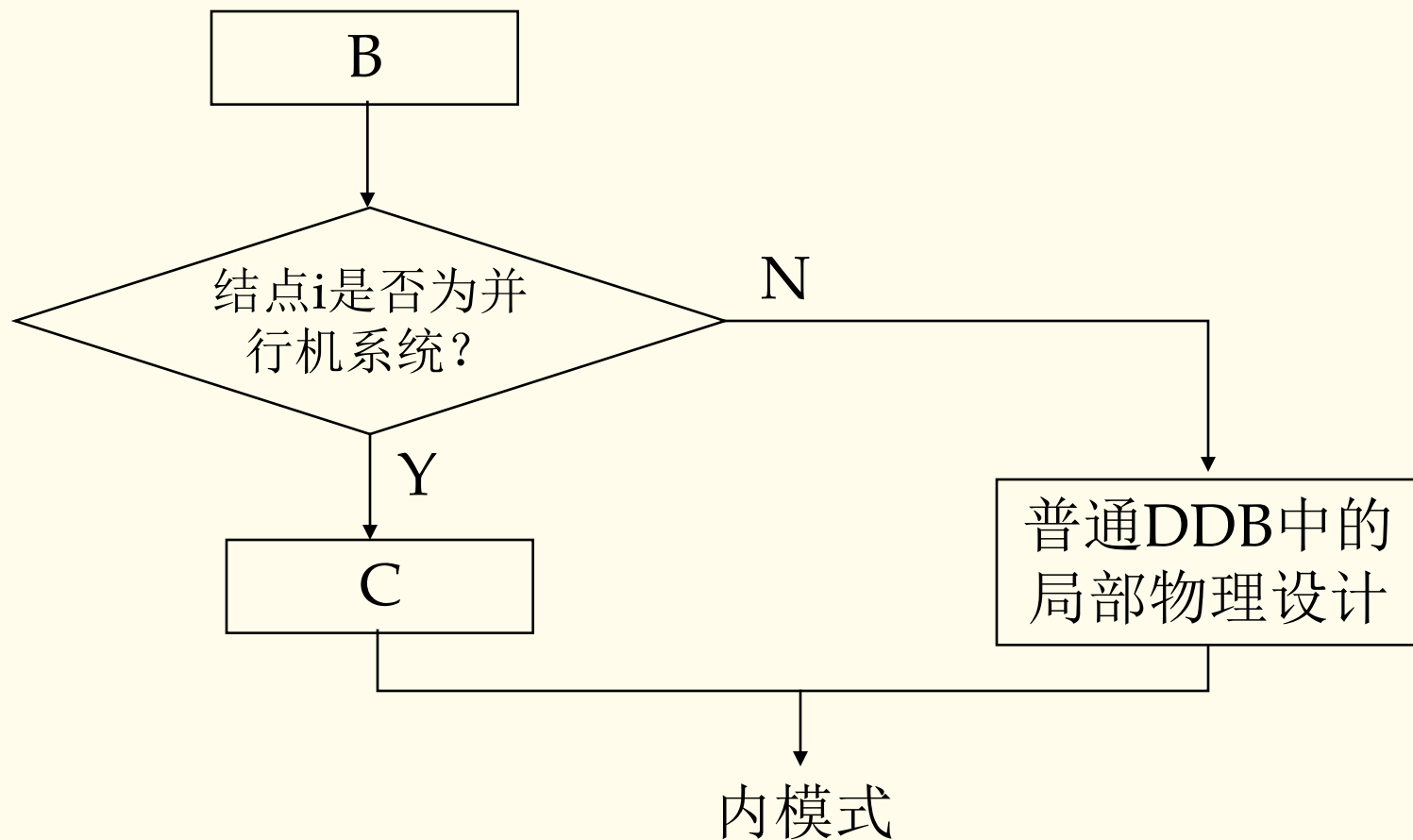
(2)基于表达式方式

将满足某一条件的元组指定存储于某个裂片内，适合于对该关系的查询大多基于分片条件的情况——分块排除。可以与方式1配合使用。

PDB与DDB关于数据的分割与分布的区别

	PDB	DDB
分割与分布的目的	提高处理的并行性，充分利用并行机的并行处理能力	提高访问的局部性，减少网上数据传输量
分割依据	所用并行机系统及PDBMS的特性，结合用户处理需求	用户的处理需求，结合所用DDBMS的特性
分布方式	在一台并行机的多个处理器上分布	在网络的多个结点上分布

带PDB的DDB系统设计流程

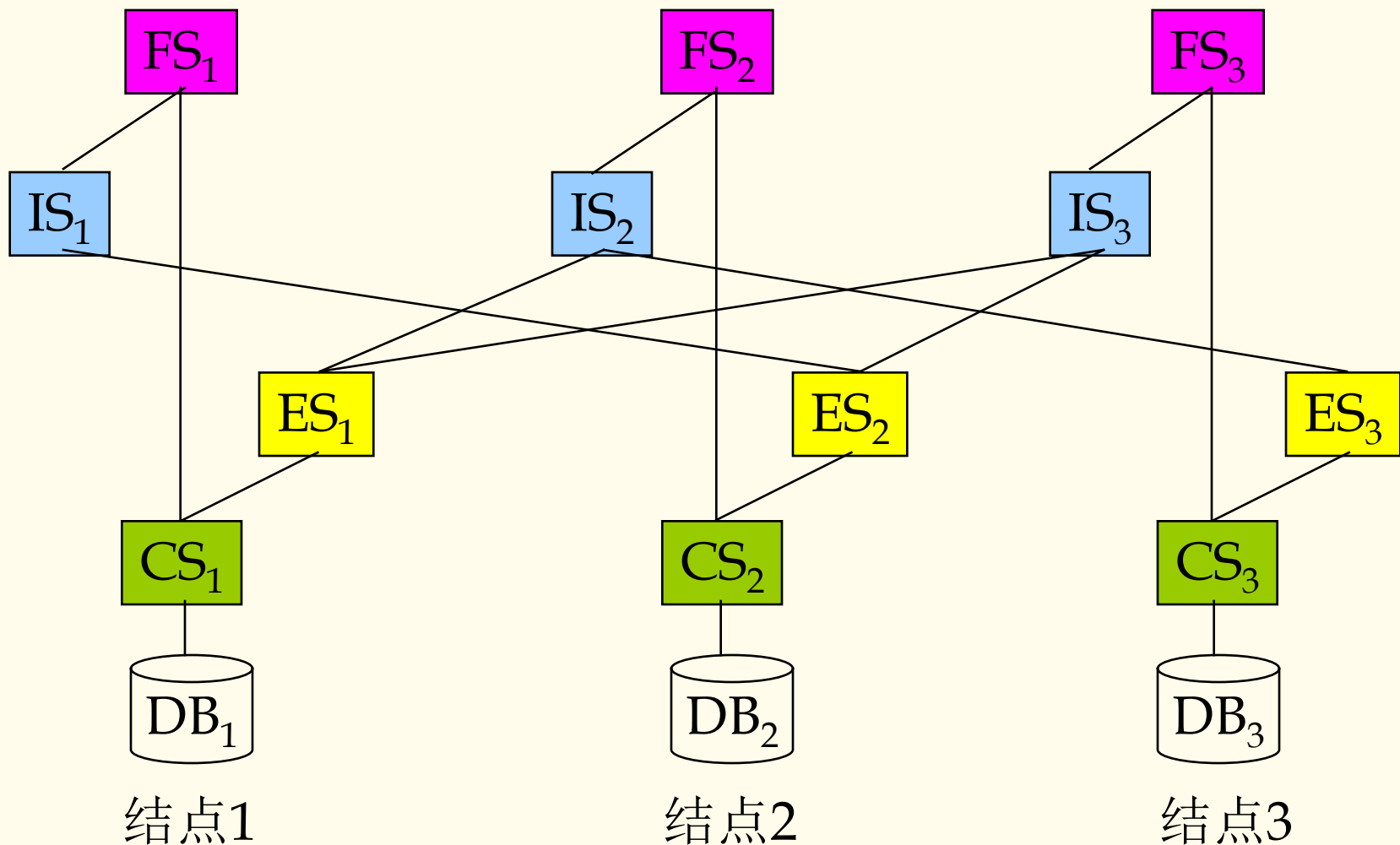





3) 联邦式数据库

- 应用中需要解决已有的、分布的、异构的多数数据库的集成。
- 各成员自治并相互协商合作的数据库系统——联邦式数据库。
- 无全局模式，联邦成员保持各自的数据模式。
- 成员之间通过协商确定输入/输出模式从而建立彼此的数据共享关系。

联邦式数据库的模式结构



- 
- $FS_i = CS_i + IS_i$
 - FS_i 是结点*i*上的用户所看到的所有可用数据。
 - IS_i 通过与其它结点的 ES_j 协商而得($j \neq i$)。
 - 用户对 FS_i 的查询 \Rightarrow 对 CS_i 和 IS_i 的查询 \Rightarrow 对相应 ES_j 的查询。
 - 对 ES_j 的查询结果 \Rightarrow 对应 IS_i 的结果， IS_i 的结果和 CS_i 的结果最后综合成 FS_i 的结果。




4.6 数据目录的分布

Catalog — — Data about data.

主要用于将用户需求转换成系统内的物理目标。

4.6.1 Contents of Catalogs

- (1)对象的类型(如关系、视图...)及其模式
- (2)分布信息(如fragment location)
- (3)存取路径
- (4)授权信息
- (5)用于决定优化的存取计划的一些统计信息



(1)~(4)不常变化，而(5)每次更新操作时就会变化。
对此：

- 定期成批更新
- 每次Update之后更新

4.6.2 目录的特征

- (1)主要是读操作
- (2)对系统的效率及数据分布的透明性很重要
- (3)对结点自治很重要
- (4)某些一致性要求不像数据那么严格
- (5)目录的分布由DDBMS的体系结构决定



4.6.3 目录的分布策略

(1) 集中式

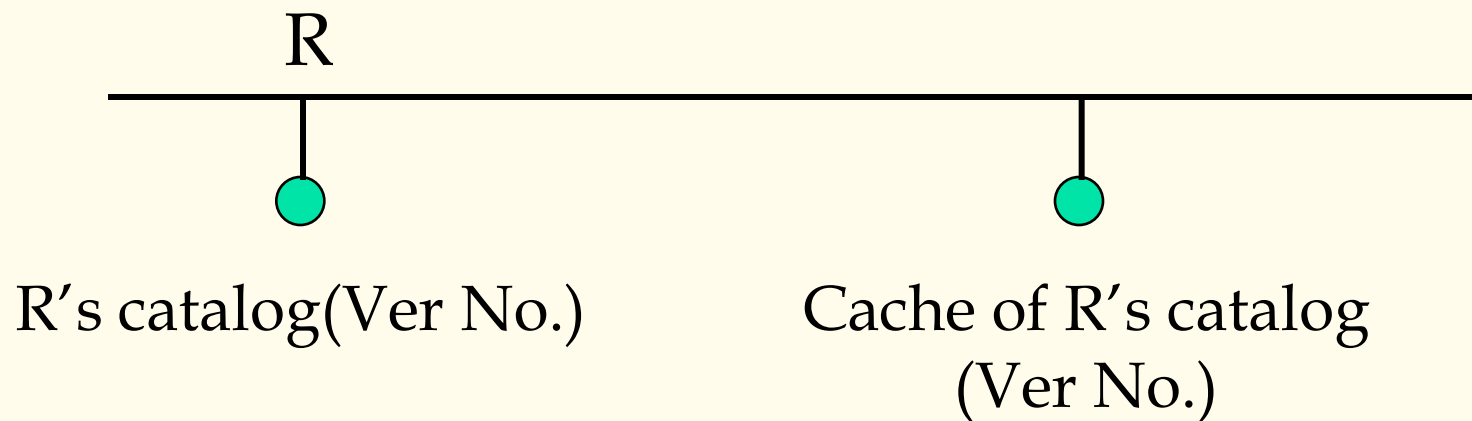
- A complete catalog stored at one site.
- Extended centralized catalog: 开始时集中于一点; 用完留下。更新时通知。

(2) 全复制目录: Catalogs are replicated at each site. Simple in retrieval. Complex in update. Poor in autonomy.

(3) 局部目录: Catalogs for local data are stored at each site. 即目录随数据存放。

如要查其它结点的数据：(广播找人)

- Master Catalog:在某结点上存放一个全部目录,使每个目录项都有两个复本。
- Cache:取得其它结点目录信息后,用完留下。通过比较版本号更新Cache的目录






(4) Different Combination of the above

对目录中的不同内容，采用上述的不同目录分布方法，从而得到不同的组合。如：

- a. 对分布信息(2)采用全复制，其它内容局部存放。
- b. 对统计信息采用局部存放，其它部分全复制。

4.6.4 R*的目录管理——结点自治

- 特点：
 - 无全局模式的概念(指目录)
 - 独立的命名和数据定义
 - 平稳地实现数据增长
- 最重要的概念——系统范围名(SWN)
<SWN> ::= User@UserSite.ObjectName@BirthSite
 - ObjectName:用户为相应对象起的名字
 - User:用户名。它可使不同用户用相同的名字访问不同的数据。

- 
- ▶ **UserSite**: 用户所在结点。它可使不同结点上的不同用户使用相同的用户名。
 - ▶ **BirthSite**: 该数据的产生结点。R* 目录无全局模式：At the birth site the information about the data is always kept even the data is migrated to other site.
 - **打印名**：用户在各种语句中所用的数据对象名。
 $\langle \text{PN} \rangle ::= [\text{User}[\text{@UserSite}].]\text{ObjectName}[\text{@birthSite}]$

▪ Name Resolution – Mapping PN to SWN

为每个用户建立一张同义词表
(用Define Synonym 语句)。

ObjectName	SWN
⋮	⋮

变换时可能有以下几种情况：

- 1) PrintName = SWN, 不需转换
- 2) 只有ObjectName:查当前结点上的当前用户的同义词表
- 3) User.ObjectName:查当前结点上User用户的同义词表
- 4) User@UserSite.ObjectName



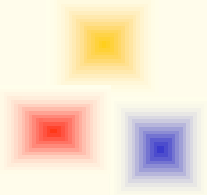
5) ObjectName@BirthSite

If no match for the ObjectName is found in (2), (3) or print name is in the form of (4) or (5), name completion is used:

补缺规则:

- A missing User is replaced by current User.
- A missing UserSite or BirthSite is replaced by current site ID.

5 查询处理





简介

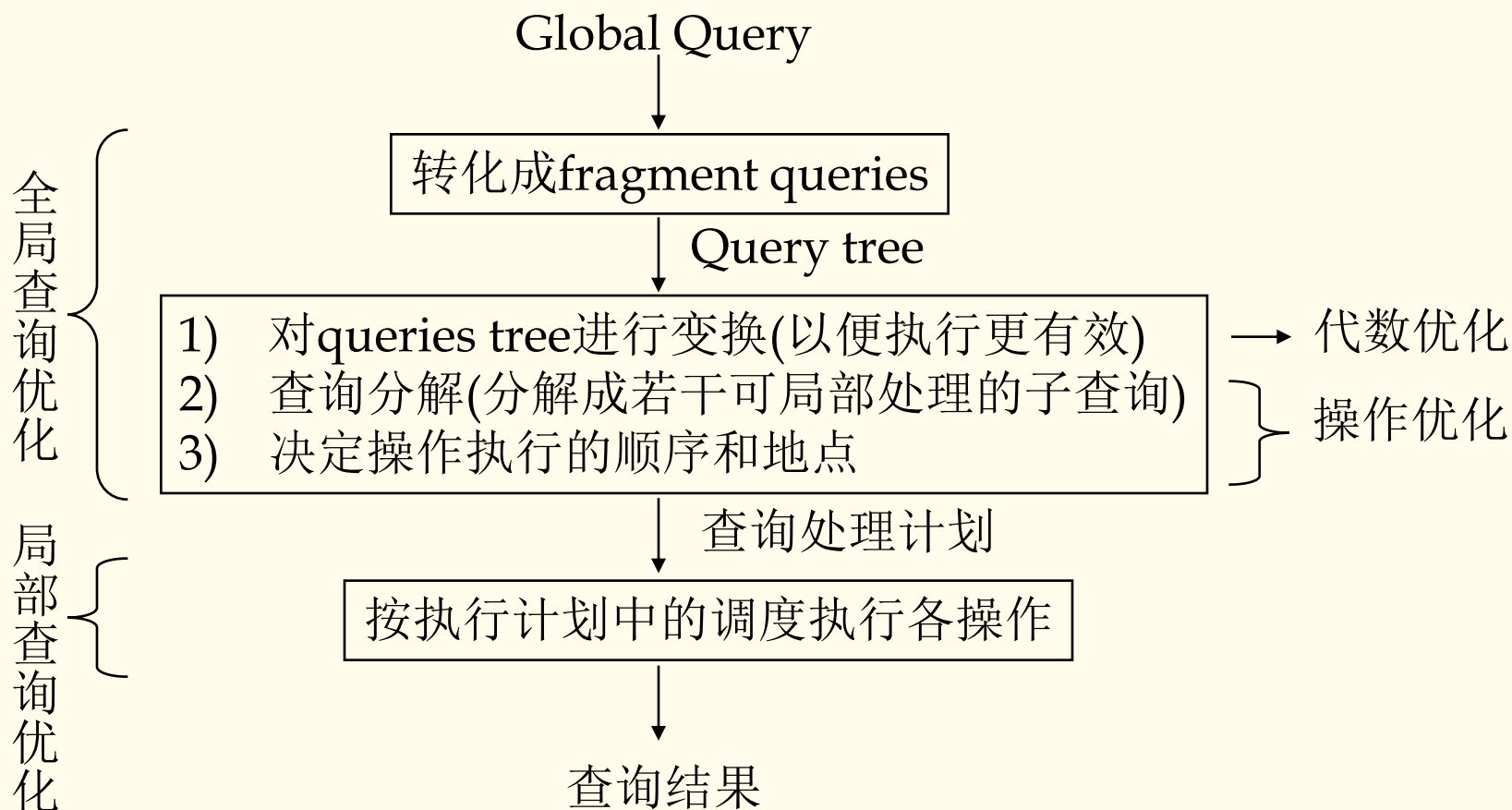
- 首先对用户所写的程序（查询）进行“改写”，然后确定最有效的具体操作方法和步骤，获得查询结果。
- 目标是以最小的代价、最快的速度得出用户查询的结果。



5.1 DDBMS查询处理概述

- 全局查询（Global Query）：
用户所写的针对全局关系的查询
- 裂片查询（Fragment Query）：
系统处理后转换的针对裂片的查询

总流程





例子:

S(SNUM,SNAME,CITY)

SP(SNUM,PNUM,QUAN)

P(PNUM,PNAME,WEIGHT,SIZE)

设分割情况如下:

$$S1 = \sigma_{CITY='Nanjing'}(S)$$
$$S2 = \sigma_{CITY='Shanghai'}(S)$$
$$SP1 = SP \bowtie S1$$
$$SP2 = SP \bowtie S2$$

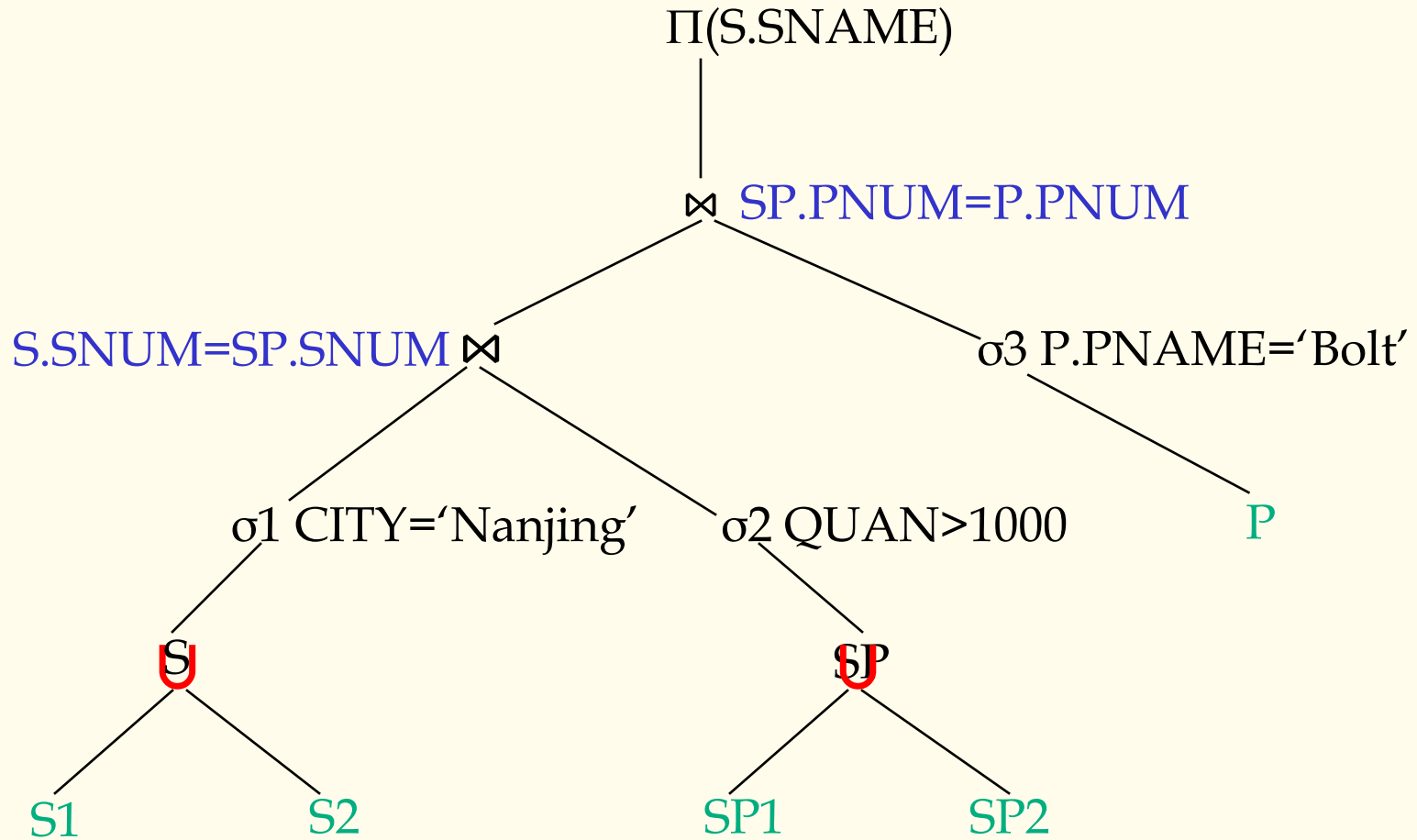


Global Query:

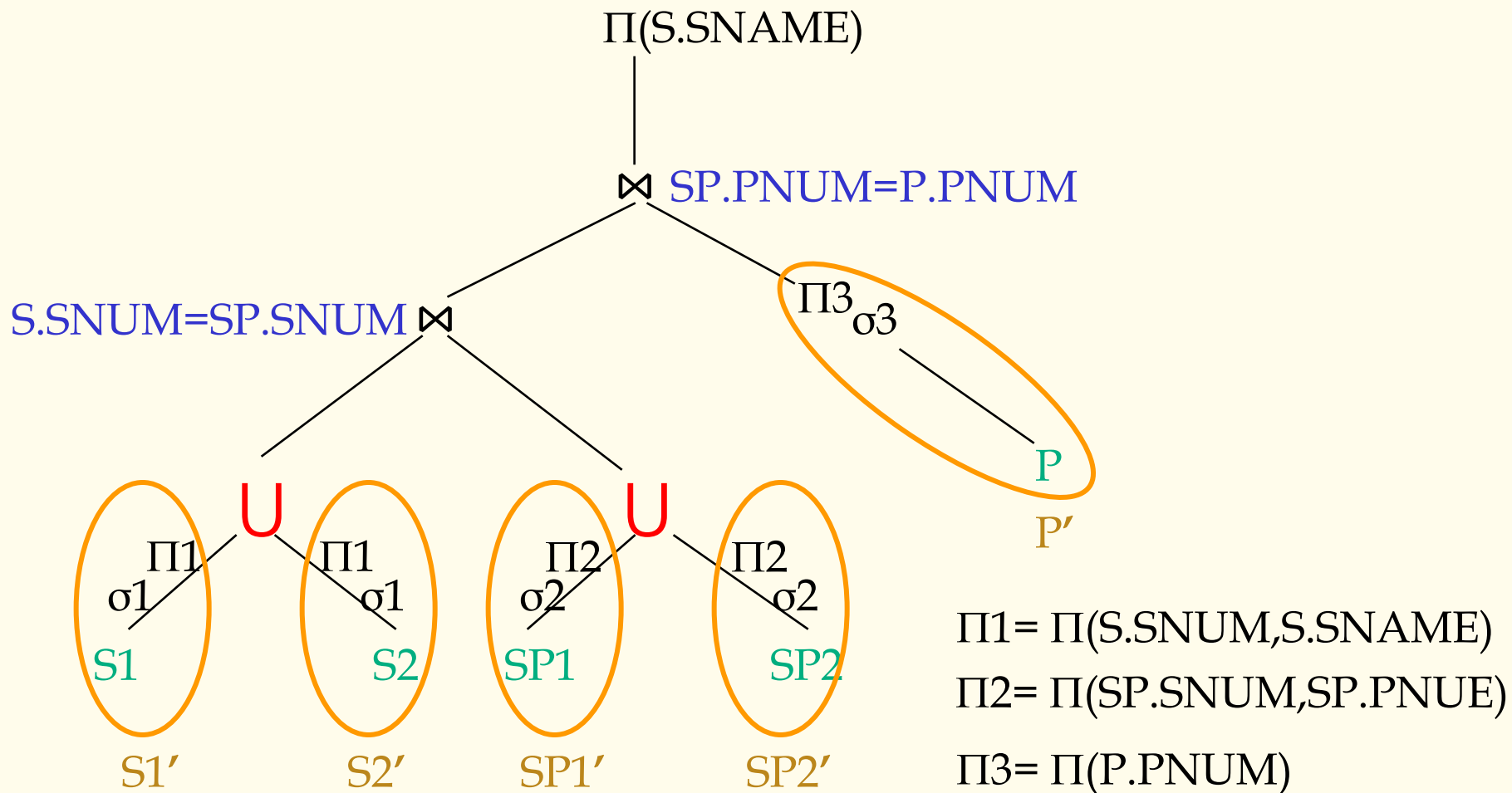
```
SELECT SNAME  
FROM S,SP,P  
WHERE S.SNUM=SP.SNUM AND  
       SP.PNUM=P.PNUM AND  
       S.CITY='Nanjing' AND  
       P.PNAME='Bolt' AND  
       SP.QUAN>1000;
```



Query tree

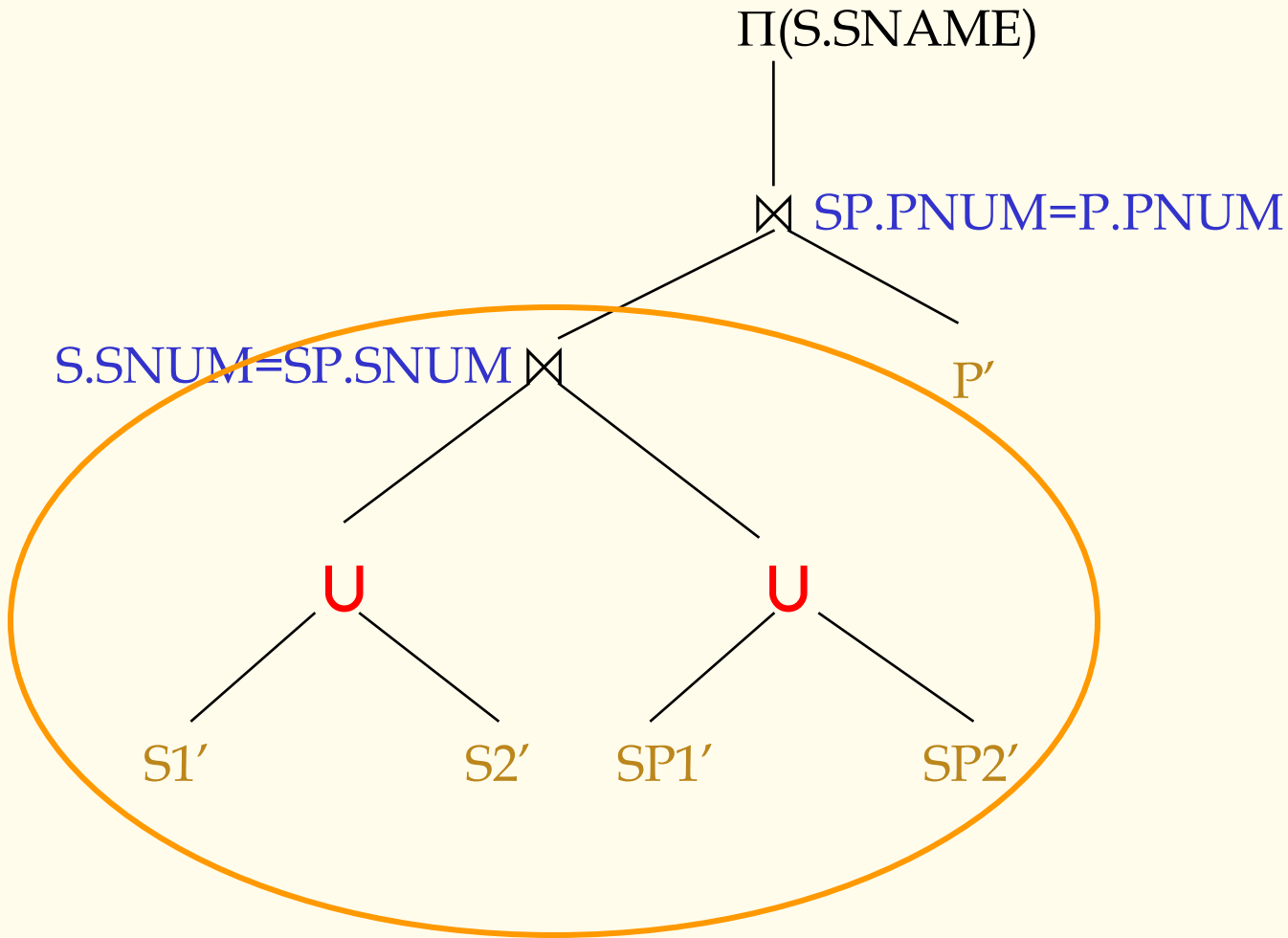


经过等价变换:



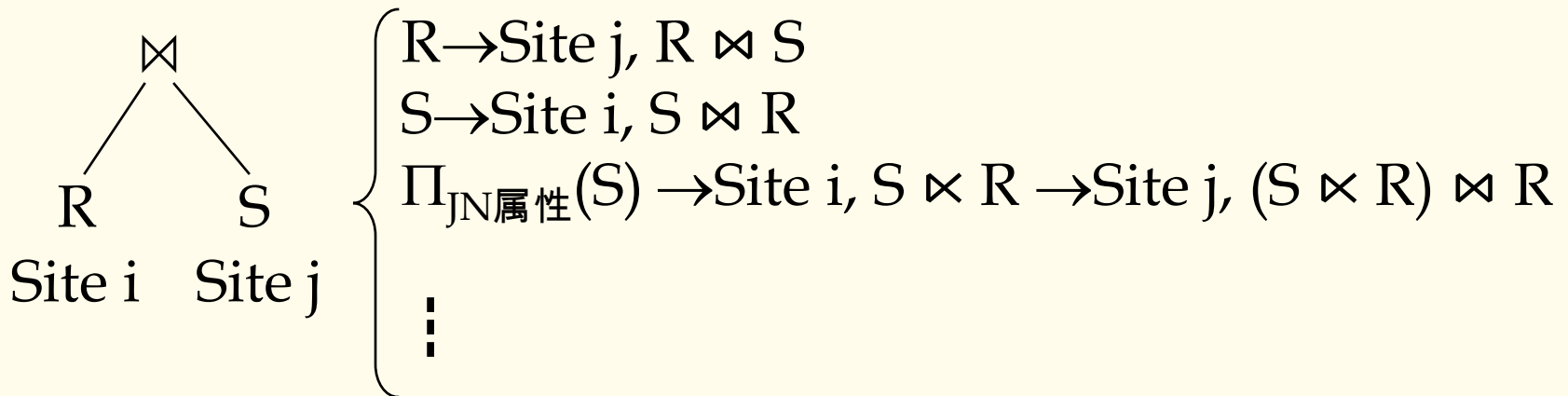


变换结果:



子树部分的操作优化:

- 先看分布JN: (1) $(S1' \cup S2') \text{ JN } (SP1' \cup SP2')$
(2) Distributed Join
- 再考虑Site Selection, 产生很多组合
- 每个Join本身, 也有很多实现方法:



查询优化就是要从这些可能的方案中找出较好的。复杂。

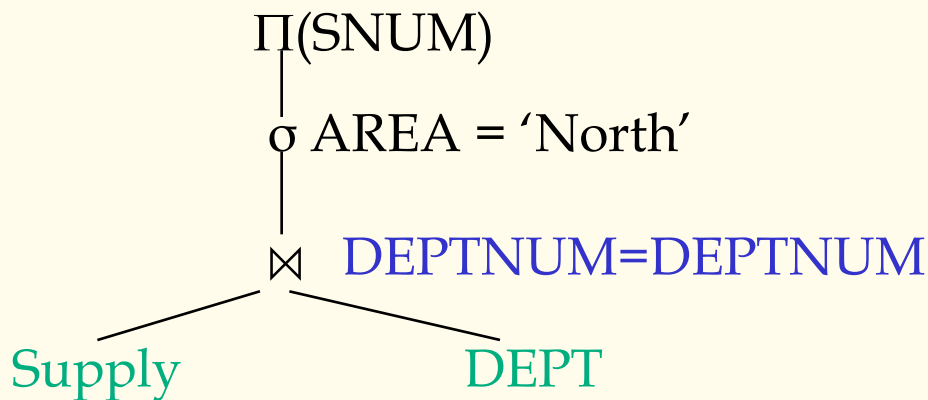
5.2 查询的等价变换

即代数优化。对原查询进行一系列的变换，将其转换成便于执行的最佳形式。但变换必须等价。

例 $\Pi_{\text{NAME,DEPT}} \sigma_{\text{DEPT}=15}(\text{EMP}) \equiv \sigma_{\text{DEPT}=15} \Pi_{\text{NAME,DEPT}}(\text{EMP})$

(1) 查询树

例 $\Pi_{\text{SNUM}} \sigma_{\text{AREA}='NORTH'}(\text{SUPPLY} \bowtie_{\text{DEPTNUM}} \text{DEPT})$



树叶：Global Relation
中间结点：一元/二元操作
叶→根：操作的执行顺序

(2) 关系代数的等价变换规则

- 1) \bowtie/\times 的交换律: $E1 \times E2 \equiv E2 \times E1$
- 2) \bowtie/\times 的结合律: $E1 \times (E2 \times E3) \equiv (E1 \times E2) \times E3$
- 3) Π 的串接律: $\Pi_{A_1 \dots A_n}(\Pi_{B_1 \dots B_m}(E)) \equiv \Pi_{A_1 \dots A_n}(E)$
 $A_1 \dots A_n$ 为 $\{B_1 \dots B_m\}$ 的子集时才合法。
- 4) σ 的串接律: $\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$
- 5) σ 与 Π 的交换律: $\sigma_F(\Pi_{A_1 \dots A_n}(E)) \equiv \Pi_{A_1 \dots A_n}(\sigma_F(E))$
如 F 中也包括不属于 $A_1 \dots A_n$ 的属性 $B_1 \dots B_m$,
则 $\Pi_{A_1 \dots A_n}(\sigma_F(E)) \equiv \Pi_{A_1 \dots A_n} \sigma_F(\Pi_{A_1 \dots A_n, B_1 \dots B_m}(E))$
- 6) 如 F 中出现的属性全是 $E1$ 中的属性, 则
 $\sigma_F(E1 \times E2) \equiv \sigma_F(E1) \times E2$



如F有形式 $F1 \wedge F2$ ，且F1中只出现E1的属性，
F2只出现E2的属性，则

$$\sigma_F(E1 \times E2) \equiv \sigma_{F1}(E1) \times \sigma_{F2}(E2)$$

如F有形式 $F1 \wedge F2$ ，且F1中只含E1的属性，
而F2含E1和E2两者的属性，则

$$\sigma_F(E1 \times E2) \equiv \sigma_{F2}(\sigma_{F1}(E1) \times E2)$$

7) $\sigma_F(E1 \cup E2) \equiv \sigma_F(E1) \cup \sigma_F(E2)$

8) $\sigma_F(E1 - E2) \equiv \sigma_F(E1) - \sigma_F(E2)$

9) 设 $A_1 \dots A_n$ 为一列属性，其中的 $B_1 \dots B_m$ 是E1的
属性， $C_1 \dots C_k$ 为E2的属性，则

5.3 将Global Queries转换成Fragment Queries

- 方法:

- 对水平分割: $R=R1 \cup R2 \cup \dots \cup Rn$

- 对垂直分割: $S=S1 \bowtie S2 \bowtie \dots \bowtie Sn$

将上式带入替换Global Relation即可。

- 带入后的正则表达式需用前述规则变换, 原则:

- 1)将一元操作尽可能向下压。

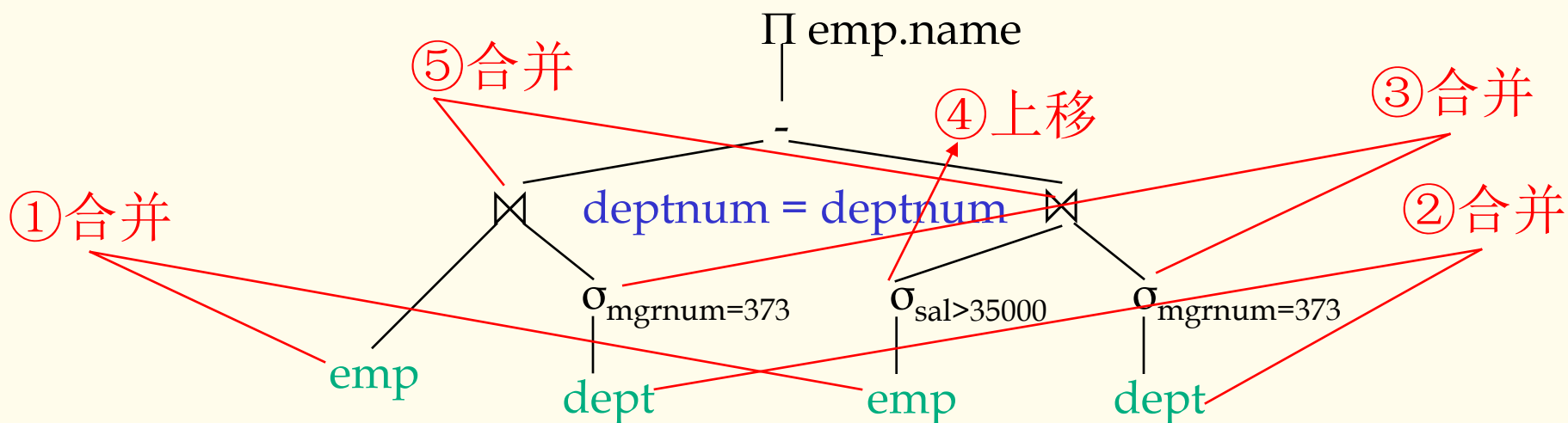
- 2)寻找并合并公共子表达式(Common Sub-expression)

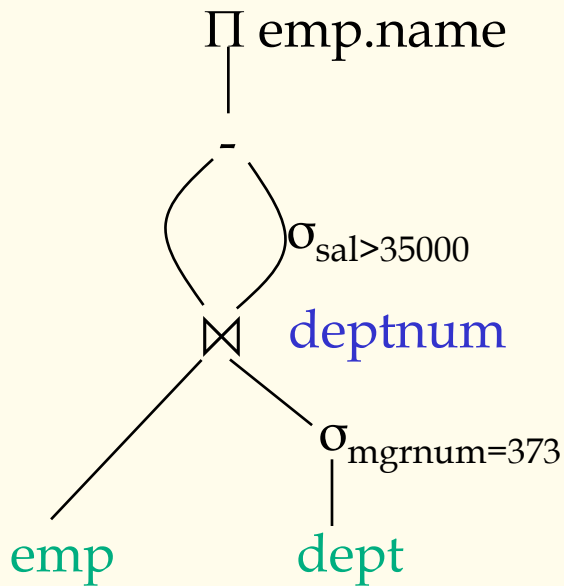
- ✓ 定义: 在查询表达式中不止出现一次的表达式。
如找出这种子表达式, 对其只计算一次, 无疑会提高效率。



- ✓ 一般方法： (1)合并操作树上的相同叶结点。
(2)合并相对于带相同操作数的相同操作的中间结点。

例： $\Pi_{emp.name}(emp \bowtie (\sigma_{mgrnum=373} dept) - (\sigma_{sal>35000} emp) \bowtie (\sigma_{mgrnum=373} dept))$





公共子表达式:

$emp \bowtie (\sigma_{mgrnum=373} dept)$

性质:

✓ $R \bowtie R \equiv R$

✓ $R \cup R \equiv R$

✓ $R - R \equiv \Phi$

✓ $R \bowtie \sigma_F(R) \equiv \sigma_F(R)$

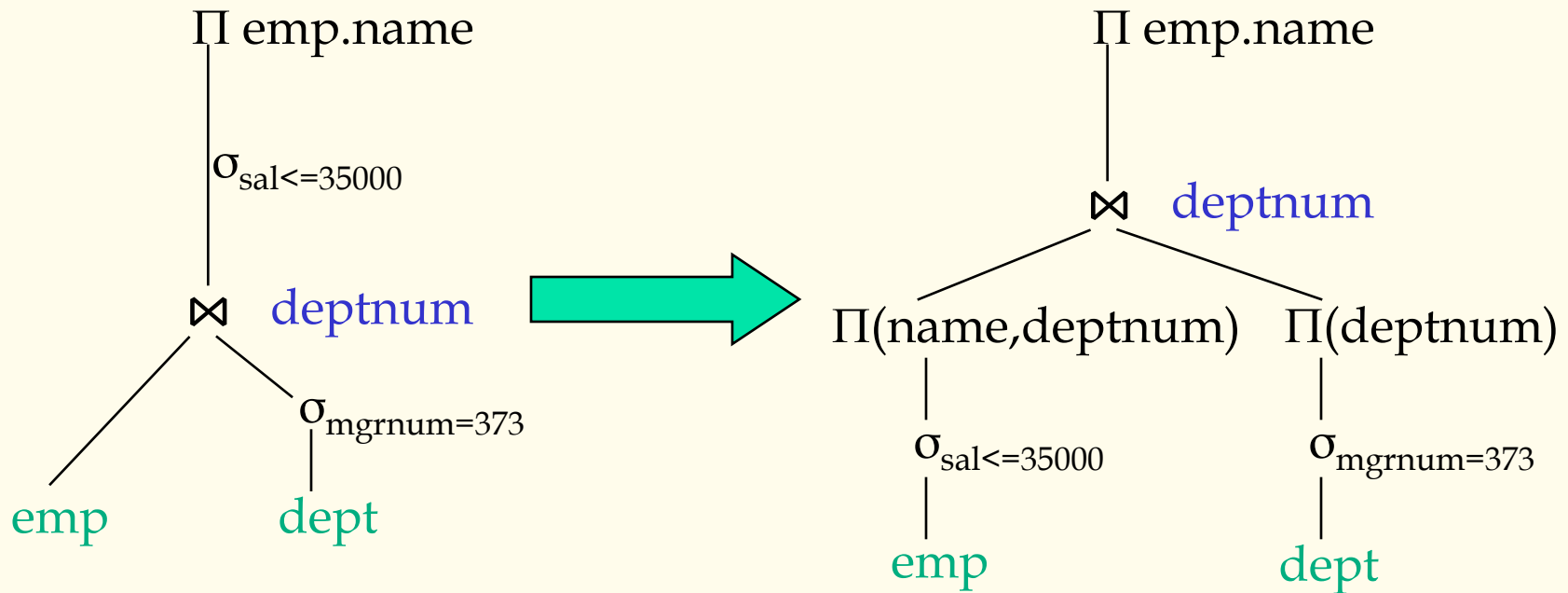
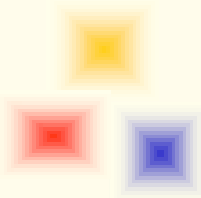
✓ $R \cup \sigma_F(R) \equiv R$

✓ $R - \sigma_F(R) \equiv \sigma_{\text{not } F} R$

✓ $\sigma_{F1}(R) \bowtie \sigma_{F2}(R) \equiv \sigma_{F1 \wedge F2}(R)$

✓ $\sigma_{F1}(R) \cup \sigma_{F2}(R) \equiv \sigma_{F1 \vee F2}(R)$

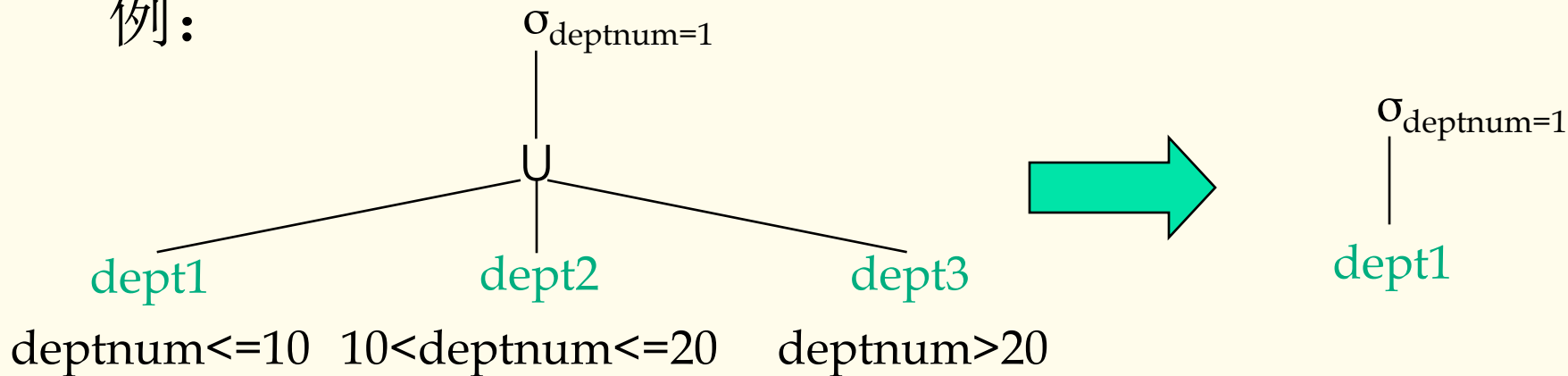
✓ $\sigma_{F1}(R) - \sigma_{F2}(R) \equiv \sigma_{F1 \wedge \text{not } F2}(R)$



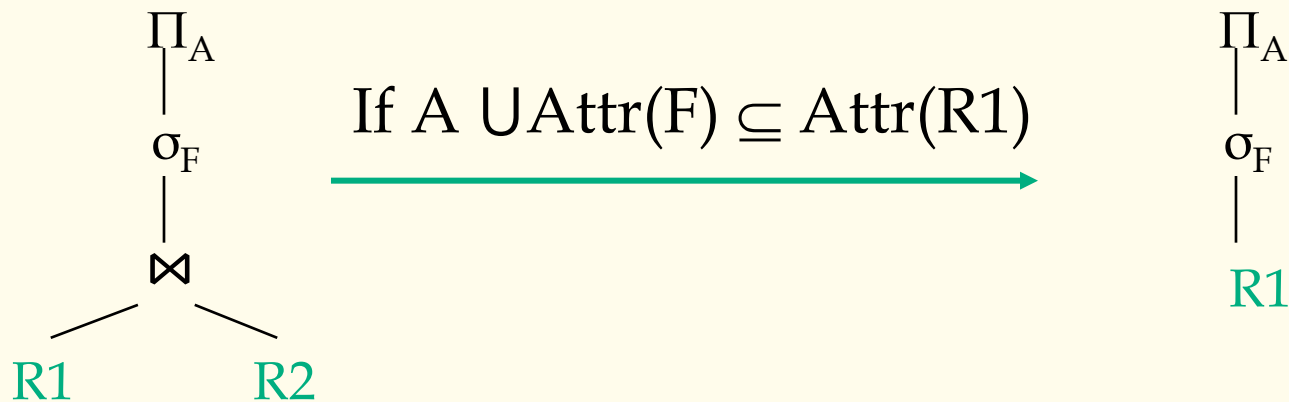
注意：最后得到的操作树就是expert级的用户一开始就能写出的。代数优化的目的就是化简这种一开始没有写成最佳形式的查询描述。

3) 确定并消除空的子表达式

例:

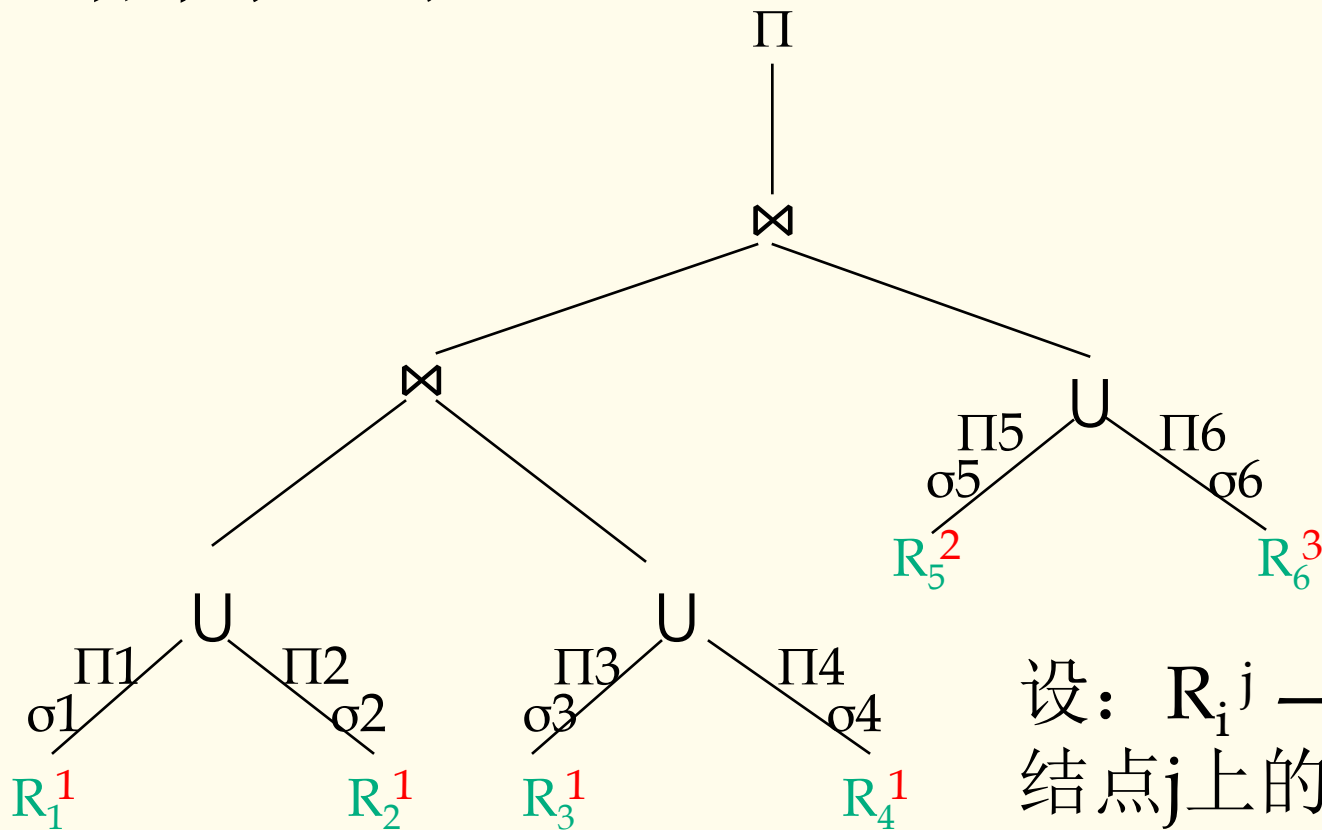


4) 消除无用的垂直分割



5.4 将查询分解成子查询

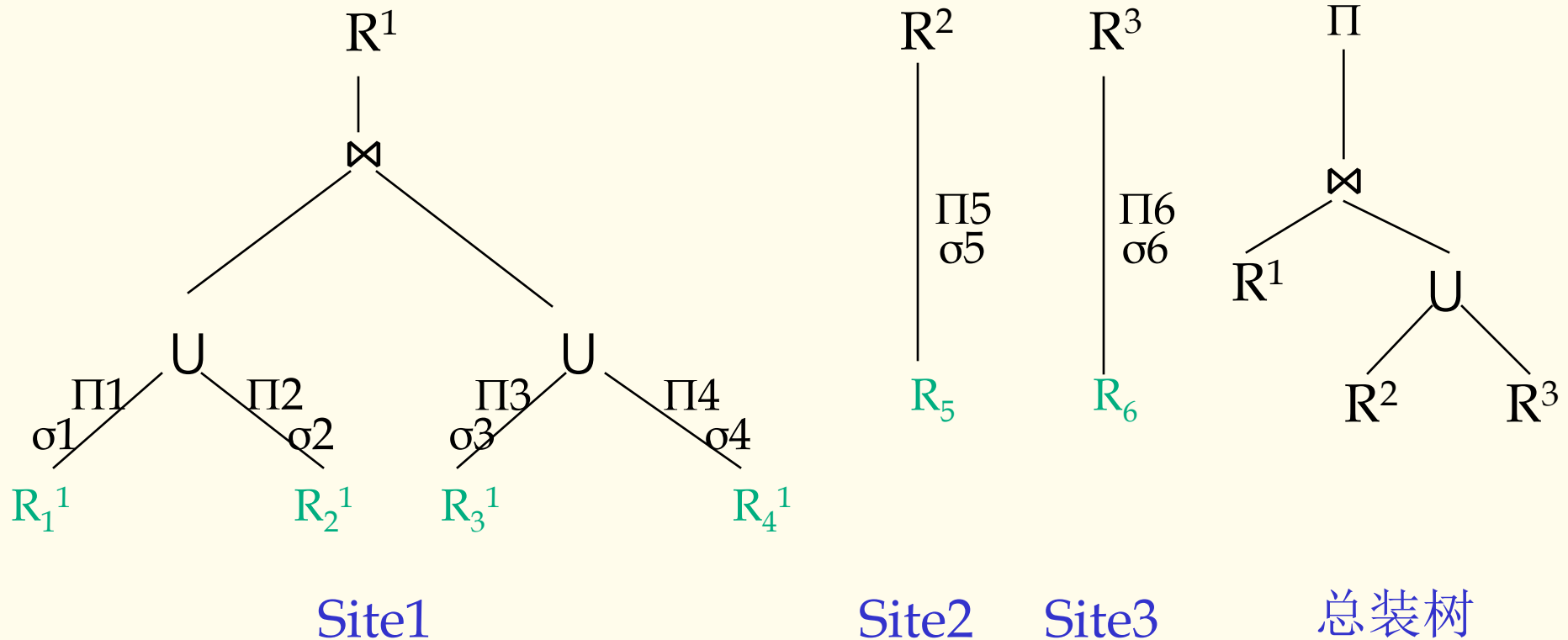
考虑到结点的情况，将查询分解成可局部处理的若干子查询：



设： R_i^j — 存放在
结点j上的裂片 R_i

分解方法:

对查询树进行后序遍历, 直到发现 j 为2时, 暂停, 得到第1棵子树, 依此类推, 得到所有子树。





5.5 二元操作执行的优化

上述局部子查询的执行由相应结点的局部DBMS负责，DDBMS的查询优化负责解决Global optimization问题，即总装树的执行。

由于一元操作经代数优化、查询分解后全部交由局部DBMS负责，所以DDBMS的全局查询优化只需考虑二元操作，主要是Join操作。

本节介绍的就是如何根据具体环境，找到一个“好”的存取策略来计算出经代数优化“改良”过的查询。



一. 全局查询优化中的主要问题

1) 决定查询所涉及的fragments的复本, 即复本选择(或Site selection)

2) Strategies for join operation

Non_distributed join:

$$(R^2 \cup R^3) \bowtie R^1$$


Distributed join:

$$(R^1 \bowtie R^2) \cup (R^1 \bowtie R^3)$$


Direct join

Use of semi_join

3) 选择每个操作的执行方法(主要对direct join而言)

- 
- **Nested loop:** 实际相当于二重循环，对外循环关系的每个元组，将内循环关系扫描一遍。关系不是以元组为单位，而是以物理块为单位从磁盘取到内存的。所以为提高效率，对于 $R \bowtie S$ ，如果以 R 为外关系， S 为内关系， b_R 为 R 的物理块数， b_S 为 S 的物理块数，系统提供 n_B 个缓冲区 ($n_B \geq 2$)，其中 $n_B - 1$ 个块为外关系的缓冲区，1 个块为内关系的缓冲区，则总的访盘次数为：

$$b_R + \lceil b_R / (n_B - 1) \rceil \times b_S$$

- 
- Merge scan: 将R、S事先按连接属性排序，则可按序比较两者的元组，各扫描一遍即可。如R、S事先未按连接属性排序，则需仔细权衡是否用此方法。(参王书p122)
 - 利用索引或散列寻找匹配元组法：在嵌套循环法中，如内关系有合适的存取路径，则可考虑使用这些存取路径取代顺序扫描，特别当连接属性上有簇集索引或散列时，最为有利。
 - 散列连接法：R、S连接属性有相同的域，用同一散列函数将R、S散列到同一散列文件中。

- 
- 上述均为集中式关系DBMS中的经典算法，在DDBMS中用direct join实现连接时，思路类似。

二. 优化方法一般有三种：

1) By cost comparison(也称exhaustive search)


2) By heuristic rule. 小系统用的较多。

例王书p124连接方法的启发式规则。

3) Combination of 1, 2: 基本同1，但可先用2去掉明显不合适的方案，减小解空间。

三. Cost estimation

Total query cost=Processing cost+Transmission cost



根据环境不同:

- For wide area network: 传输率 100bps~50Kbps, 比机内处理速度低得多, 所以Processing cost通常忽略不计。
- For local area network: 传输率可达1000Mbps, 这时两项都必须考虑。

1) Transmission cost

$$TC(x)=C_0+C_1x$$

x: 传输数据量; C_0 : 初始化代价; C_1 : 网上传送1个数据的代价。 C_0 、 C_1 依赖于网络特性。



2) Processing cost

$$\text{Processing cost} = \text{cost}_{\text{cpu}} + \text{cost}_{\text{I/O}}$$

一般 cost_{cpu} 可以忽略。

$$\text{cost of one I/O} = D_0 + D_1$$

D_0 平均寻道时间(ms); D_1 : I/O一个数据(μs)

$$\text{cost}_{\text{I/O}} = \text{no. of I/O} \times D_0$$

- ❖ 注意：精确计算query cost是不必要也是不现实的，我们的目的是在不同查询执行策略之间进行比较，所以只需在相同运行环境下对不同方案的执行代价进行估算。

5.6 用semi_join实现join操作

一. semi_join的作用

Semi_join is used to reduce transmission cost. So it is suitable for WAN only.

$$R \bowtie S = \Pi_R(R \bowtie S)$$

如果R、S分别存放在结点1、2，用 \bowtie 实现 $R \bowtie S$ 的步骤：

- 1) 将 $\Pi_A(S) \rightarrow$ 结点1，A是连接属性
- 2) 在结点1上做 $R \bowtie \Pi_A(S) = R \bowtie S$ (压缩R)
- 3) 将 $R \bowtie S \rightarrow$ 结点2
- 4) 在结点2上做 $(R \bowtie S) \bowtie S = R \bowtie S$

代价比较

- Cost of direct join = $C_0 + C_1 \min(r, s)$ — — ①

r, s — — $|R|, |S|$ (关系的大小)

- Cost of join via semi_join =

$$\min(2C_0 + C_1 s' + C_1 r'', 2C_0 + C_1 r' + C_1 s'') = 2C_0 + C_1 \min(s' + r'', r' + s'') \quad \text{--- ②}$$

s', r' — — $|\Pi_A(S)|, |\Pi_A(R)|$

s'', r'' — — $|S \text{ SJ } R|, |R \text{ SJ } S|$

- 仅当② < ①时，使用semi_join才是合算的：
(1) C_0 must be small (2) 不宜用多元semi_join
(3) R或S的规模经semi_join后应大大减小



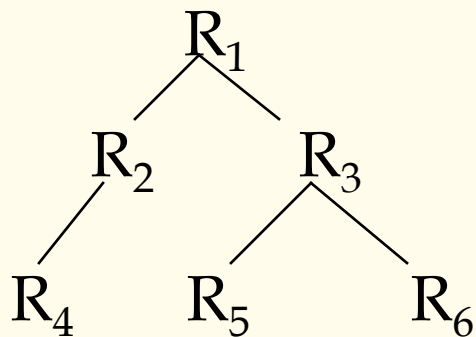
二. 关于semi_join的评论

- 1) 采用SJ对传输代价的减少是以牺牲处理代价为代价的。
- 2) Semi_join的可选方案很多
如查询 $R_1 \bowtie R_2 \bowtie R_3 \dots \bowtie R_n$, 对 R_1 做 \bowtie :
 $R_1 \bowtie R_2$, $R_1 \bowtie (R_2 \bowtie R_1)$, $R_1 \bowtie (R_2 \bowtie R_3) \dots$
要从所有可能方案中选出最好的几乎不可能。
- 3) Bernstein
 - \bowtie can be regarded as reducers.
 - Def: A chain of semi_join to reduce R is called reducer program for R.



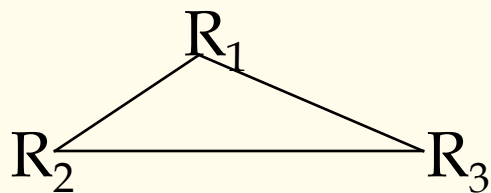
- $RED(Q,R)$: A set of all reducer programs for R in query Q.
 - Full reducer: 满足下列条件的reducer:
 - (1) $\in RED(Q,R)$
 - (2) reduce R mostly
 - 但full reducer并不是优化应该追求的目标
- 例1: Q is a query with qualification:
- $$q = (R_1.A = R_2.B) \wedge (R_1.C = R_3.D) \wedge (R_2.E = R_4.F) \\ \wedge (R_3.G = R_5.H) \wedge (R_3.J = R_6.K)$$

Query graph:



两关系间有 \bowtie 关系就用一条线连接，便可得Query graph。左图这种查询称为tree query(TQ)。

例2: $q = (R_1.A = R_2.B) \wedge (R_2.C = R_3.D) \wedge (R_3.E = R_1.F)$



左图这种查询称为cyclic query(CQ)。

例3: $q = (R_1.A = R_2.B) \wedge (R_2.B = R_3.C) \wedge (R_3.C = R_1.A)$

这是一个TQ而不是CQ，因为 $R_3.C = R_1.A$ 可由传递得到。

可以证明:

- 1) Full reducer exists for TQ.
- 2) No full reducer exists for CQ in many cases.

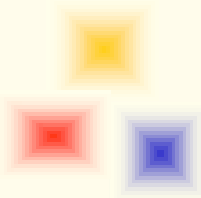
R ₁	A	B
	0	1
	3	4

R ₂	C	D
	1	2
	4	5

R ₃	E	F
	2	3
	5	0

$$q = (R_1.B = R_2.C) \wedge (R_2.D = R_3.E) \wedge (R_3.F = R_1.A)$$

查询结果为空，但R₁、R₂、R₃都无法通过SJ减小，所以不存在full reducer。



R	A	B
	1	a
	2	b
	3	c

S	B	C
	a	x
	b	y
	c	z

T	C	A
	x	2
	y	3
	z	4

$$q = (R.B=S.B) \wedge (S.C=T.C) \wedge (T.A=R.A)$$

查询结果?

$$R' = R \bowtie T = \begin{array}{c|cc} & A & B \\ \hline & 2 & b \\ & 3 & c \end{array}$$

$$S' = S \bowtie R' = \begin{array}{c|cc} & B & C \\ \hline & b & y \\ & c & z \end{array}$$

$$T' = T \bowtie S' = \begin{array}{c|cc} & C & A \\ \hline & y & 3 \\ & z & 4 \end{array}$$

$$R'' = R' \bowtie T' = \begin{array}{c|cc} & A & B \\ \hline & 3 & c \end{array}$$

$$S'' = S' \bowtie R'' = \begin{array}{c|cc} & B & C \\ \hline & c & z \end{array}$$

$$T'' = T' \bowtie S'' = \begin{array}{c|cc} & C & A \\ \hline & z & 4 \end{array}$$



$R''' = R'' \bowtie T'' = \Phi$, 所以R的full reducer:

① $R' = R \bowtie T$ ② $S' = S \bowtie R'$ ③ $T' = T \bowtie S'$ ④ $R'' = R' \bowtie T'$ ⑤
 $S'' = S' \bowtie R''$ ⑥ $T'' = T' \bowtie S''$ ⑦ $R''' = R'' \bowtie T'' = \Phi$

结论:

- 对CQ: 一般没有full reducer, 即使有, 其长度随查询中某关系的元组数线性增长($3(m-1)$)。
- 对TQ: full reducer的长度 $<n-1$ (n 为树中结点数)。
- 所以, 通过 \bowtie 实现分布环境下的连接操作不能以full reducer为优化目标。

SDD-1算法：启发式规则(王书p189~190)

5.7 Direct Join

— —R*中的连接操作实现方法

一. 两种基本的join实现方法

- Nested Loop: 集中式DBMS中相应算法的扩展

$$\text{代价} = (b_R + \lceil b_R / (n_B - 1) \rceil \times b_S) \times D_0$$

- Merge Scan: 集中式DBMS中相应算法的扩展

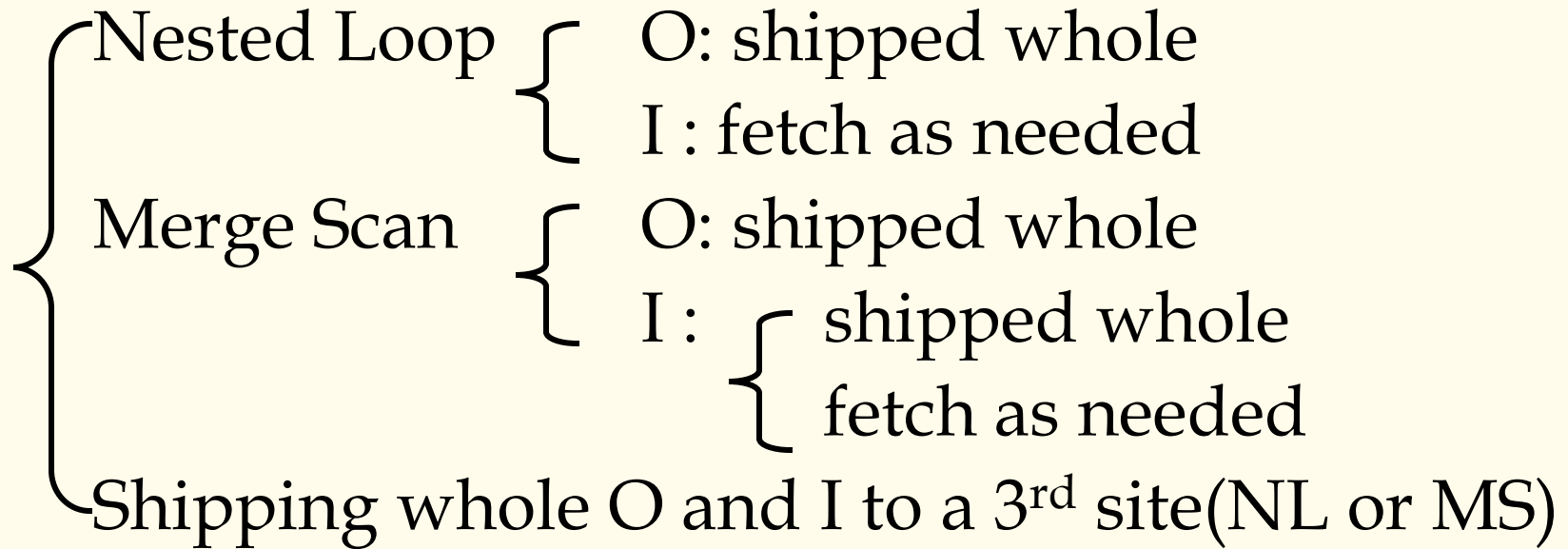
$$\text{代价} = (b_R + b_S) \times D_0 + \text{Cost}_{\text{sort}}(R) + \text{Cost}_{\text{sort}}(S)$$



二. 两种方法中关系的传递

- 1) “shipped whole” : The whole relation is shipped without selections.
 - For I, a temporary relation is established at the destination site for further use.
 - For O, the relation need not storing.
- 2) “Fetch as need” : The whole relation is not shipped. The tuples need by the remote site are sent at its request. The request message usually contains the value of join attribute. Usually there is a index on join attribute at the requested site.

三. 六种join实现策略



- It is obvious that O should be shipped whole.
- In NL, if I is shipped whole, index can't be shipped along with it, moreover temporary relation is required. Both processing cost and storage cost are high.



六种策略不包括:

- 多元join——化为多个二元join。
- Copy selection——因为R*不支持多复本。

5.8 Distributed Grouping & Aggregate Function Evaluation

```
SELECT PNUM, SUM(QUAN)
```

```
FROM SP
```

```
GROUP BY PNUM;
```

即： $GB_{PNUM, SUM(QUAN)}SP$

分布式环境下的分组及聚集函数计算，有以下结论：

- 1) 设 G_i 是按某属性集对关系 $R_1 \cup R_2$ 进行分组所得到的一个组，iff $G_i \subseteq R_j$ OR $G_i \cap R_j = \Phi$ for all i, j —— (SNC)，则：

$$GB_{G, AF}(R_1 \cup R_2) = (GB_{G, AF}R_1) \cup (GB_{G, AF}R_2)$$

例 SELECT SNUM, AVG(QUAN) FROM SP
GROUP BY SNUM;

- 如SP按供应商所在城市导出分割：满足SNC
- 如SP按零件类型导出分割：不满足SNC，因为同一供应商可能同时供应两种不同零件。

- 
- 2) If SNC does not hold, it is possible to compute some aggregate functions of global relation distributed

设 global relation: S

fragments: S_1, S_2, \dots, S_n

则 $SUM(S) = SUM(SUM(S_1), SUM(S_2), \dots, SUM(S_n))$

$COUNT(S) = SUM(COUNT(S_1), \dots, COUNT(S_n))$

$AVG(S) = SUM(S) / COUNT(S)$

$MIN(S) = MIN(MIN(S_1), MIN(S_2), \dots, MIN(S_n))$

$MAX(S) = MAX(MAX(S_1), MAX(S_2), \dots, MAX(S_n))$



5.9 更新策略

由于在DDB中任一数据都可能有多复本(便于查询),所以在更新时必须考虑多复本间的一致性。

1) Updating all strategy


The update will fail if any one of copies is unavailable.

p — — probability of availability of a copy.

n — — No. of copies

The probability of success of the update= p^n

$$\lim_{n \rightarrow \infty} p^n = 0$$

- 
- 2) Updating all available sites immediately and keeping the update data at spooling site for unavailable sites, which are applied to that site as soon as it is up again.
 - 3) Primary copy updating strategy
Assign a copy as primary copy. The remaining copies called secondary copies.
Update : update P.C, then P.C broadcast the update to S.Cs at sometimes.
P.C与S.C会有短暂不一致，如紧接着仍是更新操作没关系，如是读操作，读P.C也没问题，而读的是S.C时：




Compare the version No. of S.C with that of P.C, if version No. are equal, read S.C; else:

- (1) redirect the read to P.C
- (2) wait the update of S.C

4) Snapshot

snapshot — — a copy image not followed the changes in DB.

- Master copy at one site, many snapshots are distributed at other sites.
- Update: master copy only.

- 
- Read: $\left\{ \begin{array}{l} \text{master copy} \\ \text{snapshots} \end{array} \right\}$ is indicated by users
 - The snapshot can be refreshed:
 - (1) periodically
 - (2) forced refreshing by REFRESH command
 - 此法对更新较少的应用系统，如人口统计等应用比较适用。



6 Recovery



6.1 概述

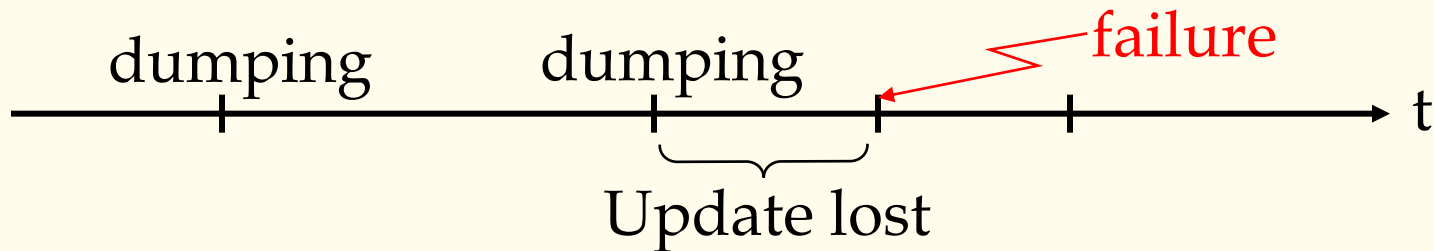
DBMS中恢复机制的主要作用是：

- (1) Reducing the likelihood of failures (防)
- (2) Recover from failures (治)

Restore DB to a consistent state after some failures.

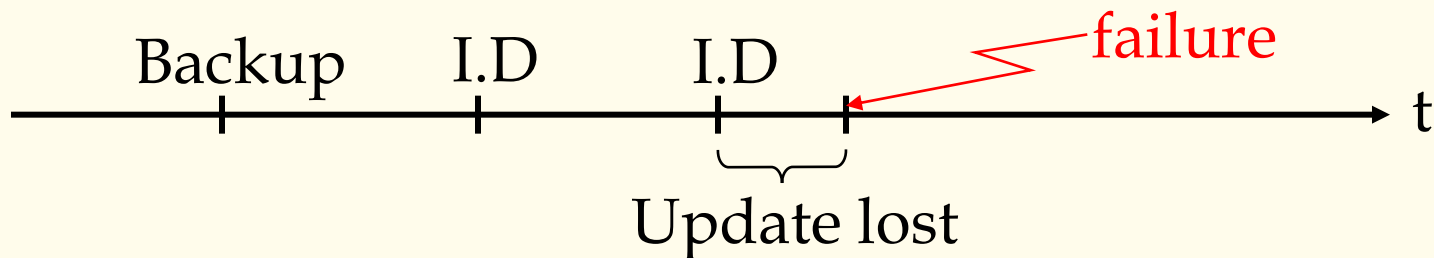
- 冗余是必须的。
- 要检测**所有可能**的故障。
- 一般的方法：

1) Periodical dumping



- 变种: Backup+Incremental dumping

I.D — — updated parts of DB



此法易于实现，开销小，但会丢失信息，常用于文件系统或小型DBMS。

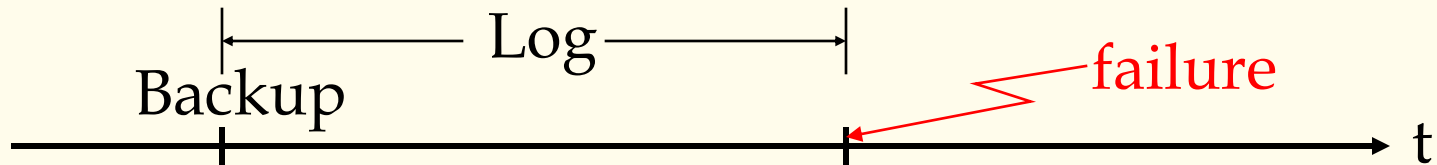


2) Backup+Log

Log : record of **all** changes on DB since the last backup copy was made.

Change: $\left\{ \begin{array}{l} \text{Old value(before image---B.I)} \\ \text{New value(after image---A.I)} \end{array} \right\}$ 记入 Log

对	update op. : B.I	A.I
	insert op. : ----	A.I
	delete op. : B.I	----





恢复时:

- 有些操作可能只做了一半，要利用Log中记录的B.I将其还原，即undo
- 对已完成但还未及写入DB的操作，用A.I恢复，即redo

It is possible to recover DB to the most recent consistent state.



3) Multiple Copies

每个数据都有多复本，发生故障时用其它复本恢复，且不会因某个复本不可用而导致系统瘫痪。

好处： (1)increase reliability

(2)recovery is very easy

问题： (1)difficult to acquire independent failure modes in CDBS.

(2)waste in storage space

所以此法不适用于集中式数据库系统。



6.2 Transaction(事务)

A transaction T is a finite sequence of actions on DB exhibiting the following effects:

- ✓ **A**tomic action: Nothing or All.
- ✓ **C**onsistency preservation: consistency state of DB→another consistency state of DB.
- ✓ **I**solation: concurrent transactions should run as if they are independent each other.
- ✓ **D**urability: The effects of a successfully completed transaction are permanently reflected in DB and recoverable even failure occurs later.



例：银行转帐，将款项s从A→B

Begin transaction

read A

A:=A-s

if A<0 then Display “insufficient fund”

Rollback /*undo and terminate */

else B:=B+s

Display “transfer complete”

Commit /*commit the update and terminate */

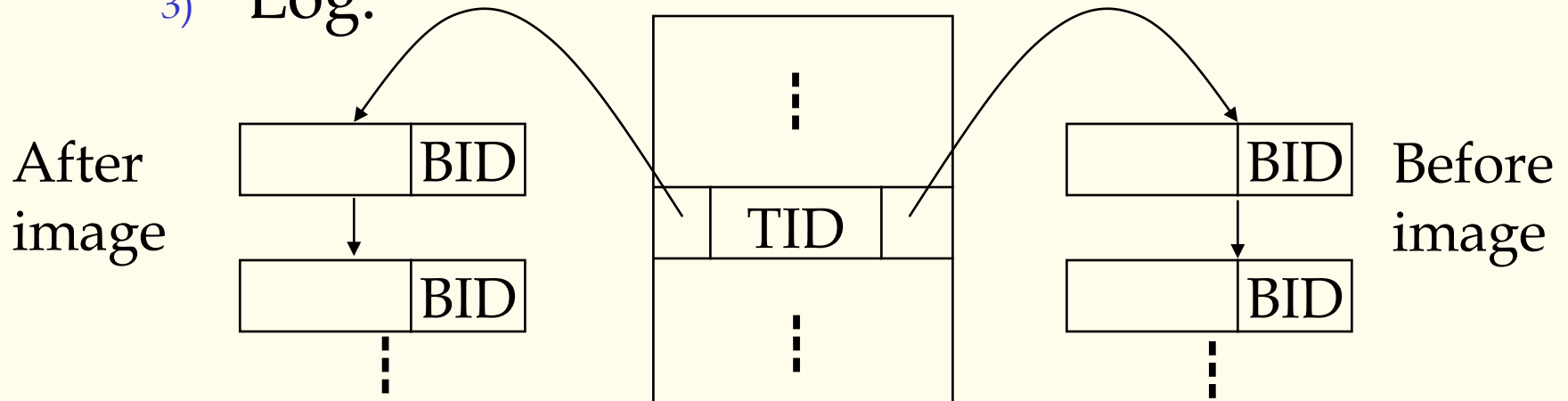
Rollback --- abnormal termination. (Nothing)


Commit --- normal termination. (All)

6.3 Some Structures to support recovery

恢复信息(如Log等), 应放在nonvolatile storage中。为便于恢复, 通常需保存如下信息:

- 1) Commit list: list of TID which have been committed.
- 2) Active list: list of TID which is in progress.
- 3) Log:





Log的保护要求高于一般数据，经常双套。
如两次dump间隔太长，Log可能越聚越多，从而引起空间问题，解决：

- 1) Free the space after commit. It is impossible to recover from disk failure.
- 2) Periodically dump to tape.
- 3) Log compression:
 - ✓ 不必为aborted T 保存Log
 - ✓ B.I are no longer needed for committed T
 - ✓ Changes can be consolidated, keep the newest A.I only.



6.4 commit Rule and Log Ahead Rule

6.4.1 Commit Rule

A.I must be written to nonvolatile storage before commit of the transaction.

6.4.2 Log Ahead Rule

If A.I is written to DB before commit then B.I must first written to log.

6.4.3 恢复策略

(1) undo和redo的特点，满足幂等性：

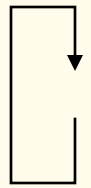
$$\text{undo}(\text{undo}(\text{undo} \dots \text{undo}(x) \dots)) = \text{undo}(x)$$
$$\text{redo}(\text{redo}(\text{redo} \dots \text{redo}(x) \dots)) = \text{redo}(x)$$



(2) 三种更新策略

a) A.I → DB before commit

TID → active list

 { B.I → Log (按Log Ahead Rule)
A.I → DB

⋮

commit { TID → commit list
delete TID from active list



在这种情况下，如果出现故障，如何恢复？

Restart时，对每个TID，查两个list:

Commit list	Active list	
	✓	undo, delete TID from active list
✓	✓	delete TID from active list
✓		nothing to do



b) A.I → DB after commit

TID → active list

□ { A.I → Log (按commit rule)
⋮

commit { TID → commit list
□ A.I → DB
delete TID from active list



在这种情况下，如果出现故障，如何恢复？

Restart时，对每个TID，查两个list:

Commit list	Active list	
	✓	delete TID from active list
✓	✓	redo, delete TID from active list
✓		nothing to do



c) A.I → DB concurrently with commit

TID → active list

$\left[\begin{array}{l} \downarrow \\ \downarrow \end{array} \right] \left\{ \begin{array}{l} \text{A.I, B.I} \rightarrow \text{Log} \quad (\text{按2 rules}) \\ \text{A.I} \rightarrow \text{DB} \quad (\text{partially done}) \end{array} \right.$

⋮

commit $\left\{ \begin{array}{l} \text{TID} \rightarrow \text{commit list} \\ \left[\begin{array}{l} \downarrow \\ \downarrow \end{array} \right] \text{A.I} \rightarrow \text{DB} \quad (\text{completed}) \\ \text{delete TID from active list} \end{array} \right.$



在这种情况下，如果出现故障，如何恢复？

Restart时，对每个TID，查两个list:

Commit list	Active list	
	✓	undo, delete TID from active list
✓	✓	redo, delete TID from active list
✓		nothing to do



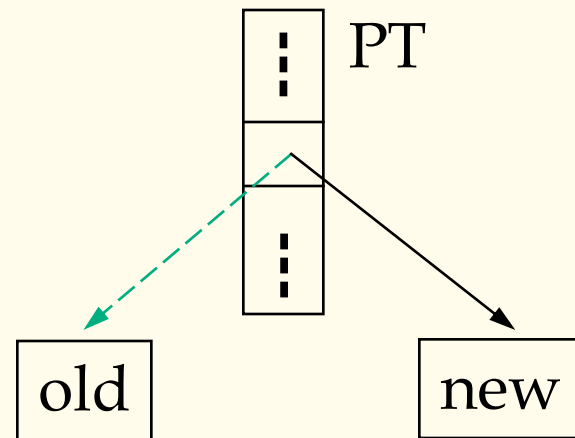
总结:

	redo	undo
a)	✘	✓
b)	✓	✘
c)	✓	✓
? d)	✘	✘

6.5 Update out of Place

- 为关系中的每一个page设置两个副本
- 为每个关系保持一张page table(PT)
- 更新某page时，异地生成new page，事务提交时拨动page table中相应项的指针，使其指向new page即可

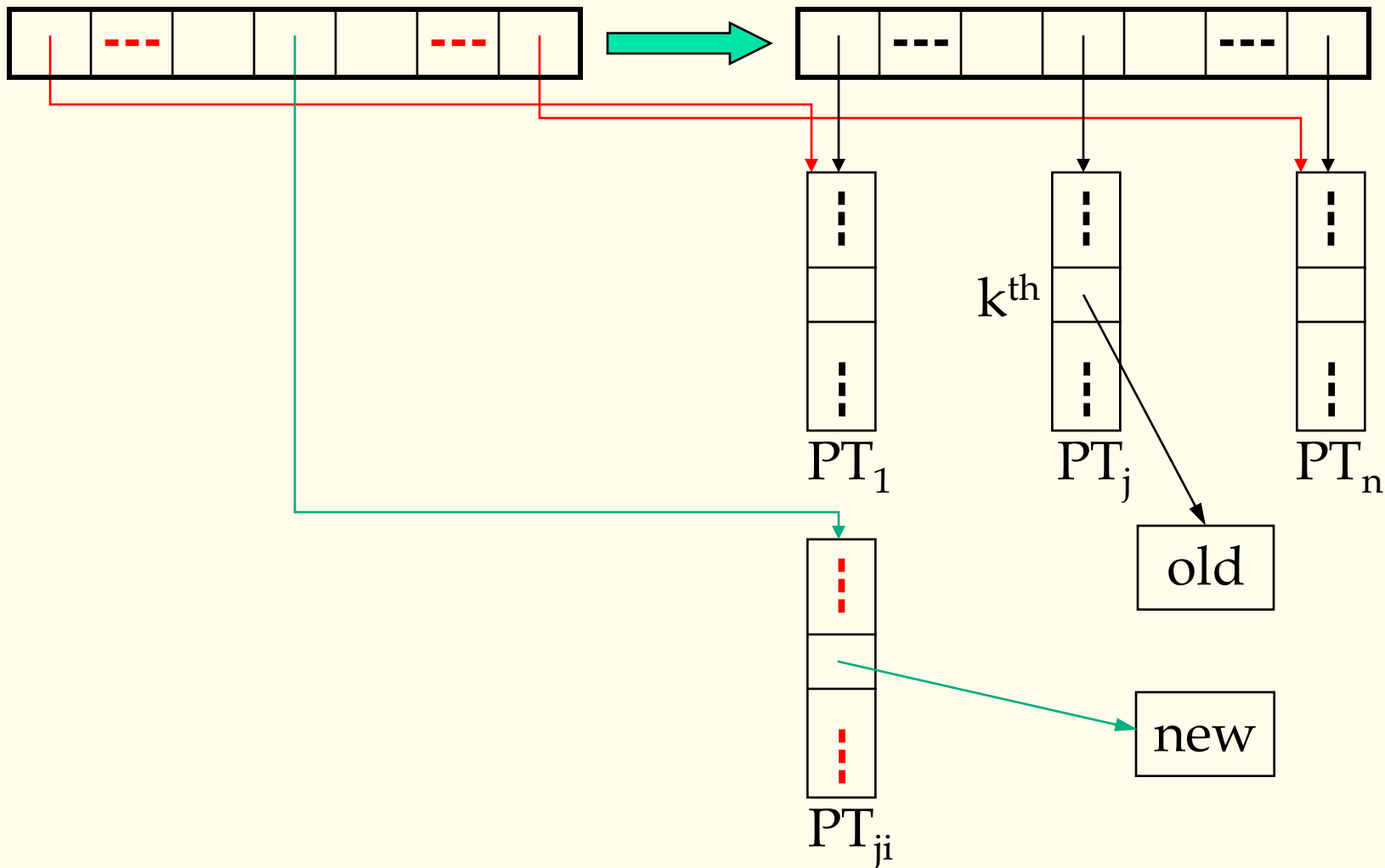
如关系R占N个page，
则它的PT长度为N



王能斌书p141: lorie's approach

Master Record(内存)

Master Record(硬盘)






6.6 Recovery Procedures

故障类型:

- 1) Transaction failure: 由于非预料原因, 该 transaction 要作废。
- 2) System failure: 系统崩溃, 但盘上DB未被破坏。如掉电。
- 3) Media failure: 硬盘故障, 盘上DB破坏了。

措施:

- 1) Transaction failure: 因为肯定在 commit 之前
 - Undo if necessary
 - Delete TID from active list

- 
- 2) System failure:
 - Restore the system
 - Undo or redo if necessary
 - 3) Media failure:
 - Load the latest dump
 - Redo according the log

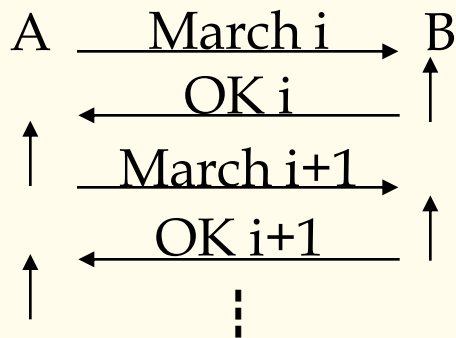


6.7 System Start Up

- 1) Emergency restart
Start after system or media failure.
Recovery is needed before start.
- 2) Warm start
Start after system shutdown. Recovery is not required.
- 3) Cold start
Start the system from scratch. Start after a catastrophic failure or start a new DB.

6.8 Two Phase Commit

- DDBMS中的事务是分布式事务，其执行的关键在于使各子事务要么同时commit，要么同时abort。
- 达到它们之间的协调要靠通信，而通信是不可靠的。
- 两将军悖论：No fixed length protocol exists.
- 解决：对信件编号。

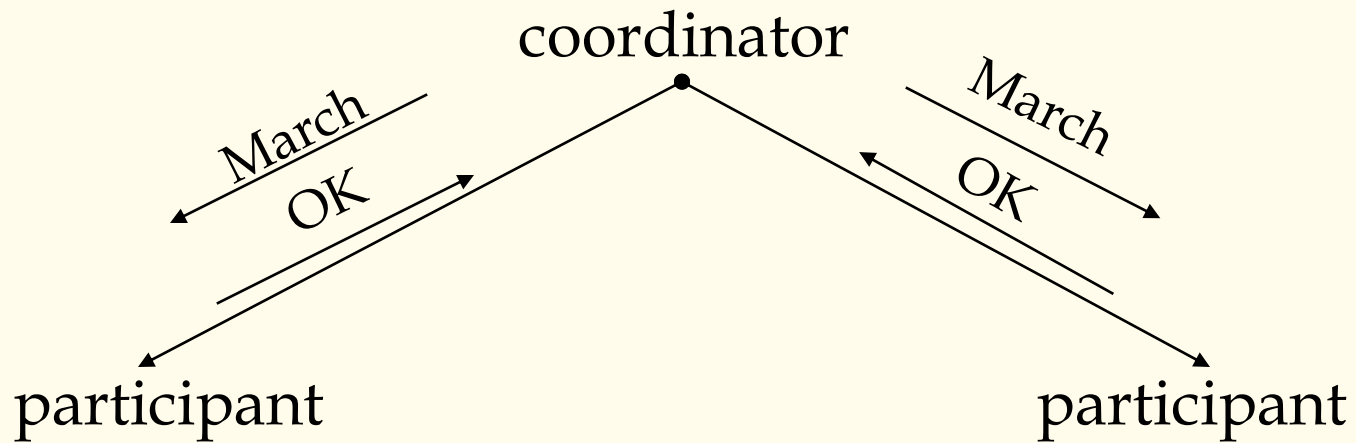


如果A未收到OK i+1

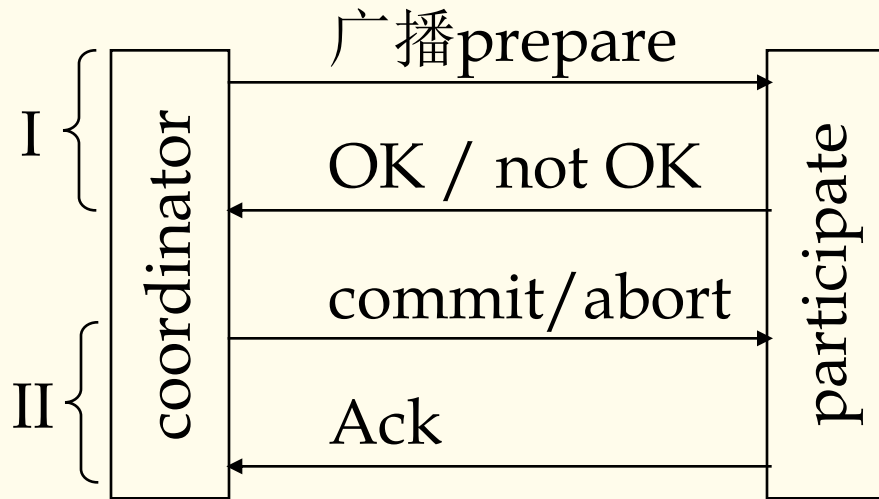
- | | | |
|------------------|---|--------------------|
| 1) B未收到March i+1 | } | A重发March i+1; 而对B: |
| 2) OK i+1丢失 | | |
- 1) 如已收到过，重发OK i+1
2) 否则，↑，发送OK i+1



- 多将军时，选其一作为coordinator



Two Phase Commit



如该子事务成功，可以commit，便回答OK，否则not OK

如所有子事务都OK，则发commit命令，否则发abort命令

OK — — if success

not OK — — if fail

commit — — all OK

abort — — any one not OK

- 各participant在回答OK之前，有自主权，可以随意abort，一旦回答OK之后，便只能等待协调者的命令了。
- 如发出OK后，协调者坏了，这时参与者只有等待，处于阻塞状态，这是2PC的弱点。

Three Phase Commit



如该子事务成功，可以commit，便回答ready，否则回答abort

Broadcast to all participants and this MSG is recorded on the disk

- 如协调者不出问题，II就白做了。
- 如发出OK后，协调者坏了，参与者之间相互联络，检查II记录在盘上的MSG，选举新协调者，如发现有参与者prepare to commit，则发commit命令，否则abort。从而解决阻塞问题。



7 并发控制



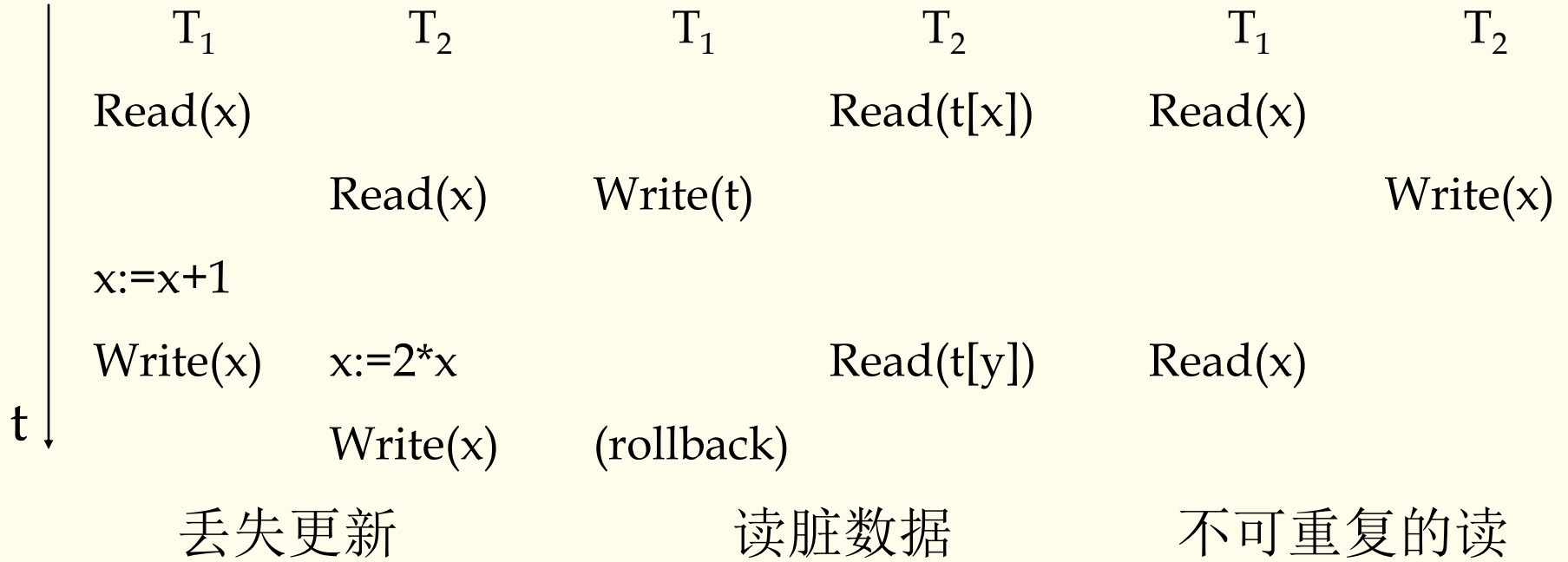
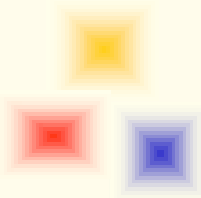
7.1 概述

多用户DBMS中，允许多个事务并发访问数据库。

7.1.1 Why concurrency?

- 1) Improving system utilization & response time
- 2) Different transaction may access to different parts of DB

7.1.2 Concurrency Problems



所以，事务并发运行时可能会有三种冲突：写—写、写—读、读—写；写—写冲突在任何情况下都应避免，写—读和读—写冲突一般情况下应避免，但某些应用场合可以容忍。

7.1.3 可串行化——并发控制的正确性准则

定义：设 $\{T_1, T_2, \dots, T_n\}$ 是一组并发运行的事务。如果对 $\{T_1, T_2, \dots, T_n\}$ 的一组调度(schedule)在数据库中产生的效果，与这组事务的某种串行执行序列的结果相同，则称这个调度是可串行化的。

问题：不同的调度 \rightarrow 不同的等价串行序列 \rightarrow 不同的执行结果？(n!)

T_A	T_B	T_C	执行结果与串行序列
	Read R1		$T_A \rightarrow T_B \rightarrow T_C$ 相同。
Read R2		Write R1	所以，可串行化。等价串
	Write R2		行序列 $T_A \rightarrow T_B \rightarrow T_C$

7.1.4 关于目标等价与冲突等价

- 调度：是系统对 n 个并发事务的所有操作的顺序的一个安排。
- 目标等价：两个调度 s_1 和 s_2 ，如果在同样的初始条件下执行，对数据库产生的效果完全相同，则称 s_1 和 s_2 是目标等价的。
- 冲突操作：R-W、W-W。冲突操作的执行顺序会影响执行效果。
- 不冲突操作：①R-R ②虽有写操作，但作用对象不同，如 $R_i(x)$ 和 $W_j(y)$ 。
- 冲突等价：凡是通过调换 s 中的不冲突操作所得的新调度，称为 s 的冲突等价调度。



- 性质：如两调度是冲突等价的，则一定是目标等价的；反之未必正确。
- 串行化也分为目标可串行化和冲突可串行化。
- 例1：对事务集 $\{T_1, T_2, T_3\}$ 的一个调度 s
 $s = R_2(x)W_3(x)R_1(y)W_2(y) \rightarrow R_1(y)R_2(x)W_2(y)W_3(x) = s'$
因为 s' 是串行调度，所以 s 是冲突可串行化的。
- 例2： $s = R_1(x)W_2(x)W_1(x)W_3(x)$ 无冲突等价调度，但却可以找到一个调度 s'
 $s' = R_1(x)W_1(x)W_2(x)W_3(x)$
与 s 目标等价。



目标可串行化的测试算法是NP难度的，冲突可串行化覆盖了绝大部分可串行化的调度实例，所以今后如无特别说明，可串行化均指冲突可串行化。

7.1.5 前趋图

有向图 $G=\langle V,E\rangle$

V ——顶点的集合，包含所有参与调度的事务。

E ——边的集合，通过分析冲突操作来决定。如果下列条件之一成立，可在 E 中加边 $T_i\rightarrow T_j$ ：

- $R_i(x)$ 在 $W_j(x)$ 之前
- $W_i(x)$ 在 $R_j(x)$ 之前
- $W_i(x)$ 在 $W_j(x)$ 之前

最后，看构造好的前趋图中是否有环路，如果有，则该调度不可串行化；否则，可串行化。



可串行化时，决定等价串行调度序列的算法：

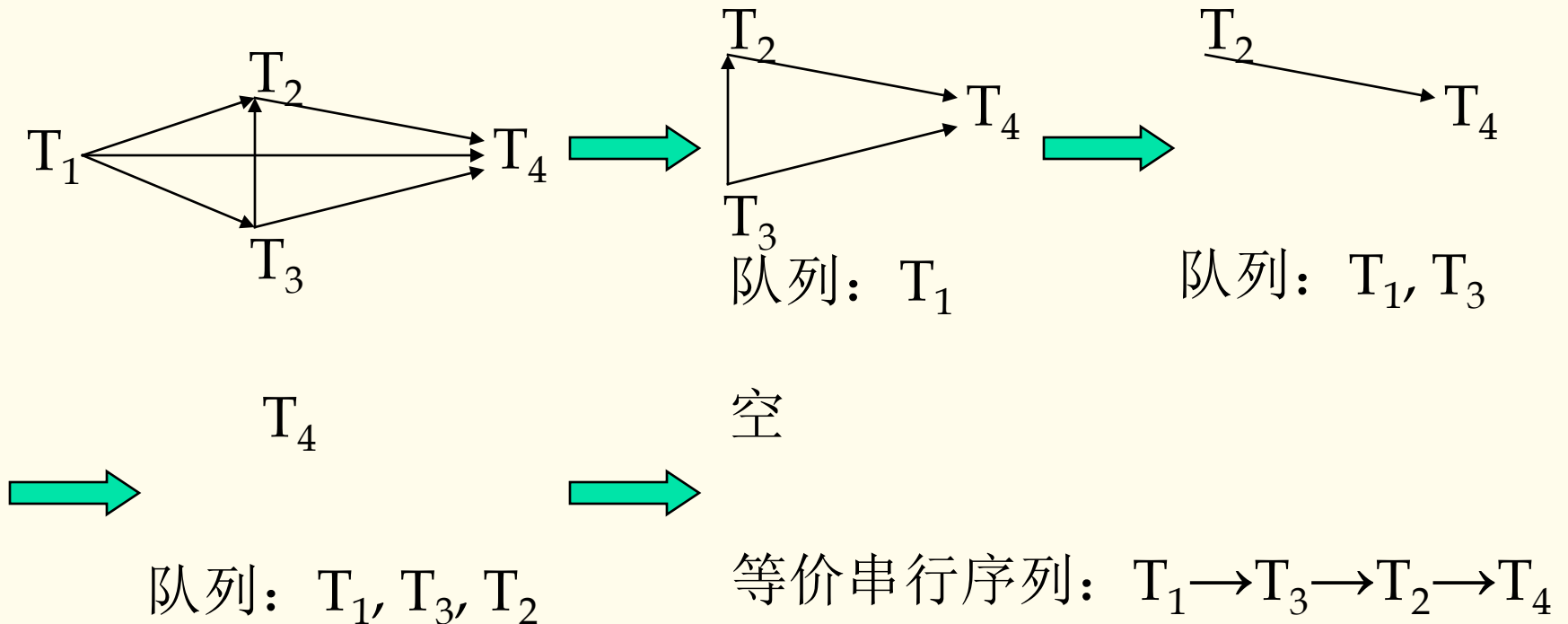
- 1) 由于无环路，必有入度为0的顶点。将它们及其有关的边从图中移去并将这些顶点存入一个队列。
- 2) 对剩下的图作同样处理，不过移出的顶点要队列中已有顶点之后。
- 3) 重复1，2直至所有顶点移入队列为止。

例对 $\{T_1, T_2, T_3, T_4\}$ 的一个调度 s

$S = W_3(y)R_1(x)R_2(y)W_3(x)W_2(x)W_3(z)R_4(z)W_4(x)$

它是否可串行化？如可串行化找出其等价的串行执行序列。

$S = W_3(y)R_1(x)R_2(y)W_3(x)W_2(x)W_3(z)R_4(z)W_4(x)$



并发控制的任务就是对并发执行的事务加以控制，使之按某种可串行化的调度序列来执行。

7.2 Lock Protocol

封锁法是最基本的并发控制方法之一，它可以有多种实现方式。

7.2.1 X locks

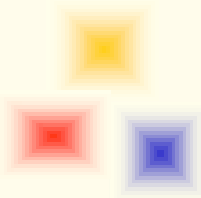
Only one type of lock, for both read and write.

相容矩阵： NL—no lock X—X lock
 Y —相容 N—不相容

待加 \ 已有	NL	X
NL	Y	Y
X	Y	N

T_A
 X_lock R
 Update R
 ⋮
 X_unlock
 R
 EOT

T_B
 X_lock R
 wait
 ↓
 X_lock R
 Read R
 ⋮



*Two Phase Locking

- Def1: In a transaction, if all locks precede all unlocks, then the transaction is called two phase transaction. This restriction is called two phase locking protocol.
- Def2: In a transaction, if it first acquires a lock on the object before operating on it, it is called well-formed.

- Theorem: If S is any schedule of well-formed and 2PL transactions, then S is serializable.
(王书p151证明)

	T ₁	T ₂
Growing phase	Lock A Lock B Lock C ⋮	Lock A Lock B Unlock A Unlock B
Shrinking phase	Unlock A Unlock B Unlock C	Lock C ⋮ Unlock C
	2PL	not 2PL

结论:

- 1) Well-formed+2PL: serializable
- 2) Well-formed+2PL+unlock update at EOT: serializable and recoverable.(不会有多米诺现象)
- 3) Well-formed+2PL+holding all locks to EOT: strict two phase locking transaction.

7.2.2 (S,X) locks

S lock — — if read access is intended.

X lock — — if update access is intended.

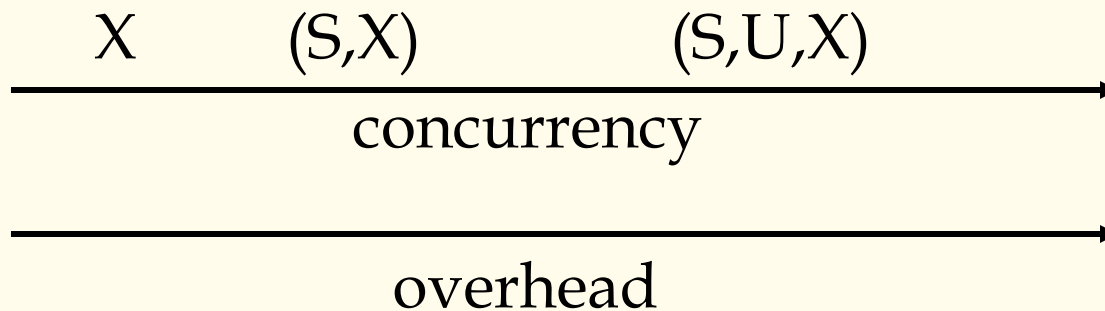
待加 \ 已有	NL	S	X
NL	Y	Y	Y
S	Y	Y	N
X	Y	N	N

7.2.3 (S,U,X) locks

U lock — — update lock.
 For an update access the transaction first acquires a U-lock and then promote it to X-lock.

目的：减少排他时间，提高并发度，减少死锁。

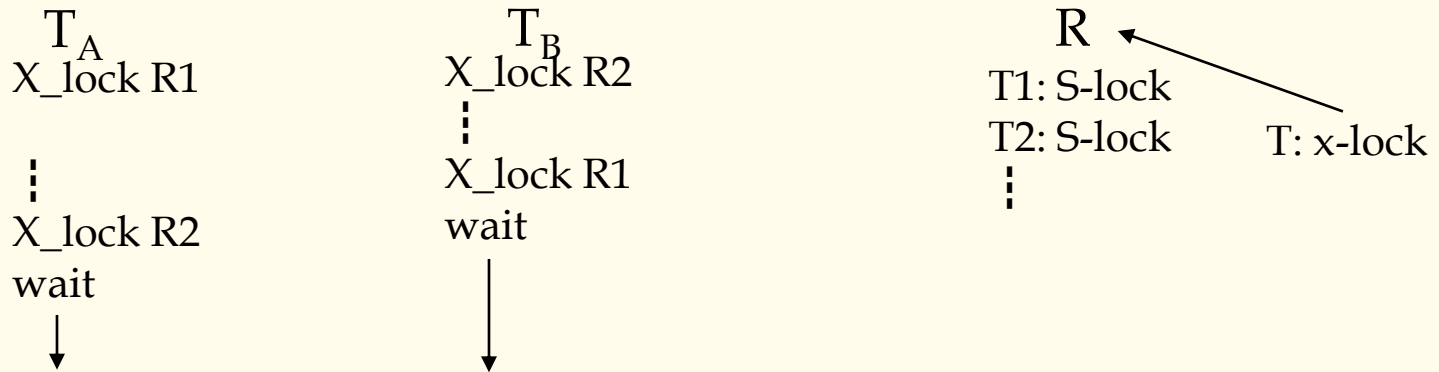
待加 \ 已有	NL	S	U	X
NL	Y	Y	Y	Y
S	Y	Y	Y	N
U	Y	Y	N	N
X	Y	N	N	N



7.3 Dead Lock & Live Lock

死锁：循环等待，谁也无法得到全部资源。

活锁：虽然其它事务都在有限长的时间内释放了资源，但某事务就是无法得到想要的资源。




- 活锁较简单，只需稍加修改调度策略，如FIFO
- 死锁：(1)防(不允许发生)；(2)治(允许，能消除)




7.3.1 Deadlock Detection

- 1) Timeout: If a transaction waits for some specified time then deadlock is assumed and the transaction should be aborted.
- 2) Detect deadlock by wait-for graph $G = \langle V, E \rangle$
 - V — set of transactions $\{T_i \mid T_i \text{ is a transaction in DBS } (i=1,2,\dots,n)\}$
 - $E = \{ \langle T_i, T_j \rangle \mid T_i \text{ waits for } T_j (i \neq j) \}$
 - 若图中有环路则说明已经发生死锁。
 - When to detect?
 - 1) whenever one transaction waits.
 - 2) periodically

- 
- What to do when detected?
 - 1) Pick a victim (youngest, minimum abort cost...)
 - 2) Abort the victim and release its locks and resources
 - 3) Grant a waiter
 - 4) restart the victim (automatically or manually)

7.3.2 Deadlock avoidance

- 1) Requesting all locks at initial time of transaction.
- 2) Requesting locks in a specified order of resource.
- 3) Abort once conflicted.
- 4) Transaction Retry



Every transaction is uniquely timestamped. If T_A requires a lock on a data object that is already locked by T_B , one of the following methods is used:

- a) Wait-die: T_A waits if it is older than T_B , otherwise it “dies”, i.e. it is aborted and automatically retried with original timestamp.
- b) Wound-wait: T_A waits if it is younger than T_B , otherwise it “wound” T_B , i.e. T_B is aborted and automatically retried with original timestamp.

上述方法中，都只有一个方向的等待，年老→年轻或年轻→年老，所以不会出现循环等待，从而避免了死锁的发生。



7.4 Lock Granularities

7.4.1 多级封锁

从减少lock开销来讲，封锁单位越大越好；从提高事务运行并发度来讲，粒度越小越好。

所以大数据库中封锁单位分几级：

DB—File—Record—Field

In this situation, if a transaction acquires a lock on a node then it acquires implicitly the same lock on each descendant of that node.

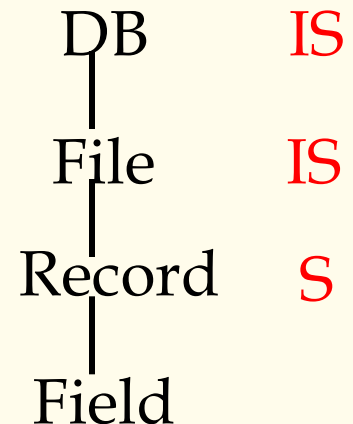
所以多级封锁有两种锁法：

- Explicit lock
- Implicit lock

7.4.2 意向锁

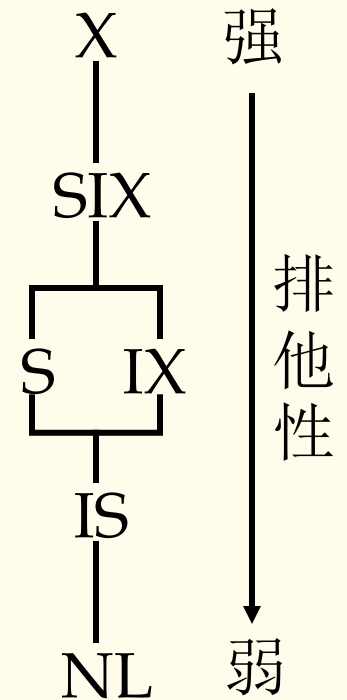
- 如何检查implicit locks?
- IBM的intention lock: 提供IS、IX和SIX三种意向锁。例如低级某对象加了S锁, 则在它所属的高级各对象上加IS锁, 作为警告信息。这样在高级某对象上要加X锁时, 就可以发现隐含的冲突。

- IS—Intention share lock
- IX—Intention exclusive lock
- SIX—S+IX



多级封锁时的相容矩阵:

待加 \ 已有	NL	IS	IX	S	SIX	X
NL	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
S	Y	Y	N	Y	N	N
SIX	Y	Y	N	N	N	N
X	Y	N	N	N	N	N

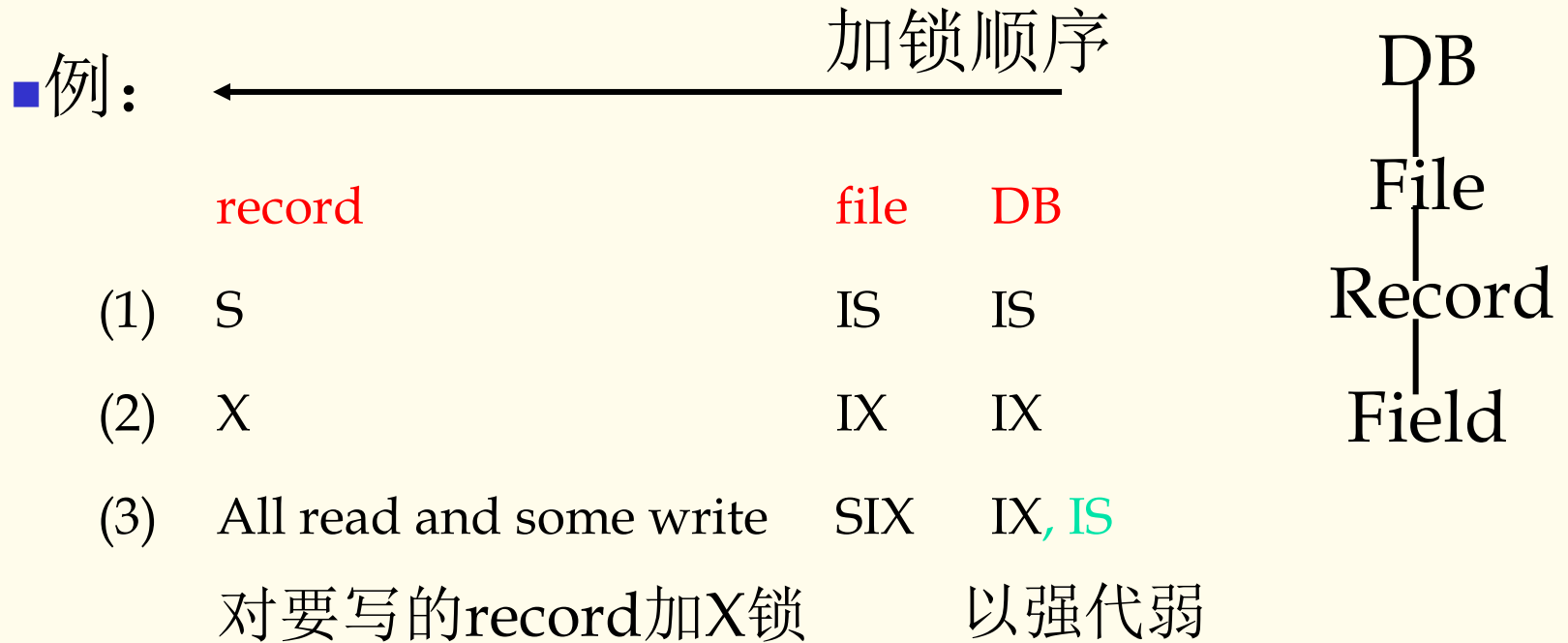


加锁时可以有强(排他性)代弱,但不能以弱代强。



加锁原则:

- Locks are requested from root to leaves and released from leaves to root.





7.5 Locking on Index (B+ Tree)

- Transactions access concurrently not only the data in DB, but also the indexes on these data, and apply operations on indexes, such as search, insert, delete, etc. So indexes also need concurrency control while multi granularity locking is supported.
- How can we efficiently lock a particular leaf node?
 - Btw, don't confuse this with multiple granularity locking!
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
 - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.



Some Useful Observations

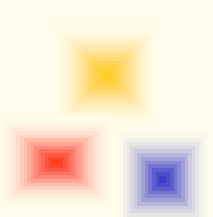
- Every access to B+ tree need a traverse from root to some leaf. Only leaves have detail information about data, such as TID, while higher levels of the tree only direct searches for leaf pages.
- One node occupies one page generally, so the lock unit of B+ tree is page. Don't need multi granularity locks, only S, X lock on page level are enough.
- B+ tree is key resource accessed frequently, liable to be the bottleneck of system. Performance is very important in index concurrency control.
- If occur conflict while traverse the tree, discard all locks applied, and search from root again after some delay. Avoid deadlock resulted by wait.



Some Useful Observations

- Originally, locks on index are only used to keep the consistency of index itself. The correctness of concurrent transactions is responsible by 2PL. From this sense, the locks on index don't need keeping to EOT, they can release immediately after finish the mapping from attribute value to tuples' addresses. But in 7.6 we will know, even **the strict 2PL has leak while multi granularity locks are permitted**. The lock on leaf of B+ tree should be kept to EOT in order to make up the leak of strict 2PL in this situation, while locks on other nodes of the tree can be released after finishing of search.

ROOT



A

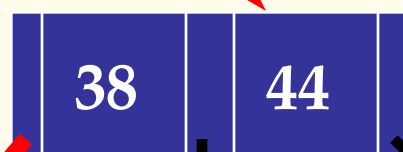
Example
Traversing
the tree



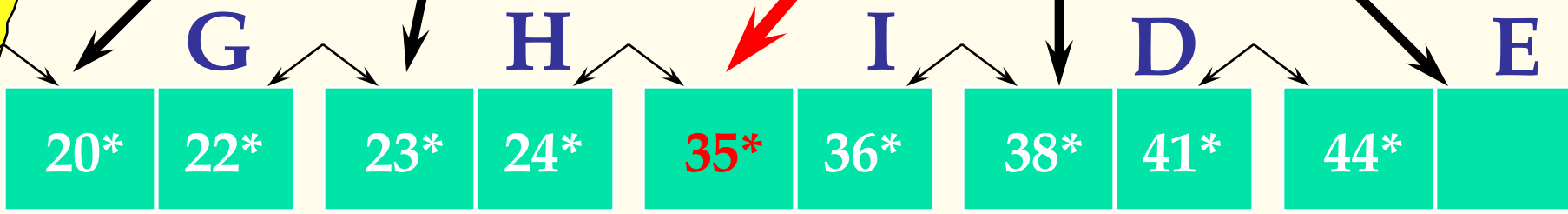
B



F



C



G

H

I

D

E



Tree Locking Algorithm

1. While traversing, apply S lock on root first, then apply S lock on the child node selected. Once get S lock on the child, the S lock on parent can be released, because traversing can't go back. Search like this until arrive leaf node. After traversing, only S lock on wanted leaf is left. Keep this S lock till EOT.
2. While inserting new index item, traverse first, find the leaf node where the new item should be inserted in. Apply X lock on this leaf node.
 - If it is not full, insert directly.
 - If it is full, split node according to the rule of B+ tree. While splitting, besides the original leaf, the new leaf and their parent should add X lock. If parent is also full, the splitting will continue.
 - In every splitting, must apply X lock on each node to be changed. These X locks can be released when the changes are finished.
 - After the all inserting process is completed, the X lock on leaf node is changed to S lock and kept to EOT.



Tree Locking Algorithm

3. While deleting an index item from the tree, the procedure is similar as inserting. Deleting may cause the combination of nodes in B+ tree. The node changed must be X locked first and X lock released after finishing change. The X lock on leaf node is also changed to S lock and kept to EOT.



7.6 Phantom and Its Prevention

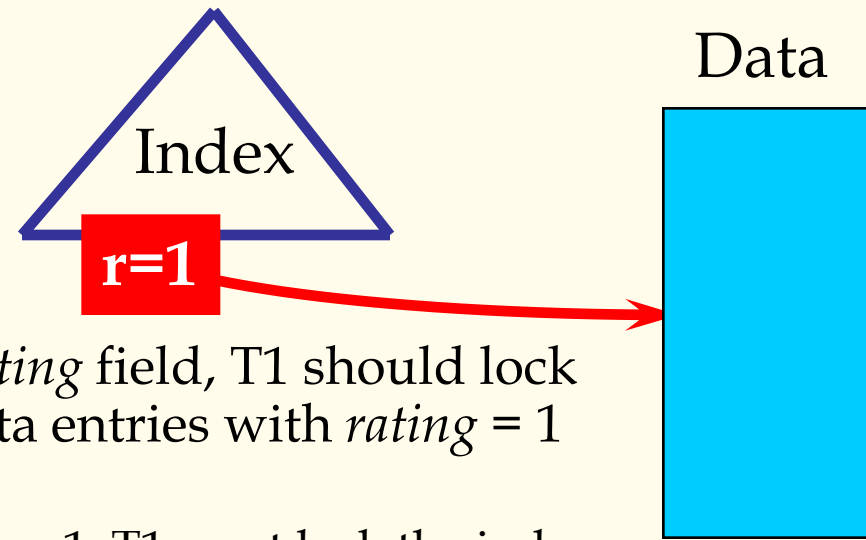
- The assumption that the DB is a fixed collection of objects is not true when multi granularity locking is permitted. Then even Strict 2PL will not assure serializability:
 - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
 - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
 - T2 also deletes oldest sailor with *rating* = 2 (and, say, *age* = 80), and commits.
 - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state where T1 is “correct”!



The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
 - Assumption only holds if no sailor records are added while T1 is executing!
 - Need some mechanism to enforce this assumption. (Index locking and predicate locking)
- Example shows that conflict serializability guarantees serializability only if the set of objects is **fixed!**
- If the system don't support multi granularity locking, or even if support multi granularity locking, the query need to scan the whole table and add S lock on the table, then there is not this problem. For example :
select s#, average(grade) from SC group by s#;

Index Locking



- If there is a dense index on the *rating* field, T1 should lock the index node containing the data entries with *rating* = 1 and **keep it until EOT**.
 - If there are no records with *rating* = 1, T1 must lock the index node where such a data entry *would* be, if it existed!
- When T2 wants to insert a new sailor (*rating* = 1, *age* = 96), he can't get the X lock on the index node containing the data entries with *rating* = 1, so he can't insert the new index item to realize the insert of a new sailor.
- If there is no suitable index, T1 must lock the whole table, no new records can be added before T1 commit of course.



Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
 - What is the predicate in the sailor example?
- In general, predicate locking has a lot of locking overhead. It is almost impossible to realize it.

7.7 Isolation Level of Transaction

- Support for isolation level of transaction is added from SQL-92. Each transaction has an access mode, a diagnostics size, and an isolation level.
- SET TRANSACTION statement

Isolation Level	Possible result			Lock demand
	Dirty Read	Unrepeatable Read	Phantom Problem	
Read Uncommitted	Maybe	Maybe	Maybe	No lock when read; X lock when write, keep until EOT
Read Committed	No	Maybe	Maybe	S lock when read, release after read; X lock when write, keep until EOT
Repeatable Reads	No	No	Maybe	According to Strict 2PL
Serializable	No	No	No	Strict 2PL and keep S lock on leaf of index until EOT



Example :

```
SET TRANSACTION READ ONLY  
ISOLATION LEVEL REPEATABLE READ;
```

```
SET TRANSACTION ISOLATION LEVEL  
{ READ COMMITTED |  
  READ UNCOMMITTED |  
  REPEATABLE READ |  
  SERIALIZABLE  
}
```

7.8 面向对象数据库管理系统中的封锁机制

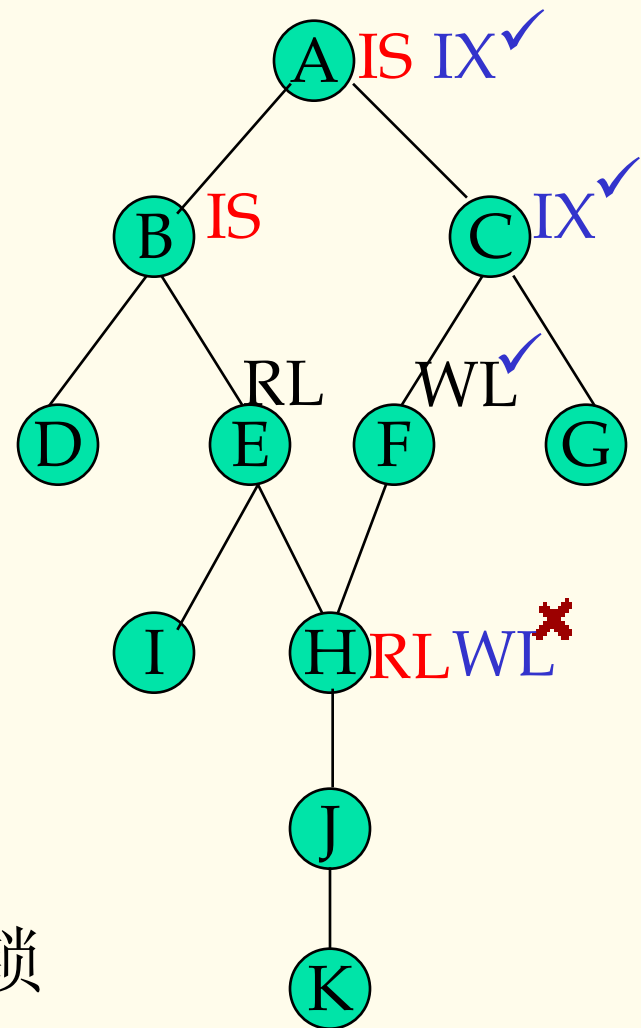
- 1) 封锁粒度：OODB中，对象一般作为最小并发控制粒度。DB一类一对象
- 2) 单级封锁：利用S, X锁直接封锁要操作的对象。只适于面向CAD这类应用的OODBMS, 对需要经常进行联想查询的场合不太合适。
- 3) 多粒度封锁：利用上节所述的S, X, IS, IX, SIX等类型的锁，是多粒度封锁的典型应用。
但这时的类级封锁只锁该类的直属实例，不包括该类的子类。对级联查询或模式修改不合适。
- 4) 复杂多粒度封锁：增加2个类层次锁RL和WL

- RL 一相当于在类及其所有子类都加了S锁
- WL一相当于在类及其所有子类都加了X锁

待加 \ 已有	NL	IS	IX	S	SIX	X	RL	WL
NL	Y	Y	Y	Y	Y	Y	Y	N
IS	Y	Y	Y	Y	Y	N	Y	N
IX	Y	Y	Y	N	N	N	N	N
S	Y	Y	N	Y	N	N	Y	N
SIX	Y	Y	N	N	N	N	N	N
X	Y	N	N	N	N	N	N	N
RL	Y	Y	N	Y	N	N	Y	N
WL	Y	N	N	N	N	N	N	N

RL(WL)加锁步骤:

- a) 在该类的任一超类链及DB上加IS(IX)
- b) 在该类加RL(WL)
- c) 自上而下检查该类的子类有无与RL(WL)冲突的锁，若无，可在最先遇到的多继承子类上加RL(WL)，子类检查结束
- d) 以上步骤如发现冲突，锁申请失败
- 5) 复杂对象加锁：点到才加锁
(按简单多粒度协议)





7.9 The Time Stamp Method

1. T.S --- A number generated by computer's internal clock in chronological order.
2. T.S for a transaction --- the current T.S when the transaction initials.
3. T.S for an data object:
 - 1) Read time (tr) – highest T.S possessed by any transaction to have read the object.
 - 2) Write time (tw) – highest T.S possessed by any transaction to have written the object.
4. The key idea of T.S method is that the system will enforce the concurrent transactions to execute in the schedule equivalent with the serial execution according to T.S order.



Read/Write operations under T.S method

5. Let R --- a data object with T.S tr and tw .
 T --- a transaction with T.S t .

Transaction T reads R :

read R

if ($t \geq tw$)

 then /* OK */

$tr = \text{Max}(tr, t)$

 else /* a transaction younger than T has already write R
 ahead of T , conflict */

 restart T with a new T.S



Transaction T writes R:

if ($t \geq tr$)

 then if ($t \geq tw$)

 then /* OK */

 write R

$tw = t$

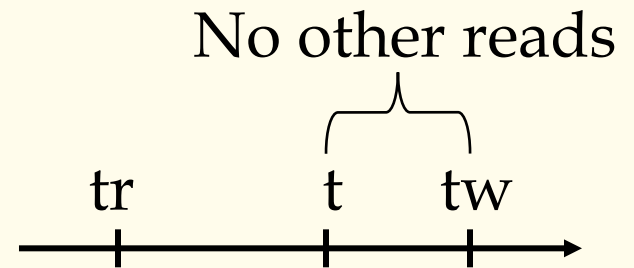
 else /* $tr \leq t < tw$ */

 do nothing

 else

 /* a transaction younger than T has already
 read R ahead of T, conflict */

 restart T with a new T.S





Remarks:

1. Compared with lock method, the most obvious advantage is that there is no dead lock, because of no wait.
2. Disadvantage: every transaction and every data object has T.S, and every operation need to update tr or tw, so the overhead of the system is high.
3. Solution:
 - Enlarge the granularity of data object added T.S. (Low concurrency degree)
 - T.S of data object are not actually stored in nonvolatile storage but in main memory and preserved for **a specified time** and the T.S of data objects whose T.S is not in main memory are assumed to be zero.



7.10 Optimistic Concurrency Control Method

The key idea of optimistic method is that it supposes there is rare conflict when concurrent transactions execute. It doesn't take any check while transactions are executing. The updates are not written into DB directly but stored in main memory, and check if the schedule of the transaction is serializable when a transaction finishes. If it is serializable, write the updating copies in main memory into DB; Otherwise, abort the transaction and try again.

The lock method and time stamp method introduced above are called “pessimistic method”.



Three phases of transaction execution:

1. *Read phase*: read data from database and execute every kind of processing, but update operations only form update copies in memory.
2. *Validate phase*: check if the schedule of the transaction is serializable.
3. *Write phase*: if pass the check successfully, write the update copies in memory into DB and commit the transaction; Or throw away the update copies in memory and abort the transaction.



Information must be reserved:

1. Read set of each transaction
2. Write set of each transaction
3. The start and end time of each phase of each transaction



Checking method while transaction ends:

When transaction T_i ends, only need to check if there is conflict among T_i and the transactions which have committed and other transactions which are also in checking phase. The transactions which are in read phase don't need to be considered.

Suppose T_j is any transaction which has committed or is being checked, T_i passes the check if it fulfills one of the following conditions for all T_j :

1. T_j had finished write phase when T_i began read phase, $T_j \rightarrow T_i$
2. The intersection of T_i 's read set and T_j 's write set is empty, and T_i began write phase after T_j finished write phase.
3. Both T_i 's read set and write set don't intersect with T_j 's write set.
4. There is not any access conflict between T_i and T_j .



7.11 Locking in DDBMS

分布式数据库中的并发控制与集中式数据库一样，要求并发事务的执行可串行化。问题：

- Multi_copy
- Communication overhead

7.11.1 write lock all, read lock one

- Read R —S_lock on any copy of R
- Write R —X_lock all copies of R
- Hold the locks to EOT

通信开销： 设 n —No. of copies



Write: n lock MSG
n lock grants
n update MSG
n ACK
[n unlock MSG]

4n

Read: 1 lock MSG
1 lock grants
1 read MSG

2

} Can be merged

7.11.2 Majority locking

- Read R — S_lock on a majority of copies of R
- Write R — X_lock on a majority of copies of R
- Hold the locks to EOT

通信开销: Majority — $(n+1)/2$



Write: $(n+1)/2$ lock MSG
 $(n+1)/2$ lock grants
 n update MSG
 n ACK

$3n+1$

Read: $(n+1)/2$ lock MSG
 $(n+1)/2$ lock grants
1 read MSG

$n+1$

- 对7.11.1，如两个事务都要写，竞争X锁，可能各自拿到一部分锁，但都不能X-lock all，所以很容易死锁，而此法只要 n 是奇数就不会发生上述这种形式的死锁，因为总有一个事务先过半数。



7.11.3 k-out-of-n locking

- Write R—X_lock on k copies of R, $k > n/2$
- Read R—S_lock on $n-k+1$ copies of R
- Hold the locks to EOT
- ✓ 对读写冲突： $k+(n-k+1)=n+1 > n$ ，所以至少可以在一个复本上检测出冲突。
- ✓ 对写写冲突： $2k > n$ ，所以也能保证检测出冲突。
所以前两种方法实际上是它的两个特例：
- ✓ 7.11.1就是 $k=n$ ；7.11.2就是 $k=(n+1)/2$
- ✓ K可在 $(n+1)/2 \sim n$ 间浮动，k越大对读越有利。

7.11.4 Primary Copy Method

R—data object

Assign the lock responsibility for locking R to a given site. This site is called primary site of R.

通信开销:

Write: 1 lock MSG	Read: 1 lock MSG
1 lock grants	1 lock grants
n update MSG	1 read MSG
n ACK	
<hr/>	<hr/>
2n+1	2

此法效率较高但易受损害。为此有很多变种。经常和主复本更新策略配合使用。

7.11.5 Global Deadlock



- T₁和T₂均为分布式事务，分别在结点A、B上各有两个子事务。
- T_{1A}、T_{1B}必须同时commit
- T_{2A}、T_{2B}必须同时commit

上图所示就是一个全局死锁。如何发现这种死锁？
全局等待图：在原来的基础上增加EXT结点，如果事务T是分布式事务，在其它结点有子事务，且T在当前结点等待图的链头，则增加EXT→T；如果在当前结点等待图的链尾，则增加T→EXT

全局等待图的处理方法:

例如某结点上: $EXT \rightarrow T_i \rightarrow T_j \rightarrow \dots \rightarrow T_k \rightarrow EXT$

1) 找到另一结点: $EXT \rightarrow T_k \rightarrow T_l \rightarrow \dots \rightarrow T_x \rightarrow EXT$

2) if $T_x = T_i$: global deadlock is detected.

if $T_x \neq T_i$: merge two wait-for graphs:

$EXT \rightarrow T_i \rightarrow T_j \rightarrow \dots \rightarrow T_k \rightarrow T_l \rightarrow \dots \rightarrow T_x \rightarrow EXT$

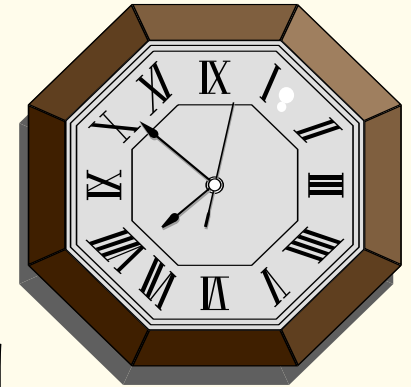
3) 重复第1步, 看 T_x 是否会像 T_k 那样因符合上面条件而产生全局死锁, 如各结点上的等待图都检查过而未产生全局回路——无死锁发生。

7.12 Time Stamp Technique in DDBMS

7.12.1 全局时间戳

- 为避免各结点到来的事务的时间戳相同，设置：
Global T.S = Local T.S + Site ID
- 各结点时钟可能不同，不要紧，关键是保证：
time of receipt \geq time of delivery
- 解决：
 $t_{\text{at receipt site}} := \max(t_1, t_2)$
 t_1 — current T.S at receipt site
 t_2 — T.S of MSG

7.12.2 读写操作



- 1) Write—update t_w of all copies.
- 2) Read—update t_r of the copy read.
- 3) When writing we check T.S of all copies. If $t < t_r$ for any copy the transaction should be aborted. When reading we check T.S of the copy read. If $t < t_w$ the transaction should be aborted.

