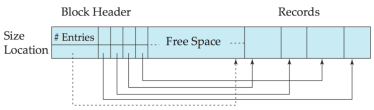
Created by wth

Buffer Manager

边长记录

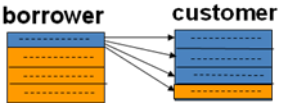


Buffer-Replacement Policies

B) MRU strategy (Most recently used， 最近最常用策略)– system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.

LRU can be a bad strategy

borrower |×| customer



设 M=5 (buffer has 5blocks), 1 for borr. , 3 for cust. ,1 for out

对 customer，LRU 的块可能是下面快要用的块(循环)，而最近刚用过的块则暂时不用，当空间不够时倒是可以将其覆盖的, 故 LRU 策略不佳

B^+-Tree

1.All paths from root to leaf are of the same length---balanced tree

2.Each node that is not a root nor a leaf has between ⌈n/2⌉ and n children. (指针数)

3.A leaf node has between ⌈(n–1)/2⌉ and n–1 values (search keys)

4. HT = log(number of keys) / log(⌈n/2⌉)

Special cases:

1.If the root is not a leaf, it has at least 2 children.

2.If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.

3. B^+树中非叶节点的 search-key 存在与叶节点中的重复（也允许不重复）。即一个 search-key 允许在 B^+树中出现 1 次或 2 次。

Procedure find(value K)

set L= root node
  while L is not a leaf node
    Begin let V_i= 大于 K 的最小键值
      if 没有符合条件的 V_i
      then set L = P_m 指向的 node
       else set L = P_i 指向的 node(沿左指针)
    End
  if L node 中存在 V_i= K
  then P_i 指向所需的 record or bucket
  else no record exists
End procedure

Insertion

1) Find the leaf node in which the search-key value would appear

2) If the search-key value is already there in the leaf node, record is added to data file [and a pointer is inserted into the bucket (对非顺序文件)].

3) If the search-key value is not there, then add the record to the data file and create a bucket if necessary. Then update B^+tree:

  a) If there is room in the leaf node,
insert :(pointer, key-value) pair in the leaf node

  b) Otherwise, split the node.

Splitting a node:

(1) take the m pairs (search-key value, pointer), including the one being inserted, in sorted order. Place the first ⌈ m/2 ⌉ in the original node, and the rest in a new node. （将 m 个**键值对**排序，分裂为 2 组，前半放原节点，后半放新节点）

(2) let the new node be p, and let k be the least key value in p. Insert (k, p) pair in the parent of the node being split. If the parent is full, split it and propagate the split further up. (新节点的键值对插入父节点，并排序。若已满，再分裂，向上)

(2)' The splitting of nodes proceeds upwards till a node that is not full is found. In the worst case the root node may be split increasing the height of the tree by 1.

Deletion

1) Find the record to be deleted, and remove it from the main file and from the bucket (if present) (删除数据记录)

2) Remove (pointer, search-key value) from the leaf node if there is no bucket or if the bucket has become empty (若无桶或空桶，则删除叶节点中的键值对(p,k), 否则说明还有相同键值的其他记录存在，不能删除该索引项)

3) If the leaf node has too few entries (< ⌈(n-1)/2⌉) due to the removal, and if the number of search-key values in node:

a) + sibling ≤ n-1

merge into one node, delete the sibling.

a) the entries in the node and a sibling fit into a single node, then (若可与相邻的兄弟节点合并为一个节点，则取消右节点，并删除对应的父节点中的项(K_{i-1}, P_i))

(1)Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.

(2) Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

b) + sibling > n-1

Redistribute with sibling

b) Otherwise, if the entries in the node and a sibling cannot fit into a single node, then （与相邻的兄弟节点中的键值对排序后重组，并更新父节点）

(1) Redistribute the search-key values and pointers between the node and a sibling such that both have more than the minimum number of entries.

(2) Update the corresponding search-key value in the parent of the node.

Index Definition in SQL

Create an index:

create index <index-name> on <table-name> (<attribute-list>);

E.g.: create index b-index on branch(branch-name);

Use create unique index to indirectly specify and enforce the condition that the search key is a candidate key.

E.g.: create unique index A on account(account-number);

Not really required if SQL unique integrity constraint is supported.

To drop an index

drop index <index-name>

Selection Operation

（1）basic algorithms : File scan –do not use index

Algorithm A1 (linear search 线性搜索).

Cost estimate = b_r.

If selection is on a key attribute, cost = (b_r /2)

A2 (binary search).

Assume that the blocks of a relation are stored contiguously

Cost estimate =⌈$\log_2(b_r)$⌉ — cost of locating the first tuple by a binary search on the blocks.

If selection is not on a key attribute, need plus the number of blocks containing records that satisfy selection condition.

Cost = ⌈$\log_2(b_r)$⌉ + ⌈sc(A, r) / f_r⌉ -1

(2) Selections Using Indices and equality

A3 (primary index on candidate key, equality, 利用主索引，码属性的等值比较).

Cost = HT_i + 1, 其中 HT_i 为索引树高(the height of tree)

A4 (primary index on non-candidate key, equality, 主索引,非码属性的等值比较)

Retrieve multiple records. --- duplicate Records will be on consecutive blocks

Cost = HT_i + number of blocks containing retrieved records

A5 (equality on search-key of secondary index,利用辅助索引的等值比较)

(1)Retrieve a single record if the search-key is a candidate key:

Cost = HT_i + 1

(2)Retrieve multiple records if search-key is not a candidate key

Cost = HT_i + number of records retrieved

（3）Selections Involving Comparisons

A7 (secondary index, comparison, 基于辅助索引的比较).

For $\sigma_{A\geq v}(r)$ use index to find first index entry A ≥ v and scan index sequentially from there to the end, to find pointers to records.

Cost= (index + scan sequentially on leaf nodes + number of records)

(4) Implementation of Complex Selections

A8 (conjunctive selection using one index).

Step1: Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$. （从 n 个条件 $\theta_1 \cdots \theta_n$ 中选择代价最小的 θ_i 先执行，返回元组放内存，然后对这些元组施行其他 θ_i , 这里的关键因素是合取）

Step2: Test other conditions on tuples in memory buffer fetched by step1.

A9 (conjunctive selection using composite index).

Use appropriate composite (multiple-key) index if available.（若正好存在组合索引: 则使用该组合索引– composite index）,于是可用 A3~A5 算法.

A10 (conjunctive selection by intersection of identifiers, 通过识别符的交集实现合取选择---先根据各合取项通过索引取得对应的元组的指针，而非元组数据本身，然后对这些指针合取运算，再根据指针取出元组数据，节省了许多取数据的时间. **适用于每个子条件都有索引**).

A11 (disjunctive selection by union of identifiers， 通过识别符的并集实现析取选择). Applicable if all conditions have available indices. Otherwise use linear scan.

Join Operation

(1) Nested-Loop Join(嵌套循环连接)

To compute the theta join: r |x|$_\theta$ s

In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is ： cost = $n_r * b_s + b_r$

nr + br disk seeks

If the smaller relation fits entirely in memory, use that as the inner relation ：

In the best case, cost = b_r + b_s disk accesses.2seeks

(2) Block Nested-Loop Join

Worst case : cost= b_r/(M-2) * b_s + b_r blocks accesses.

2*br/(M-2)* seeks

Let the relation with smaller number of blocks be outer relation

Best case: b_r + b_s block accesses（s 全部且始终在内存）2seeks

Let the relation with smaller number of blocks be inner relation

(3) Indexed Nested-Loop Join

For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.

Worst case:

Nr 是记录个数（外关系，较小的时候用这个比较有效率）， c 是 B+树层数。

Tt: transfer time, Ts: seek time, c: B+ tree seek time

+ Tt + Ts

Cost = $n_r * c + b_r *$(Tt+Ts)

(4) Merge-Join(sort-merge join,排序归并连接)

Cost = $b_r + b_s$ + the cost of sorting(if unsorted)

= $b_r+b_s+ 2 * b_r\lceil \log_{M-1}(b_r / M)\rceil + b_r b$

transfer: br + bs, seek: br/x + bs/y,

xy 具体取值求个导算算

(5) hash join

如果内存块的数量的平方小于较小的关系的内存块数量+1，需要递归的 partition

bb 每个 partition 的缓冲块数，nh 是 partition 数量

transfer: 3(br + bs) + 4 * nh

seek: 2(br / bb + bs / bb)

recursive:

$2(b_r + b_s)\lceil \log_{(M/bb)-1}(b_s/M)\rceil + b_r + b_s$ block transfers + $2(\lceil b_r/b_b\rceil + \lceil b_s/b_b\rceil)\lceil \log_{(M/bb)-1}(b_s/M)\rceil$ seeks

Statistical Information for Cost Estimation

(1) Catalog Information about relation

n_r: number of tuples in a relation r.

b_r: number of blocks containing tuples of r.

f_r: blocking factor of r — i.e., the number of tuples of r that fit into one block.

l_r : number of bytes for a tuple in r

l_s: number of bytes for a tuple in s

V(A, r): number of distinct values that appear in r for attribute A; same as the size of $\prod_A(r)$.

SC(A, r): selection cardinality of attribute A of relation r ; average number of records that satisfy equality on A.

(2) Catalog Information about Indices

F_i : average fan-out of internal nodes of index i, for tree-structured indices such as B+-trees.

HT_i : number of levels in index i — i.e., the height of index i.

For a balanced tree index (such as B+-tree) on attribute A of relation r, HT_i = ⌈$\log_{Fi}$(V(A,r))⌉.

For a hash index, HT_i is 1.

Estimate the size of selection

(3.a)Equality selection $\sigma_{A=v}(r)$

SC(A, r) : number of records that will satisfy the selection

⌈SC(A, r)/f_r⌉ — number of blocks that the records satisfying the selection will occupy

(3.b) Selections Involving Comparisons

Selections of the form $\sigma_{A\leq V}$ ：

Let c denote the estimated number of tuples satisfying the condition.

If min(A,r) and max(A,r) are available in catalog

C = 0 if v < min(A,r)

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

(Assume the values are uniformly distributed)

In absence of statistical information,(e.g v is not available at the time of optimization) c is assumed to be n_r / 2.

(3.c) Complex Selections

The selectivity of a condition θ_i (选择 σ_{θ_i}的中选率) is the probability that a tuple in the relation r satisfies θ_i . If S_i is the number of tuples in r that satisfy θ_i , the selectivity of θ_i is S_i / n_r

Conjunction: $\sigma_{\theta_1\wedge\theta_2\wedge\ldots\wedge\theta_n}(r)$.

The estimate for number of tuples in the result is:

$$n_r * \frac{S_1 * S_2 * \ldots * S_n}{n_r^n}$$

Negation: $\sigma_{\neg\theta}(r)$.

Estimated number of tuples:

n_r –size(σ_θ(r)) = n_r .(1 - S_i / n_r)

Disjunction:$\sigma_{\theta_1\vee\theta_2\vee\ldots\vee\theta_n}(r)$.

Estimated number of tuples:

$$n_r * \left(1-(1-\frac{s_1}{n_r})*(1-\frac{s_2}{n_r})*\ldots*(1-\frac{s_n}{n_r})\right)$$

Estimation of the Size of Joins

1) The Cartesian product r x s produces n_r.n_s tuples; each tuple occupies l_r + l_s bytes.

If R ∩ S = ∅, then r |x|s is the same as r x s.

2) If R ∩ S is a key for r, then a tuple of s will join with at most one tuple from r.

therefore, the number of tuples in r|x|s is no greater than the number of tuples in s.（若连接属性是 r 的候选码，则连接结果得到的行数≤s 的行数 ）

3) If R ∩ S forms a foreign key in r referencing s, number of tuples in r|x|s is exactly the same as the number of tuples in r.若连接属性是 r 的外码并参照 s，则结果行数 = r 的行数, 即结果行数最多为参照关系的行数

4) If R ∩ S = {A} is not a key for R nor S.

If we assume that each value of A appears with equal probability, then every tuple t in R produces tuples in R S, the number of tuples in R S is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

If V(A, r)≠ V(A,s), （存在悬空的元组即不匹配的元组，） the lower of these two estimates is probably the more accurate one.

Equivalence Rules

$\sigma_{\theta_1}(E_1 \ |x|_{\theta_2} E_2) = E_1 \ |x|_{\theta_1.\theta_2} E_2$

6.(a) (自然连接的结合律):

$(E_1 |x| E_2) \ |x| \ E_3 = E_1 \ |x| \ (E_2 \ |x| \ E_3)$

(b) Theta joins are associative

$(E_1|x|_{\theta_1} E_2)|x|_{\theta_2\ .\ \theta_3} E_3 = E_1|x|_{\theta_1.\ \theta_3}(E_2|x|_{\theta_2} E_3)$

where θ_2 involves attributes from only E_2 and E_3.

7. (选择的分配律：先连后选 = >先选后连)

(a) all attributes in θ_0 involve only attributes of one of the expressions being joined.

$\sigma_{\theta 0}(E_1 |x|_\theta E_2) = (\sigma_{\theta 0}(E_1)) |x|_\theta E_2$

(b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$\sigma_{\theta 1 \wedge \theta 2}(E_1 |x|_\theta E_2) = (\sigma_{\theta 1}(E_1))|x|_\theta (\sigma_{\theta 2}(E_2))$

performing the selection as early as possible reduces the size of the relation to be joined !!!

Join Ordering★让连接结果较小的连接运算先进行

Log-Based Recovery

（1）Deferred Database Modification

1) 未提交的事务不需恢复。

2) 已提交但数据未真正写到磁盘上数据库中的事务必须恢复---借助于 log 中的信息进行恢复---向前恢复---REDO

Recovery With Concurrent Transactions

When the system recovers from a crash:

ACID 性质：

1. Atomicity. Either all operations of the transaction are properly reflected in the database or none are.
2. Consistency. Execution of a transaction in isolation preserves the consistency of the database.
3. Isolation. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished.

4. Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Recoverable schedule（可恢复调度） — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti appears before the commit operation of Tj.

Lockpoint 是最后一次获得锁的时间。

strict 2pl

事务在提交/回滚之前一直拿着 X 排他锁

能够防止级联回滚

Rigorous 2pl

事务在结束之前拿着所有锁不放

能够按照事务提交的顺序进行串行化

Rigorous two-phase locking：Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.

Intention Lock

Intention locks are put on all the ancestors of a node before that node is locked explicitly.

	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

Multiple Granularity Locking Scheme

Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase ---2PL).
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i（解锁自下而上）

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order(加锁自顶向下，解锁自下而上，且遵守 2PL 协议)

Chapter 10 XML

Introduction

XML: **Extensible Markup Language**

Comparison with Relational Data

Inefficient: tags, which in effect represent schema information, are repeated

Better than relational tuples as a data-exchange format

* not like relational tuples, XML data is self-documenting due to presence of tags
* Non-rigid format: tags can be added
* Allows nested structures
* Wide acceptance, not only in database systems, but also in browsers, tools, and applications

Structure of XML Data

Basic Structure

* Mixture of text with sub-elements is legal in XML. Useful for document markup, but discouraged for data representation

More on XML Syntax

* Elements without subelements

<account number= "A-101" balance= "200" />

* To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below.

<![CDATA[<account> … </account>]]>

Namespaces

unique-name:element-name

Avoid using long unique names all over document by using XML Namespaces

<bank XmlnsFB= "http://www.FB.com" >
<FB:branch> …

XML Document Schema

DTD: Widely used

XML Schema: Newer, increasing use

Document Type Definition (DTD)

DTD constraints: What elements can occur/ What attributes can/must an element have/ What subelements can/must occur inside each element, and how many times.

DTD does not constrain: data types. All values represented as strings in XML

DTD syntax:

<!DOCTYPE bank [
<!ELEMENT element (subelements-specification) >
<!ATTLIST element (attributes) >
]>

Element Specification in DTD

Subelements can be specified as

* names of elements
* **#PCDATA** (parsed character data), i.e., character strings
* **EMPTY** (no subelements) or **ANY** (anything can be a subelement)

Subelement specification may have regular expressions

<!ELEMENT bank ((account | depositor)+)>
<!ELEMENT message (to+ from, cc*, sub, text)>
<!ELEMENT to (#PCDATA)>

"|" - alternatives, "+" - 1 or more occurrences
"*" - 0 or more occurrences, "?" - 0 or 1 occurrences

Attribute Specification in DTD

For each attribute:

<!ATTLIST name1 type1 default1|#REQUIRED|#IMPLIED… >

Type of attribute
- **CDATA**
- **ID** (identifier) or **IDREF** (ID reference) or **IDREFS** (multiple IDREFs)

Whether
- mandatory (#**REQUIRED**)
- has a default value (value),
- or neither (#**IMPLIED**)

<!ATTLIST account acct-type CDATA "checking" >
<!ATTLIST customer customer-id ID #REQUIRED accounts IDREFS #REQUIRED>

IDs and IDREFs

* An element can have at most one attribute of type ID
* The ID attribute value of each element in an XML document must be distinct
* An attribute of type IDREF must contain the ID value of an element in the same document
* An attribute of type IDREFS contains a set of (0 or more) ID values…

Limitations of DTDs

* All values are strings, no integers, reals, etc.
* Difficult to specify unordered sets of subelements
* IDs and IDREFs are untyped

XML Schema

* XML Schema addresses the faults of DTDs.
* XML Schema is itself specified in XML syntax
* XML Schema is integrated with namespaces
* XML Sche is more complicated than DTDs

<xs:schema xmlns:xs=… >
<xs:element name= "bank" type= "BankType" />
<xs:element name= "account" >
<xs:complexType> <xs:sequence>
<xs:element name= "" type= "xs:string" /> …
</xs:sequence> </xs:complexType> </xs:element>
…. definitions of customer and depositor …
<xs:complexType name= "BankType" >
</xs:complexType> </xs:schema>

<xs:keyref name = "AK" refer= "aKey" >
<xs:selector xpath = "/bank/account" />
<xs:field xpath = "account_number" />
</xs:keyref>

Querying and Transformation

Standard XML querying/translation lang

XPath / XSLT / XQuery

Tree Model of XML Data

* Element nodes have children nodes, which can be attributes or subelements
* Text in an element is modeled as a text node child of the element
* The root node has a single child, which is the root element of the document

XPath

Result of path expression: set of values that along with their containing elements/attributes match the specified path.

* Path( ) returns without the enclosing tags
* Selection predicates may follow any step in a path, in [ ]

/bank-1/customer/account[balance>400] /bank-1/customer/account[balance]

returns account containing a balance subele

Attributes are accessed using "@"

/bank-1/cust/acc [bal>400]/@account-number

Functions in XPath

count(), and, or, not(),

IDREFs can be referenced using function id()

More XPath Features

* Operator "|" used to implement union, cannot be nested inside other operators

"//" in path can be used to skip levels of nodes
"//" alone, specifying "all descendants" (of last step)
".." alone specifies the parent(of last step)

doc(name) returns the root of a named doc

XQuery

XQuery uses a **for…let…where…order by…result** syntax

for: SQL from

let: allows temporary variables

where: SQL where

order by: SQL order by

result: SQL select

for $x in /bank-2/account

let $acctno := $x/@account-number

where $x/balance > 400

return <account-number> $acctno </account-number>

Joins & Nested Queries…

Storage of XML Data

Flat File/ XMLDB/ Relational databases…

<book_pairs>
for $a in /bookstore/book
$b in /bookstore/book
where $a/author[1]=$b/author[1] and $a/price>$b/price
return <book_pair>
{<first_author> data($a/author[1])
</first_author>
$a/title $b/title}
</book_pair>
</book_pairs>

Example

DTD

ex1

<!DOCTYPE research-proj [
<!ELEMENT research (project +, developer +)>
<!ELEMENT project (pname, budget, from, to)>
<!ATTLIST project
pid ID # REQUIRED
members IDREFS # REQUIRED >
<! ELEMENT pname (#PCDATA)> <! ELEMENT budget (#PCDATA)> <! ELEMENT from (#PCDATA)>
<! ELEMENT to (#PCDATA)> <!ELEMENT developer(dname,age)>
<!ATTLIST developer
did ID # REQUIRED >
<! ELEMENT dname (#PCDATA)>
<! ELEMENT age (#PCDATA)>
]>

ex2

<!DOCTYPE parts [
<!ELEMENT part (name, subpartinfo*)>
<!ELEMENT subpartinfo (part, quantity)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
]>

corresponding **XML schema**

<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="parts" type="partsType"/>
<xs:complexType name="partType">
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="subpartinfo" type="subpartinfoType" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="subpartinfoType">
<xs:sequence>
<xs:element name="part" type="partType"/>
<xs:element name="quantity" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:schema>

An **XML data** corresponding to this DTD

<parts>
<part>
<name> watermelon </name>
<subpartinfo>
<part>
<name> pulp </name>
</part>
<quantity> 1 </quantity>
</subpartinfo>
</part>
<part>
</parts>

路径表达式

bookstore 选取 bookstore 元素的所有子节点

/bookstore 选取根元素 bookstore. 假如路径起始于正斜杠(/)，则此路径始终代表到某元素的绝对路径.

bookstore/book 选取所有属于 bookstore 的子元素的 book 元素。

//book 选取所有 book 子元素，而不管它们在文档中的位置。

bookstore//book 选择所有属于 bookstore 元素的后代的 book 元素，而不管它们位于 bookstore 之下的什么位置。

//@lang 选取所有名为 lang 的属性。

/bookstore/book[1] 选取属于 bookstore 子元素的第一个 book 元素。

/bookstore/book[last()-1] 选取 bookstore 子元素的倒数第二个 book 元素

/bookstore/book[position()<3] 选取最前面的两个属于 bookstore 元素的子元素的 book 元素。

//title[@lang] 选取所有拥有名为 lang 的属性的 title 元素。

//title[@lang= 'eng'] 选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。

/bookstore/book[price>35.00]/title 选取所有 bookstore 元素中的 book 元素的 title 元素，且其中的 price 元素的值须大于 35.00。

bookstore/* 选取 bookstore 元素的所有子节点

//title[@*] 选取所有带有属性的 title 元素。

//book/title | //book/price 选取所有 book 元素的 tilte 和 price 元素。

Aries 其他特性：

嵌套的顶层动作：允许某些操作即使回滚也不被撤销（通过虚拟 CLR）

回复的独立性：使某页独立恢复

保存点：允许部分回滚到某个保存点，对于死锁处理特别有用

细粒度的封锁：允许索引元素级的封锁而不是页级的封锁，大大提高并发性

恢复最优化：重做的顺序和时间非常科学。