W2 物理存储媒介和磁盘

磁盘

Access Time

文件组织（定长、变长、聚集）

W3a 系数

顺序索引（密集/稀疏，主/次级索引，多层索引）

W3b B+ 树

W3c 哈希索引

W4a 关系代数符号

W4b 选择查询

选择成本

外部排序

W4c 连接查询

五种连接

其他运算

W5a 查询优化

物化 / 流水线

等价替换

W5b 选择规模估计

连接规模估计

不同值的个数估计

评估计划的选择

W6a 事务管理

冲突 / 视图可串行

可恢复 / 无级联调度

W6b 并发控制、基于锁的协议、死锁预防检测和多粒度

W6c 日志和检查点

W8a XML

元素，属性和名字空间

XPath, XQuery and XSLT

W8b Semantic Web, RDF and SPARQL

W9 数据库连接

集中式和客户机-服务器系统

事务系统进程

ODBC & JDBC

W10 OODB 面向对象的数据库系统

复杂的 SQL

W11 分布式数据库

类型（同质 / 异质）

存储数据（分片 / 复制）

分布式查询 & 连接 & 优化 & 事务 & 锁协议

故障恢复

W12 NoSQL 和大数据存储

最终一致性

四种类型的存储

四种大数据存储系统

分布式文件系统

W13 数据挖掘和市场篮子分析

重要缩写

ACID: Atomicity, Consistency, Isolation, Durability

API: Application Programming Interface 应用程序接口（预先定义）

BASE: basically available, soft state, eventually consistent 最终一致性

CDT: Change Data Table 异步复制，由日志生成

CNF: Conjunctive Normal Form

DOM: Document Object Model 文档对象模型

DTD: Document Type Definition 文档定义类型（可以定义 *XML*文档的类型）

EJB: Enterprise JavaBeans

ETL: Extract-Transform-Load 提取转换负载

E-R feature: Entity relationship

FLWOR: XQuery (for … let … where … order by …return…)

GFS: Google File System

HDFS: Hadoop File System

HF: Horizontal fragments

HTML: Hyper Text Markup Language 超文本标记语言

ID: identifier

IDREF: ID reference

*JDBC: Java Database Connectivity Java*数据库连接

ODBC: Open Database Connectivity 开放式数据库连接

OLAP: Online Analytical Processing 联机分析处理

OODB: Object-Oriented Databases 面向对象的数据库

O-R feature: Object-Relational

ORM: Object-Relational Mapping 对象到关系映射

OWL: Web Ontology Language 万维网本体语言

PCDATA: parsed character data 被解析的字符数据

RAM: Random Access Memory 随机存取存储器

RDF: Resource Description Framework 资源描述框架（在*Web*上表示信息的框架）

RDFS: Resource Description Framework Schema 提供描述*RDF*词汇表的基本功能

RPC: Remote procedure call 远程过程调用

SAX: Simple API for XML 简单应用程序接口

SGML: Standard Generalized Markup Language 通用标识语言标准

SPARQL: Simple Protocol and RDF Query Language

SOAP: Simple Object Access Protocol 简单对象访问协议

SQLJ: 嵌入式 *SQL*

UDDI: Universal Description, Discovery, and Integration 通用描述、发现和集成

URI: Uniform Resource Identifier 统一资源标识符（标识某一万维网资源）

VF: Vertical fragments

W3C: World Wide Web Consortium 万维网联盟

*WSDL: Web Services Description Language Web*服务描述语言

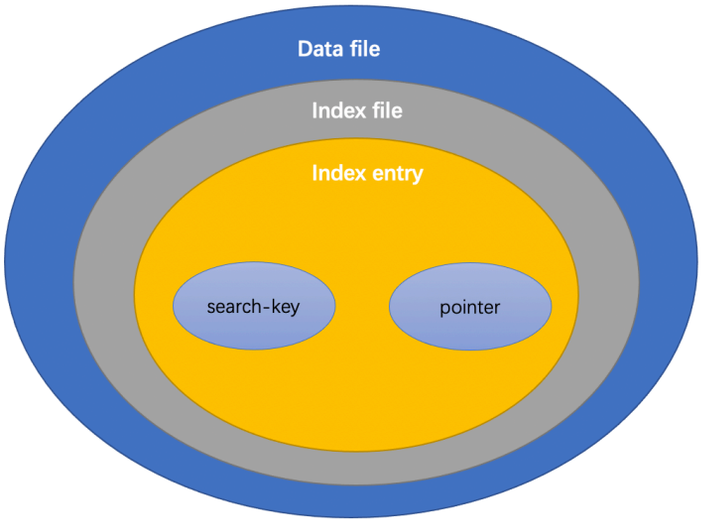
XML: Extensible Markup Language 可扩展标记语言

XSLT: XML Stylesheet Language Transformations 一种样式转换标记语言

Week3a 索引

1. 背景
   1. 不同块上分散的记录效率低
   2. 索引：加速访问，但更新索引会增加数据库修改的开销，因为当文件被修改时，文件上的每个索引都必须更新
   3. Sequential scan using primary index 是有效的

2. 术语解析
   1. Data file：磁盘上，存储记录的块的集合
   2. Index file：一种数据结构，允许数据库系统在**Data file**中更有效的查找特定记录
   3. Index entry/Record：组成 **Index file**



   4. Search key：一个属性或属性集，用于File中查找记录
   5. Pointer：指针，连接 **Index file** 中的Search key 与 Data file 中带Search key 的**记录**

对索引技术的评估必须基于：

1. Access types：访问类型，找到具有特定属性值的记录**或**找到属性值在特定范围的记录
2. Access time：在查询中找到一个特定数据项（集）所需的时间
3. Insertion time：插入新数据项所用时间（包括插入到正确位置+更新索引结构）

4. Deletion time：删除新数据项所用时间（包括找到待删除项+更新索引结构）

5. Space overhead：空间开销，索引结构所占用的额外存储时间

3. 两种基本索引类型

Ordered index：顺序索引：索引按存储搜索码的顺序被排序

Dense index 稠密索引：每个搜索码值都有一个索引项

Sparse index 稀疏索引：部分搜索码值才有索引项，必须聚集

与稠密索引相比，稀疏索引占用空间更小，必须顺序，可能无法直接找到记录

折中：为每个块，建立一个索引项的稀疏索引；对应于块中的最小搜索码值

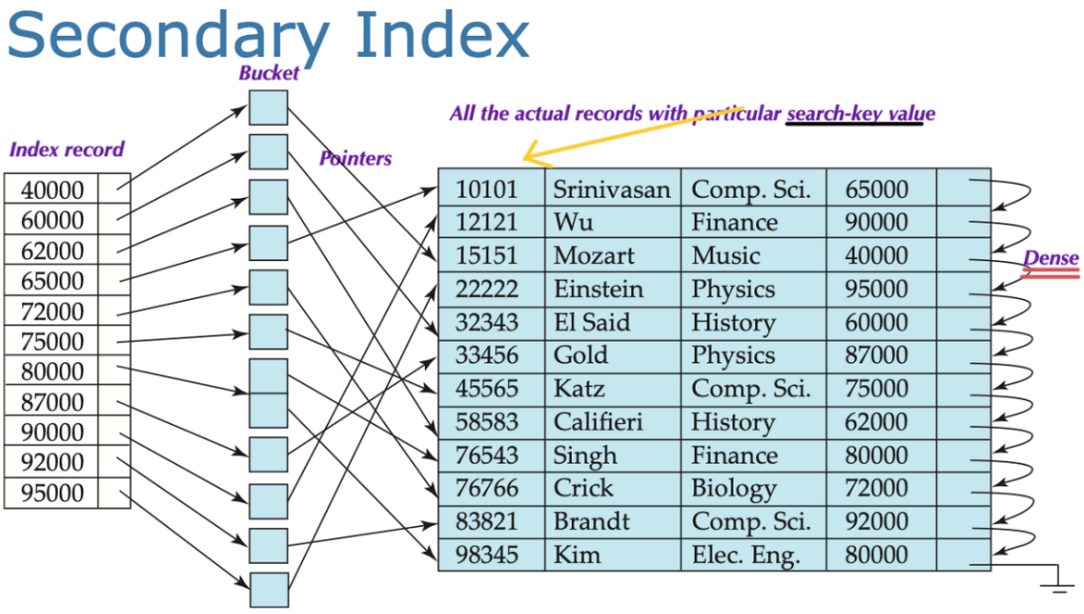
clustering/primary index 聚集/主索引：搜索码的逻辑顺序决定文件记录的物理顺序

可以是稀疏（主索引的搜索键通常是但不一定是主键）

non-clustering/secondary index 非聚集/辅助索引：二者顺序不同

必须稠密，不可以稀疏

系数记录指向一个**包含指针的 bucket**



Index-sequential file 索引顺序文件：带主索引的有序顺序文件

4. 多级索引

如果主索引过大不能放入内存中，就很贵；当需要时，就必须从磁盘中取出索引块

解决措施：将保存在磁盘上的主索引视为一个顺序文件，并在其上构造稀疏索引

外部索引 – 主索引的稀疏索引

内部索引 – 主索引文件

从内层索引上构造稀疏的外层索引

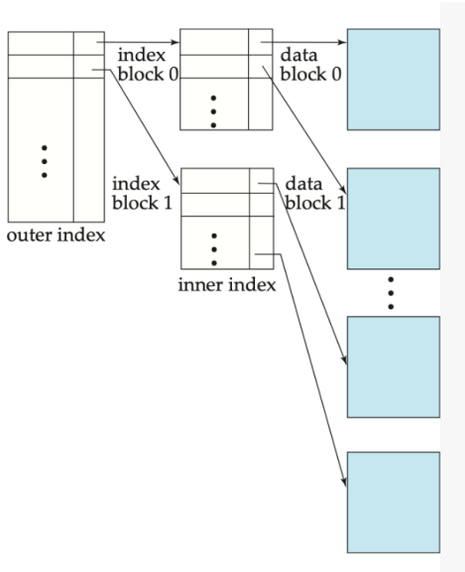
首先在外层索引使用二分法搜索

找到最大搜索码值小于或等于

所需所搜码值的记录

指针指向内层索引块

扫描这个块



Week3b B+ 树

1. 性质
    1. 每个块一个节点
    2. 最多包含 n-1 个搜索码值和 n 个指针

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

    3. 树根到树叶每条路径长度相同
    4. **根节点**：至少两个孩子
    5. **非叶节点**：至少一半的指针被使用
    6. **叶节点**：包含的值最少为 $\lceil (n-1)/2 \rceil$ keys
    7. 非叶节点由**指针**控制，是逻辑层面，构成稀疏索引（逻辑关闭不用物理关闭）
    8. 树高最大为 $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$. 根节点下为 $2 * \lceil n/2 \rceil$ 再往下 $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$

节点大小一般等于磁盘块大小

可以有效地处理对主文件的插入和删除

因为索引可以在logarithmic time内重组

2. B+ 树的更新

当一条记录从关系中插入或删除时，建立在该关系上的索引必须相应更新

插入

查找搜索码的叶节点

如果叶节点中已存在搜索码

添加记录到文件（如果必要添加指针指向桶）

如果搜索码不存在

添加记录到主文件（如果必要创建桶）

如果叶节点中有空间，插入一堆搜索码和指针

如果叶节点中没有空间，分裂

分裂叶节点

将 n 个搜索码按顺序存在内存区域M中

值分为两半，前 ⌈n/2⌉ 不动，后一半放入新节点

新节点为P，K为P的最小搜索码，插入（K, P）到父节点中

分裂非叶节点

若父节点N已满，N复制到内存M，此时N有$n+1$个指针和n个搜索码

按顺序将（K, P）插入M

将 $P1 \ldots P\lceil n/2\rceil$ 从M放入N

将 $P\lceil n/2\rceil+1 \ldots Pn+1$ 放入N'

将 $(K\lceil n/2\rceil, N')$ 放入**N的父节点**

删除

找到记录并将它从主文件和桶中删除

如果没有桶或桶为空，从叶节点删除（搜索码，指针）

如果删完之后节点不够而且兄弟节点可以装进去，就**合并节点**

合并节点**对父节点只删不改**

将两个节点中的所有搜索关键字值插入到一个节点中，然后删除另一个节点

非叶节点：从父节点复制两节点中间值到合并后的

叶/非叶节点：删除(Ki–1, Pi)，Pi 是指向已删除节点的指针

如果删完之后指针太少，但旁边没有空位，就**重新分配指针**

叶节点：用兄弟节点的值，并替换父值

非叶节点：插入父节点的旧值到未满节点，从同级中移除一个值，并将适当的值插入父级

Week3c 散列索引

1. Static Hashing 静态散列

1. bucket 桶：存储一条或多条记录的存储单元，就是磁盘块
2. K表示搜索码值的集合，B表示所有桶地址的集合
3. h指散列函数，表示从K到B的函数，用于访问，插入和删除
4. 必须按顺序搜索整个桶去定位一条记录（因为不同的键值可能会被映射到相同桶中）

散列函数

最坏情况：所有的搜索码被映射到相同的桶中，访问时间和搜索码数成正比

理想的散列函数是：统一（桶值数相同），随机（不依赖文件中搜索键值实际分布）

公式：如果桶数为N，$h(value) = (a*value + b) \bmod N$

桶溢出处理

1. 发生原因：桶的数量不足，某个桶分配到的记录比其他桶多
2. 桶溢出只能被减少，不能避免
3. 溢出链：一个给定桶的所有溢出桶用一个链接列表链接在一起

散列索引

散列不仅可以用于文件，也可以用于索引结构的创建

散列索引将搜索码和指针存储进散列文件结构，通常散列索引为辅助索引

不足

静态散列要求固定桶地址集合B，但数据库会随时间而变大

根据当前文件大小选散列 - 初始桶数太少 - 文件增长 - 溢出过多 - 性能下降

根据未来文件预计大小算则散列函数 - 浪费初始空间

数据库缩小 - 空间再次浪费

如果使用新的哈希函数定期重新组织文件（昂贵，耗时）

解决措施：允许动态修改存储桶数

2. Dynamic Hashing 动态散列

只使用哈希函数的前缀索引 i，有i位需要用来正确定位，随着数据库大小而增减

每个桶 j 存储一个值 ij

ij 表明该桶共同的散列前缀长度

桶地址表中指向桶j的表项编号（指针数）为 $2^{(i-i_j)}$

查询

找到包含搜索键 Kj 的桶

计算 $h(Kj) = X$

使用 X 的第一位 i

插入时桶的分裂

如果 i > ij，多于一个指针指向j桶

分配一个新桶z，使得 ij = iz = ij+1

更新原本指向j桶的后半部分指针指向z

删除j桶的记录并重新排列记录

如果 i = ij，只有一个指针指向j桶

如果达到限制b，或这个插入中发生太多的拆分，创建一个溢出桶

否则，增加i，桶地址大小翻一倍

指向该桶的指针翻一倍

重新排列记录

删除值

如果桶变空，可以删除，必要时更新桶的地址表

如果桶非空，可以合并

只能合并具有相同ij值和相同ij-1索引值的桶

减少存储桶地址表的大小是一个昂贵的操作

只有当存储桶的数量远远小于表的大小时才应该执行

3. 优点与缺点

Extendable Hashing

优点：性能不会随文件增长而降低，最小化空间开销

缺点：查找涉及附加的间接层，桶地址表本身很大，更改桶地址表本身很贵

Ordered Indexing vs Hashing

顺序索引更适合于范围查询

散列索引更适合规定值查询

Week4a 关系代数

1. 一元运算：选择、投影和更名运算

选择运算

- 基本格式 $\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$

- 可以在选择谓词中进行比较 $=, \neq, >, \geq, <, \leq$

- 可以谓词合并 $\wedge$ (and), $\vee$ (or), $\neg$ (not)

- 举例 $\sigma_{branch_name=\text{"Perryridge"}}(account)$

投影运算

- 基本格式 $\Pi_{A_1, A_2, \ldots, A_k}(r)$ 只挑选这几个A

- 去除重复行，例如 $\Pi_{account_number,\ balance}(account)$

更名运算

- 基本格式 $\rho_x(E)$

- 举例：重新命名表Department的属性Name和Age为A和B `ρ (A, B) (Department)`

- 也可以同时更改表名和属性名 $\rho_{x(A_1, A_2, \ldots, A_n)}(E)$

- 举例：重新命名表Project为Pro，属性为P, Q和R `ρ Pro(P, Q, R) (Project)`

- 返回表达式E的结果，并赋给它名字x，同时将各属性更名为 A1, A2…

2. 二元运算

合并运算

- 基本格式 $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$

- 有意义的情况：两个关系的属性个数相同 + 第i个属性类型相同
- 举例 $\Pi_{customer_name}(depositor) \cup \Pi_{customer_name}(borrower)$

集合差运算

- 基本格式 $r - s = \{t \mid t \in r \text{ and } t \notin s\}$

- 用途：找出在r中而不在s中的元组
- 有意义的情况：两个关系的属性个数相同 + 第i个属性类型相同

笛卡尔积运算

- 基本格式 $r \times s = \{t\,q \mid t \in r \text{ and } q \in s\}$

- 用途：r中的每一个都与s中每一个对应

3. 附加的关系代数运算

集合交运算

- 基本格式 $r \cap s = \{t \mid t \in r \text{ and } t \in s\}$

- 有意义的情况：两个关系的属性个数相同 + 第i个属性类型相同

自然连接运算

- 只考虑相同属性的相同值相连接

除法运算

- 基本格式 $r \div s$

· 查询时通常会包含“ ***for all*** “

· *s*有的值*r*必须全部包括，并不显示*s*有的属性

赋值运算

· 给临时变量赋值

Set-Intersection	**Set-Intersection Operation**
	▪ Notation: $r \cap s$
	▪ Defined as:
	▪ $r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$
	▪ Assume:
	  ▪ r, s have the *same arity*
	  ▪ attributes of r and s are compatible
	▪ Note: $r \cap s = r - (r - s)$

Natural-Join Operation	**Natural-Join Operation**
	▪ Notation: $r \bowtie s$
	▪ Let r and s be relations on schemas R and S respectively. Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
	  ▪ Consider each pair of tuples（关系行） t_r from r and t_s from s.
	  ▪ If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
	  ▪ t has the same value as t_r on r
	  ▪ t has the same value as t_s on s
	▪ Example:
	$R = (A, B, C, D)$
	$S = (E, B, D)$
	  ▪ Result schema = (A, B, C, D, E)
	  ▪ $r \bowtie s$ is defined as:
	$\Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$

Division Operation	▪ Property
	  ▪ Let $q = r \div s$
	  ▪ Then q is the largest relation satisfying $q \times s \subseteq r$
	▪ Definition in terms of the basic algebra operation
	Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$
	$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$
	To see why
	  ▪ $\Pi_{R-S,S}(r)$ simply reorders attributes of r
	  ▪ $\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$ gives those tuples t in $\Pi_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.
	▪ Notation: $r \div s$
	▪ Often suited to queries that include the phrase "for all".
	▪ Let r and s be relations on schemas R and S respectively where
	  ▪ $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
	  ▪ $S = (B_1, \ldots, B_n)$
	The result of $r \div s$ is a relation on schema
	$R - S = (A_1, \ldots, A_m)$
	$r \div s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$
	Where tu means the concatenation of tuples t and u to produce a single tuple 啥意思???

Assignment Operation	▪ The assignment operation ($\leftarrow$) provides a convenient way to express complex queries.
	  ▪ Write query as a sequential program consisting of
	  ▪ a series of assignments
	  ▪ followed by an expression whose value is displayed as a result of the query.
	  ▪ Assignment must always be made to a temporary relation variable.
	▪ Example: Write $r \div s$ as
	$temp1 \leftarrow \Pi_{R-S}(r)$
	$temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$
	$result = temp1 - temp2$
	  ▪ The result to the right of the $\leftarrow$ is assigned to the relation variable on the left of the $\leftarrow$.
	  ▪ May use variable in subsequent expressions.

Week4b 选择

1. 查询代价的度量

cost = 查询花费的总运行时间，**传送磁盘块数 + 搜索磁盘次数**

一次传输b个块以及执行S次磁盘搜索将消耗 $b * t_T + s * t_S$

访问路径

也称为索引结构，它们提供了一条可以定位和访问数据的路径

功能：检索满足所需条件的元组所需的块I/O数

每个关系都有一个或两个表作为输入

两种可能的方法：文件扫描，与选择条件匹配的索引

2. 等值选择运算

file scan 文件扫描，扫描文件并检索满足选择条件的记录，块连续存储

A1 线性搜索

b_r block transfers + 1 seek

扫描每一个文件块，并测试所有记录，可用于任何文件

$$t_s + b_r * t_T$$

平均情形 $t_s + (b_r/2) * t_T$

这种线性搜索经常适用，不管：选择条件、文件中的记录顺序、索引的可用性

A2 二进制搜索

适用条件：选择为等式 + 属性按顺序排列

通过在块上进行二进制搜索来定位第一个元组的代价 $\lceil \log_2(b_r) \rceil * (t_T + t_S)$

index scan 索引扫描，选择谓词决定了使用哪个索引（选择条件须在索引搜索键上）

A3 主索引在候选键，等值比较

只需检索单个记录 $Cost = (h_i + 1) * (t_T + t_S)$

A4 主索引在非键，等值比较

$Cost = h_i * (t_T + t_S) + t_S + t_T * b$

检索多条记录，记录必须连续存储

b是包含匹配记录的块数

A5 辅助索引在键和非键上，等值比较

如果键是候选键，检索单条记录 $Cost = (h_i + 1) * (t_T + t_S)$

如果键不是候选键，检索多条记录 $(h_i + n) * (t_T + t_S)$

n匹配的记录可能在不同块上

3. 比较选择运算

使用线性文件扫描或二进制搜索

使用主索引

$\sigma_{A \geq V}(r)$ 使用索引找到第一个大于等于V的，从那里依次扫描

$\sigma_{A \leq V}(r)$ 不用索引，只顺序扫描关系

直到找到第一个大于V的元组就停止

使用辅助索引

$\sigma_{A \geq V}(r)$ 使用索引找到第一个大于等于V的，从那里依次扫描，找指针

$\sigma_{A \leq V}(r)$ 扫描索引叶找指针，直到找到第一个大于V的元组就停止

由于连续的记录可能存在于不同的磁盘块中

因此每取一条记录可能需要一次I/O操作

一次I/O操作都需要一次磁盘搜索和一次块传输

因此如果记录数很大，线性搜索可能更有用

4. **连接选择**

· **Conjunctive Selections** $\sigma_{\theta 1 \wedge \theta 2 \wedge \cdots \wedge \theta n}(r)$

使用一个索引

使用成本最小的

在将元组提取到内存缓冲区后测试它的其他条件

第一个条件的选择很重要，必须使用估计

使用多值索引

如果可用，请使用适当的复合（多键）索引

- ***Disjunctive Selections*** $\sigma_{\theta 1 \vee \theta 2 \vee \cdots \vee \theta n}(r)$

使用线性扫描

或者索引扫描：对每个条件使用相应的索引，并对所有获得的记录指针集进行并集

- ***Negation*** $\sigma_{\neg\theta}(r)$

线性扫描

如果索引适用，找出r不在元组中的

- ***Duplicate elimination***

通过哈希函数和排序实现

排序时，重复项将彼此相邻，重复项可以删除

优化：也可在运行过程中删除，或在外部排序归并的中间合并步骤中删除

哈希函数：相似的重复的将进入同一桶

- ***Evaluating Projection***

投影将删除不在“选择属性”列表中的列

但是很贵

5. 排序

通过在排序码上建立索引，然后使用该索引按序读取关系，可以完成对关系的排序

仅在逻辑上通过索引对关系排序

可能导致每读取一个元组就要访问一个磁盘块

不适合内存时：*External Sort-Merge* 外部排序归并算法

外排序：对不能全部放在内存中的关系的排序，最常用的算法是外部排序归并算法

M：*memory size*，内存缓冲区中可用排序的块数（内存缓冲区能容纳的磁盘块数）

算法实施：

创建多个排序好的归并段 run

合并归并段，假定归并段总数N小于M

使用 N个 内存块去缓冲输入段，1个 内存块缓冲输出

读每个归并段的第一个块到内存

在所有缓冲块中按序挑选第一个元组

把该元组作为输出写出，并将其从缓冲块中删除

如果输出缓冲区已满，则将其写入磁盘

直到所有缓冲块都空

如果N大于等于M，则在第一阶段可能产生M个甚至更多的归并段

每趟归并用M-1个归并段作为输入

头M-1个归并段进行归并得到一个归并段作为下一趟的输入

接下来的M-1个归并段类似的进行归并

如此下去，直到所有的初始归并段都处理过为止

此时归并段数目减少到原来的1/（M-1）个归并段

直到归并段数目小于M，得到归并段输出结果

代价分析

br：关系r中用于存储的磁盘块数

bb：缓冲区大小，把bb个缓冲块分配给每个归并段

当不适合内存时候

传输成本

初始归并段总数 br/M

总共需要的归并趟数 $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$.

初始创建以及每个过程中的块传输 **2br**

磁盘块传输总次数 $b_r(2\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil + 1)$

搜索成本

每次从一个归并段读取bb块数据

运行生成阶段，一次读一次写 $2\lceil b_r/M\rceil$

合并阶段，每次合并需要 $2\lceil b_r/b_b\rceil$

搜索总次数 $2\lceil b_r/M\rceil + \lceil b_r/b_b\rceil(2\lceil \log_{\lfloor M/bb\rfloor-1}(b_r/M)\rceil - 1)$

适合内存时： 采用标准排序技术，无归并段和外部归并排序，**block transfers = 2br**

Week4c 连接运算 $b * t_T + s * t_S$

1. Nested-loop join 嵌套循环连接（迭代记录）

最简单的连接，独立于任何

由两个嵌套的for循环构成

与选择算法中的先行文件扫描算法类似，不要求有索引，且不管连接条件算法均可用

r为外层关系，s为内层关系

如果外部输入很小，内部输入很大且已预先建立索引，连接更有用

tr · ts 表示将tr和ts元组的属性值拼接而成的一个元组，放入结果集前需删除重复属性

代价很大，因为该算法逐个检查两个关系中的每一对元组

nr 表示r中元组，ns 表示s中元组

block transfers: $n_r * b_s + b_r$

seeks: $n_r + b_r$

小关系为外层关系

但如果小关系能够被放入内存，就作为内层关系，成本为 $b_r + b_s$ block transfers and 2 seeks

2. Block Nested-loop join 块循环连接（迭代块）

上一个的变种，读块而不是读元组

内部关系的每一块都与外部关系的每一块成对

block transfers: $b_r * b_s + b_r$

seeks: $2 * b_r$

最好情况 $b_r + b_s$ block transfers plus 2 seeks.

3. Indexed Nested-loop join 索引嵌套循环连接

上面两种方法的改进，可以代替文件扫描，如果：

是自然连接或等值连接

内层循环的连接属性上有索引（也可以构造）

连接成本 $b_r + n_r * c$ block transfers and seeks

c：遍历索引，为r中的某个元组找到所有匹配的s元组

C 可以被估计为 使用连接条件对关系s进行单次选择操作的代价

$c = hi + 1$

连接的时间代价：原公式基础上✖ $(t_T + t_S)$

如果两个表都有索引，谁tuple少谁是外关系

4 Merge-join 归并连接

先按两个关系的公共属性排序，类似归并排序算法的归并阶段

只能用于等值连接和自然连接

每个块只需要读一次

磁盘块传输次数是两个文件的块数之和 $b_r + b_s$

磁盘搜索次数为 $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

如果关系未排序，需要加上排序的成本

对关系r进行排序

块传输 $b_r (2 \lceil \log_{\lfloor M/bb \rfloor -1} (b_r / M) \rceil + 1)$

加上结果写回的 br 次块传输（适用内存不用写回）

磁盘搜索 $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor -1} (b_r / M) \rceil - 1)$

加上结果写回的 $\lceil b_r / b_b \rceil$ 次磁盘搜索

对关系s进行排序，同理

最后归并两个关系，将三个数据相加

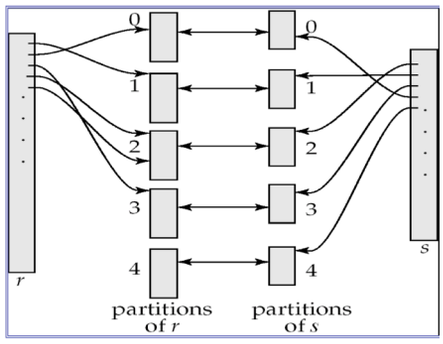
5. Hash-join 散列连接

可用于实现自然连接和等值连接

基本思想：把两个关系的元组划分成在连接属性上具有相同散列值的元组集合

假设A是关系r中元组，B是关系s中元组，h是元组属性ID上的散列函数

当且仅当 h(A) = h(B) 时 A和B 才需要比较



散列函数h所选择的分区n的数目为 $\lceil b_s/M \rceil * f$

f 近似于 *1.2*

以便*Si*放入内存中，*ri*不需要

当不需要递归划分时 $3(b_r + b_s) + 4 * n_h$ block transfers, and
$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n_h$ seeks

当题目中给出“ *the cost of writing partially filled blocks needs to be considered* ”

时计算 4nh

nh 为 build input 的另一个

如果主存中可以容纳整个构造输入关系时，成本缩减至 $b_r + b_s$ *and 2 seeks.*

具体操作

Hash-Join

用散列函数h来划分两个关系的元组

- Also only 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.

关系r的元组的划分　　自然连接中r和s的公共属性

  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1, \ldots, s_n$ denotes partitions of *s* tuples 关系s的元组的划分
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- General idea:
  - Partition the relations according to this
  - Then perform the join on each partition r_i and s_i
    - There is no need to compute the join between different partitions 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.

可能存在散列值相同，但ID值不同的情况

6. 其他运算

消除重复

Other Operations: Duplicate Elimination

- **Duplicate elimination** can be implemented via hashing or sorting. 通过哈希或排序实现
  - On sorting duplicates will come adjacent 排序时，重复项相邻 to each other, and all but one set of duplicates can be deleted.
  - *Optimisation:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge. 运行期间
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection:**
  - perform projection on each tuple;
  - followed by duplicate elimination.

聚合运算

Other Operations: Aggregation

- **Aggregation** 聚合 can be implemented similarly to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimisation:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group. 将聚合值保留在组中到目前为止找到的元组上
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

集合运算

Other Operations : Set Operations

- **Set operations** ($\cup$, $\cap$ and -): can either use variant of merge-join after sorting, or variant of hash-join.
- Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not in it.
          2. At the end, add the tuples in the hash index to the result.
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is in the hash index, delete it from the index.
          2. At the end, add remaining tuples in the hash index to the result.

Week5a 表达式合集

1. 物化和流水线

Materialisation 物化

输入是关系或者被计算好的关系

一次执行一个操作，将中间结果作为临时关系被存储于磁盘上

优点：任何情况都适用

缺点：可能需要很大存储空间，将结果写入磁盘并读回它们的成本可能相当高（我们的计算公式都忽略了将最终结果写入磁盘的成本）

因此总成本 = 单个操作成本的总和 + **写入中间结果到磁盘的成本**

解决措施：两个输出缓冲区（一个满了，另一个填满的同时将它写入磁盘）

Pipelining 流水线：只算最终成本

同时执行几个操作，不需要磁盘存储临时结果

可能不适用于一些情况：如排序和哈希连接

两种执行方法

生产者驱动：operator急切地生成元组并将其传给父级，缓冲区在operators之间，孩子把元组放在缓冲区，父级删除；如果区满，孩子等待

需求驱动：系统反复请求顶层操作的下一个元组，每个操作根据需要从孩子请求元组，为了产出下一个元组；调用间，操作必须保持“state”，以便知道下一次返回什么

2. 基于成本的查询优化

evaluation plan 查询计划：定义了每个操作所用算法和如何协调这些操作的执行

cost-based query optimisation 基于成本的查询优化：找到逻辑等价表达式

equivalent 等价：两个关系表达式会产生相同的元组集

equivalent rule 等价规则：两个不同形式的表达式是等价的

A1. 合取运算分解单个选择运算 $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$

A2. 选择运算交换律 $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$

A3. 投影只需要最后一个 $\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$

A4. 选择运算与笛卡尔积

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$
$$\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$$

A5. 等值/自然连接交换律 $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

A6. 等值连接结合律 $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$

θ2 只涉及 E2和E3 的属性

自然连接结合律 $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

A7. 选择运算与等值连接

选择条件中的所有属性只涉及参与的表达式之一 $\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$

θ1 只涉及 E1，θ2 只涉及 E2 $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$

A8. 投影运算与等值连接

θ只涉及 L1和L2的并集 $\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\prod_{L_1}(E_1)) \bowtie_\theta (\prod_{L_2}(E_2))$

L3是E1出现在θ中但不在L1 ∪ L2中，L4是E2出现在θ中但不在L1 ∪ L2中

A9. 集合的并与交满足交换律 $\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \prod_{L_1 \cup L_2}((\prod_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\prod_{L_2 \cup L_4}(E_2)))$

A10. 集合的并与交满足结合律

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$$
$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

A11. 选择运算与∪ ∩ - 有分配律

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

A12. 投影运算与∪ 有分配律

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Week5b 表达式统计结果的估计

1. 基本定义

nr 关系r的元组数

br 关系r中的磁盘块数

lr 元组的大小

fr 一个磁盘块能容纳的元组数

V（A, r） 属性A的非重复值个数，相当于 $\prod_A(r)$.

如果A是主码，V（A, r）= nr

假设r的元组物理上存储在一个块中 $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$

2. 选择运算估计元组个数

等值选择 $\sigma_{A=v}(r)$

满足选择条件的记录个数： $n_r / V(A,r)$

如果相等条件在主键上：size = 1

非等值比较（以小于为例） $\sigma_{A\le v}(r)$

c：满足条件的估计元组数

min(A, r) / max(A, r)：属性A的最大最小值

如果 v < min(A, r)，c = 0

如果 v >= max(A, r)，c = nr

否则： $c = n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$

3. **连接运算估计**元组个数

笛卡尔乘积 *r*s* 包含 *nr*ns* 个元组，每个元组占 *sr+ss* 字节

如果R ∩ S = ∅，r ⋈ s 相当于 r x s

如果R ∩ S 是R上键，s的元组与r中最多一个元组联接，r ⋈ s 不会超过 s 中元组数

如果R ∩ S 是S上的键，r ⋈ s 不会超过 r 中的元组数

如果R ∩ S 是S参照R的外键，r ⋈ s 就是 s 中的元组数（**in谁就是谁**）

如果R ∩ S 是R参照S的外码，r ⋈ s 就是 r 中的元组数

如果R ∩ S 既不是是S的码也不是R的码

假定 R ∩ S = {A}

R中的元组数 $\frac{n_r * n_s}{V(A,s)}$

S中的元组数 $\frac{n_r * n_s}{V(A,r)}$

两个里面较小的可能更精确

4. 投影估计 $\prod_A(r) = V(A,r)$

5. 集合运算

相同关系上 $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$

不同关系上（都不精确，但但提供大小的上限）

$r \cup s$ = *size of r + size of s*

$r \cap s$ = *minimum size of r and size of s.*

$r - s$ = r

6. 不同取值个数的估计

θ令A取特定值，$V(A, \sigma_\theta(r)) = 1$

θ令A取一组特定值，$V(A, \sigma_\theta(r))$ = 值的个数

θ令A是比较的形式，$V(A, \sigma_\theta(r)) = V(A,r) * s$（S是该选择的中选率）

在别的情况下 $\min(V(A,r), n_{\sigma_\theta(r)})$

$V(A, r \bowtie s)$ 的估计

如果A的所有属性都来自r， $\min(V(A,r), n_{r \bowtie s})$

如果A的属性A1来自r，A2来自s， $\min(V(A1,r)*V(A2-A1,s), V(A1-A2,r)*V(A2,s), n_{r \bowtie s})$

7. 评估计划的选择

当选择评估计划时，必须考虑所选评估技术之间的作用

单独为每个操作选择最便宜的算法可能不会产生最佳的整体算法

查询优化程序的两种方法：

cost-based 基于成本的优化：搜索所有计划，并以成本为基础选择

使用动态规划进行优化，将计算过的子集存起来，等需要时候直接使用

伪代码	

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

时间复杂度	With dynamic programming time complexity of optimisation with bushy trees is $O(3^n)$.
空间复杂度	Space complexity is $O(2^n)$ as the number of subsets of the S is 2^n.

采用等价规则的基于代价的通用优化器

优点：可以处理各种各样的查询结构，易于扩展新的规则到优化器

缺点：枚举过程昂贵

为了使方法高效需要以下技术：

- To make the approach work efficiently requires the following: 为了使方法高效需要以下技术：
  - A space-efficient representation of expressions
  - Efficient techniques for detecting duplicate derivations of the same expression
  - dynamic programming based on memoisation
  - avoid generating all possible equivalent plans

1. 一种节省空间的表达式表达形式，来避免应用等价规则是创建相同子表达式的多个副本
2. 检测相同表达式重复推导的有效技术
3. 基于缓存的动态规划形式
4. 避免产生所有可能的执行计划技术

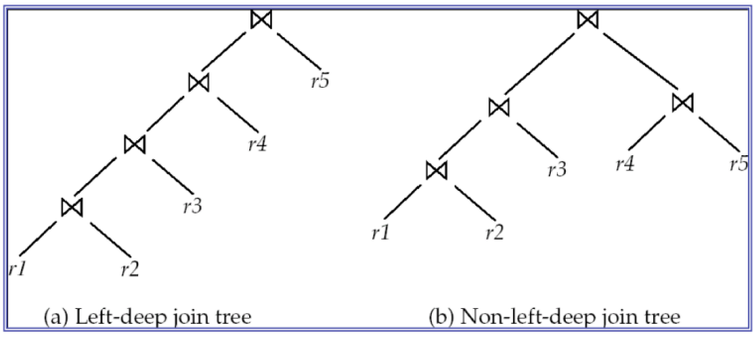
heuristic 启发优化：更便宜

先计算选择（减少元组数）

先计算投影（减少属性数）

执行最小的选择和连接

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform the most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.

左深连接树	In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join. 

左深连接的成本优化	▪ To find best left-deep join tree for a set of *n* relations:
	  ▪ Consider *n* alternatives with one relation as right-hand side input and the other relations as left-hand side input.
	  ▪ Modify optimisation algorithm:
	  ▪ Replace "**for each** non-empty subset *S1* of *S* such that *S1* ≠ *S*"
	  ▪ By: **for each** relation r in S, let S1 = S – r .
	▪ If only left-deep trees are considered, time complexity of finding best join order is O(n!), with dynamic programming this can be reduced to $O(n\ 2^n)$
	  ▪ Space complexity remains at $O(2^n)$
	▪ Cost-based optimisation is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

查询优化的结构

只考虑连接次序	Many optimisers considers only left-deep join orders. ▪ Plus heuristics to push selections and projections down the query tree ▪ Reduces optimisation complexity and generates plans amenable to pipelined evaluation.

集成	Some query optimisers integrate heuristic selection and the generation of alternative access plans.
	▪ Frequently used approach
	  ▪ heuristic rewriting of nested block structure and aggregation
	  ▪ followed by cost-based join-order optimisation for each block
	▪ Some optimisers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure

Week6a 事务

1. 事务概念

事务 transaction：访问并可能更新各种数据项的一个程序执行单元，如资金转账

不一致状态 inconsistent state：由于故障，系统状态不再反映真实状态

ACID 性质

Consistency 一致性：事务的执行不改变前后数据（之和）

Atomicity 原子性：某个事务的所有操作要么反映，要么全部不反映

Durability 持久性：一旦事务完成，任何系统故障都不会引起数据丢失

事务成功完成后，它对数据库所做的更改仍然存在，即使存在系统故障

Isolation 隔离性：每个事务都感觉不到系统中有其他事务在并发执行

事务状态

Active 活动状态：事务在执行时的初始状态

Partially committed 部分提交状态：最终提交前，完成最后一条语句后

Committed 提交状态：成功完成后

Failed 失败状态：无法再正常执行

Aborted 中止状态：事务被回滚**并且数据库已经恢复到事务开始之前的状态**之后

Terminated 结束状态：事务已经提交或中止

并发执行

概念：事务处理系统允许多个事务并发执行

优点：提高吞吐量和磁盘利用率；减少等待时间

并发控制机制：保证**隔离性**，控制事务之间的交互，以防止它们破坏数据库一致性

数据访问

physical blocks：留在磁盘上的块

buffer blocks：暂时在主存储器的块，

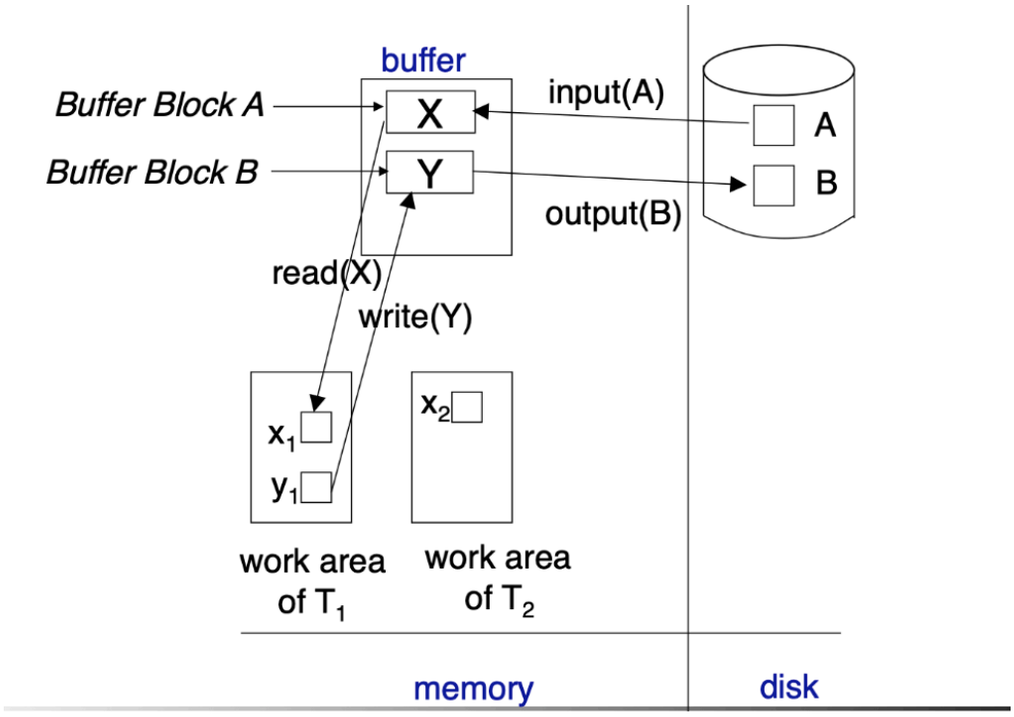
块的移动涉及到

input(B)：将 physical blocks 传送到 main memory (buffer)

output(B)：将 buffer blocks 传送回 disks，并替换相应的 physical blocks

read(X)：将块从 buffer blocks 传送到 work area of T1

write(X)：将块从 work area of T1 传送回 buffer blocks，不一定立即更新磁盘块上



2. 调度

schedule 调度：并发事务的执行顺序

一组事务的调度必须包含这些事务的所有指令

必须保持指令顺序，在每个单独事务中的

serial 串行：属于同一事务的指令在调度中紧挨在一起

serialisable 可串行性：如果一个（可能是并行的）调度等于串行调度

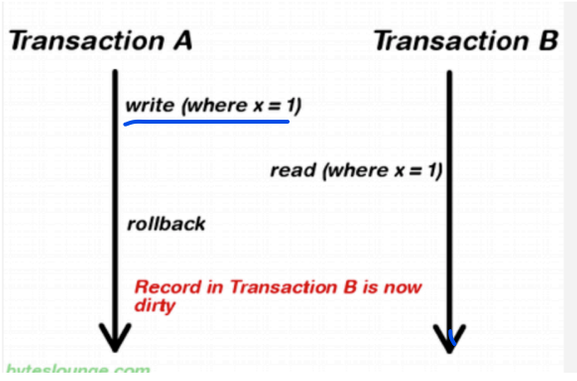
conflict 冲突：不同事务指令在相同的数据项上操作 + 有一个指令是 write(Q)

li和lj之间的冲突迫使他们之间形成（逻辑）时间顺序

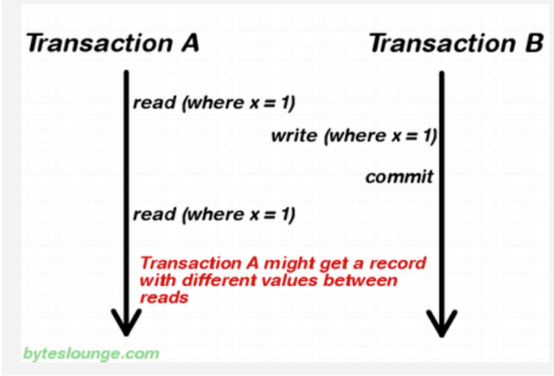
如果li和lj在一个调度中是连续的，且没有冲突，那么它们在调度中可以交换

三种冲突类型：

write-read: T2 读取 T1 修改但尚未提交的数据，dirty read



read-write: T1 只读，T2 读写提交，T1 再读，unrepeatable read



T1	T2
write(*Steven*)	
	write(*Paul*) write(*Steven*) commit
write(*Paul*) commit	

write-write: T1 写，T2 写提交，T1 再写，blind write

conflict equivalent 冲突等价：调度S通过一系列非冲突指令转换成S'

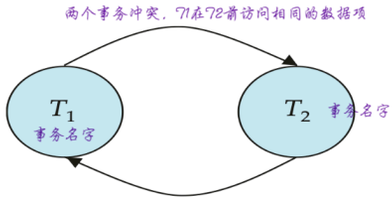
则S与S'冲突等价

conflict serialisable 冲突可串行：一个调度S与一个串行调度冲突等价

即一个调度S通过一系列非冲突指令转换成串行调度S'

则称S是冲突可串行

判定



为S构造一个有向图（优先图），G=（V, E）

加进去的边必须是三种冲突之一

如果图是非循环，就是冲突可串行，可通过拓扑排序得到

两个计划可能产生相同的结果，但不是冲突可串行化的

view equivalent 视图等价

考虑一个相同事务集的两个调度S和S'

针对每个事务项Q，需要满足以下三种情况：

· S中Ti读Q初始值，S'中Ti也要读Q初始值

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

· S中Ti执行read(Q)，该值是由Tj执行write(Q)产生；那么在S'中，Ti的read(Q)读取的值也是由Tj同一个write(Q)产生

· S中有事务执行了最终的write(Q)，则S'中该事务也必须执行最终的write(Q)

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

Schedule S1

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

Schedule S2

***view serialisable* 视图可串行**：一个调度S与一个串行调度视图等价

冲突可串行也是视图可串行，但反之不是

测试：

冲突可串行性的优先图测试不能直接用于测试视图的可串行性

测试视图可串的扩展在优先图的大小上具有指数级的成本

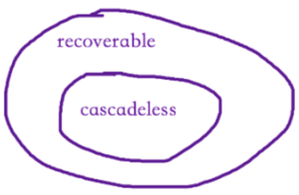
检查一个调度是否是视图可串行化的问题属于NP - complete 问题

***recoverable schedule* 可恢复调度**：如果T2读取T1写的数据，T1必须在T2**提交**前提交

***cascadeless schedule* 无级联调度**：如果T2读取T1写的数据，T1必须在T2**读前**提交

cascading rollback 级联回滚：因为单个事务故障导致一系列事务回滚

T_{10}	T_{11}	T_{12}
read(A) read(B) write(A)		
	read(A) write(A)	
		read(A)



Week6b 锁与并发调度

1. 锁的机制

好的事务是：冲突或视图可串行，可恢复或无级联调度

lock 锁：控

concurrency control protocols 并发控制协议：确保可串行

向**并发控制管理器**发出锁请求，控制器负责授予锁

只有在请求被批准后，事务才能继续

exclusive(X) 排他锁：能读能写，只能持有排他锁

shared(X) 共享锁：只能读，可以在之上加排他锁

locking protocol 封锁协议：事务遵从一组规则，规定了事务何时对数据项进行加锁

但不保证串行性

deadlock 死锁：互相等待释放，哪个事务都不能正常执行，须回滚其中之一

starvation 饿死：某个事务永远无法取得进展或重复被选择回滚

源于并发控制器设计的不好，所以它被设计防止饿死

最常见的解决方案是在选择受害者的成本因素中包括回滚次数

2. **两阶段封锁协议：确保冲突可串行性（反之不一定）** 2PL

该协议要求每个事务分两个阶段提出加锁和解锁申请

growing phase 增长阶段：事物可以获得锁，不能解锁

shrinking phase 缩减阶段：事物可以释放锁，不能获得锁

lock point 封锁点：该事务获得其最后加锁的位置（增长阶段结束点）

事务**按锁点顺序为可串行化顺序**，但会发生死锁和级联回滚

***strict two-phase locking* 严格两阶段封锁协议**：避免级联回滚

事务持有的排他锁必须在事务 *commit/abort* 后释放

从而防止其他事务读

***rigorous two-phase locking* 加强两阶段封锁协议**：事务按提交顺序串行化

事务提交前不得释放任何锁

缺点：降低事务的并发性

lock conversions 锁转换：对两阶段封锁协议的优化，**只产生冲突可串行**

upgrade：将共享锁升级为排他锁，发生在增长阶段

downgrade：将排他锁降级为共享锁，发生在缩减阶段

仍然依赖于程序员插入各种锁指令

仅在需要时变更，获得更高的并发度

如果排他锁直到事务提交后再释放，则无级联

自动获得锁：不用显式的锁调用

- 当*Ti*进行*read(Q)*前，自动产生*lock-S(Q)*
- 当*Ti*进行*write(Q)*前，检查是否持有共享锁

  若有，*upgrade(Q)*；否则 *lock-X(Q)*
- 当一个事务 *commit/abort* 后，释放所有锁

lock manager 锁管理器：封锁的实现，实现为一个单独的过程

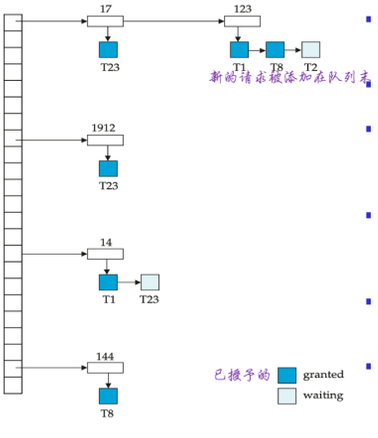
事务向其发送锁定和解锁请求

它针对锁请求返回授予锁，或在死锁时要求回滚

请求中的事务等待，直到请求被批准

维护一个数据结构（锁表），记录已授予的锁和待批准的锁

lock table 锁表：锁管理器为目前已加锁的每个数据项维护一个链表



- 锁表通常实现为内存中的哈希表，根据被锁定的数据项的名称编制索引
- 每个数据项都有一个已授予锁或等待授予锁的事务列表
- 新请求被添加到数据项的请求队列的末尾
- 解锁请求会导致请求被删除，然后检查以后的请求是否可以被授予
- 如果事务中止，则删除事务的所有正在等待或已授予的请求
- **保证了无饿死**

3. 基于图的协议

两阶段锁定的替代方案

给每个数据项施加一个偏序：$D = \{d1, d2, \ldots, dh\}$

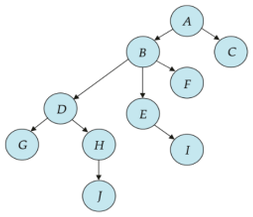
$di \rightarrow dj$ 表明任何同时访问$di \rightarrow dj$的事务都必须在访问dj之前访问di

集合D现在可以看作是一个有向无环图，称为 database graph

tree protocol 树协议：一种简单的图协议，只允许排他锁

除了首次加锁，Q被加锁的前提是Ti持有Q的父项上的锁

可以随时解锁



Ti已锁定和解锁的数据项随后无法重新锁定

解锁可能比2PL更早发生

保证了**冲突可串行和防止死锁**

但不保证可恢复和无级联（事务结束前不允许释放排他锁）

优点：不产生死锁，不需要回滚；可较早释放锁，减少等待时间，增加并发性

缺点：不保证可恢复和无级联，可能给不访问的数据加锁

4. 死锁处理

死锁预防协议确保了系统永远不会进入死锁状态

预防策略：

在每个事务执行前，锁定所有数据项

对所有数据项强制部分排序，要求事务只能按（基于图的协议）指定顺序锁定

使用 timestamps 时间戳

wait-die 非抢占技术：$T1$所申请的锁被$T2$持有，若$T1$更老，$T1$等待；若$T1$更新，$T1$回滚

wound-die 抢占技术：$T1$所申请的锁被$T2$持有，若$T1$更新，$T1$等待；若$T1$更老，$T2$回滚

可能发生不必要的**回滚，避免饿死**

lock timeout-based schemes 锁超时

事务只在指定的时间内等待锁。之后，等待超时，事务回滚

因此防止死锁

会**发生饿死**

死锁检测：如果系统没有采用能保证不产生死锁的协议

死锁可以描述为 *wait-for graph*，*Ti* → *Tj* 意味着*Ti*正在等待*Tj*释放一个数据项

当*Ti*请求当前由 *Tj* 保存的数据项时，将边（*Ti* → *Tj* ）插入到等待图中

只有当*Tj* 不再保存*Ti*所需的数据项时，才会移除该边缘

当且仅当等待图有一个循环时，系统处于死锁状态

必须定期调用死锁检测算法来查找循环

当死锁被检测出来，需要进行三个步骤：

选择回滚的牺牲者：使事务回滚带来的代价最小

回滚：决定回滚多远

total rollback 彻底回滚：中止事务，重新开始

partial rollback 部分回滚：只回滚到可以解除死锁处，更有效

饿死：同一事务总是被当作牺牲者，在成本系数中包括回滚次数以避免饿死

5. 多粒度 Multiple Granularity

允许不同大小的数据项，并定义数据粒度的层次结构

其中小粒度嵌套在较大的粒度中

也能被表示为树形，但是与树形协议不同

这里的非叶节点表示与后代相联系的数据

树形协议的每个节点是一个相互独立的数据项

当事务显式锁定树中的一个节点时，它隐式地以相同的模式锁定节点的所有后代

granularity of locking 锁的粒度：树中的级别

fine granularity 更低：高并发性，高锁开销

coarse granularity 更高：低并发性，低锁开销

intention lock mode 意向锁

避免搜索整棵树

如果一个节点加上意向锁，意味着要在树的较低层进行显式加锁

因此，事务不必搜索整棵树就能判定能否成功给一个节点加锁

希望给Q加锁的事务必须遍历从根到Q的路径

在遍历树的过程中，该事务给各节点加上意向锁

三种额外的锁：

intention-shared IS 共享意向：在树的较低级别显式锁定，但仅使用共享锁

intention-exclusive IX 排他意向：使用排他锁或共享锁在较低级别显式锁定

shared and intention-exclusive SIX：以该节点为根的子树显式的加了共享锁，并在树的更底层显式的加排他锁

multiple granularity locking scheme 多粒度封锁协定：事务按如下规则给Q加锁

1. 必须遵循锁相容矩阵
2. 事务Ti必须先封锁树的根节点，并且可以加任意类型的锁
3. 仅当Ti对Q的父节点具有IX或IS锁时，Ti对结点Q可加S或IS锁
4. 仅当Ti对Q的父节点具有IX或SIX锁时，Ti对结点Q可加X, SIX或IX锁
5. 仅当Ti未曾对任何结点解锁时，Ti可对结点加锁（也就是说，Ti为两阶段）
6. 仅当Ti当前不持有Q的子节点的锁时，Ti可对结点Q解锁

从根到叶获取，从叶到根释放

锁粒度升级：如果某个特定级别的锁太多，请切换到更高粒度的S或X锁

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

	冲突可串行	无死锁	无饿死	可恢复	无级联
两阶段封锁协议	√				
严格两阶段	√			√	√
强两阶段	√			√	√
锁转换	√				
锁管理器			√		
基于图的协议	√	√			
树形协议	√	√			
死锁预防 抢占/非抢占		√	√		
锁超时		√			

Week6c 故障恢复系统

1. 故障分类

本章只考虑如下三种故障

1. *Transaction failure* 事务故障

逻辑错误：由于某些内部错误情况，事务无法完成

系统错误：由于系统进入一种不良状态，如死锁，事务无法继续执行

2. *System crash* 系统崩溃

电源、硬件或软件故障

易失性存储器内容丢失，非易失性仍完好无损

fail-stop assumption 假定非易失性存储内容不会因系统崩溃而损坏

3. *Disk failure* 磁盘故障

由于磁头损坏或故障造成磁盘块上的内容丢失

破坏被假定为可检测的：磁盘驱动程序使用校验和来检测故障

恢复算法：故障发生后仍保证数据库一致性和事务原子性

- 正常事务处理时，保证有足够的信息用于故障恢复
- 故障发生后，将内容恢复到保证原子、一致和持久性

2. 存储器分类

volatile storage 易失性存储：无法在系统崩溃后幸存

如 *main memory, cache memory*

non-volatile storage 非易失性存储：可以在系统崩溃后幸存

如 *disk, tape, flash memory, non-volatile (battery backed up) RAM*

但仍会失败，丢失数据

stable storage 稳定存储：可以在任何故障中幸存

近似于在不同的非易失性介质上保持多个拷贝

副本可以放在远程站点上，以防发生火灾或洪水等

稳定存储的实现

数据传输过程中：将信息写入第一个物理块；当第一次写入成功完成时，将相同的信息写入第二个物理块。只有在第二次写入成功完成后，输出才会完成

输出操作期间：首先找到不一致的块（在非易失性存储器上记录正在进行的磁盘写入，使用此信息可查找可能不一致的块，并仅比较这些块的副本，用于硬件RAID系统），如果检测到不一致块的任一副本有错误，则用另一个副本覆盖它；如果两者都没有错误，但是不同，则用第一个块覆盖第二个块（反）

3. 数据访问

Work-area：保存Ti所访问及更新的所有数据项的拷贝

Ti的数据项X的本地副本 xi

read(X):

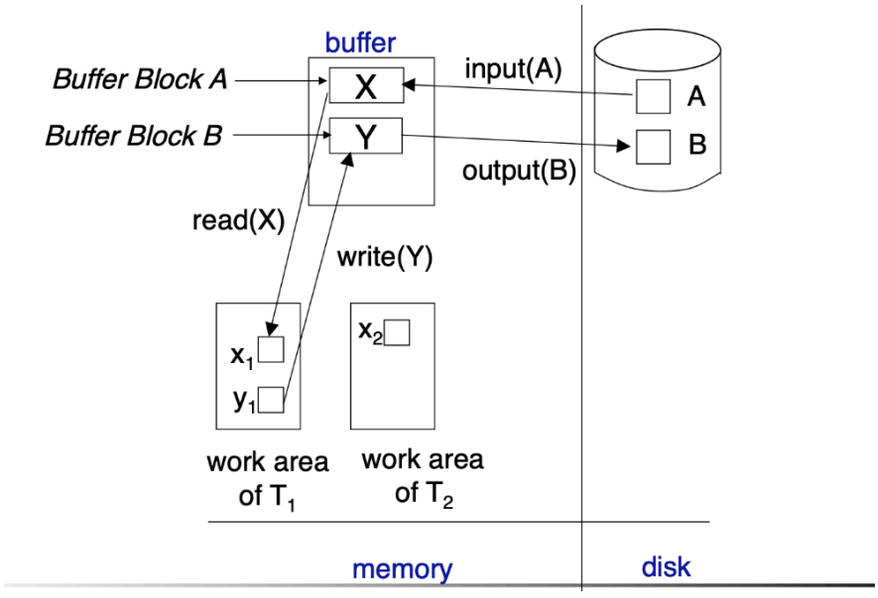
将数据项X值赋给局部变量xi

在第一次访问X之前必须执行

write(X):

将局部变量Xi值赋给缓冲区X

可以在事务提交之前的任何时候执行



4. 日志

为了保证原子性，在修改数据库之前

先向稳定存储器输出信息，描述修改（如，日志），而不修改数据库本身

日志：最广泛的记录数据库修改的结构

保存在稳定存储器中，维护数据库上更新活动的记录

日志记录：**事务在对数据库修改前创建日志记录**，日志是一系列日志记录

<Ti start> 事务Ti开始

<Ti commit> 事务Ti提交

<Ti abort> 事务Ti中止

<Ti, X, V1, V2>：更新一个日志记录，在事务Ti执行write(X)操作前

写操作前X的值是V1，写操作后X的值是V2

使用日志的两种方法：

延迟修改技术：仅在事务**提交时**对缓冲区/磁盘执行更新，有存储本地副本的开销

立即修改技术：数据库修改在事务仍然活跃时发生

允许对缓冲区进行未提交事务的更新

或者磁在事务**提交前**对磁盘本身更新

必须在写入数据库项之前写入更新日志记录

假设日志记录直接输出到稳定存储器

在事务提交之前或之后，可以随时将更新的块输出到稳定存储

块的输出顺序可能与它们的写入顺序不同

committed 已提交：当事务的提交日志记录被输出到稳定存储时

此时 *write(X)* 可能仍在缓冲区，并在稍后输出

假定事务*Ti*修改了一个数据项，在*Ti*提交或中止之前，不允许其他事务修改该数据项

并发事务共享一个磁盘缓冲区和一个日志

缓冲区块可以由一个或多个事务更新数据项

未提交事务的更新不应对其他事务可见

可以使用严格两阶段封锁来完成：获取更新项排他锁并保持锁直到事务结束

***undo* 撤销**：将*Ti*更新过的所有数据项的值都**恢复成旧值**

只包含 <Ti start>

撤销 <Ti, X, V1, V2>，将V1写入X

从Ti的最后一项记录往回走

写出一个特殊的日志记录 **<Ti, X, V1>**

当撤销完成，写入 **<Ti abort>**

redo 重写：将Ti更新过的所有数据项的值都**设置成新值**

包含 <Ti start> 和 <Ti commit>/<Ti abort>

重写 <Ti, X, V1, V2>，将V2写入X

从Ti的第一项记录向前

重做会重做所有原始操作，包括恢复旧值的步骤

不需要再写日志

故障发生时候：

如果日志只包含 **<Ti start>**，**undone**

如果日志包含 **<Ti start>**，**<Ti commit>** 或 **<Ti abort>**，**redone**

如果日志先 **undone** 再 **<Ti abort>** 写入，故障发生，故障恢复后需要 **redone**

checkpoints 检查点：简化恢复过程，避免不必要的重做

将当前驻留在主内存中的所有日志记录输出到稳定存储器中

写 <checkpoint L> 到稳定存储器，L 指检查点时还活动的所有事务列表

执行检查点时，所有更新都将停止

在恢复过程中，只需要考虑

检查点之前启动但尚未完成的事务 + 在检查点之后启动的事务

因为在检查点之前提交或中止的事务已将其所有更新输出到稳定存储

失败时，从日志末尾往回扫描以找到最近的 <checkpoint L> 记录

只有在 L 中或在检查点之后启动的事务才需要 redo / undo

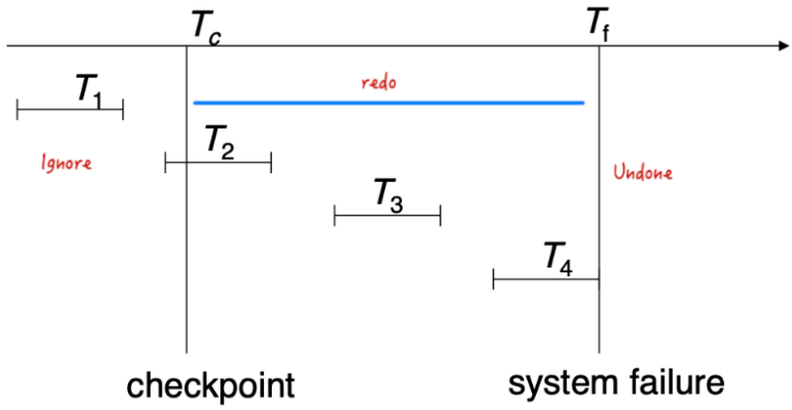
继续往回扫描，直到为L中的每个事务Ti找到一个记录<Ti start>

T1: 忽略

T2 and T3: redone

T4: undone

but before failure point need to be redo



5. 故障恢复算法

正常操作期间：

记录日志：<Ti start>，<Ti, Xj, V1, V2>，<Ti commit>

日志回滚：从末尾往回扫描，重写 <Ti, Xj, V1>，一旦发现 <Ti start>，扫描停止，写 <Ti abort>

故障恢复期间：

redo 阶段：从最后一条checkpoint向前扫描，重播所有事务

找到最后一个<checkpoint L>，令 undo-list = L

发现 <Ti, Xj, V1, V2>，用V2重写Xj

发现 <Ti start>，添加 Ti 到 undo-list

发现 <Ti commit/abort>，删除 undo-list中Ti

undo 阶段：从failure point 往回扫描

当Ti 在 undo-list

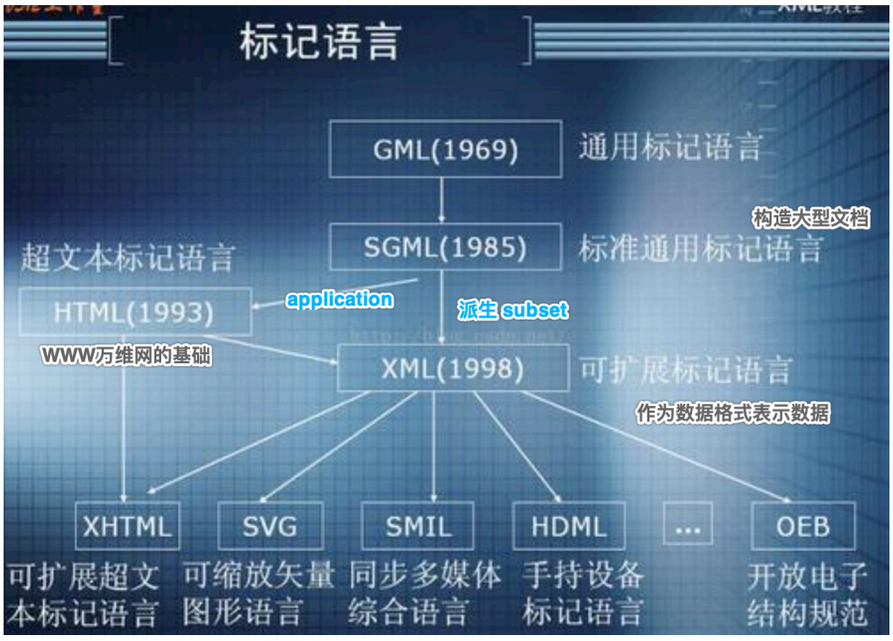
发现 <Ti, Xj, V1, V2>，执行撤销 用V1重写Xj，写日志记录 <Ti, Xj, V1>

发现 <Ti start>，写日志记录 <Ti abort>，将Ti从undo-list中删除

直到undo-list中为空，撤销阶段结束，正常事务

Week8a XML 可扩展性标记语言

一、XML 的动机



extensible：用户可以添加新的标签，并分别指定如何处理

markup 标记：文档中任何不想用于打印输出的部分

tag 标签：封闭在尖括号内的

概念：

用户可以添加新标记，并分别指定如何处理标记以进行显示

被 the WWW Consortium (W3C) 定义

交换数据（而不仅是文档）指定新标记和创建嵌套标记结构的能力

XML与HTML相比：没有指定的标签集，每个用户可以选择自己需要的标签集

所以XML用于数据表示和交换

HTML用于文档格式化

XML的优势（与关系数据相比）

· 利于数据交换以及文件中存储复杂结构信息

· 标签使得消息是 self-documenting自描述，不需要参考模式就可以理解文本含义

· 文档格式不严格，可以根据需要忽略一些标签，添加新标签

· 允许嵌套结构，尽管有冗余

· 被广泛接受

· 缺点：标签名称在整个文档中被反复使用，效率低

二、XML 的结构

markup 标记：文档中任何不想用于打印输出的部分

tag 标签：封闭在尖括号内的

element 元素：一对互相匹配开始和结束的标签，必须正确被嵌套 <course> … <title> …. </title> </course

root 根：独立的根元素来包含文档里的所有其他元素

in the context of：文本在开始和结束标签间出现

attributes 属性：在开始标签中以 *name=value* 的形式出现

一个元素可以有很多属性

但每个属性名只能出现一次

在元素开始的标签内

如 <course **course_id = "CS-101" credits="4"**>

attributes vs subelements

在文档构造环境中，区别很重要

属性是*markup* 的一部分，是隐式的

子元素是可以重复的、文档内容的一部分

在数据交换应用中，区别不重要

属性用于元素标识符，子元素用于文档

- <course course_id= "CS-101"> … </course>
- <course>
  <course_id>CS-101</course_id> …
  </course>

namespace 名字空间：在每个标签或属性前面加上资源标识符（如网址），避免重复

格式：*unique-name: element-name*

在根元素中使用 <university **xmlns:yale="http://www.yale.edu"**>

在元素中即 <**yale: course**>

三、XML 文档模式

数据库模式约束可以存储的信息以及存储值的数据类型

XML文档不需要任何相关模式或限制信息类型，但**数据交换**需要模式

指定XML模式的两种机制

Document Type Definition (DTD) 文档定义类型

可以使用DTD指定XML文档的类型

约束数据结构（什么元素可以出现，元素可以/必须具有哪些属性，哪些子元素可以/必须出现在每个元素内部，以及出现多少次）

不限制数据类型，所有都是字符串

根元素使用正则表达式

<!ELEMENT university ((department | course | instructor | teaches)+)>

| ：或

+ ：一个或多个

*：零个或多个

? ：零个或一个

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept name, building, budget)>
    <!ELEMENT course ( course id, title, dept name, credits)>
    <!ELEMENT instructor (IID, name, dept name, salary)>
    <!ELEMENT teaches (IID, course id)>
    <!ELEMENT dept name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
]>
```

元素的表达式

<! ELEMENT department (dept_name, building, budget)>

子元素的表达式

<! ELEMENT dept_name (#PCDATA)>

#PCDATA ：文本数据

EMPTY ：这个元素没有内容

ANY：子元素没有限制

属性的表达式

<!ATTLIST course course_id CDATA #REQUIRED>

类型声明

CDATA ：属性包含字符数据

ID：提供该元素唯一标识符，一个元素最多只允许一个属性为ID

IDREF：对一个元素的引用，这个属性必须包含出现某个元素的ID属性上的值

IDREFS：允许以空格分开的一个引用列表

默认声明

#REQUIRED：每个元素在该属性上必须指定一个值

有一个默认值

#IMPLIED：没有提供默认值

- <!ATTLIST course course_id CDATA #REQUIRED>, or
- <!ATTLIST course
  course_id ID #REQUIRED
  dept_name IDREF #REQUIRED
  instructors IDREFS #IMPLIED >

University DTD with ID, IDREF, and IDREFS attribute types. DTD

```
<!DOCTYPE university-3 [
   <!ELEMENT university ( (department|course|instructor)+)>
   <!ELEMENT department ( building, budget )>
   <!ATTLIST department
        dept_name ID #REQUIRED >
   <!ELEMENT course (title, credits )>
   <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
   <!ELEMENT instructor ( name, salary )>
   <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
   ··· declarations for title, credits, building,
        budget, name and salary ···
]>
```

XML data with ID and IDREF attributes

XML文档事例示例

```
<university-3>
    <department dept_name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept_name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id="CS-101" dept_name="Comp. Sci"
            instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    ....
    <instructor IID="10101" dept_name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    ....
</university-3>
```

DTD的局限性

· 单个文本元素和属性类型不能进一步限定，所有都是字符串

· 很难指定子元素的无序集合，顺序在数据交换中不重要

· ID和IDREF中缺乏类型限定，导致两个子元素中的属性互相引用

XML Schema 更完善的模式语言

解决了DTD的不足

· 更多的内置类型，整数、字符串、布林、小数等，也限制最大最小值

· 允许用户自定义类型，也可以是复杂类型

· 更多功能：唯一性和外键约束

· 允许使用继承的形式来扩展复杂类型

名字空间的整合

该模式由XML语法指定，为了避免与用户定义标签发生冲突

在XML schema标签上增加名字空间前缀“ **xs:** “

例如 **<xs: schema xmlns:xs="http://www.w3.org/2001/XMLSchema">**

更标准，更复杂，也更冗长

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

四、查询和转换

从一个 XML schema 到另一个的转换

重要性：

从体积庞大的XML数据中提取信息

模式之间的转换数据

标准的查询/转换语言有三种

XPath：简单的路径表达式，是XQuery的基础

XSLT：主要用于文档格式化

XQuery：查询XML数据的标准语言

树模型：查询和转换的基础

XML文档被建模为树，节点对应于元素和属性

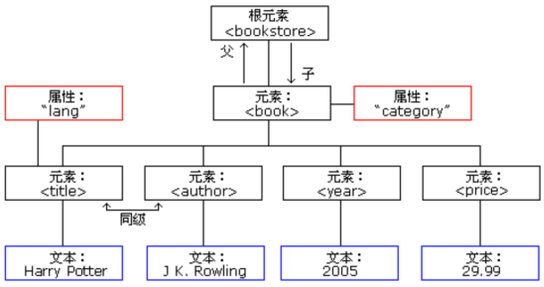
元素节点有子节点，子节点可以是属性或子元素

元素的文本内容可以被建模为元素的文本子节点

子节点根据其在XML文档中的顺序进行排序

元素和属性节点（根节点除外）只有一个父节点，即元素节点

根节点有一个子节点，它是文档的根元素



XPath：通过路径表达式选择XML文档的部分内容

path expressions：路径表达式，***/university-3/instructor/name***

<name>Srinivasan</name>
<name>Brandt</name>

去除标签：***/university-3/instructor/name/text()***

selection predicates：包含在方括号中，***/university-3/course[credits >= 4]***

@：访问属性值，***/university-3/course[credits >= 4]/@course_id***

函数：

计算元素数量 ***/university-2/instructor[count(./teaches/course)> 2]***

and 和 or 也可以在谓词中使用，not() 用于否定

IDREFS 的引用：id("foo") 返回属性的类型为ID且值为"foo"的节点

如 ***/university-3/course/id(@dept_name)***

返回被course元素的dept_name属性引用的所有department元

表达式结果合并：

```
/university-3/course[@dept name="Comp. Sci"] |
/university-3/course[@dept name="Biology"]
```

| 不能嵌套在其他操作符之中

结点依照他们在文档中出现的顺序被返回

跳过多层节点：***/university-3//name***

找出任意位置下所有的name元素

以命名文档的根 ***doc("university. xml")/university/department***

XQuery：XML标准查询语言，被 World Wide Web Consortium (W3C) 标准化

包含五个部分：for … let … where … order by …return…，即 **FLWOR**

for ⇔ SQL **from**
where ⇔ SQL **where**
order by ⇔ SQL **order by**
return⇔ SQL **select**
let allows temporary variables, and has no equivalent in SQL

For子句：指定在 XPath 表达式结果范围内变动的变量

Let子句：可选。可以只使用 for 和 return

- find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

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

Joins

```
for $c in /university/course,
$i in /university/instructor,
$t in /university/teaches
where $c/course_id= $t/course id and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

等价于

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches[ $c/course_id= $t/
course_id
    and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

嵌套查询

- The following query converts data from the flat structure for university information into the nested structure used in university-1

```
<university-1>
{   for $d in /university/department
    return <department>
              { $d/* }
              { for $c in /university/course[dept name = $d/dept name]
                return $c }
           </department>
}
{   for $i in /university/instructor
    return  <instructor>
              { $i/* }
              { for $c in /university/teaches[IID = $i/IID]
                return $c/course_id }
            </instructor>
}
</university-1>
```

- $c/* denotes all the children of the node to which $c is bound, without the enclosing top-level tag

Order by子句：可以被用于任何表达式末尾，如返回按name排序的instructors

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

使用 ***order by $i/name descending*** 按降序排列

可以在多个层次的嵌套中进行排序

按dept_name对department排序，并按course_id排序courses

```
<university-1> {
  for $d in /university/department
  order by $d/dept name
  return
        <department>
       { $d/* }
       { for $c in /university/course[dept name = $d/dept name]
         order by $c/course id
         return <course> { $c/* } </course> }
     </department>
} </university-1>
```

用户定义函数

- User defined functions with the type system of XML Schema

```
declare function local:dept_courses($iid as xs:string)
        as element(course)*
{
   for $i in /university/instructor[IID = $iid],
       $c in /university/course[dept_name = $i/dept name]
   return $c
}
```

类型对于函数参数和返回值是可选的

* (as in decimal*) 表示该类型的一系列值

where子句谓词的普遍量化与存在量化

- **some** $e **in** *path* **satisfies** *P*
- **every** $e **in** *path* **satisfies** *P*
- Add **and fn:exists($e)** to prevent empty $e from satisfying **every** clause

XQuery 还支持 If-then-else 子句

XSLT：一个 stylesheet 存储文档的格式选项，通常与文档分开

例如HTML样式表可以为标题等指定字体颜色和大小

是一种通用的转换语言，例如 translate XML to XML, and XML to HTML

templates: XSLT 转换的规则，它将使用XPath的选择与结果的构造结合起来

五、XML应用程序接口

用于处理单个XML文档的应用

但不适合查询大的XML数据的集合，此时更适合使用上面两个

SAX (Simple API for XML)：基于解析器模型，用户提供用于分析事件的事件处理程序

例如开始和结束元素

DOM (Document Object Model)：基于文档对象模型

把XML内容解析为树，其每个元素用结点表示

提供了很多函数用来遍历树

也提供了用来更新文档的函数

- E.g.: Java DOM API provides Node class with methods
  getParentNode( ), getFirstChild( ), getNextSibling( )
  getAttribute( ), getData( ) (for text node)
  getElementsByTagName( ), ...

六、XML的应用

· 用于存储复杂结构数据

如用户的使用偏好设置，需要有大量的域被记录

XML避免了大量的前期设计工作、创建词法分析器、读取文件等工作

基于XML的表示现在已被广泛用作存储文档、电子表单数据以及作为办公应用软件包的一部分的其他数据

两种使用最广泛的可编辑文档表示形式：

Open office软件包以及其他包公软件包所支持的ODF开放文档格式

微软办公软件包所支持的OOXML格式

· 标准化数据交换格式

如表示化学工业需要有关化学制品信息的标准，如沸点和熔点，ChemML

如在互联网上书写数学符号和公式的置标语言，MathML

如描述广泛传感器和测量过程标准的SensorML

· Web服务数据交换标准

外部组织和部分并不希望别人用SQL直接访问它们的数据库

但愿意通过预定义接口提供受限形式的信息

如通过支持Web服务，旅行网站整合了各个独立的服务，用户只需于这一个网站交互就可以进行行程预定，而不需要练习多个独立的网站

如基于HTTP协议的远程方法调用

SOAP简单对象访问协议为过程调用定义了一个标准，跨不同数据库应用的程序调用过程，使用XML来表示过程的输入和输出

Web服务：提供这样一个SOAP过程集合的站点

Web服务描述语言—WSDL用于描述Web服务

通用描述、发现和集成—UDDI定义如何创建一个Web服务的目录

· 数据中介

连接不同系统的通用数据表示格式

如比较购物，其中关于商品、库存、价格以及运送费用的数据是从销售特定商品的各种不同网站中提取出来的

再如提供对客户所有账户的集中式管理，如银行账号、信用卡、退休金账户

六、XML数据存储

· 一种方式是将其以文档形式存储在文件系统中

· 另一种是构建专用数据库在存储

· 另外也可以把XML数据转换为关系表示，并将其存储到关系数据库中

非关系的数据存储

存储在平面文件中：访问和查询数据相对容易，但缺乏数据隔离、原子性等

创建XML数据库：支持DOM模型和声明式查询，目前没有商业级系统

关系数据存储

利于被现有应用程序访问

数据必须被转化成关系格式，是成熟数据库系统，但增加转换数据和查询开销

存储方法：

存储为字符串

对于大的XML文档，其顶层元素有很多子元素

可以通过将每个子元素作为字符串存储到一个单独的元组中来处理

也可以通过把不同类型的元素存储在不同的关系中，并将一些关键元素的值存储为关系的属性以便索引

一些数据库系统支持函数索引，它使用函数的结果作为键值

优点：可以存储任何XML数据，即使没有DTD；只要文档中有许多顶级元素，字符串与完整文档相比都很小

缺点：需要解析字符串来访问元素内部的值，解析很慢

树表示法

每个元素/属性都有唯一标识符 ***nodes (id, parent_id, type, label, value)***

id：元组的标识

parent_id：父结点标识

type：元素或属性

label：元素的标签名/属性名

value：元素或属性的文本值

可以添加额外的属性 ***position*** 来记录子级的顺序

优点：可以存储任何XML数据，即使没有DTD

缺点：数据被分解成很多部分，增加空间开销；将子元素重组为元素需要大量的连接运算

映射到关系

已知模式的XML元素被映射为关系和属性

未知模式的元素按字符串存储或按树存储

所有只出现一次的子元素都可以成为关系属性

多次出现的子元素被表示进分离的表

发布和分解XML数据

发布：将关系数据转换为XML格式的过程

分解：将XML文档转换为要插入一个或多个关系的元组集的过程

支持XML的数据库系统可以自动发布和分解

在关系数据库中的本地存储

一些关系数据库支持XML的本地存储

这类系统将XML数据存储为字符串或更有效的二进制表示

而不需要将数据转换为关系形式

创建特殊用途的索引来索引XML数据

Week8b 语义Web与RDF

一、语义网的基本特征

万维网的扩展

给文档添加能够被计算机所理解的语义

更好的协调计算机和人

使用它的原因：需要通过Internet合并数据，人类知道如何组合这些信息

通过建立一个 technology stack 来支持 web of data

最终目标是使计算机能够做更多有用的工作，并开发能够支持网络可信任交互的系统

更重视计算机的理解和处理，并具有一定的判断、推理能力

从而使整个互联网成为一个通用的信息交换媒介

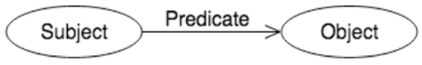
二、资源描述框架

RDF (Resource Description Framework)：在Web上表示信息的框架，有助于数据合并

核心结构是一个 tripes，由 subject, predicate, object 组成

一组 tripes 组成一个 RDF graph

每个 tripes 都由 node-arc-node 组成



RDFS：提供基本功能，描述RDF词汇表

OWL：万维网本体语言，建立在RDF与RDFS的顶部，提供额外功能，知识表示

是一种语义网语言

旨在表示关于事物、事物组和事物之间关系的复杂的知识

一种基于计算逻辑的语言

表示的知识可以被计算机程序利用

Ontologies：RDF and OWL documents

本体是是概念化的明确声明

概念模式定义数据上的关系，而本体定义术语来表示知识

在语义Web环境下，本体和知识库有时可以互换使用（但有一些细微的区别）

三、关联数据

Linked data：在网上收集相互关联的数据集

对现有数据库（关系、XML、HTML数据库等）进行转换或动态访问

数据之间的关系应该是可用的

设置查询端点以更方便地访问该数据

关联数据设计原则：

使用 URIs 作为事物名字

使用 HTTP URIs 让人们找到这些名字

如果人们查询URIs，使用标准(RDF*, SPARQL)提供有用信息

包括一些到其他URIs 的连接，以至于人们可以找到更多东西

四、SPARQL: Protocol and RDF Query Language

为RDF开发的一种查询语言和数据获取协议

可用于表示不同数据源之间的查询

包含查询必需的和可选的图形模式，及其连接和析取的功能

还支持可扩展值测试和通过源RDF图约束查询

查询的结果可以是结果集或RDF图

查询基于（三重）模式，SPARQL引擎将返回与这些模式匹配的所有三元组的资源

Week9 数据库连接

一、centralised systems 集中式系统

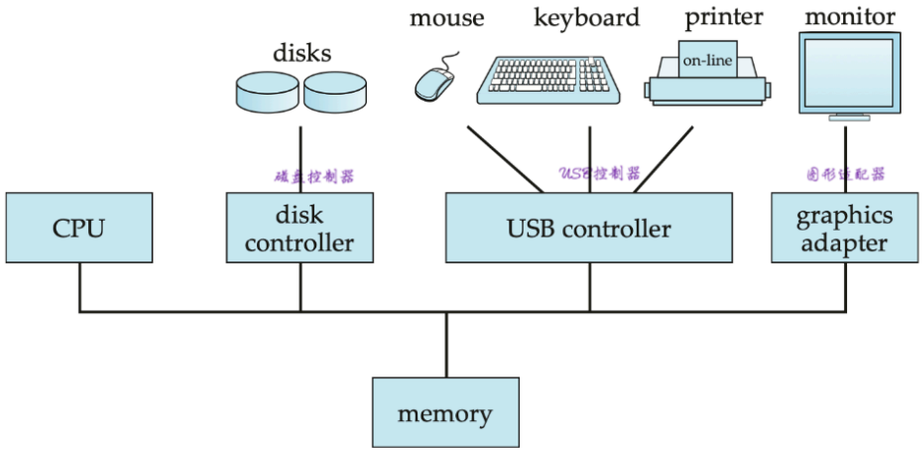
定义：在单个计算机系统上运行，不与其他计算机系统交互

分类：

通用计算机系统：一到几个CPU和若干设备控制器，它们通过一个公共总线连接起来，提供对共享内存的访问

单用户系统（如个人计算机或工作站）：桌面单元，单用户，通常只有一个CPU和一个或两个硬盘；操作系统只能支持一个用户

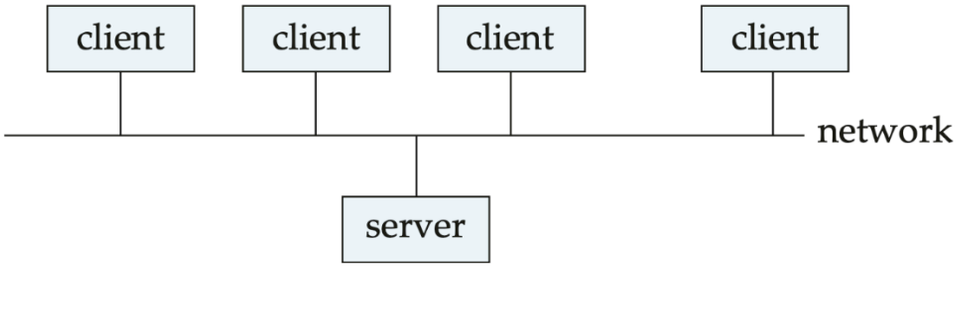
多用户系统：更多磁盘、更多内存、多个CPU和多用户操作系统。为连接到系统终端的大量用户提供服务。通常称为server系统。



二、client-server systems 客户-服务器体系结构

连接到集中式系统的终端被个人计算机所替代

集中式系统现在起到服务器系统的作用，它满足客户系统的需求

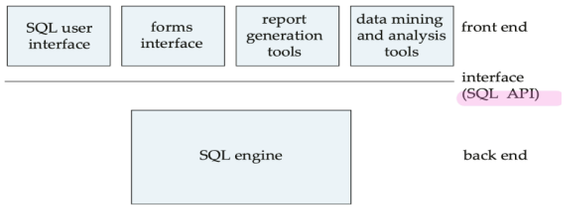


数据库提供的功能分为：前端和后端

后端：存取结构、查询计算和优化、并发控制以及故障恢复

前端：SQL用户界面、表格界面、报表生成工具以及数据挖掘和分析工具

前端和后端间的接口通过SQL或API 实现



用连接到后端服务器的工作站或**个人计算机网络取代**大型机的优势：

成本更低

资源定位和设施扩展的灵活性

更好的用户界面

维护更简单

三、server system architecture 服务器系统体系结构

大致可分为两类

***transaction servers* 事务服务器**：用于关系数据库系统

也称为 query / transaction server / SQL Server systems

客户端向服务器发送请求，服务器处理事务，结果返回客户端

- Requests are specified in SQL, and communicated to the server through a *remote procedure call* (RPC) mechanism.
- Transactional RPC allows many RPC calls to form a transaction.
- *Open Database Connectivity* (ODBC) is a C language application program interface standard from Microsoft for connecting to a server, sending SQL requests, and receiving results.
- JDBC standard is similar to ODBC, but for Java.

典型的包括多个在共享内存中**访问数据的进程**

server process 服务器进程：接受用户查询、执行并返回结果的进程

lock manager process 锁管理器进程：包括锁授予、释放和死锁检测

database writer process 数据库写进程：将修改过的缓冲块输出到磁盘

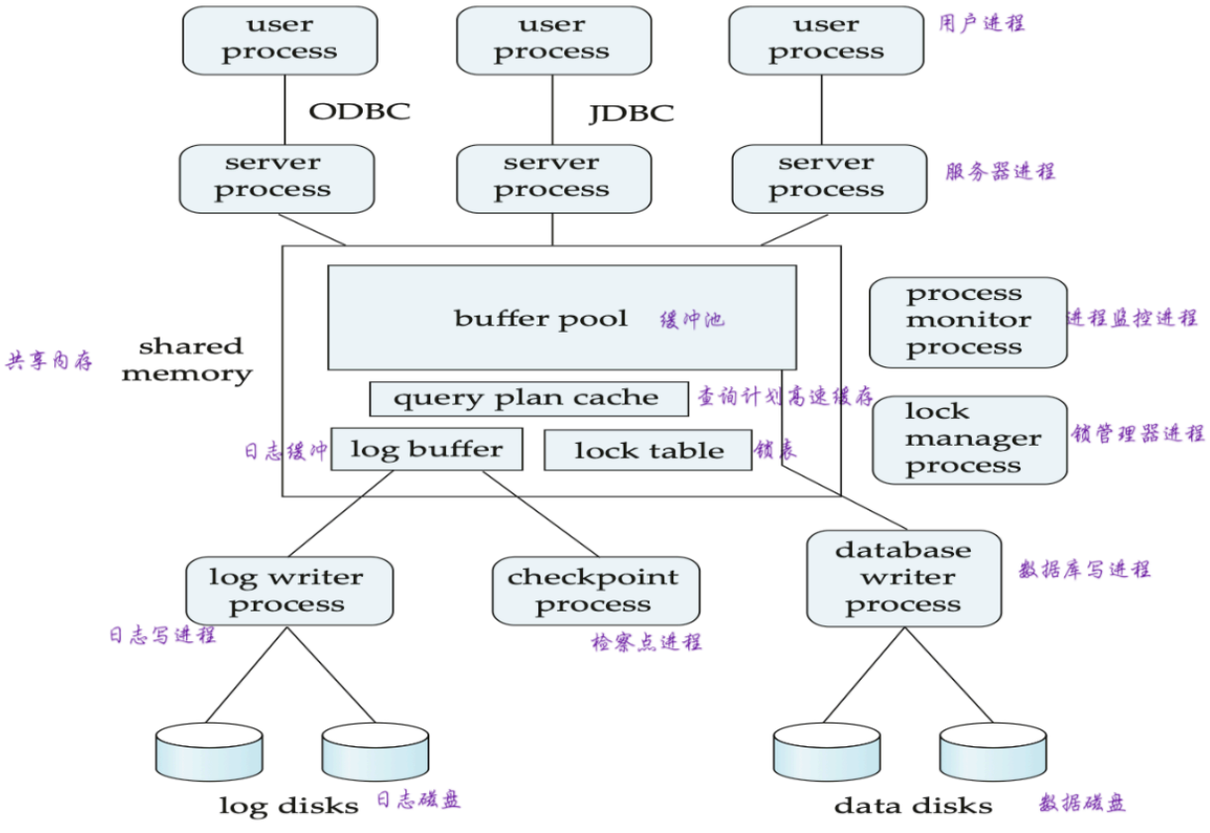
log writer process 日志写进程：将日志记录从缓冲区输出到稳定存储器

checkpoint process 检查点进程：定期执行检查点操作

process monitor process 进程监控进程：监控其他进程

共享内存包含所有的共享数据：如缓冲池、日志缓冲区、查询计划高速缓存

所有的数据库进程都能访问其中的数据



***mutual exclusion* 互斥**：确保没有两个进程同时访问或修改同一个数据结构

互斥机制 解决措施：

operating system semaphores 信号量：每个进程一个信号量，代表每个进程目前的状态

atomic instructions 原语：计算机硬件所支持，开销更小，如*test-and-set*

为了避免消息传递的开销，服务器进程**直接更新锁表**来实现封锁

而不需要向锁管理器发送锁请求

仍然需要锁管理器进程用于**死锁检测**

***data servers* 数据服务器**：使用*high-speed LANs*，用于面向对象数据库系统

客户机的处理能力与服务器相当

要执行的任务是计算密集型的

把数据传送到客户机，并在客户机上进行所有的处理，再把数据传回到服务机

这种体系结构要求客户端具有完整的后端功能

引发问题：

page-shipping vs item-shipping 页面传送和项传送

更小的传输项意味着更多的信息

页面传送：数据通信的单位是粗粒度的，例如页面。如果有多个项驻留在一页上，那么页面传送也可以看作是一种预读取

项传送：是细粒度的，例如元组

prefetching 预读取：有些项在请求之前就取过来

locking 锁：对于传送给客户机的数据项，通常由服务器给他们授予锁

锁降级：服务器可以要求客户端将预读取项上的锁传回服务器

该锁就可以分配给其他客户端。

由于消息延迟，从服务器请求和获取锁的开销很高

- Overhead of requesting and getting locks from server is high due to message delays.
- Can grant locks on requested and prefetched items; with page shipping, transaction is granted lock on whole page.
- Locks on a prefetched item can be called back by the server, and returned by client transaction if the prefetched item has not been used.
- Locks on the page can be **deescalated** to locks on items in the page when there are lock conflicts. Locks on unused items can then be returned to server.

data caching 数据高速缓冲存储：事务之间的数据可以高速缓存客户端

缓存一致性：在使用数据之前，检查数据是否是最新的

检查可以在请求锁定数据项时进行

需要与服务器交换一条消息以检查数据有效性，并得到该项的锁

lock caching 锁高速缓存：一定条件下，锁也可以高速缓存在客户机中

事务可以在本地获取缓存锁，而无需与服务器联系

当服务器接收到冲突的锁请求时，它会从客户端calls back锁

一旦没有本地事务在使用它，客户端将返回锁

四、从应用程序访问数据库

SQL命令可以从host languages（例如，C++或java）程序中调用

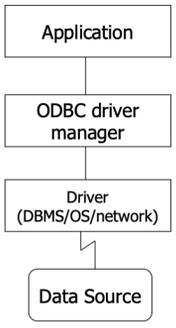
- SQL statements can refer to host variables (including special variables used to return status).
- Must include statement to *connect* to right database.

两种主要的集成方法

在host languages中嵌入SQL（例如，Pro*C、嵌入式SQL、SQLJ）

创建特殊的API来调用SQL命令（例如：JDBC、ODBC、PHP）

ODBC: Open Database Connectivity



用于访问数据库的API - application programming interface

通过在一个程序中使用ODBC语句，可以访问许多不同数据库中的文件

如access、dBase、DB2、Excel和Text

它允许程序使用SQL请求来访问数据库，而不必知道数据库的专有接口

ODBC处理SQL请求并将其转换为单个数据库系统能够理解的请求

使用它需要：一个ODBC软件 + 每个要访问的数据库的单独模块或驱动程序

JDBC: Java Database Connectivity

用于将用Java编写的程序连接到关系数据库中的数据 Java API

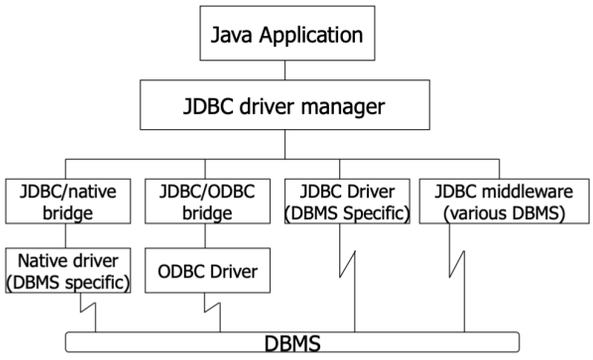
由一组用Java编程语言编写的类和接口组成

为工具/数据库开发人员提供一个标准API

并使使用纯 Java API 编写数据库应用程序成为可能

如何使用：

与数据库建立连接



发送SQL语句

处理结果

可以使用JDBC接口访问ODBC可访问的数据库

ODBC：在应用程序间使用

JDBC：被Java程序员用来连接到数据库，允许基于SQL的数据库访问

JDBC API

支持数据库访问的两层和三层模型

二层模型：a Java applet or application 直接与数据库交互

三层模型：引入一个用于执行业务逻辑的中级服务器

中间层：保持对数据访问控制

用户可以使用一个易于使用的 higher-level API，它由中间层转换成适当的低级调用

JDBC 使用步骤

1. Importing Packages

```
// Program name: JDBCExample.java
// Purpose: Basic selection using prepared statement
//
//Import packages

import java.sql.*;  //JDBC packages
import java.math.*;
import java.io.*;
import oracle.jdbc.driver.*;
```

2. Registering JDBC Drivers

```
class JDBCExample {

public static void main (String args []) throws SQLException
  {

//Load Oracle driver
DriverManager.registerDriver (new
  oracle.jdbc.driver.OracleDriver());
```

3. Opening connection to a Database

```
//Prompt user for username and password

 String user;
 String password;

 user = readEntry("username: ");
 password = readEntry("password: ");

//Connect to the local database
 Connection conn = DriverManager.getConnection
  ("jdbc:oracle:thin:@aardvark:1526:teach",
      user,
      password);
```

4. *Creating a Statement Object*

```
// Query the hotels table for resort = 'palma nova'

PreparedStatement pstmt = conn.prepareStatement
   ("SELECT hotelname, rating FROM hotels WHERE
   trim(resort) = ?");
pstmt.setString(1, "palma nova");
```

5. *Executing a Query, Returning a Result Set Object*
6. *Processing the Result Set*

```
ResultSet rset = pstmt.executeQuery();

//Print query results
while (rset.next ())
    System.out.println (rset.getString(1)+" "+
                rset.getString(2));
```

7. *Closing the Result Set and Statement Objects*
8. *Closing the Connection*

```
// close the result set, statement, and the connection
    rset.close();
    pstmt.close();
    conn.close();
  }
```

五、映射数据类型

有一些指定给SQL的数据类型需要映射到Java数据类型

如果用户希望Java能够处理它们

三种转换方式：

直接等价，如 SQL INTEGER 直接等价于 Java int data type

转换等价：如 SQL CHAR, VARCHAR, and LONGVARCHAR 可以转换为 the Java String data type

不可等价：SQL数据类型是唯一的，需要为它们的SQL等价物专门创建一个特殊的Java数据类对象

SQL DATE 转换为 Java Date object 被 java.Date 定义

WEEK 10 基于对象的数据库

传统的关系模型：只支持有限的类型系统

对象 - 关系数据模型

概念：关系模型的扩展，包含了复杂数据类型，非原子值和面向对象

优势：试图在扩展建模能力的同时，保持关系基础，特别是对数据的声明式访问，同时扩展建模能力；对现存关系语言向上兼容（较新的在较老的上执行）

SQL: 1999 为 SQL 增加了一个扩展类型系统

允许 collection（如嵌套关系）和大型对象类型

允许结构类型（嵌套记录结构，如复合属性）和继承

允许面向对象，包括对象标识符和引用

目前尚未在任何数据库系统中完全实现

结构类型 *Structured types / user-defined types*

允许直接表示 E-R 设计中的复合属性

如 一个由成分属性firstname 和 lastname 构成的复合属性 **name**

final 和 not final表明

是否可以创建 subtypes

```
create type Name as
    (firstname    varchar(20),
    lastname      varchar(20))
final
```

创建一个包含复合属性 **name** 的 **person** 表

可以引用 name.firstname 返回

```
create table person (
    name        Name,
    address     Address,
    dateOfBirth date )
```

创建一个 **Person** 表

表的行是用户定义类型 **PersonType**

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
create table person of PersonType;
```

另一种替代方法是 unnamed row types

和前面的表等价

```
create table person_r(
    name        row(firstname varchar(20),
                    lastname varchar(20)),
    address     row(street    varchar(20),
                    city      varchar(20),
                    zipcode   varchar(20)),
    dateOfBirth date)
```

如何访问一个复合属性的成分属性

component attributes of a composite attribute

```
select name.lastname, address.city
from person;
```

方法声明可以作为结构类型定义的一部分

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
method ageOnDate(onDate date)
    returns interval year;
```

单独创建方法的主体

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

指出这个方法是对哪个类型定义的

可以用 **ageOnDate()** 方法查询每个人的年龄

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

构造器函数被用于创建结构类型的值

如为 **Name** 类型声明一个构造器

```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

使用

new Name('John', 'Smith')

创建该类型的值

也可以创造一系列新元组，**Name** 上也有一个构造器

```
insert into Person values
    (new Name('John', 'Smith'),
    new Address('20 Main St', 'New York', '11001'),
    date '1960-8-22');
```

类型继承

假定一个 **person** 类型定义

```
create type Person
    (name varchar(20),
    address varchar(20))
```

为那些是 **student** 和 **teacher** 的 **person** 存储额外信息

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

create type Teacher
    under Person
    (salary integer,
     department varchar(20));
```

子类型可以重新定义方法，通过在方法声明中 *overriding method*

多重继承：一个类型可以被声明为多个类型的子类型

定义一个类型 **Teaching_Assistant**

助教既可以是 **student** 也可以 **teacher**

```
create type Teaching_Assistant
    under Student, Teacher
```

为了避免同时在 **student** 和 **teacher** 出现的 **department** 冲突

用 **as** 子句重新给它命名

```
create type TeachingAssistant
    under Student with (department as student_dept),
        Teacher with (department as teacher_dept);
```

每个值必须有一个 ***a most-specific type***：一般为子类型

如 **student** 和 **person** 中的 **student**

表继承

从子类型创建的表可以进一步指定为子表

定义一个 **people** 表

表 **students** 和 **teachers** 作为它的子表

```
create table people of Person;
create table students of Student under people;
create table teachers of Teacher under people;
```

添加到子表的元组对超级表上的查询自动可见

对 **people** 的查询也查询 **students** 和 **teachers**

对 **people** 的更新/删除也 **students** 和 **teachers**

使用 **only people** 只针对 **people**

概念上多重继承可用于表

但现阶段还不支持

```
create table teaching_assistants
of TeachingAssistant
    under students, teachers;
```

所以我们不能创建一个人即是 **students** 也是 **teachers**

超表和子表的一致性要求

correspond: 子表和父表中的元组在所有的继承属性上具有相同的值

1. 父表的每个元组（如**people**）最多可以对应每个子表（如**students** 和 **teachers**）中的一个元组（避免了**students**中两个元组同时对应**people**）
2. 所有互相对应的元组必须由同一个元组派生出来（避免了**people**中一个元组同时对应**students** 和 **teachers**）
3. 每个实体必须有一个最具体的类型

SQL中的数组和多重集合类型

集合：只允许每个元素最多出现一次

多重集合：一个无序集合，一个元素可能出现多次

数组：中的元素是**有序**排列的

```
create type Publisher as 定义了名为publishers的类型
  (name        varchar(20), 它包括两个成分
  branch       varchar(20));
create type Book as 定义了名为Book的结构类型
 (title          varchar(20),一个最多可以存储10个作者姓名的数组
  author_array   varchar(20) array [10],
  pub_date       date,
  publisher      Publisher,
  keyword-set    varchar(20) multiset);
create table books of Book; 创建一个包含类型为Book的元组的books表
```

存储作者名字使用数组是因为，作者名字的排序是有意义的

创建集合体值

创建一个数组的值　　**array** ['Silberschatz',` Korth',` Sudarshan']

创建多重集合　　**multiset** ['computer', 'database', 'SQL']

创建 **books** 关系中定义的元组　　('Compilers', **array**['Smith', 'Jones'], **new** *Publisher*('McGraw-Hill', 'New York'), **multiset**['parsing', 'analysis'])

别忘记声明 **Publisher** 的**函数构造器**

插入进关系

```
insert into books
values ('Compilers', array['Smith', 'Jones'],
                     new Publisher('McGraw-Hill', 'New York'),
                     multiset['parsing', 'analysis']);
```

用索引访问和更新数组，如 **author_array[1]**

查询集合体值

查找所有以 **database** 为关键字的书

```
select title
from books
where 'database' in (unnest(keyword-set ))
```

如果我们知道一本书有三位作者，我们就可以写

```
select author_array[1], author_array[2], author_array[3]
from books
where title = `Database System Concepts'
```

如果我们用“**title, author_name**“的对 对应每本书和书的每个作者

```
select B.title, A.author
from books as B, unnest (B.author_array) as A (author )
```

为了防止对数组解除嵌套而导致的元素顺序信息丢失

使用“**with ordinality as**“生成额外的属性记录数组中元素的位置

```
select title, A.author, A.position
from books as B,
        unnest(B.author_array) with ordinality as A(author, position);
```

解除嵌套

定义：将一个嵌套关**系转换成更少或没有**以关系为值的属性的形式的过程

在 **books** 关系中，**author_array** 和 **keyword_set** 两个属性是集合体

title 和 **publisher** 两个属性不是

想把该关系转化为一个单一的平面关系，不包含嵌套和结构类型作为属性

使用以下查询

```
select B.title, A.author, B.publisher.name as pub_name,
        B.publisher.branch as pub_branch, K.keyword
from books as B, unnest(B.author_array ) as A (author ),
        unnest (B.keyword_set ) as K (keyword )
```

from 子句中的变量 **B** 被声明为以 **books** 为取值范围

变量 **A** 被声明为以 **B** 的 **author_array** 为取值范围

变量 **K** 被声明为以 **B** 的 **keyword_set** 为取值范围

flat_books 如下，满足 **1NF**

title	*author*	*pub_name*	*pub_branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

嵌套

定义：将 **1NF**关系转化为嵌套关系

使用：**collect()**

在属性 **keyword** 上对关系 **flat_books** 进行嵌套

select *title*, *author*, *Publisher*(*pub_name*, *pub_branch*) **as** *publisher*,
  collect(*keyword*) **as** *keyword_set*
from *flat_books*
group by *title*, *author*, *publisher*;

执行结果

title	*author*	*publisher*	*keyword_set*
		(*pub_name*,*pub_branch*)	
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

同时嵌套 **keyword** 和 **author**

select *title*, **collect**(*author*) **as** *author_set*,
  Publisher(*pub_name*, *pub_branch*) **as** *publisher*,
    collect(*keyword*) **as** *keyword_set*
from *flat_books*
group by *title*, *publisher*;

另一种创建嵌套关系的方法：

在 **select** 子句中使用子查询，优势是在子查询中可以使用 **order by**

从而以创建数组所需的顺序来产生结果

```
select B.title,
      array (select author
             from authors as A
             where A.title = B.title
             order by A.position) as
          author_array,
      B.Publisher (pub-name, pub-branch) as publisher,
      multiset (select keyword
                from keywords as K
                where K.title = B.title) as keyword_set
from books4 as B
```

SQL中对象标识和引用类型

定义一个 **Department** 类型的表 **departments**

它有一个 **name** 域和对 **Person** 类型引用的 **head** 域 - 引用被限制在**people**表

```
create type Department (
    name varchar(20),
    head ref(Person) scope people);

create table departments of Department;
```

也可以省略为

```
create table departments of Department
    (head with options scope people)
```

被引用的表必须有一个属性来存储元组的标识符

如 **people** 表通过一个 **ref is** 子句声明自引用属性**self-referential attribute**

```
create table people of Person
ref is person_id system generated;
```

初始化一个引用属性：

1. 创建带有空引用的元组
2. 单独设置其引用

```
insert into departments
    values ('CS', null);

update departments
    set head = (select p.person_id
                from people as p
                where name = 'John')
where name = 'CS';
```

除了系统生成标识符外，还有一种是允许**用户生成标识符**

对象标识符的类型必须被指定为 被引用表的类型定义的一部分

而且表定义中必须指定**引用是用户生成的**

```
create type Person
(name varchar(20)
 address varchar(20))
ref using varchar(20)
create table people of Person
ref is person_id user generated
```

当向表 **people** 插入元组时

必须提供**唯一**标识符值（**people** 表、父表、子表均不能有同样标识符的值）

```
insert into people (person_id, name, address ) values
('01284567', 'John', `23 Coyote Run')
```

这样我们在向 **people** 表插入元组时就可以使用标识符的值

而不用单独查询来获取标识符

```
insert into departments
values(`CS', `02184567')
```

第三种方法是使用**已有主键作为标识符**

这种方法必须指明

引用是派生出来的

还有一个自引用属性名

```
create type Person
    (name varchar (20) primary key,
     address varchar(20))
   ref from (name)
create table people of Person
   ref is person_id derived
```

这样在向插入表 **departments** 元组时，就可以使用

```
insert into departments
    values(`CS',`John')
```

SQL: 1999 使用 - > 符号对引用取内容

所以我们使用以下查询找出各部门符人的名字和地址

```
select head->name, head->address
from departments
```

Path expression 路径表达式：形如 “head–>name”

引用的优点：

由于 **name** 是 **people** 表中属性，引用可以隐藏连接操作

否则，**head** 字段就会被声明为 **people** 表的外码

要找出 **name** 和 **address** ，就要显式的连接 **people** 和 **departments**

所以引用大大简化了查询

O-R特性的实现

Object-Relational DS 是对已有关系数据库系统的扩展

因此它支持的复杂数据类型可以转化为关系数据库的较简单的类型系统

子表的储存（无需复制所有继承字段），二选一：

1. 每个表只存储 *primary key* 和局部定义属性（被定义在该表中的属性），继承属性可以通过基于主键的与其父表的连接得到
2. 每个表存储所有继承和局部定义属性。当插入一个元祖时，它仅仅存储在它所插入的那个表中，它的每个父表通过推断判断它的存在。因为不需要连接，所以可以更快的访问元组中的所有属性。

持久化编程语言

定义：一种程序设计语言，通过构造来处理持久性数据

为什么需要它：

1. 对数据库的访问只是任何实际应用程序的一个组成部分
2. 虽然 *DML*，如*SQL* 是有用的
3. 但它是实现其他组件的必需，如用户界面或与其他计算机的通信

将数据库语言与编程语言连接的传统方法是：在编程语言中嵌入*SQL*

	persist programming languages	embedded SQL languages
翻译	持久化编程语言	嵌入型SQL语言
宿主语言和DML类型系统	相同	不一致
更新数据	程序员不必关心	需要关心，负责转换
显式代码	无需，可直接操纵持久性数据	需要，以从数据库获取数据
将数据放入内存并存回磁盘	不需要	需要

生成持久对象的方法：

按类持久：在类声明处显式声明

按创建持久：特殊语法，用来创建持久对象

按标志持久：在创建后，让对象持久

按可达性持久：如果对象被显式声明、或由持久对象可达，则持久

对象标识和指针

对象标识的持久程度：

过程内部：在单个过程的执行期间持久

程序内部：在单个程序的执行期间持久

程序之间：跨越程序之间，但如果磁盘上的数据存储格式变化则失效

持久的：程序之间 + 数据存储重组，依然保持

已经实现的持久版本：

C+ (ODMG C++, ObjectStore)

Java (Java Database Objects JDO)

对象-关系映射

定义：系统使用映射，从关系中获取相关数据并构造对象

建立在传统关系数据库之上

实现者提供了从对象到关系的映射（对象是暂时的，没有永久的对象标识）

对象从数据库中被更新，更新后对象通过*update/insert/delete*语句存回数据库

Hibernate ORM system 被用于：

Java 持久化 *API* 的实现

提供用于启动/结束事务、获取对象等的 *API*

提供直接在对象模型上操作的查询语言，转换为*SQL*的查询

局限性：开销，大批量更新

集中数据库类型的比较

	Relation system	Persistent-programming-language-based OODBs	Object-relational systems	Object-relational mapping systems
翻译	关系系统	基于持久化程序语言的面向对象的数据库系统	对象关系系统	对象关系映射系统
数据类型	简单	复杂	复杂	复杂
查询语言	强大	与程序语言集成	强大	与程序语言集成
高度保护	是		是	
高性能		是		
备注				关系数据库上构建

尽管现实世界这些界限可能很模糊

WEEK 11 分布式数据库

1. 分布式数据库系统

特征：loosely coupled sites，不共享物理组件；每个站点数据库彼此独立，事务可以访问一个或多个站点上的数据。数据存储在多个站点上，每个站点由一个可以独立运行的DBMS管理

数据特征：位置很重要（影响查询优化、并发控制和恢复），受一些因素制约

独立性：用户不必知道数据的位置（现实不可能）

原子性：用户应该能够编写访问和更新多个站点数据的事务；提交保留，中止回滚

2. 分布式数据库的类型

Homogeneous 同构数据库

所有站点都使用相同的DBMS软件，它们共同合作处理用户请求

Heterogeneous 异构数据库

不同站点运行不同DBMS，通过连接支持访问数据，站点之间可能并不了解

Gate protocols：向外部应用程序公开DBMS功能的API

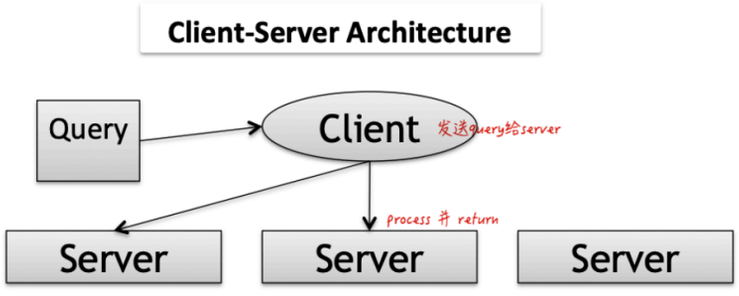
ODBC & JDBC

3. 分布式数据库的结构

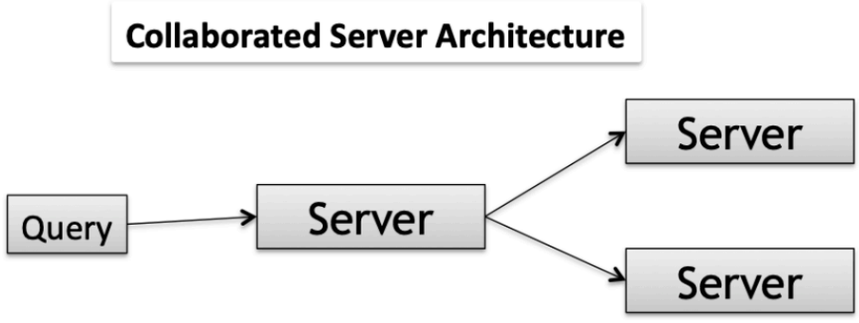
Client Server：具有一个或多个客户机进程和一个或多个服务器进程的系统

客户机将查询发送到服务器，服务器处理该查询，并将结果返回给客户机

所有查询都在服务器上执行



Collaborating Server：能够对本地数据运行查询并跨多个服务器执行事务

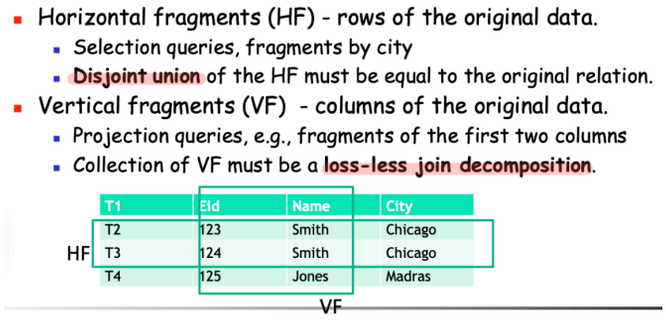


Middleware：一个数据库服务器可以跨多个服务器管理查询和事务

对来自其他服务器的数据执行关系操作但不维护任何数据的层

4. 分布式数据存储

存储关系r的两种方法



Fragmentation 分片：把关系分为较小关系，每个片存储在不同的站点上

优势：降低存储在不同站点上数据的传递成本

Horizontal fragments：选择查询，*disjoint union of the HF* 必须等于原关系

Vertical fragments：投影查询，*VF* 的合并必须 *loss-less join decomposition.*

Replication 复制：存储一个关系或分片的多个副本，整个关系存在一个或多个站点

优势：提高可用性（复制站点出故障用另一个），更快评估查询（使用关系的本地副本而不是遥远站点），两种类型

4. 同步复制和异步复制

用户应该能够更新数据而不必担心关系存储在哪里

名称	同步复制	异步复制
英文	*Synchronous replication*	*Asynchronous replication*

定义	在主数据中心中创建或更新新数据的同时，复制到辅助远程位置	数据只在预定的时间段内复制（可以是每小时、每天或每周）
优点	保证零数据丢失	提高网络性能和可用性
缺点	总体性能降低，成本高	关键数据丢失
应用	即时灾难恢复和完全不需要数据丢失的项目	存储不太敏感的数据和能够容忍部分数据丢失的项目 广泛应用于商业分布式数据库管理系统 用户必须了解分布式数据库
事务	事务必须在副本全部更新后提交	允许先提交事务，再更新修改后的关系的副本
距离	距离较近	较远（只要网络连接可用）
成本	贵	划算
恢复点目标 RPO	0	15分钟到几个小时
性能	较低	较高

二者融合：一个关系划分为几个片，每个片可以有几个副本

同步复制技术

	Voting technique	*Read-any, write-all technique*
步骤	事务必须写入大部分副本才能修改对象； 读取至少足够的副本以确保其中一个副本是最新的（最高的版本号）	一个事务只能读一个副本 但可以写所有副本 写较慢，读很快
缺点	效率低：因为读一个对象要读多个副本，读取次数>更新次数	
适用情况		读 > 写
成本		在更新的事务提交前，必须锁定所有副本； 事务将锁请求发送到远程站点并等待授予锁，在很长一段时间内，它继续持有所有锁； 如果存在站点或通信故障，则在恢复所有站点之前，事务无法提交

异步复制技术

同步复制成本高所以异步复制受欢迎，尽管不同的副本可能具有不同的值

用户必须知道他们正在阅读的是哪个副本，并且该副本可能会在短时间内不同步

区别是有多少个副本是 *updatable* 或 *master copies*

Peer-to-Peer Asynchronous Replication

可以将多个副本指定为可更新

对主副本的更改必须以某种方式传播到其他副本

Conflict resolution 用于处理不同站点的更改

每个*master*只允许更新关系的一个片段，任何两个由不同*master*更新的片段不相交

更新权限一次由一个*master*持有

Primary Site Replication

关系的一个副本是 *master copy*

关系的一个副本是主副本

- **Primary site** – one copy of a relation is the master copy

整个关系的副本在其他站点创建。它们无法更新

- **Secondary site**- replicas of the entire relation are created at other sites. They cannot be updated.
- Users register/publish a relation at the primary site and subscribe to a fragment of the relation at the secondary site.
- Changes to the primary copy transmitted to the secondary copies are done in two steps.
  - First capture changes made by committed transactions, then apply these changes. 应用这些更改

捕获被提交事务所做更改

捕获

基于日志的捕获：通过更好

- **Log Based Capture** - the log maintained for recovery is used to generate a Change Data Table (CDT)

由DBMS调用的一种过程

- **Procedural Capture** – A procedure that is invoked by the DBMS which takes a snapshot of the primary copy
- Log based capture is generally better because it deals with changes to the data and not the entire database. However it relies on log details which may be system specific.

应用

定期获取

The Apply process at the secondary site periodically obtains a snapshot of the primary copy or changes to the CDT table from the primary site, and updates the copy.

- Period can be timer or user's application program based.

副本可以是修改后的关系的视图

- Log-Based Capture plus continuous Apply minimises delay in propagating changes.
- Procedural Capture plus application-driven Apply is the most flexible way to process updates.
  - Used in data warehousing applications

5. 数据仓库（**异步复制**实例：副本很少更新）

从许多站点创建巨型数据仓库

在某些位置创建并使用所有数据的副本，使用副本，而不是去个别来源

启用复杂的决策支持查询

副本很少更新

数据源被不同的DBMS控制

创建复制副本时需要清理数据并删除不匹配项

6. 分布式查询

假设一个数据库 Sailor (sid, sname, rating, age)

查找 age 的平均数

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3 AND S.rating < 7
```

Distributed Query Processing: Non-Join Distributed Queries

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3
  AND S.rating < 7
```

tid	sid	sname	rating	age	
T1			4		stored at Shanghai
T2			5		stored at Tokyo
T3			9		

水平分片：假设 rating<5 的在上海，>=5 的在东京，必须算两站点 sum() 和 count()

假设改为 WHERE S.rating>6 的在上海，只看东京站点

垂直分片：sid 和 rating 在上海，sname 和 age 在东京，tid 两站点都在 相连

由于关系被复制到多个站点，请根据本地成本选择站点

分布式数据库的连接

- ***Fetch as needed***

LONDON	PARIS
Sailors (S)	Reserves (R)
500 pages	1000 pages

假设关系 ***Sailors*** 存在伦敦，关系 ***Reserves*** 存在巴黎

使用 the block nested loops join， ***Sailors*** 为外关系

成本 = 500 D + 500 * 1000 (D + S)

D = the cost of read/write blocks, S = the cost to ship a block

若查询未在伦敦提交，则必须将运费结果添加到查询站点

也可以使用 index nested loops join in London

Distributed Queries

Sailors (Sid, name, rating, age, tid)

Sid	sname	Rating	age	Tid
22	Shanghai	4	45	
29	Tokyo	5	35	

Example query with a relation S (fragmented at Shanghai and Tokyo sites):

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3 AND S.rating < 7
```

- Horizontally Fragmented
  - Tuples with rating < 5 at Shanghai, >= 5 at Tokyo. When calculating average, must compute sum and count at both sites
  - If WHERE contained just S.rating > 6, just one site.
- Vertically Fragmented
  - *sid* and *rating* at Shanghai, *sname* and *age* at Tokyo, *tid* at both.
  - Joining two fragments by a common *tid* and execute the query over this reconstructed relation
- Replicated
  - Since relation is copied to more than one site, choose a site based on local cost.

举个例子	▪ Joins of relations across different sites can be very expensive.
	▪ Fetch as needed 不同站点之间的关系连接可能非常昂贵
	  ▪ For example: Sailors relation is stored at London and Reserves relation is stored in Paris. There are 500 blocks of Sailors and 1,000 blocks of Reserves
	  ▪ Use the block nested loops join in London with Sailors as the outer join, and for each Sailors block, fetch all Reserves blocks from Paris.
	  ▪ The cost is: 500 D + 500 * 1000 (D + S)
	  ▪ 500D = the time to scan Sailors
	  ▪ 500*1000 (D+S) = for each Sailor's block the cost of scanning and shipping all of Reserves
	  ▪ D is the cost of read/write blocks
	  ▪ S is the cost to ship a block
	  ▪ If query was not submitted at London, must add cost of shipping result to query site.
	  ▪ Can also do index nested loops join in London, fetching matching Reserves tuples for each Sailors tuple as needed.

另一个例子	SELECT *
	FROM Sailors S, Reserves R
	WHERE S.sid = Reserves.sid
	▪ Fetch as Needed, block nested loop join, Sailors as outer:
	  ▪ Cost: 500 D + 500 * 1000 (D+S)
	  ▪ D is cost to read/write page; S is cost to ship page.
	  ▪ S is very large, so cost is essentially 500,000S
	▪ Ship to One Site – merge join, Ship Reserves to London.
	  ▪ Assume only one relation can be read in memory (sorting can be done in memory), cost = 3*(500+1000)
	  ▪ Cost: 1000S + 4500D, essentially 1000S as D<<S.

· ***Semi-Join***

假设把 ***Reserves*** 传送到伦敦，减少传送的元组数：

步骤一：在伦敦计算 ***Sailors*** 在连接上的投影，并把它传送到巴黎

步骤二：在巴黎将 ***Sailors*** 投影与 ***Reserves*** 相连，把结果传送回巴黎

结果叫做 *reduction of Reserves with respect to Sailors*

步骤三：在伦敦计算传送回的值

权衡：计算成本 和 对于运送整个**Reserves**关系的运输投影

如果对 ***Sailors*** 有选择更有用

Semi Join

- Three steps in reducing the number of Reserves tuples to be shipped.
  - At London, compute projection of Sailors onto the join attribute and ship this projection to Paris. (sids)
  - At Paris, join Sailors projection with Reserves. Ship the join result to London. This result is called 'reduction of Reserves with respect to Sailors'. 关于
  - At London, compute the join of the reduction of Reserves with sailors.
- Idea: tradeoff the cost of computing and shipping projection for cost of shipping full Reserves relation. 计算成本
  - Especially useful if there is a selection on Sailors, and answer desired at London.

· ***Bloom Join***

和半连接很像，只不过第一步不是投影而是 ***bit-vector***

Bloom Join

- Bloom Join is similar to Semi Join but there is a bit-vector shipped in the first step instead of a projection.
- At London, compute a bit-vector of some size k:
  - Hash each tuple of Sailors (using sid) into range 0 to k-1.
  - If some tuple hashes to i, set bit i to 1 ($0 \le i \le k-1$).
  - Ship bit-vector to Paris.
- At Paris, hash each tuple of Reserves (using sid) similarly, and discard tuples that are hashed to 0 in Sailors bit-vector (no Sailors tuples hash to the *i*th partition).
  - Result is called 'reduction of Reserves with respect to Sailors'.
- Ship bit-vector reduced Reserves to London.
- At London, join Sailors with reduced Reserves.
- Bit-vector cheaper to ship, almost always efficient.

查询优化：

· 考虑所有的计划，选择最便宜的；如有多个副本，选择成本最少的

· 如果各个站点运行不同的DBMS，全局查询规划必须保留每个本地站点的自治性

· 使用新的分布式连接方法

· 查询站点构建一个全局计划，也附加描述每个站点的处理的本地计划

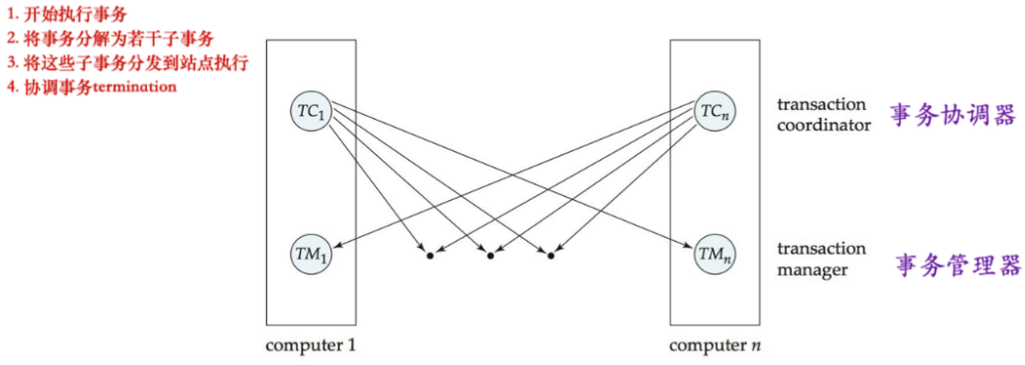
7. 分布式事务

事务在一个站点提交，但可以在其他站点访问数据

每个站点包含两个子系统

transaction coordinator：协调在该站点上发起的各个事务

local transaction manager：管理访问存储在一个局部站点中的数据的事务，以确保 ACID（如负责并发控制方案和维护一个用于恢复目的的日志）



并发控制和恢复

假设使用带死锁检测的严格2PL，即读取对象前先请求对该对象的共享锁，当事务完成时，才能释放所有排他锁

分布式封锁协议

并发控制协议决定何时获得和释放锁

锁管理可以分布在多个站点上

单一锁管理器：系统维护位于单个选定站点，所有封锁和解锁请求都在该站点处理，优点（实现简单、死锁处理简单），缺点（瓶颈，脆弱性）

主副本：每个对象只有一个副本被指定为主副本，锁定/解锁请求由主站点的锁管理器处理，而不管副本存储在何处

分布式锁管理器：副本请求由该站点的锁管理器处理，优点（实现简单、减轻瓶颈、低开销），缺点（死锁处理更复杂）

分布式死锁预防

每个站点都维护局部等待图，图中的循环表示死锁

即使局部图不包含循环，也可能存在全局死锁

当站点*S1*的事务*Ti*需要站点*S2*的资源时，发送请求，如果被*Tj*占用，插入*Ti->Tj*到*S2*

死锁检测算法

集中式死锁检测：将所有局部图发送到负责死锁检测的一个站点，并一块

层级式死锁检测：将站点组织成一个层次结构，并将局部图发送给结构中的父级

请求超时：如果事务等待时间太长，就终止

分布式恢复和并发控制

分布式更复杂，例如通信连接的故障；正在执行子事务的远程站点的故障

所有子事务必须提交或根本不提交

需要提交协议 - 最常见的是 *Two-Phase Commit*

在每个站点维护一个日志，提交协议操作也会被额外记录

Commit Protocols	*Two Phase Commit (2PC)*
Coordinator	发起事务的站点
Subordinates	执行子事务的其他站点
启动	当用户决定提交事务时，*commit*命令被发送到事务的*Coordinator* 这将启动*2PC*：

两阶段	▪ Phase 1: Coordinator sends a prepare message to each subordinate; When subordinate receives a prepare message, it decides to abort or commit its sub transaction; Subordinate forces writes an abort or ready log record and sends a no or yes message to coordinator accordingly. ▪ Phase 2: If coordinator receives a yes message from all subordinates, force-writes a commit log record and sends commit message to all subordinates. Else force-writes a an abort log record and sends an abort message; Subordinates force-write abort or commit log record based on the message they receive. ▪ In some implementations, after the two phases, subordinates send acknowledgement message to coordinator; After coordinator receives ack messages from all subordinates it writes an end log for the transaction.

故障后的恢复

会有一个恢复进程读取日志，并处理崩溃时执行提交协议的所有事务

参与站点的故障

- Examine its own logs, if there is commit or abort log record for transaction T, then redo/undo T respectively.
- The log contains a <ready T> record, repeatedly contact the coordinator or other active sites to find the status of T, then performs redo/undo accordingly and write commit/ abort log records depending on coordinator's response;
- The log contains no control records (abort, commit, ready) concerning T, undo.

协调器的故障

- participating sites must decide the fate of T.
- If an active site contains a <commit T>/<abort T> record in its log, then T must be committed/aborted.
- If some active site does not contain a <ready T> , preferable to abort T.
- If active sites have a <ready T> record in their logs, but no additional control records (such as <abort T> or <commit T>), it is impossible to determine if a decision has been made, and what that decision is, until the coordinator recovers. (***in-doubt transaction***)

Blocking problem

背景：如果使用了锁定，可疑事务T可能会在活动站点上持有数据锁。这种情况是不可取的，因为协调器可能需要数小时或数天才能再次激活。这种情况称为阻塞问题，因为T被阻塞，等待协调器站点的恢复

解决措施：

- Use **<ready T, L>** log record, where **L** is a list of all write locks held by the transaction T when the log record is written. At recovery time, after performing local recovery actions, for every in-doubt transaction T, all the write locks noted in the <ready T, L> log record (read from the log) are reacquired.
- Can also be solved using the 3-phase commit protocol (under certain situations)

从网络分区恢复

如果协调器和它的参与者在一个分区：失败对提交协议没有影响

如果属于多个分区：从其中一个分区中的站点看，其他分区中的站点似乎出现了故障

不在包含协调器的分区中的站点只需执行协议来处理协调器的故障

协调器和站点在同一分区：协调器遵循提交协议，假设其他分区中的站点出现故障

WEEK 12 大数据和NoSQL

1. 大数据

特征

Volume	海量数据，terabytes, petabytes, exabytes
Velocity	数据进出速度，实时、流播
Variety	数据来源和类型多样，非关系数据（如嵌套关系、文档、XML数据、web数据、图形、多媒体、灵活模式或无模式）
Veracity	信息的正确性和准确性，数据质量和可靠性
Value	使用机器学习、数据挖掘、统计、可视化、决策分析技术，提供知识和业务价值

传统的RDBMS是不够的，因为

数据冗余不一定会导致更新异常（因为我们不会在事务完成后更改它们）

相反，冗余可以显著提高性能，避免连接

RDBMS不能有效地处理多值属性和复合属性

连接操作非常昂贵

使用2PL确保ACID很昂贵，且不改现有或最多只添加新数据不需要ACID

2. Not-only SQL

概念：泛指非关系数据库，分布式，不提供ACID的数据库设计模式，关系的补充

目的：与关系数据结构相比，存储更简单或更类似于面向对象编程语言的数据结构

优点：

灵活模式或无模式	避免不必要的复杂性，减少成本
巨大的可扩展性	因为数据之间无需关系，软化ACID，允许水平扩展
高吞吐量	高读写性能
宽松的一致性	提高了性能和可用性
安装	更快，更便宜

不同于ACID的四大特性 **BASE**（主要集中在系统可用性上，以放松一致性为代价）

Basically Available	基本可用性	允许某些部分出现故障，其余部分依然可用
Soft state	软状态	软化了关系数据库中的ACID的属性，以允许水平扩展

Eventually consistent	最终一致性	接受客户端可能读取不一致（即过时）数据的时间段，保证结束

最终一致性：

水平分布系统的理想性质：一致性（总的顺序）和可用性，无法保证分区

一种一致性模型，为分布式计算实现高可用性

分为三种情况：

强一致性：更新完成后，任何后续访问都将返回更新后的值

弱一致性：在返回更新的值之前，需要满足许多条件

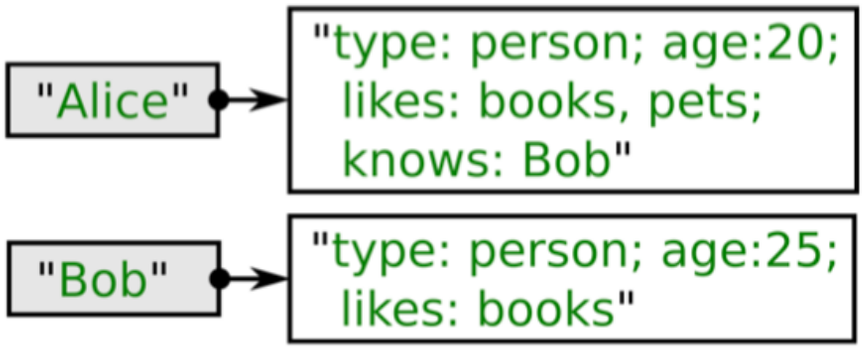
最终一致性：如果没有对给定的数据项进行新的更新，那么最终对该项的所有访问都将返回上次更新的值，以实现高可用性

3. 根据数据存储的方式分类：

Key-value Stores	键值存储：通过键快速找到值，存储不管值的格式 优点：降低复杂性，更快速高效工作，高度可扩展 缺点：查询可能性的灵活性降低，有限的查询接口
Wide-Column Stores	列存储：按列存储 优点：方便存储结构化和半结构化数据、数据压缩、列查询
Document Stores	文档存储：可以对某些字段建立索引，实现关系数据库功能
Graph Database	图存储：图形关系的最佳存储

键值存储：大量小型记录

每个key是唯一的，value可以是复杂数据结构，但含义需要被程序员解释



记录可以在多台计算机上分区，以及被复制到多台机器，以保证故障恢复

查询从系统到适当机器

记录还可以跨多台计算机进行复制，以确保即使一台计算机发生故障也能保持可用性

需要确保更新应用于所有副本

键值对中的值的含义必须由程序员解释

以 array["key"] = value 或 hash table 的形式存于主内存，使用 a key value 去访问

没有**外键和连接**，数据可以**水平分区和分布**

存储系统通常无法解释存储值

JSON object:

```
{
  "ID": "22222",
  "name": {
      "firstname: "Albert",
      "lastname: "Einstein"
      },
  "deptname": "Physics",
  "children": [
      { "firstname": "Hans", "lastname":
"Einstein" },
      { "firstname": "Eduard", "lastname":
"Einstein" }
  ]
}
```

存储地址：

- Key-value stores may store
  - Un-interpreted bytes, with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - Wide-table (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- Some key-value stores support multiple versions of data, with timestamps/version numbers

典型的操作（无修改）：

INSERT *new Key-Value pairs (or* ***put****)*

LOOKUP *value for a specified key (or* ***get****)*

DELETE *key and the value associated with it*

有些系统还支持对键值的范围查询 *range queries*

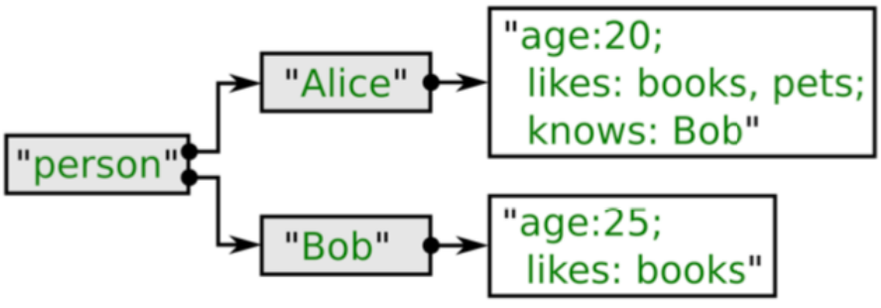
但是，键值存储不是完整的数据库系统，原因：

对事务性更新没有/有限的支持

应用程序必须自己管理查询处理

但这也使得构建可扩展的数据存储系统更加容易

列存储：数据以表的形式存储



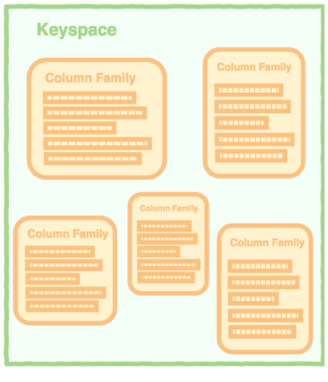
一个表有一个 ***row-key*** 和一组预定义 ***column-family columns***

表中每个 ***row*** 都由一个 ***row-key value*** 独立标示

表中每个 ***column family*** 都有一组大而灵活的 ***columns***（列数可能会随行改变）

每个列都有一个名称和一个或多个值

他们都被存储在 *keyspace* 中

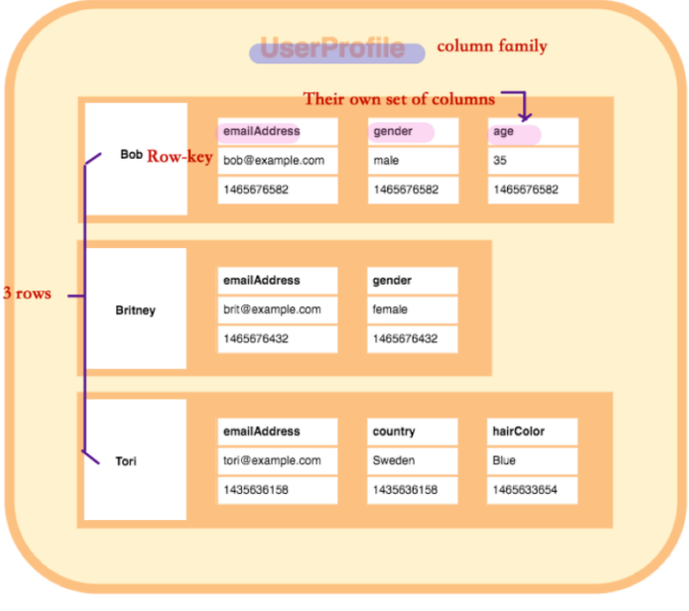


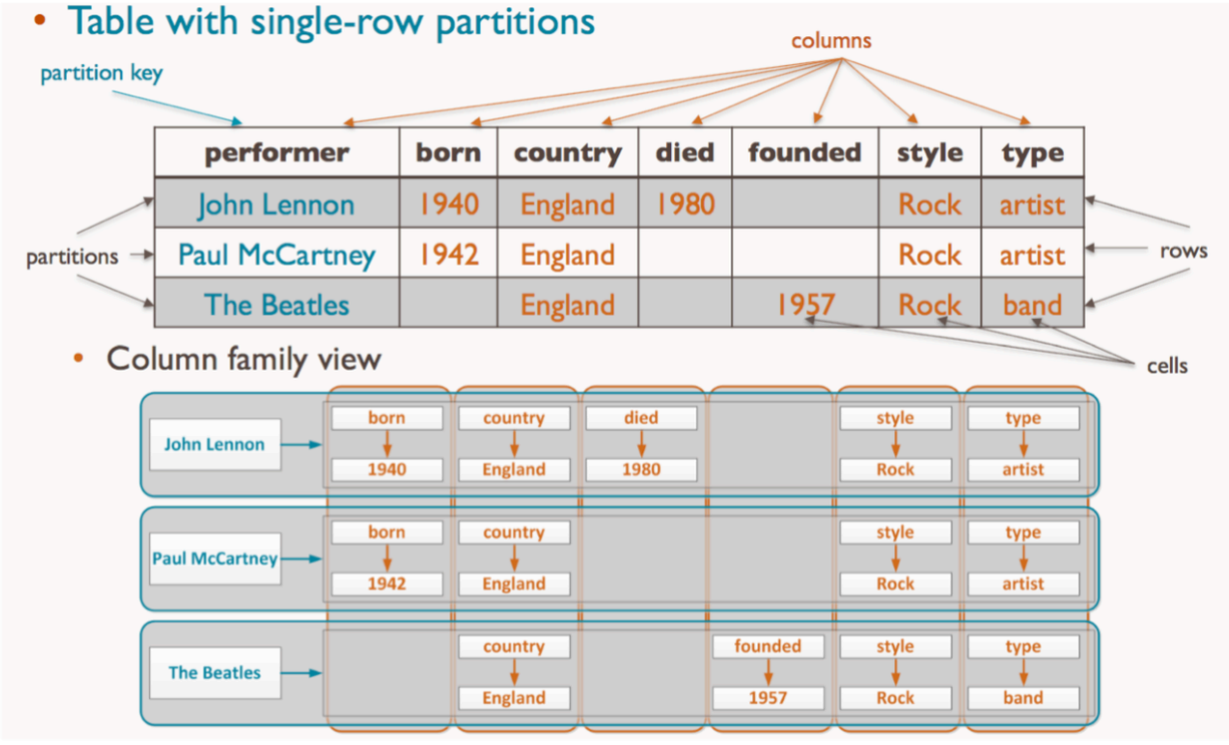
面向列的数据库系统存储数据

以 ***column family of data*** 存储

便于数据压缩*compression*

举个例子





例如：谷歌的*BigTable*

一种稀疏的、分布的、持久的多维排序映射

被应用于多个*Google*应用程序

映射由 *a row key, column key, and a timestamp* 索引，每个值都是未解释的字节数组

图存储：最适合表示具有大量互连的数据，尤其是它们和所表示的数据一样重要时

例如，社会关系或地理数据

允许对图结构进行查询

举例：

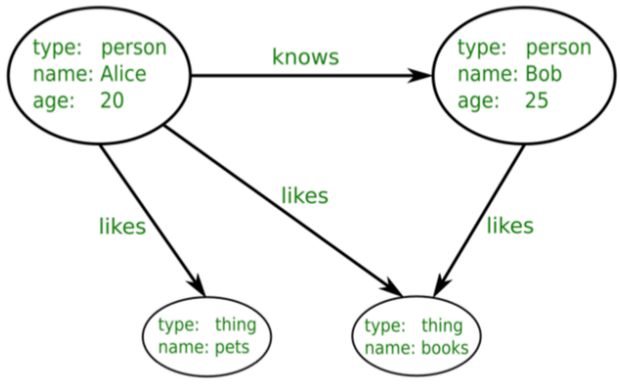
节点之间的关系或最短路径

*RDF*图和链接数据

谷歌知识图

边被用来存储关系信息

节点存储其他属性信息



文件存储：数据存储在所谓的文档中，即某种结构化数据格式的任意数据

模式语言不能很好的表达 *Object- Relationship-Attribute semantics in ER model.*

数据格式通常是固定的，但结构是灵活的

在基于JSON的文档存储中，具有完全不同属性集的文档可以存储在一起

JSON作为数据格式

因为系统理解格式

可以直接查询属性

如 *name* 或 *age*

people

```
{
  name:  "Alice",
  age:   20,
  likes:  ["books", "pets"],
  knows: "Bob"
}
```

```
{
  name:  "Bob",
  age:   25,
  likes:  ["books"]
}
```

4. NoSQL 与 SQL 的对比

性质	SQL	NoSQL
Schema 架构	预定义，固定的 模式演化困难	不一定预定义，可选的 可以是半结构化或非结构化
数据类型	**Flat relation** 定长记录	**Tree/graph structured data.** 变长记录，多值属性 可随时添加新 tag
数据持久性	数据库存储在 **disk drive** 上 数据持久 与内存存储相比，访问速度非常慢	数据库存储在 **in-memory** 上 使用指针和哈希 访问速度快 需要convert data in memory from/to disk drive 使得数据持久性
OLTP or OLAP	**OLTP,** mission critical online transaction applications	**OLAP,** data warehouse and data analytics
数据库状态	仅保留当前 如果需要历史数据，则需要使用带有时间段的temporal database来存储	保留历史数据，时间/日期维度数据分析
访问数据的语言	标准DBMS**声明性**查询语言SQL 一次操作一组元组	不同的 **imperative** programming languages 编写程序
更新数据	频繁	主要是新数据 很少更新、删除和添加
查询优化	**Query optimiser** of RDBMS	**Programmer** 对每个程序做优化
DBMS	**RDBMS**	不是，仅仅是 **data stores**
查询答案	返回精确值	返回最佳猜测，类似于数据挖掘

ACID	是，一致性是OLTP应用程序最重要的问题	注重速度性能，使用最终一致 如果没有更新，则不需要ACID
连接操作	涉及很多连接，可能很慢	避免或没有，使用冗余数据加快处理速度
特殊用户查询	Write **SQL** programs or RDB keyword query search	需要 **Programmer** 编写程序
分布 & 并行处理	被限制	可以 数据可以被水平及垂直分区并分布到节点 然后并行处理分区数据 对此类应用有效

5. 大数据存储系统

1. 分布式文件系统：适用于数百万个大文件

将数据存储在大量计算机中，但提供单个文件系统视图

应用：大型数据密集型应用程序（高度可扩展性）

方式：在低成本、不可信赖的计算机上提供大量冗余存储

文件被复制以处理硬件故障

检测故障并从中恢复　　实现了GFS的基本思想

例子：Google File System (GFS)，Hadoop File System (HDFS)

Hadoop File System Architecture

分布式文件系统的物理结构由计算机集群中的多个节点构成

这些节点分为两类：

由一个 Namenode 和一定数量的 Datanodes 组成

	NameNode	DataNodes
名称	一个名称节点	若干个数据节点
作用	中心服务器 管理该系统命名空间和客户对文件的访问	一个节点运行一个数据节点进程 负责处理文件系统客户端读写请求 接受名称节点统一调度
存储数据类型	元数据	文件内容
保存地方	内存	磁盘

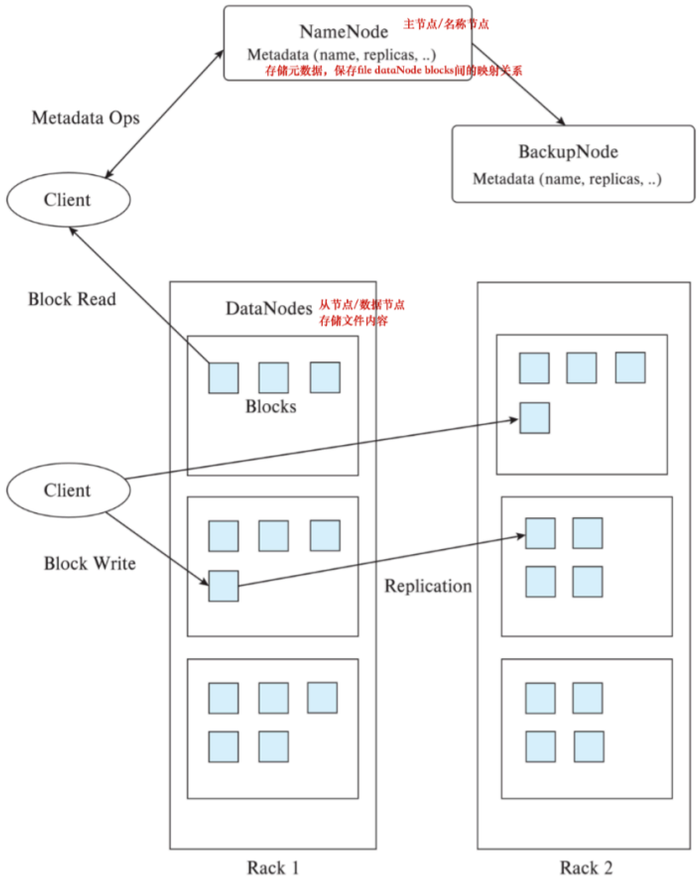
功能	保存文件、block、datanode之间的映射关系	维护了 block id 到 datanode 本地文件的映射关系
映射	将a filename 映射到 list of Block IDs 将每个 Block IDs 映射到包含块副本的DataNodes	将 Block IDs 映射到磁盘上的物理位置

整个群集有一个**Namespace**，由Namenode 负责管理，记录了文件各个块所在数据节点的位置和信息

Files 被分解为 blocks

每个 block 被复制进多个Datanodes

Client 从 NameNode 查找blocks 的位置，或直接从DataNode访问数据



优点：数据一致性

write-one-read-many 访问模型

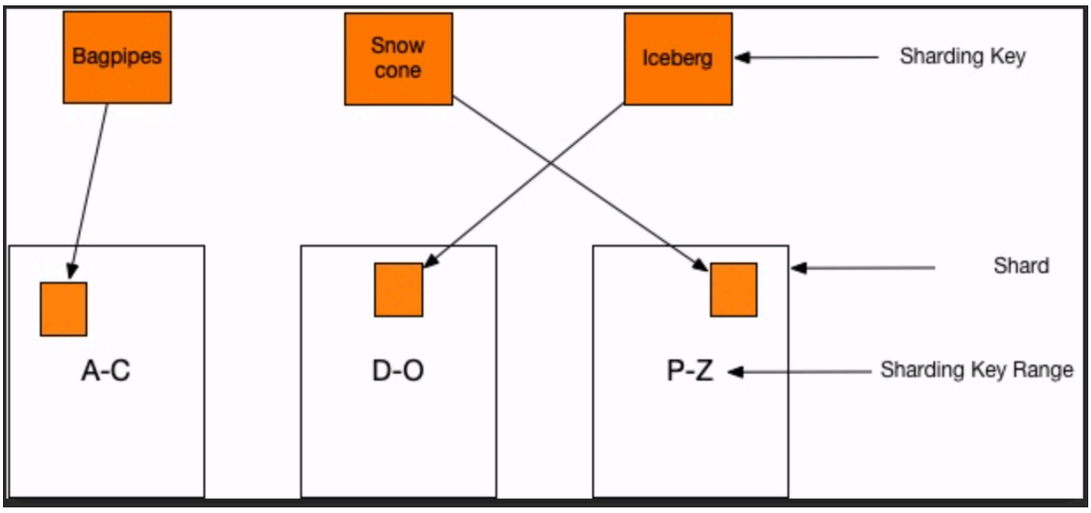
客户端只能附加到现有文件

缺点：高开销和性能差数十亿个较小元组

2. 跨多个数据库分片

Sharding：跨多个数据库分区数据

分区通常在 *partitioning attributes /partitioning keys / shard keys* 上进行，如 *user ID*



应用程序必须跟踪哪些记录在哪个数据库上，并向该数据库发送查询/更新

优势：扩展性好，易于实现

劣势：

不透明：应用程序必须处理查询的路径，查询跨越多个数据库

数据库过载：很难将其部分负载移出

数据库越多，失败的可能性就越大：需要保留副本以确保可用性，成本高

3. 键值存储系统：*NoSQL*

4. 并行分布式数据库：

支持可扩展的数据访问，四种方法：

方法一：在应用服务器上使用memcache或其他缓存机制，以减少数据库访问（可扩展性有限）
方法二：跨多个单独的数据库服务器分区（“*shard*”）数据
方法三：使用现有的并行数据库（并行数据库是为决策支持而不是OLTP而设计的）
方法四：大规模并行键值数据存储（分区、高可用性等对应用程序完全透明）

方法二四不支持很多跨分区的关系特性，如连接和完整性约束

应用：多台机器集群，较小规模（10-100），不能提供简单的可扩展性

复制：在机器发生故障时用于确保数据可用性，但通常在失败时重新启动查询

*Map reduce*系统 可以继续执行查询，解决故障

定义	可靠、可扩展的并行计算平台，是并行分布式数据库的抽象
功能	获取大数据，并使用并行分布式计算将大数据转换为小规模或常规大小的数据
程序员	提供核心逻辑（*map() reduce()* 函数）
系统	负责计算、协调等并行化
数据存储/访问	通常使用分布式文件系统或键值存储来完成

举例	大型文档集合中每个单词出现次数
	解决措施：
	在*workers*间分配文件
	每个*worker* 解析文件以查找所有单词，映射函数输出 *(word, count)* 对
	基于单词，在*workers*间将*(word, count)* 对分区
	对于每个*worker*的每个单词，<u>*reduce function locally add up counts*</u>
	
	map(String record): 输入记录字符串
	for each word in record 每个单词一条记录
	emit(word, 1);
	
	// First attribute of emit above is called **reduce key**
	// In effect, group by is performed on reduce key to create a
	// list of values (all 1's in above code). This requires **shuffle step** across machines.
	// The reduce function is called on list of values in each group
	
	reduce(String key, List value_list):
	String word = key
	int count = 0;
	for each value in value_list:
	count = count + value
	Output(word, count);

编程模型

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp.
- Input: a set of key/value pairs
- User supplies two functions:
  - **map**(k,v) → list(k1,v1)
  - **reduce**(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs
- For our example, assume that system
  - Breaks up files into lines, and
  - Calls map function with value of each line
    - Key is the line number

示例：日志处理

MapReduce Example: Log Processing

- Given log file in following format:

```
2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...
```

- Goal: find how many times each of the files in the *slide-dir* directory was accessed between 2013/01/01 and 2013/01/31.
- Options:
  - Sequential program too slow on massive datasets
  - Load into database expensive, direct operation on log files cheaper
  - Building parallel program is possible, but very laborious
  - Map-reduce paradigm

具体代码

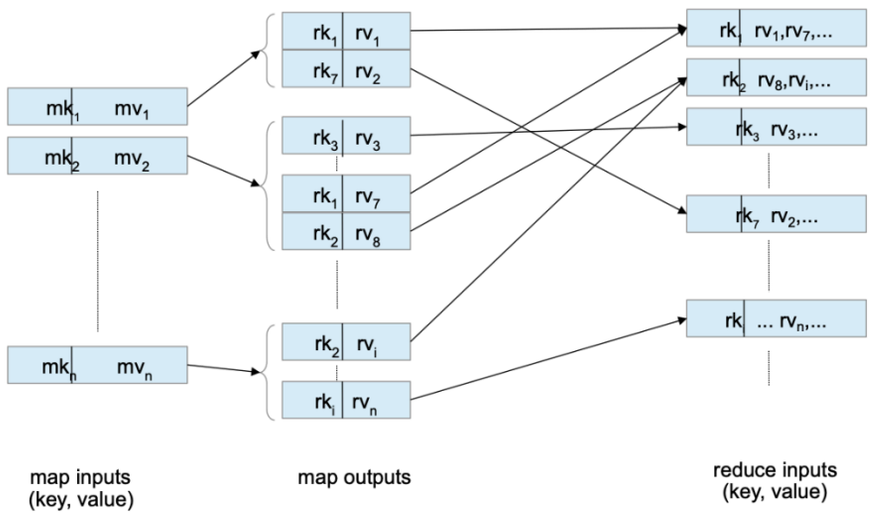
MapReduce Example: Log Processing cont'd

```
map(String key, String record) {
    String attribute[3];
    //break up record into tokens (based on space character), and store
     the tokens in array attributes
    String date = attribute[0];
    String time = attribute[1];
    String filename = attribute[2];
    if (date between 2013/01/01 and 2013/01/31
            and filename starts with "/slide-dir/")
        emit(filename, 1).
}

reduce(String key, List recordlist) {
    String filename = key;
    int count = 0;
    For each record in recordlist
      count = count + 1
    output(filename, count)
}
```

键值流动

Flow of keys and values in a map-reduce task



方法的具体实现	**Hadoop MapReduce**
	- Google pioneered map-reduce implementations that could run on thousands of machines (nodes), and transparently handle failures of machines
	- Hadoop is a widely used open source implementation of Map Reduce written in Java
	  - Map and reduce functions can be written in several different languages.
	- Input and output to map reduce systems such as Hadoop must be done in parallel
	  - Google used GFS distributed file system
	  - Hadoop uses Hadoop Distributed File System (HDFS)
	  - Input files can be in several formats
	  - Text/CSV
	  - compressed representation such as Avro, ORC and Parquet
	  - Hadoop also supports key-value stores such as Hbase, Cassandra, MongoDB, etc.

Hadoop 类型	- Generic Mapper and Reducer interfaces both take four type arguments, that specify the types of the
	  - input key, input value, output key and output value
	- Map class in next slide implements the Mapper interface
	  - Map input key is of type LongWritable, i.e. a long integer
	  - Map input value which is (all or part of) a document, is of type Text.
	  - Map output key is of type Text, since the key is a word,
	  - Map output value is of type IntWritable, which is an integer value.

Hadoop Code in Java: Map Function	`public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>`
	`{`
	`private final static IntWritable one = new IntWritable(1);`
	`private Text word = new Text();`
	`public void map(LongWritable key, Text value, Context context)`
	`throws IOException, InterruptedException`
	`{`
	`String line = value.toString();`
	`StringTokenizer tokenizer = new StringTokenizer(line);`
	`while (tokenizer.hasMoreTokens()) {`
	`word.set(tokenizer.nextToken());`
	`context.write(word, one);`
	`}`

Hadoop Code in Java: Reduce Function	
	public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
	public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException
	{
	int sum = 0;
	for (IntWritable val : values) {
	sum += val.get();
	}
	context.write(key, new IntWritable(sum));
	}
	}

Hadoop Code in Java: Overall program	
	public class WordCount {
	public static void main(String[] args) throws Exception {
	Configuration conf = new Configuration();
	Job job = new Job(conf, "wordcount");
	job.setOutputKeyClass(Text.class);
	job.setOutputValueClass(IntWritable.class);
	job.setMapperClass(Map.class);
	job.setReducerClass(Reduce.class);
	job.setInputFormatClass(TextInputFormat.class);
	job.setOutputFormatClass(TextOutputFormat.class);
	FileInputFormat.addInputPath(job, new Path(args[0]));
	FileOutputFormat.setOutputPath(job, new Path(args[1]));
	job.waitForCompletion(true);
	}
	}

Mapreduce vs Database	▪ Map Reduce is widely used for parallel processing ▪ Google, Yahoo, and 100' s of other companies ▪ Example uses: compute PageRank, build keyword indices, do data analysis of web click logs, …. ▪ Allows procedural code in map and reduce functions ▪ Allows data of any type ▪ Many real-world uses of MapReduce cannot be expressed in SQL ▪ But many computations are much easier to express in SQL ▪ Map Reduce is cumbersome for writing simple queries ▪ Relational operations (select, project, join, aggregation, etc.) can be expressed using Map Reduce ▪ SQL queries can be translated into Map Reduce infrastructure for execution ▪ Apache Hive SQL, Apache Pig Latin, Microsoft SCOPE

W13 Data Mining a

三个主要趋势

Data Warehousing：将来自多个源的数据整合到一个大型存储库中

特征：跨度时间长，经常添加摘要信息（*from gigabytes to terabytes*），交互响应时间被期望于复杂查询，临时更新不常见

存在问题：

Heterogeneous Sources：可以访问来自各种源和存储库的数据

Semantic Integration：当从多个源获取数据时，必须消除不匹配

Load, Refresh, Purge：必须加载数据，定期更新，并清除太旧的数据

Metadata Management：必须跟踪仓库中所有数据的来源、加载时间和其他信息

Online Analytical Processing (OLAP)：交互式数据分析

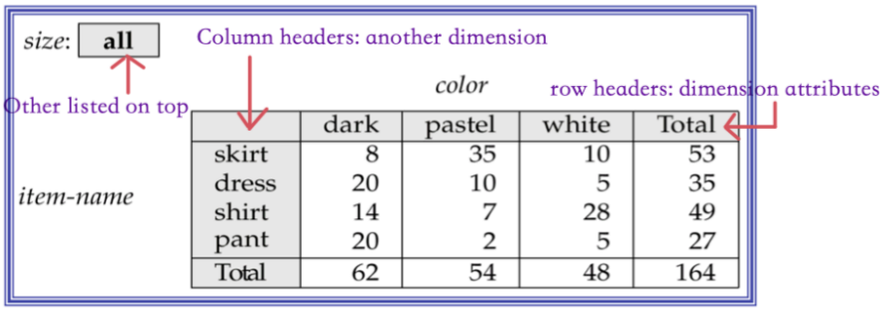
定义：允许以在线方式以不同方式汇总和查看数据

multidimensional data：可以建模为维度属性和度量属性的数据

measure attributes：测量一些值，可以被聚合，如属性“number“

dimension attributes：定义 *measure attributes* 的维度

cross-tabulation / pivot-table：单元格中值是指定单元格维度属性的值

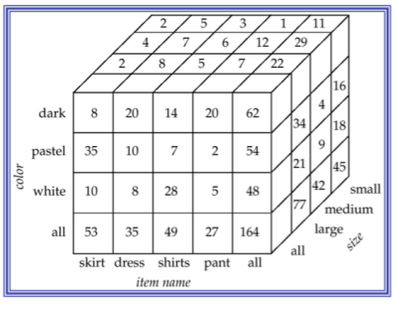


	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

item-name	*color*	*number*
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	**all**	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	**all**	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	**all**	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	**all**	27
all	dark	62
all	pastel	54
all	white	48
all	**all**	164

cross-tabs：能够被表示为关系，使用值*all*来表示聚合

data cube：*cross-tab*的多维泛化



OLAP Queries：

pivoting: 更改交叉表中使用的维度

slicing: 只为固定值创建交叉表

```
SELECT grouping attributes and aggregations
FROM fact table joined with zero or more dimension tables
WHERE certain attributes are constant
GROUP BY grouping attributes
```

rollup: 从细粒度数据移动到粗粒度数据，如从正方体到单元格

drill down: 从粗粒度数据移动到细粒度

Hierarchy on dimension attributes: 允许以不同的详细程度查看维度

例如：年-季度-月-天 / 地区-国家-州-城市

cross-tabs 可以很容易地扩展来处理这种结构

可以 drill down 或 roll up 一个结构

Data Mining: 探索和分析大量的数据，以发现有用的最终可以理解的数据模式

标准：

valid: 模式普遍适用

novel: 我们事先并不知道这种模式

useful: 以根据模式设计行动

understandable: 可以解释和理解模式

data mining model: 对数据集某一方面的描述

功能：为一组指定的输入生成输出值

最重要的类型之一：预测

例如：**clustering**, regression, **classification**

Classification 分类

目标：预测一个新项所属的类

模型要求：高精度，可理解，可快速构建大型数据库

随机变量/预测变量：X1, …, Xk，值域 dom(Xi)

因变量：y，值域 dom(y)

P：概率分布，dom(X1) x … x dom(Xk) x dom(Y)

D：training database，P 的随机样本

d：预测器，是个函数，d: dom(X1) … dom(Xk) -> dom(Y)

<u>变量类型</u>

Numerical：有序的，可以用实线表示（例如年龄、收入）

Nominal or categorical：没有任何自然顺序的有限集合（如职业种族）

Ordinal：有序，但值之间的绝对差异未知（比例偏好，伤害严重程度）

<u>Regression Problem</u>：y是numerical

Y: the dependent variable, such as Spent $

d: a regression function

r: 是从P随机抽取的记录

mean squared error rate of d: $RT(d,P) = E(r.Y - d(r.X_1, \ldots, r.X_k))^2$

Problem definition: Given dataset D that is a random sample from probability distribution P, find regression function d such that RT(d,P) is minimized.

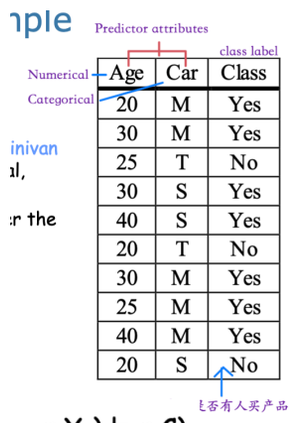
<u>Classification Problem</u>：y是categorical, such as yes/no

C代替Y: the class label

|dom(C)| = J，即类的数量

d: a classifier

r: 是从P随机抽取的记录



the misclassification rate of d: $RT(d,P) = Prop(d(r.X_1, \ldots, r.X_k) != r.C)$

Problem definition: Given dataset D that is a random sample from probability distribution P, find classifier d such that RT(d,P) is minimized.

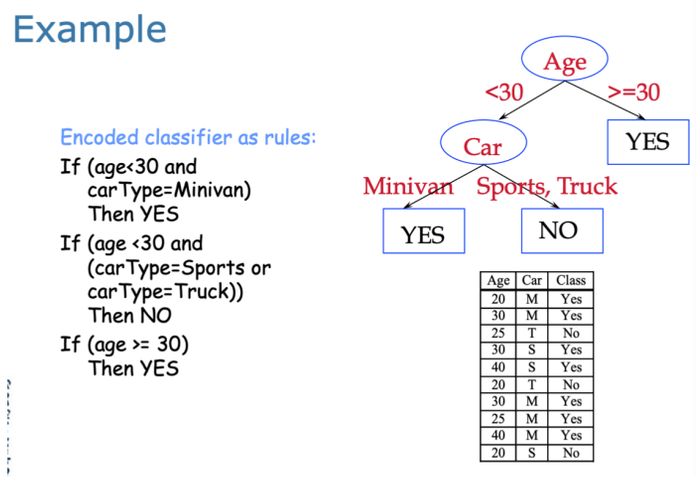
Decision trees: 分类的一个方法

决策树 T 以树的形式 encodes d (a classifier or regression function)

分类问题：

在dom(c)中，每个叶节点都用一个类标签 c 标记

每个内部节点都有一个关联的 splitting predicate，常见的是二进制



Age	Car	Class
20	M	Yes
30	M	Yes
25	T	No
30	S	Yes
40	S	Yes
20	T	No
30	M	Yes
25	M	Yes
40	M	Yes
20	S	No

构造：

Training set: 分类已知的数据样本，如信用卡的训练集是已知客户

Greedy top down: 贪心算法，从根开始递归向下构造树

Leaf node: 节点上的所有（或大部分）项属于同一类，或所有的属性都被考虑过了，不可能再进行进一步的分区，否则：

partitioning attribute: 如 bad, average, good

partitioning condition: 如 >10，树的每个内部节点将数据划分为组

最优划分

选择要划分的最佳属性和条件，从“不纯的”所有训练实例的集合开始到得到“纯的”叶节点结束

纯度测量方法：

一个训练实例集S，假设有k个类，实例数 |S|

属于类i的实例所占比例为 pi = (# of instances of class i in S)/ |S|

Gini measure $\text{Gini}(S) = 1 - \sum_{i=1}^{k} p_i^2$

Entropy $\text{Entropy}(S) = -\sum_{i=1}^{k} p_i \log_2 p_i$

当所有实例都在一个类中时，两个实例的值都为0

如果每个类具有相同数量的实例，两者达到最大值（*Gini measure 1–1/k*）

当一个集合S被分割成多个集合Si时，可以用*average entropy or information*测量集合纯度

$$\text{purity}(S_1, S_2, \ldots, S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} \text{purity}(S_i)$$

Information-gain $\text{Information-gain}(S, \{S_1, S_2, \ldots, S_r\}) = \text{purity}(S) - \text{purity}(S_1, S_2, \ldots S_r)$

split 的成本，*intrinsic information* $\text{Information-content}(S, \{S_1, S_2, \ldots, S_r\}) = -\sum_{i=1}^{r} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$

最优化分是指能提供<u>最大信息增益比</u>的分割，即

$$\text{Information-gain ratio} = \frac{\text{Information-gain}(S, \{S_1, S_2, \ldots, S_r\})}{\text{Information-content}(S, \{S_1, S_2, \ldots, S_r\})}$$

寻找最优化分

<u>*Continuous-valued attributes* 连续取值属性</u>：可以按有意义的顺序进行排序

二进制拆分：

- Sort values, try each as a split point, e.g. if values are 1, 10, 15, 25, split at ≤1, ≤ 10, ≤ 15
- Pick the value that gives best split

多路拆分：相同属性上的一系列二进制拆分

<u>*Categorical attributes* 类别取值</u>：没有意义排序

二进制拆分：尝试将所有可能的值分成两组，并选择最佳值

多路拆分：每个属性值对应一子节点，适用少数几个可区分值，如性别

决策树构造算法

```
Procedure GrowTree (S: Data set, δp: Real, δs: Nat)
  Partition (S, δp , δs );                                  截止值

Procedure Partition (S)
  if ( purity (S ) < δp or |S| < δs ) then
      return;
  for each attribute A    对于每个属性A
      evaluate splits on attribute A;   计算属于属性A的划分
  Use best split found (across all attributes) to partition
      S into S1, S2, ...., Sr,   使用找到的最好划分（根据所有属性得到）把S分成
  for i = 1, 2, ....., r
      Partition (Si , δp , δs);
```

W13 Data Mining b

Clustering 聚类 (L_p Norm)

定义：把多维空间中的给定数据 分成若干 *clusters*

方法：使用 *nearness* 来衡量，如按主题分类文档、按相似兴趣分类用户群

clusters：把记录分成k个集合作为输出，这样一个组中的记录比其他更相似

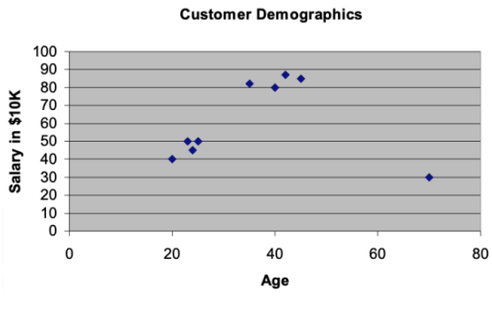
为每个 *cluster* 分配代表点 *representative points*

labeling of each record with each cluster number

unsupervised learning：没有记录标签被给予能够学习

使用：探索性数据挖掘，预处理步骤（如异常值检测）

例如



输入：两个numerical变量

Age	Salary
20	40
25	50
24	45
23	50
40	80
45	85
42	87
35	82
70	30

需要定义记录间的相似性

使用 "right" similarity (distance) 函数

· 缩放或正常化所有属性：如秒、小时、天

· 指定不同的权重以反映属性的重要性

· 选择适当的措施

Distance 的属性：Metric Spaces

度量空间指：a set **S** with a global distance function **d**

对于**S**中每两个点**x**和**y**，距离**d(x,y)**是非负实数

度量空间也必须满足

- d(x,y) = 0 iff x = y
- d(x,y) = d(y,x) (symmetry) 对称
- d(x,y) + d(y,z) >= d(x,z) (triangle inequality)

两边之和大于第三边

Minkowski Distance：关于距离

- Consider two records $x=(x_1,\ldots,x_d)$, $y=(y_1,\ldots,y_d)$, Minkowski Distance is defined by:

$$d(x,y)=\sqrt[p]{|x_1-y_1|^p+|x_2-y_2|^p+\cdots+|x_d-y_d|^p}$$

- Special cases:
  - p=1: Manhattan distance

$$d(x,y) = |x_1-y_1|+|x_1-y_2|+\ldots+|x_d-y_d|$$

  - p=2: Euclidean distance

$$\sqrt[2]{|x_1-y_1|^2+|x_2-y_2|^2+\cdots+|x_d-y_d|^2}$$

K-means Clustering 算法

选择 k 初始平均值

将每个点分配给具有最接近平均值的 cluster

计算每个 cluster 的新平均值

迭代直到 k 表示稳定

We want to cluster the data into three groups.

- Randomly assign three initial means (centroid)
  - μ_1=(20, 40), μ_2=(40, 80), μ_3=(70, 30)
- Each mean represents a cluster. Assign each sample into a cluster based on its distance to the mean.
  - C_1={(20, 40), (25, 50), (24, 45), (23, 50)}
  - C_2={(40, 80), (45, 85), (42, 87), (35, 82)}
  - C_3={(70, 30)}
- Compute and update new mean for each cluster:
  - μ_1=(23, 46.25), μ_2=(40.5, 83.5), μ_3=(70, 30)
- Repeat previous two steps until changes are less a pre-defined threshold.

Age	Salary
20	40
25	50
24	45
23	50
40	80
45	85
42	87
35	82
70	30

Market Basket Analysis 市场聚类分析

Itemset 是一组 *item*

支持度：包含 { } 的记录数 / 总记录数，*minsup* 是最小支持度

frequent itemsets 频繁项集：支持度大于 *minsup* 的 *Itemset*

Apriori property 指一个频繁项集的子集也是频繁项集

Given a minsup = 0.6, compute all frequent itemsets.

0.6 0.4 0.4

{pen}, ~~{ink}~~, ~~{milk}~~, {pen, ink}, {pen, milk},

$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

- Support of an itemset never exceeds the support of its subsets
- This is known as the anti-monotone property of support

co-occurrences：80%的客户同时购买X、Y和Z项目

- XY has support s if P(X, Y) = s
- (#of transactions contain X and Y/total of transactions)
- Same as X => Y

association rules：所有购买X、Y的客户中，60%也购买Z项目

使用两个方法精简：

- Support of a rule:
  - X => Y has support s if P(X, Y) = s
  - (#of transactions contain both X and Y/total of transactions)
- Confidence of a rule:
  - X => Y has confidence c if P(Y|X) = c
  - (#of transactions contain X and Y/# of transactions contain X)

sequential patterns：60%第一次购买X的客户也会在三周内购买Y

- Treat each transaction as a market basket
  - Example rule: {Pen} => {Milk}
    Support = 75%
    Confidence = 75%
  - Another example: {Ink} => {Pen}
    Support = 75%
    Confidence = 100%
- Treat each customer as a market basket
  - Example rule: {Pen} => {Milk}
    Support = 100%
    Confidence = 100%
  - Another example: {Ink} => {Pen}
    Support = 66.67%
    Confidence = 100%

TID	CID	Date	Item	Qty
111	201	5/1/99	Pen	2
111	201	5/1/99	Ink	1
111	201	5/1/99	Milk	3
111	201	5/1/99	Juice	6
112	105	6/3/99	Pen	1
112	105	6/3/99	Ink	1
112	105	6/3/99	Milk	1
113	106	6/5/99	Pen	1
113	106	6/5/99	Milk	1
114	201	7/1/99	Pen	2
114	201	7/1/99	Ink	2
114	201	7/1/99	Juice	4

a priori algorithm 第一个过程只考虑单个项目的集合；第二个步骤考虑两个

在一个过程结束时，所有有足够支持的集合都被输出为 *large itemsets*

支持度太少的将被取消

一旦一个集合被消除，它的任何超集都不需要考虑

在某些步骤 *i* 的末尾，如果没有一组大小 *i* 具有足够支持度

就不考虑*i+1*大小集合，然后计算终止

应用：

- 直接营销
- 医疗保险欺诈检测
- 货架规划
- 网站布局
- 交叉销售