Database Management Systems

Winter 2003

CMPUT 391: Revision

Dr. Osmar R. Zaïane

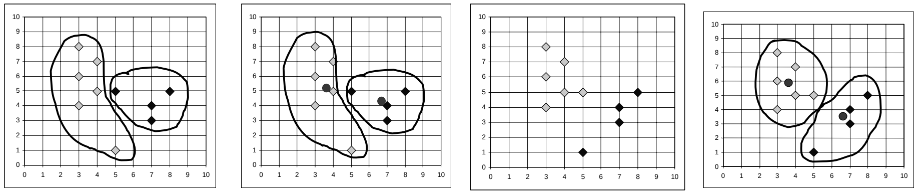
University of Alberta

Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- Data Base Recovery and Security
- Object-Oriented Databases
- Inverted Index for IR
- Spatial Data Management
- XML and Databases
- Data Warehousing
- **Data Mining**
- Parallel and Distributed Databases

The *K-Means* Clustering Method

- Given *k*, the *k-means* algorithm is implemented in 4 steps:
  1. Partition objects into *k* nonempty subsets
  2. Compute seed points as the centroids of the clusters of the current partition. The centroid is the center (mean point) of the cluster.
  3. Assign each object to the cluster with the nearest seed point.
  4. Go back to Step 2, stop when no more new assignment.



Example for Algorithm (ID3)

- All attributes are categorical
- Create a node N;
  - if samples are all of the same class C, then return N as a leaf node labeled with C.
  - if attribute-list is empty then return N as a leaf node labeled with the most common class.
- Select split-attribute with highest information gain
  - label N with the split-attribute
  - for each value A_i of split-attribute, grow a branch from Node N
  - let S_i be the branch in which all tuples have the value A_i for split- attribute
    - if S_i is empty then attach a leaf labeled with the most common class.
    - Else recursively run the algorithm at Node S_i
- Until all branches reach leaf nodes

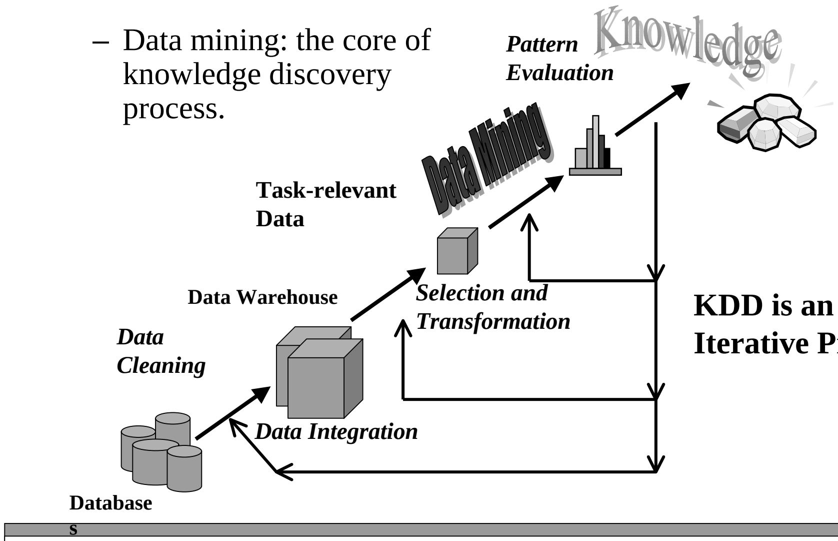
The Apriori Algorithm

C_k: Candidate itemset of size k
L_k : frequent itemset of size k

```
$L_1$ = {frequent items};
for ($k$ = 1; $L_k$ !=$\emptyset$; $k$++) do begin
    $C_{k+1}$ = candidates generated from $L_k$;
    for each transaction $t$ in database do
        increment the count of all candidates in
        $C_{k+1}$ that are contained in $t$
    $L_{k+1}$ = candidates in $C_{k+1}$ with min_support
    end
return $\cup_k L_k$;
```

The KDD Process

– Data mining: the core of knowledge discovery process.

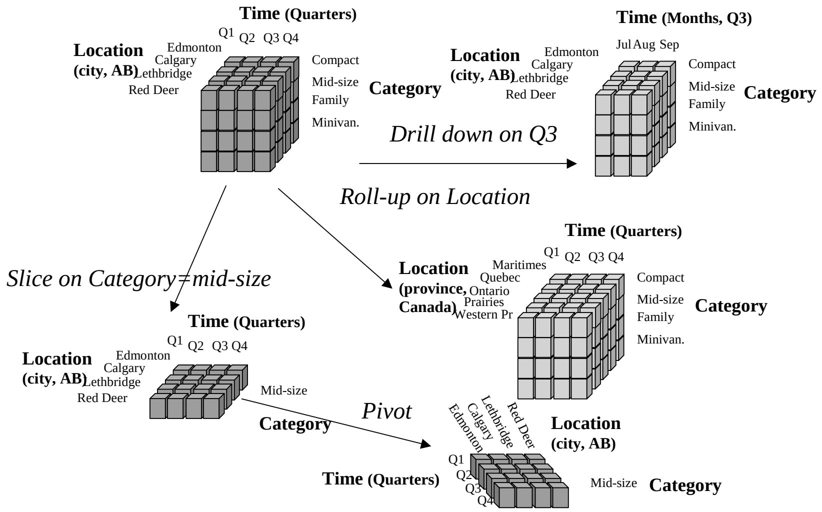


KDD is an Iterative Process

Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- Data Base Recovery and Security
- Object-Oriented Databases
- Inverted Index for IR
- Spatial Data Management
- XML and Databases
- **Data Warehousing**
- Data Mining
- Parallel and Distributed Databases

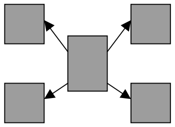
Data Warehouse OLAP Example



Data Warehouses Design (con't)

- Modeling data warehouses: dimensions & measurements

Star schema: A single object (fact table) in the middle connected to a number of objects (dimension tables)



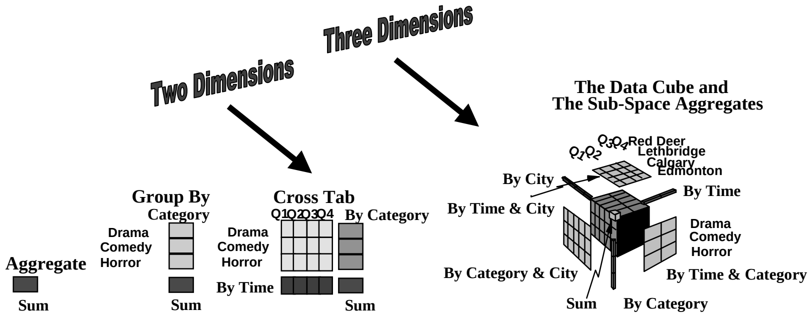
Each dimension is represented by one table
➔Un-normalized (introduces redundancy).

Ex: (Edmonton, Alberta, Canada, North America)
(Calgary, Alberta, Canada, North America)

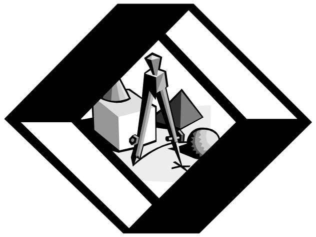
Normalize dimension tables ➔ **Snowflake schema**

Aggregation in Data Warehouses

Multidimensional view of data in the warehouse:
Stress on aggregation of measures by one or more dimensions



Issues



- Scalability
- Sparseness
- Curse of dimensionality
- Materialization of the multidimensional data cube (total, virtual, partial)
- Efficient computation of aggregations
- Indexing

Course Content

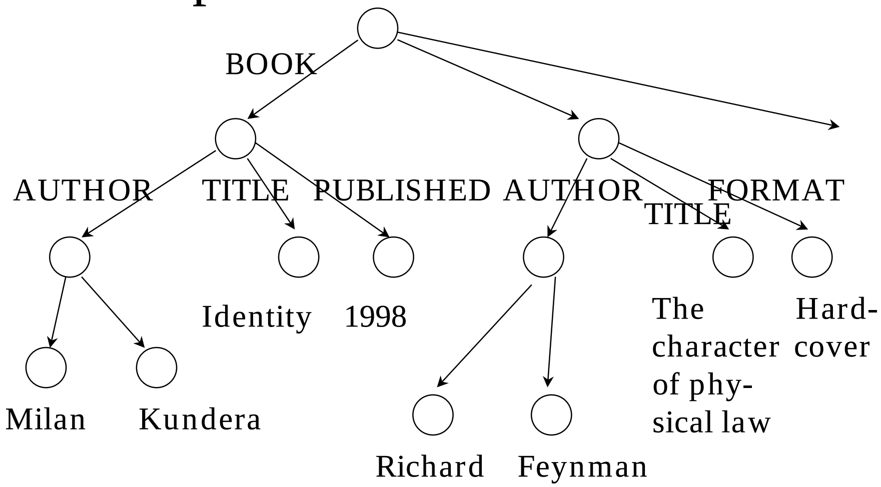
- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- Data Base Recovery and Security
- Object-Oriented Databases
- Inverted Index for IR
- Spatial Data Management
- **XML and Databases**
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases

Semi-Structured Data

- Structured data is rigidly organized & well defined ➔ predictable
- Unstructured data is disordered and unruly ➔unpredictable
- Semi-structured data is organized enough to be predictable
  - Data is organized in semantic entities
  - Similar entities are grouped together

  But
  - Entities in the same group may not have the same attributes
  - The order of the attributes is not necessarily important
  - The presence of some attributes may not always be required
  - The size of same attributes of entities in a same group may not be the same
  - The type of the same attributes of entities in a same group may not be of the same type.
- An HTML document is an example of semi-structured data

Example: Booklist Data in OEM



What is Special with XML

- It is a language to markup data
- There are no predefined tags like in HTML
- Extensible ➔ tags can be defined and extended based on applications and needs
  - Elements / tags
  - Attributes
  - Example: <BOOK page="453">…</BOOK>

Example1: Business Letter

```
<?xml version = "1.0"?>
<LETTER> <Urgency level="1"/>
 <contact type = "from">
              <name>John Doe</name>
              <address>123 Main St.</address>
             <city>Anytown</city>
             <province>Somewhere</province>
             <postalcode>A1B 2C3</postalcode>
</contact>
<contact type = "to">
             <name>Joe Schmoe</name>
             <address>123 Any Ave.</address>
              <city>Othertown</city>
             <province>Otherplace</province>
             <postalcode>Z9Y 8X7</postalcode>
</contact>
<paragraph>Dear Sir,</paragraph>
<paragraph>It is our privilege to inform you about our new  database managed with XML.
This new system will allow you to reduce the load of your inventory list server by  having the
client machine perform the work of sorting  and filtering the data.</paragraph>
<paragraph>Sincerely, Mr. Doe</paragraph>
</LETTER>
```

DTD Example for business letter

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE LETTER [
<!ELEMENT LETTER (Urgency, contact+, paragraph+)>
<!ELEMENT Urgency (EMPTY)>
<!ATTLIST Urgency level CDATA #IMPLIED>
<!ELEMENT contact (name, address, city, province,  postalcode,
phone?, email?)>
<!ATTLIST contact type CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
…
]>
```

+ means one or more

Empty means no end tag

? means optional

CDATA means string #IMPLIED means that the attribute value is unspecified.

#PCDATA is parsed character data, it means that the element contains text

Querying XML Data

- Goal: High-level, declarative language that allows manipulation of XML documents.
- Manipulation means the retrieval of documents, sub-documents and elements and attribute values, as well as the generation of new XML documents.
- There are many languages proposed by researchers. One standard emerged recently (X-Query adopted by W3C)
- Lorel (1997), XML-QL (1999), XQL (1999), Quilt(2000), XQuery (2001), …
- Also XPath (lightweight XML query language) and XSLT (transformation language for XML)

XPath with Selection Conditions (Xpath Queries)

- Select all student who took a course in F2000

//Student[Course/@Semester="F2000"]

- Select undergrad students with last name starting with 'B'

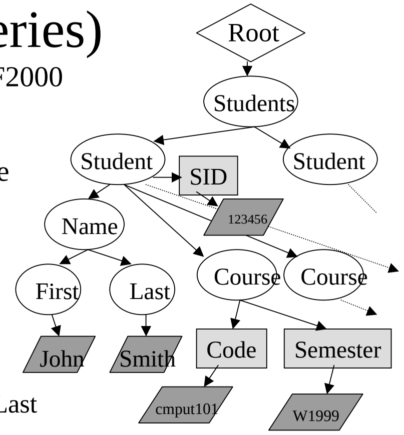
//Student[Status="UndG" AND
starts-with(.//Last, "B")]

- Select last names of students who took cmput391

//Student[Course/@Code="cmput391"]/Name/Last

- Select all students who took cmput391 in F2001

//Student[Course/@Code="cmput391" AND   Course/@Semester="F2001"]



XQuery Basics

- General structure:
  - [FOR | LET] variable declarations
  - WHERE   condition
  - RETURN   document
  - variable declaration:
    "$" VarName "in" Expression
- Variable binding
  – FOR binds a variable to each element satisfying the expression
  – LET binds a variable to the whole collection of elements that satisfy the expression

Example – FOR clause

- Assume the previous document is stored at www.outbookstore.com/books.xml
- QUERY: Find the last names of all authors

```
FOR
  $l IN doc(www.ourbookstore.com/books.xml)//AUTHOR/LASTNAME
RETURN
  <RESULT> $l </RESULT>
```

ANSWER

```
<RESULT><LASTNAME> Feynman</LASTNAME></RESULT>
<RESULT><LASTNAME> Narayan</LASTNAME></RESULT>
<RESULT><LASTNAME> Narayan</LASTNAME></RESULT>
```

Example – LET clause

```
LET
  $l IN doc(www.ourbookstore.com/books.xml)//AUTHOR/LASTNAME
RETURN
  <RESULT> $l </RESULT>
```

ANSWER

```
<RESULT>
  <LASTNAME> Feynman</LASTNAME>
  <LASTNAME> Narayan</LASTNAME>
  <LASTNAME> Narayan</LASTNAME>
</RESULT>
```

Example: WHERE clause

```
FOR
  $1 IN doc(www.ourbookstore.com/books.xml)/BOOKLIST/BOOK
WHERE $1/PUBLISHED = "1980"
RETURN
    <RESULT> $1/AUTHOUR/FIRSTNAME, $1/AUTHOR/LASTNAME
    </RESULT>
```

ANSWER

```
<RESULT>
  <FIRSTNAME> Richard <.FIRSTNAME>
  <LASTNAME>Feynman</LASTNAME>
</RESULT>
<RESULT>
  <FIRSTNAME> R.K.<.FIRSTNAME>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
```

Example: Nested Queries & Grouping

```
FOR $l IN DISTINCT
    doc(www.ourbookstore.com/books.xml)/BOOKLIST/BOOK/PUBLISHED
RETURN
<RESULT>
  $1,
  FOR $a IN DISTINCT doc(www.ourbookstore.com/books.cml)
          /BOOKLIST/BOOK[PUBLISHED=$1]/AUTHOUR/LASTNAME
  RETURN $a
</RESULT>
```

ANSWER

```
<RESULT>
  <PUBLISHED> 1980 </PUBLISHED>
  <LASTTNAME> Feynman<LASTNAME>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
<RESULT>
  <PUBLISHED> 1981<PUBLISHED>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
```

Example: Join & Aggregation

```
FOR $a IN DISTINCT
    doc(www.ourbookstore.com/books.xml)/BOOKLIST/BOOK/AUTHOR
LET $t IN
    doc(www.ourbookstore.com/books.xml)//BOOK/[AUTHOR=$a]/TITLE
RETURN
<RESULT>
          $a/LASTNAME, <TotalBooks> count(distinct($t)) </TotalBooks>
</RESULT>
SORT BY (LASTNAME descending)
```

ANSWER (e.g.)

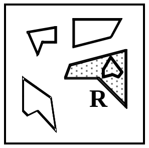
```
<RESULT>
   <LASTTNAME>Narayan<LASTNAME>
   <TotalBooks> 5 </TotalBooks>
</RESULT>
<RESULT>
   <LASTNAME>Feynman</LASTNAME>
   <TotalBooks> 2 </TotalBooks>
</RESULT>
```

Course Content

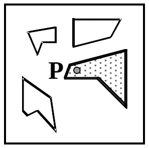
- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- Data Base Recovery and Security
- Object-Oriented Databases
- Inverted Index for IR
- **Spatial Data Management**
- XML and Databases
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases

Basic Spatial Queries

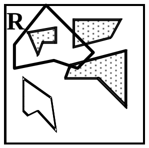
- ***Containment Query***: Given a spatial object R, find all objects that completely contain R. If R is a Point: ***Point Query***
- ***Region Query***: Given a region R (polygon or circle), find all spatial objects that intersect with R. If R is a rectangle: ***Window Query***
- ***Enclosure Query***: Given a polygon region R, find all objects that are completely contained in R
- ***K-Nearest Neighbor Query***: Given an object P, find the k objects that are closest to P (typically for points)



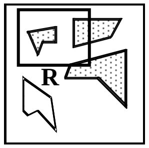
Containment Query



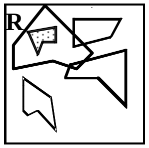
Point Query



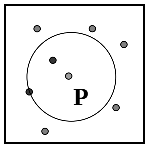
Region Query



Window Query



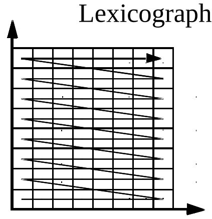
Enclosure Query



2-nn Query

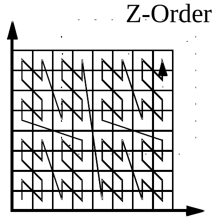
Space Filling Curves

Lexicographic Order



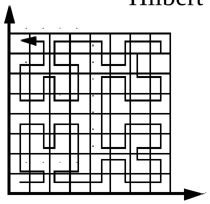
56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Z-Order



21	23	29	31	53	55	61	63
20	22	28	30	52	54	60	62
17	19	25	27	49	51	57	59
16	18	24	26	48	50	56	58
5	7	13	15	37	39	45	47
4	6	12	14	36	38	44	46
1	3	9	11	33	35	41	43
0	2	8	10	32	34	40	42

Hilbert-Curve



63	62	49	48	47	44	43	42
60	61	50	51	46	45	40	41
59	56	55	52	33	34	39	38
58	57	54	53	32	35	36	37
5	6	9	10	31	28	27	26
4	7	8	11	30	29	24	25
3	2	13	12	17	18	23	22
0	1	14	15	16	19	20	21

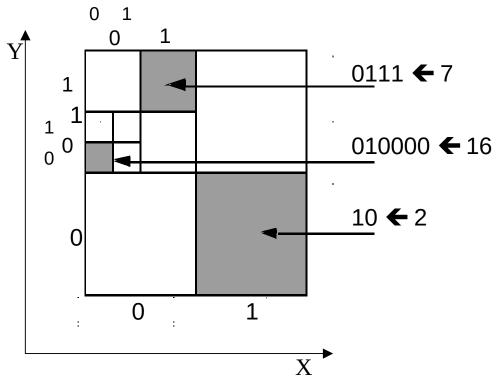
- Z-Order preserves spatial proximity relatively good
- Z-Order is easy to compute

Z-Order – Z-Values

- Coding of Cells
  - Partition the data space recursively into two halves
  - Alternate X and Y dimension
  - Left/bottom ← 0
  - Right/top ← 1

  - **Z-Value: (c, l)**
    c = decimal value of the bit string
    l = level (number of bits)

    if all cells are on the same level, then l can be omitted



Z-Order – Mapping to a B⁺-Tree

- Linear Order for Z-values to store them in a B⁺-tree:

Let (c_1, l_1) and (c_2, l_2) be two Z-Values and let $l = \min\{l_1, l_2\}$.

The order relation $\le_Z$ (that defines a linear order on Z-values) is then defined by

$$(c_1, l_1) \le_Z (c_2, l_2) \text{ iff } (c_1 \text{ div } 2^{(l_1 - l)}) \le (c_2 \text{ div } 2^{(l_2 - l)})$$

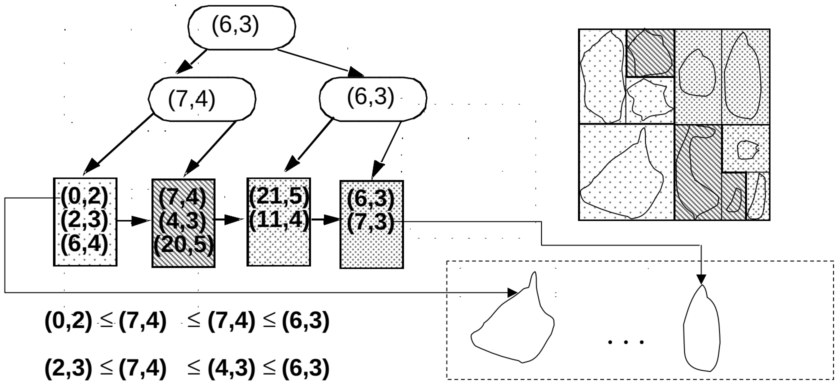
Examples:

$(1,2) \le_Z (3,2)$,

$(3,4) \le_Z (3,2)$,

$(1,2) \le_Z (10,4)$

Mapping to a B⁺-Tree - Example



Exact representations stored in a different location

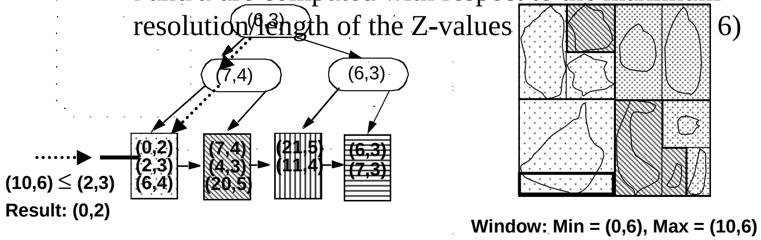
(0,2) ≤ (7,4) ≤ (7,4) ≤ (6,3)

(2,3) ≤ (7,4) ≤ (4,3) ≤ (6,3)

(6,4) ≤ (7,4) ≤ (20,5) ≤ (6,3)

Mapping to a B⁺-Tree – Window Query

- Window Query ← Range Query in the B+-tree
  - find all entries (Z-Values) in the range [l, u] where
    - l = smallest Z-Value of the window (bottom left corner)
    - u = largest Z-Value of the window (top right corner)
    - l and u are computed with respect to the maximum resolution length of the Z-values 6)

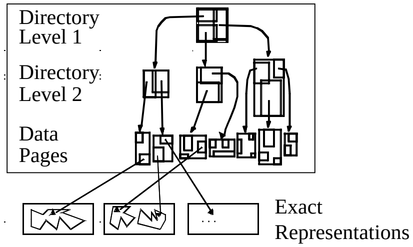


Window: Min = (0,6), Max = (10,6)

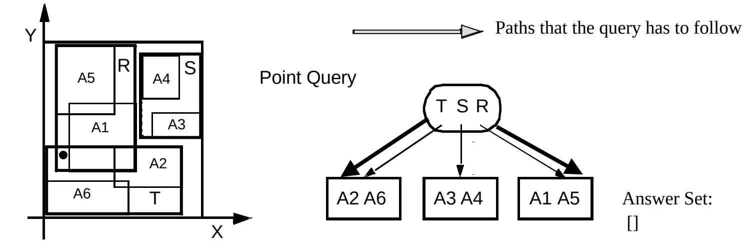
The R-Tree – Properties

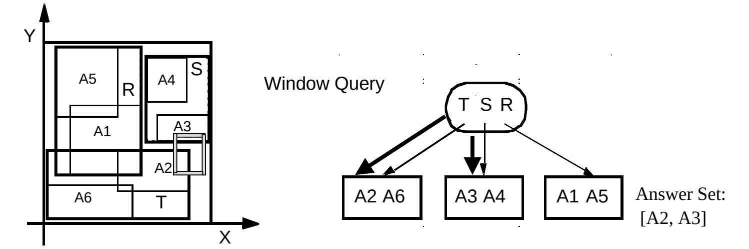
- Balanced Tree designed to organize rectangles [Gut 84].
- Each page contains between *m* and *M* entries.
- Data page entries are of the form (*MBR*, *PointerToExactRepr*).
  - MBR is a minimum bounding rectangle of a spatial object, which PointerToExactRepr is pointing to
- Directory page entries are of the form (*MBR*, *PointerToSubtree*).
  - *MBR* is the minimum bounding rectangle of all entries in the subtree, which *PointerToSubtree* is pointing to.
- Rectangles can overlap
- The height *h* of an R-Tree for *N* spatial objects:

$$h \leq \lceil \log_m N \sum +1$$



The R-Tree – Queries

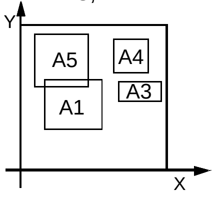




R-Tree Construction – Optimization Goals

- Overlap between the MBRs
  - ∑ spatial queries have to follow several paths
  - ∑ try to minimize overlap
- Empty space in MBR
  - ∑ spatial queries may have to follow irrelevant paths
  - ∑ try to minimize area and empty space in MBRs

M = 3, *m* = 2



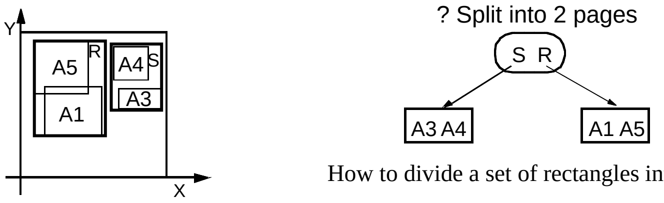
Start: □ empty data page (= root)

Insert: A5, A1, A3, A4 ∑

A5, A1, A3, A4 * (overflow)

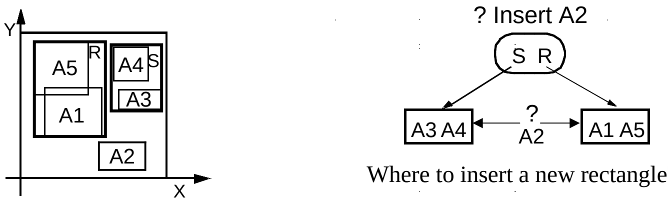
R-Tree Construction – Important Issues

- Split Strategy



How to divide a set of rectangles into 2 sets?

- Insertion Strategy



Where to insert a new rectangle?

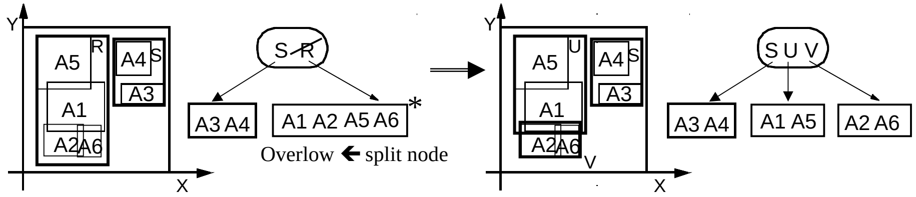
R-Tree Construction – Insertion Strategies

- Dynamic construction by insertion of rectangles *R*
  - Searching for the data page into which *R* will be inserted, traverses the tree from the root to a data page.
  - When considering entries of a directory page *P*, 3 cases can occur:
    1. *R* falls into exactly one *Entry*.*MBR*
       🡐 follow *Entry*.*Subtree*
    2. *R* falls into the MBR of more than one entry e_1 , ... , e_n
       🡐 follow E_i.*Subtree* for entry e_i with the smallest area of e_i.*MBR*.
    3. *R* does not fall into an *Entry*.*MBR* of the current page
       🡐 check the increase in area of the *MBR* for each entry when enlarging the *MBR* to enclose *R*. Choose *Entry* with the minimum increase in area (if this entry is not unique, choose the one with the smallest area); enlarge *Entry*.*MBR* and follow *Entry*.*Subtree*
- Construction by "bulk-loading" the rectangles
  - Sort the rectangles, e.g., using Z-Order
  - Create the R-tree "bottom-up"

R-Tree Construction – Split

- Insertion will eventually lead to an overflow of a data page
  - The parent entry for that page is deleted.
  - The page is split into 2 new pages - according to a *split strategy*
  - 2 new entries pointing to the newly created pages are inserted into the parent page.
  - A now possible overflow in the parent page is handled recursively in a similar way; if the root has to be split, a new root is created to contain the entries pointing to the newly created pages.

M = 3, *m* = 2

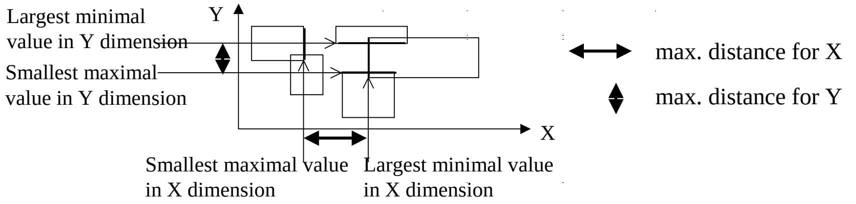


R-Tree Construction – Splitting Strategies

- Overflow of node *K* with |*K*| = *M*+1 entries 🡐 Distribution of the entries into two new nodes K_1 and K_2 such that $|K_1| \geq m$ and $|K_2| \geq m$
- *Exhaustive algorithm*:
  - Searching for the "best" split in the set of all possible splits is too expensive (O(2^M) possibilities!)
- *Quadratic algorithm*:
  - Choose the pair of rectangles R_1 and R_2 that have the largest value $d(R_1, R_2)$ for empty space in an MBR, which covers both R_1 und R_2.
    $d\ (R_1, R_2) := \text{Area}(\text{MBR}(R_1 \cup R_2)) – (\text{Area}(R_1) + \text{Area}(R_2))$
  - Set $K_1 := \{R_1\}$ and $K_2 := \{R_2\}$
  - Repeat until STOP
    - if all R_i are assigned: STOP
    - if all remaining R_i are needed to fill the smaller node to guarantee minimal occupancy *m*: assign them to the smaller node and STOP
    - else: choose the next R_i and assign it to the node that will have the smallest increase in area of the MBR by the assignment. If not unique: choose the K_i that covers the smaller area (if still not unique: the one with less entries).

R-Tree Construction – Splitting Strategies

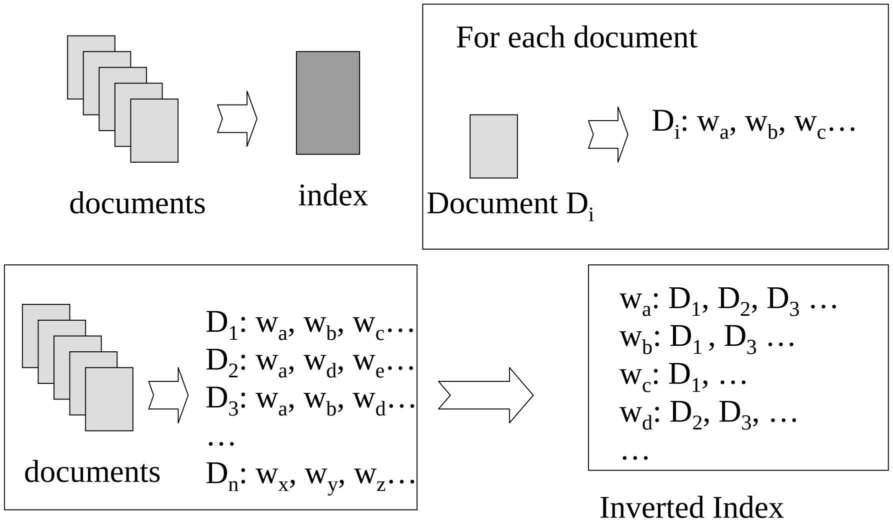
- *Linear algorithm*:
  - Same as the quadratic algorithm, except for the choice of the initial pair: Choose the pair with the largest distance.
    - For each dimension determine the rectangle with the largest minimal value and the rectangle with the smallest maximal value (the difference is the *maximal distance/separation*).
    - Normalize the maximal distance of each dimension by dividing by the sum of the extensions of the rectangles in this dimension
    - Choose the pair of rectangles that has the greatest normalized distance.
      Set $K_1 := \{R_1\}$ and $K_2 := \{R_2\}$.


Largest minimal value in Y dimension
Smallest maximal value in Y dimension
Y
X
Smallest maximal value in X dimension
Largest minimal value in X dimension
max. distance for X
max. distance for Y

Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- Data Base Recovery and Security
- Object-Oriented Databases
- **Inverted Index for IR**
- Spatial Data Management
- XML and Databases
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases

Creating an Index



documents

index

For each document

Document D_i

D_i: w_a, w_b, w_c…

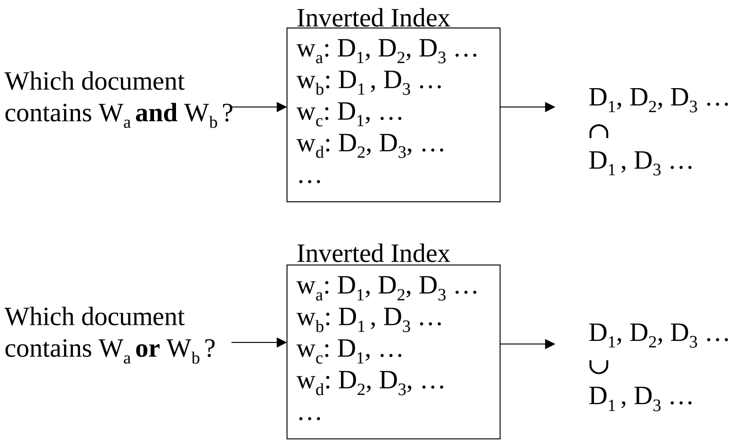
documents

D_1: w_a, w_b, w_c…
D_2: w_a, w_d, w_e…
D_3: w_a, w_b, w_d…
…
D_n: w_x, w_y, w_z…

w_a: D_1, D_2, D_3 …
w_b: D_1 , D_3 …
w_c: D_1, …
w_d: D_2, D_3, …
…

Inverted Index

Querying



Which document contains W_a **and** W_b ?

Inverted Index
w_a: D_1, D_2, D_3 …
w_b: D_1 , D_3 …
w_c: D_1, …
w_d: D_2, D_3, …
…

D_1, D_2, D_3 …
$\cap$
D_1 , D_3 …

Which document contains W_a **or** W_b ?

Inverted Index
w_a: D_1, D_2, D_3 …
w_b: D_1 , D_3 …
w_c: D_1, …
w_d: D_2, D_3, …
…

D_1, D_2, D_3 …
$\cup$
D_1 , D_3 …

Signature Files

- Index structure (the signature file) with one data entry for each document
- Hash function hashes words to bit-vector.
- Data entry for a document (the signature of the document) is the OR of all hashed words.
- Signature S1 matches signature S2 if S2&S1=S2

Signature Files: Query Evaluation

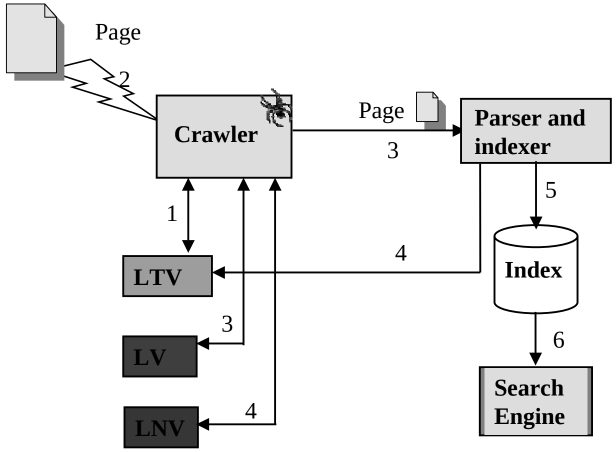
- Boolean query consisting of conjunction of words:
  - Generate query signature Sq
  - Scan signatures of all documents.
  - If signature S matches Sq, then retrieve document and check for false positives.
- Boolean query consisting of disjunction of k words:
  - Generate k query signatures S1, …, Sk
  - Scan signature file to find documents whose signature matches any of S1, …, Sk
  - Check for false positives

Signature Files: Example

Word	Hash
Agent	010
James	100
Mobile	001

RID	Document	Signature
1	Agent James	110
2	Mobile agent	011

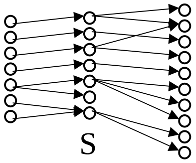
Search Engine General Architecture



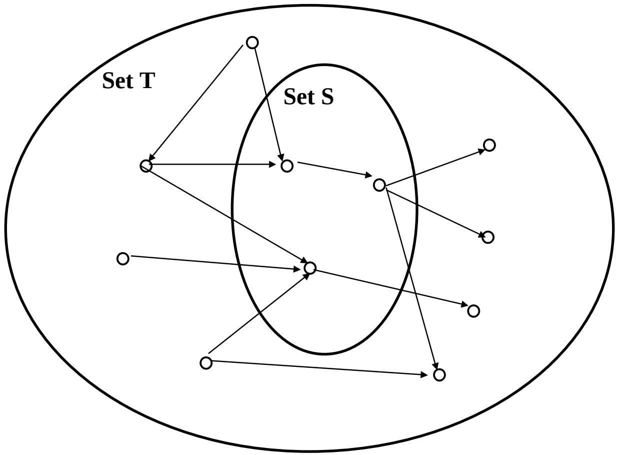
Steps of HITS Algorithm

- Starting from a user supplied query, HITS assembles an initial set S of pages:

  The initial set of pages is called root set. These pages are then expanded to a larger root set T by adding any pages that are linked to or from any page in the initial set S.



- HITS then associates with each page p a hub weight h(p) and an authority weight a(p), all initialized to one.



- HITS then iteratively updates the hub and authority weights of each page.

Let p → q denote “page p has an hyperlink to page q”. HITS updates the hubs and authorities as follows:

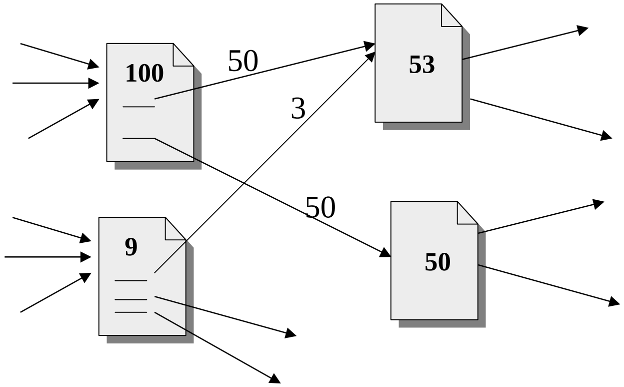
$$a(p) = \sum_{q \rightarrow p} h(q)$$

Good authorities are linked by good hubs

$$h(p) = \sum_{p \rightarrow q} a(q)$$

Good hubs link to good authorities

Idealized PageRank Calculation



Each Page *p* has a number of links coming out of it C(*p*) (C for citation), and number of pages pointing at page p_1, p_2 ….., p_n.

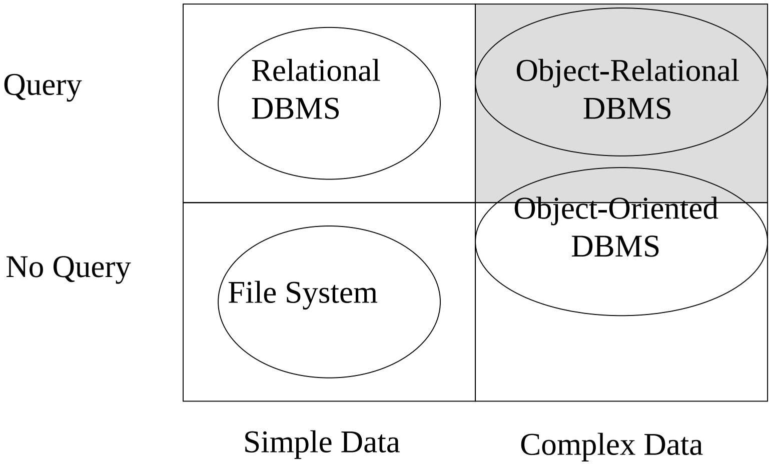
PageRank of P is obtained by

$$PR(p) = (1-d) + \sum_{k=1}^{n} \frac{PR(p_k)}{C(p_k)}$$

Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- Data Base Recovery and Security
- **Object-Oriented Databases**
- Inverted Index for IR
- XML
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases
- Other Advanced Database Topics

DBMS Classification Matrix



ODL in OODBMS

- ODL supports atomic types as well as set, bag, list array and struct type
- Interface defines a class

```
interface Movie (extent Movies key movieName)
        { attribute date start;
          attribute date end;
          attribute string   movieName;
          relashionship Set<Theater> ShownAt inverse Theater::nowShowing;
        }
interface Theater (extent Theaters key theaterName)
        { attribute string theaterName;
          attribute string address;
          attribute integer ticketPrice;
          relationship Set <Movie> nowShowing inverse Movie::shownAt;
          float numshowing() raises(errorCountingMovies);
        }
```

OQL Examples

Find the movies and theaters such that the theaters show more than one movie.

```
SELECT mname: M.movieName, tname: T.theaterName
FROM Movies M, M.shownAt T
WHERE T.numshowing() >1
```

Use of path expression
T is bound to each theater
Related to movie M by
relationship shownAt

Method of Objects can be called
everywhere in the query

Find the different ticket prices and the average number of movies shown at theaters with that ticket price.

Partitioning in OQL

```
SELECT T.ticketPrice,
            avgNum:AVG(SELECT P.T.numshowing() FROM partition P)
FROM Theaters T
GROUP BY T.ticketPrice
```

Java Binding Example

```
import org.odmg.*;
import java.util.Collection;

Implementation impl = new com.vendor.odmg.Implementation();
Database db = impl.newDatabase();
Transaction txn = impl.newTransaction();
try {
    db.open(“movieDB", Database.OPEN_READ_WRITE);
    txn.begin();
    OQLQuery query = new OQLQuery(
       "select t from Theaters t where t.ticketprice < $1");
    query.bind(uInput1()); //bind $1 to a user specified value
    Collection result = (Collection) query.execute();
    Iterator iter = result.iterator();
    while ( iter.hasNext() ) {
        Theater theater = (Theater) iter.next();
        theater.ticketprice = theater.ticketprice * 1.5;
    }
    txn.commit();
    db.close();
}
//exception handling would go here ...
```

Common Structured Types

- Type constructors are used to combine atomic types and user defined types to create more complex structures:
- ROW(n_1, t_1, …n_n,t_n) : tuple of n fields
- listof(base): list of base-type items
- ARRAY(base): array of base-type items
- setof(base): set of base-type items without duplicates
- bagof(base): multiset of base-type items

Not all collection types supported by all systems

Built-in Operators for Structured Types

- Path expression
- Comparisons of sets ($\subset\subseteq=\supseteq\supset \in\cup\cap-$)
- Append and prepend for lists
- Postfix square bracket for arrays
- -> for reference type

Object-Relational Features of Oracle

Object table: table in which each row represents an object.

```
CREATE TYPE person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TABLE person_table OF person;

INSERT INTO person_table VALUES
       person("John Smith", "1-800-555-1212");

SELECT VALUE(p) FROM person_table p
        WHERE p.name = "John Smith";
```

Object-Relational Features of Oracle

Methods

```
CREATE TYPE Rectangle_typ AS OBJECT (
  len NUMBER,
  wid NUMBER,
  MEMBER FUNCTION area RETURN NUMBER,
);

CREATE TYPE BODY Rectangle_typ AS
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
     RETURN len * wid;
  END area;
END;
```

Object-Relational Features of Oracle

REF datatype: reference to other objects

```
CREATE TABLE people (
  id            NUMBER(4)
  name_obj      name_objtyp,
  address_ref   REF address_objtyp
                  SCOPE IS address_objtab);
```

can be "scoped" for more efficient access

De-referencing (assume X is an object of type people)

`X.deref(address_ref).street`

In Oracle also implicitly: `X.address_ref.street`

Obtaining references

```
SELECT REF(po) FROM purchase_order_table po
WHERE po.id = 1000376;
```

Object-Relational Features of Oracle

Collection types / nested tables

```
CREATE TYPE PointType AS OBJECT (
     x NUMBER,
     y NUMBER);
CREATE TYPE PolygonType AS TABLE OF PointType;
CREATE TABLE Polygons (
     name   VARCHAR2(20),
     points PolygonType)
     NESTED TABLE points STORE AS PointsTable;
```

The relations representing individual polygons are not stored directly as values of the points attribute; they are stored in a single table, PointsTable

Object-Relational Features of Oracle

Collection types / VARRAYS

```
CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);
```

- The VARRAYs of type PRICES have no more than ten elements, each of datatype NUMBER(12,2).
- Creating an array type does not allocate space. It defines a datatype, which you can use as:
  - the datatype of a column of a relational table.
  - an object type attribute.

Object-Relational Features of Oracle

Type Inheritance / Subtyping

```
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;

CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER, major VARCHAR2(30)) NOT FINAL;

CREATE TYPE Employee_typ UNDER Person_t
( empid NUMBER, mgr VARCHAR2(30));

CREATE TYPE PartTimeStud_t UNDER Student_t
( numhours NUMBER);
```

To permit subtypes,
the object type must be defined as not final.

Object-Relational Features of Oracle

Method Overloading and Overriding

```
CREATE TYPE MyType_typ AS OBJECT (...,
  MEMBER PROCEDURE Print(),
  MEMBER PROCEDURE foo(x NUMBER), ...)
  NOT FINAL;

CREATE TYPE MySubType_typ UNDER MyType_typ
(...,
  OVERRIDING MEMBER PROCEDURE Print(),
  MEMBER PROCEDURE foo(x DATE), ...);
```

MySubType_typ contains two versions of foo( ): one inherited version, with a NUMBER parameter, and a new version with a DATE parameter

Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- **Data Base Recovery and Security**
- Object-Oriented Databases
- Inverted Index for IR
- XML
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases
- Other Advanced Database Topics

Recovery and the ACID properties

Atomicity: All actions in the transaction happen, or none happen.

Consistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions.

Durability: If a transaction commits, its effects persist.

- The **Recovery Manager** is responsible for ensuring two important properties of transactions: ***Atomicity*** and ***Durability***.
- Atomicity is guaranteed by undoing the actions of the transactions that did not commit (rollback aborted transaction).
- Durability is guaranteed by making sure that all actions of committed transactions survive crashes and failures.

Write-Ahead Log

- The update record must always be appended to the log before the database is updated. The log is referred to as a ***write-ahead log***.
- The Write-Ahead Logging Protocol:
  - ←Must force the log record for an update *before* the corresponding data page gets to disk.
  - ←Must write all log records for a transaction *before commit*.
  - #1 guarantees Atomicity.
  - #2 guarantees Durability.
- Exactly how is logging (and recovery!) done?

  We'll study the ARIES algorithms.

Checkpoints

- The system periodically appends a checkpoint record to the log that lists the current active transactions.
- The recovery process must scan backward at least as far as the most recent checkpoint.
- If T is named in the checkpoint, then T was still active during crash→ continue scan backward until *begin-transaction* T.

Possible Execution Strategies

- **Steal / No-force**

BM may have written some of the updated pages into disk. RM writes a commit

- Steal / force

BM may have written some of the updated pages into disk. RM issues a *flush* and writes a commit

- No-steal / no-force

None of the updated pages have been written. RM writes a commit and sends unpin to BM for all pinned pages.

- No-steal / force

None of the updated pages have been written. RM issues a *flush* and writes a commit

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted transaction?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all transactions that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all transactions that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Access Controls

- A security policy specifies who is authorized to do what.
- A security mechanism allows us to enforce a chosen security policy.
- Two main mechanisms at the DBMS level:
  - Discretionary access control
  - Mandatory access control
  - Role-based access control

GRANT Command

GRANT privileges ON object TO users [WITH GRANT OPTION]

- The following privileges can be specified:
  - SELECT: Can read all columns (including those added later via ALTER TABLE command).
  - INSERT(col-name): Can insert tuples with non-null or non-default values in this column.
    - INSERT means same right with respect to all columns.
  - DELETE: Can delete tuples.
  - UPDATE: Can update tuples.
  - REFERENCES (col-name): Can define foreign keys (in other tables) that refer to this column.
- If a user has a privilege with the GRANT OPTION, can pass privilege on to other users (with or without passing on the GRANT OPTION).
- Only owner can execute CREATE, ALTER, and DROP.
  - GRANT UPDATE (*rating*) ON Sailors TO Dustin
    - Dustin can update (only) the *rating* field of Sailors tuples.

Typical Security Classes

- Objects (e.g., tables, views, tuples)
- Subjects (e.g., users, user programs)
- Security classes:
  - Top secret (TS), secret (S), confidential (C), unclassified (U): TS > S> C > U
- Each object and subject is assigned a class.
  - Subject S can ***read*** object O only if class(S) >= class(O) (no reads in higher security)
  - Subject S can ***write*** object O only if class(S) <= class(O) (no writes in lower security)

Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- Concurrency Control
- **Data Base Recovery and Security**
- Object-Oriented Databases
- Inverted Index for IR
- XML
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases
- Other Advanced Database Topics

Transaction Properties

The acronym ACID is often used to refer to the four properties of DB transactions.

- **A**tomicity (all or nothing)
  - A transaction is *atomic*: transaction always executing all its actions in one step, or not executing any actions at all.
- **C**onsistency (no violation of integrity constraints)
  - A transaction must preserve the consistency of a database after execution. (responsibility of the user)
- **I**solation (concurrent changes invisible➔serializable)
  - Transaction is protected from the effects of concurrently scheduling other transactions.
- **D**urability (committed updates persist)
  - The effect of a committed transaction should persist even after a crash.

Scheduling Transactions

- *A Schedule* is a sequential order of the instructions (R / W / A / C) of *n* transactions such that the ordering of the instructions of each transaction is preserved. (execution sequence preserving the operation order of individual transaction)
- *Serial schedule:* A schedule that does not interleave the actions of different transactions. (transactions executed consecutively)
- *Non-serial schedule*: A schedule where the operations from a set of concurrent transactions are interleaved.

S1

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

S2

T1:	A=A+100, B=B-100	
T2:		A=1.06*A,B=1.06*B

Scheduling Transactions (continue)

- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A non-serial schedule that is equivalent to some serial execution of the transactions.

  (Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”: read an object modified by uncommited transaction.):

Aka:
Uncommitted Dependency
Dirty read problem

- T1 transfers $100 from A to B
- T2 adds 6% to A and B
- Avoid cascading aborts

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A),R(B),W(B), C	

Serializability

- The objective of *serializability* is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- It is important to guarantee serializability of concurrent transactions in order to prevent inconsistency from transactions interfering with one another.
- In serializability, the ordering of read and write operations is important (see conflict of operations).
- See the following schedules how the order of R/W operations can be changed depending upon the data objects they relate to.

Dependency Graph

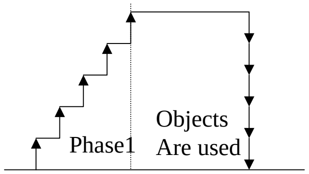
- *Dependency graph (or precedence graph)*:
  - One node per transaction;
  - edge from *Ti* to *Tj* if *Tj* reads/writes an object last written by *Ti*.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.

Strict Two-Phase Locking

- Transaction holds locks until the end of transaction (just before committing)



a.k.a.
Conservative 2PL

Deadlock Prevention

- Assign priorities based on timestamps (i.e. The oldest transaction has higher priority).
- Assume T_i wants a lock that T_j holds. Two policies are possible:
  - Wait-Die: If T_i has higher priority, T_i allowed to wait for T_j; otherwise (T_i younger) T_i aborts
  - Wound-wait: If T_i has higher priority, T_j aborts; otherwise (T_i younger) T_i waits
- If a transaction re-starts, make sure it has its original timestamp

Timestamping

- Each transaction is assigned a globally unique timestamp (starting time using a clock)
- Each data object is assigned
  - a write timestamp wts (largest timestamp on any write on x)
  - a read timestamp rts (largest timestamp on any read on x)
  - a flag that indicates whether the transaction that last wrote x committed.
- Conflict operations are resolved by timestamp ordering.
- A concurrency control protocol that orders transactions in such a way that older transactions get priority in the event of conflict.

Timestamp Ordering

- A Transaction T_i wants to read x: $R_i(x)$
  - if $ts(T_i) < wts(x)$ then reject R_i (x): rollback T_i (abort)
  - else accept $R_i(x)$; $rts(x) \leftarrow max(ts(T_i), rts(x))$

If $ts(T_i) < wts(x)$ => some other transaction T_k that started after T_i wrote a new value to x.
Since the read(x) of T_i should return a value prior to the write operation of T_k T_i is aboted (it is too old)

Timestamp Ordering

- A Transaction T_i wants to write x: W_i (x)
  - if $ts(T_i)<rts(x)$ then reject W_i (x): rollback T_i (abort)
  - if $ts(T_i)<wts(x)$ then ignore after accept $W_i(x)$ [Thomas write rule]
  - else accept $W_i(x)$; $wts(x) \leftarrow ts(T_i)$

If $ts(T_i) < rts(x)$ => some other transaction T_k that started after T_i has read an earlier value of x.
If T_i is allowed to commit, T_k should have read the new value that T_i is attempting to write. Thus T_i is too old to write.

Make sure a transaction has a new larger timestamp if it is re-started
This protocol guarantees serializability and is deadlock-free