

공간데이터 관리 및 응용 – Midterm

Ch01. SDBMS

기존의 **traditional non-spatial DBMS** 특징:

- 1) failure 간의 지속성
- 2) 데이터의 동시적 접근을 허용
- 3) MM에 맞지 않는 거대한 데이터셋에서 search query를 할 수 있는 확장성
- 4) non-spatial query에는 효율적, spatial query에는 비효율적
- 5) simple data type (ex) number, strings, date 를 지원함. (spatial data type 지원 X, 모델링하기 어려움)

Spatial Query? 위치 좌표 정보가 필요한 Query <-> Non-spatial query (좌표 정보 필요 X query)

ex) 미니애폴리스에서 10 마일 이내의 모든 서점 이름...

→ 좌표에 대한 data 질의 처리를 spatial query라고 함.

Non-spatial Data? 좌표가 필요가 없는.. (ex) textual data

ex) 이름, 전화번호, 이메일..

Spatial Data? 좌표 정보 필요함.

ex) Census Data (인구조사 데이터), Weather/Climate Data..

대표적인 spatial data 사용되는 예시 :

- Army Field Commander
- Insurance Risk Manager
- Medical Doctor
- Molecular Biologist
- Astronomer

SDBMS는 다음 특징을 갖는 **software module** :

- 다른 **underlying DBMS**와 함께 사용될 수 있다 (일반적으로 relational DB)
- **spatial data model**, **spatial abstract data type(ADT)**, **query language**들을 지원한다.
- **spatial indexing**, **spatial operation**을 처리하기 위한 **효율적인 알고리즘**, query optimization을 위한 **domain specific rule** 등을 지원한다.

ex) Oracle Spatial data cartridge, ESRI SDE

- o underlying DBMS로 relational DB를 활용 (Oracle 8i~11g DBMS)
- o **spatial data types** (ex) polygon, **operation** (ex) overlap 등을 지원하고, **SQL3 query language**로 호출 가능. (이때 SQL1,2 : 관계형 DB에서 쓰이는 QL, SQL3 : spatial ADT, operation들을 활용할 수 있도록 확장된 것)

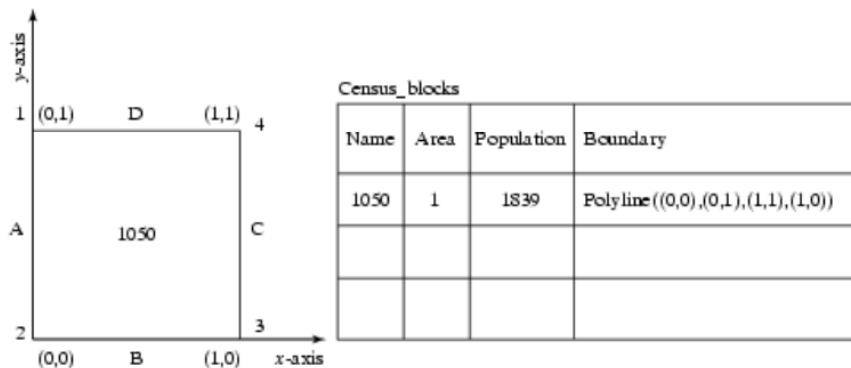
SDBMS 사용 예시)

spatial dataset을 SDBMS table에 저장할 때, table **census_blocks**을 생성.

```
create table census_blocks (
    name string,
    area float,
    population number,
    boundary polyline);
```

census_blocks : SDBMS에서 spatial data를 지원하는 table (non-spatial DBMS에서는 지원 X)

Polyline datatype을 활용한 DBMS (SDBMS, RDBMS에서는 불가..)



위의 사각형 1050 을 저장할때, SDBMS 에서는 꼭짓점 좌표들이 Polyline() 의 형식으로 저장된다. RDBMS 에서는 지원 x.

- 관계형 DB 는 atomic 해야한다.

위의 사각형 정보를 저장할 때, RDBMS 에서는 3 개의 table 에 거쳐서 16 개의 row 가 필요하다. (여러 table join 비효율적)

Census_blocks

Name	Area	Population	boundary-ID
340	1	1839	1050

Polygon

boundary-ID	edge-name
1050	A
1050	B
1050	C
1050	D

Edge

edge-name	endpoint
A	1
A	2
B	2
B	3
C	3
C	4
D	4
D	1

Point

endpoint	x-coor	y-coor
1	0	1
2	0	0
3	1	0
4	1	1

위와 같이 Census_blocks 의 한 사각형이 4 개의 Polygon (edge 들), Edge(꼭짓점), Point(x,y 좌표) table 들로 구성되어 join 되어야함.

DBMS 진화과정:

File System -> 1) Network DBMS , 2) Hierarchical DBMS -> Relational DBMS (기존의 관계형 DB, 활용성 높음)

Object-Oriented Systems (다양한 ADT 를 지원하지만 활용성은 떨어짐)

RDBMS + OODBMS -> 장점만 합쳐서 Object-Relational ORDBMS (ex Oracle 에서는 관계형 DB 가 main but object-oriented 도 지원)

Post-relational DBMS : 사용자가 지정하는 ADT 지원!! (Spatial datatype 도 추가 가능)

- Object Oriented DBMS (OODBMS : 객체 지향)
- **Object Relational DBMS (ORDBMS)**: 객체관계형/ 질의 처리가 간편해서 확실히 더 많이 쓰임) – 더 대중화됨.

GIS (Geographic Information System): 다양한 분석함수를 통해 spatial data 를 분석하고 시각화하는데 활용되는 software

- **GIS 가 large spatial data set 을 store, search, query, share 하기 위해 SDBMS 를 활용함** (단독으로 set 기반 data 를 관리하거나 set based operation 을 처리하기 어렵다)

GIS 는 application 이기 때문에, GIS 외에도 SDBMS 를 사용할 수 있는 app 들이 많이 있음 (전에 본 Astronomy, Genomics, ..)

Ch02. Spatial Concepts and Data Models

Data Model ? Specify **structure** or **schema** of data set.

- query ability, redundancy, consistency, storage space requirements.. 등을 초기분석함.
- 여러 application 들 사이에서 **reuse of shared data**.
- exchange of data across org.
- ex) GIS : spatial set 을 set of layer 로 정리함
- ex2) Database : dataset 을 collection of table 로 정리 (관계형 DBMS)

Data Model 두 가지 종류

1) Generic data models

- developed for business data processing
- support simple ADT.

- **spatial ADT** (ex) polygon) 는 사용하기 불편함. (확장성 필요 -> Application Domain specific)

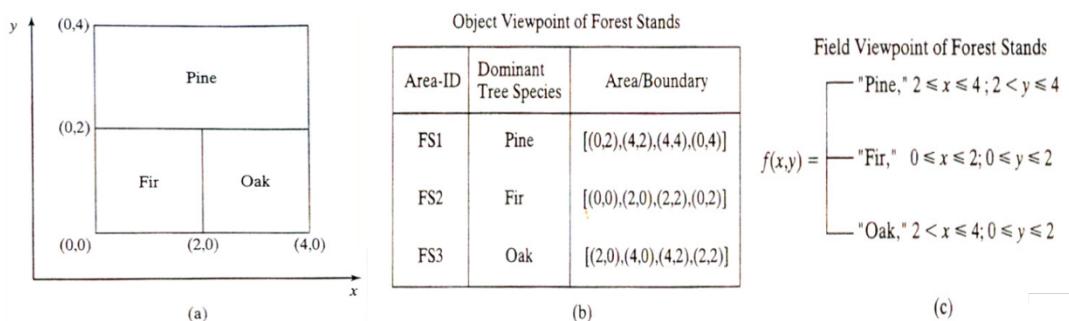
2) Application Domain Specific

- spatial 지원할 수 있도록 확장된 것.

Spatial Model 종류:

1) **Field based** : x, y 를 사용한 어떤 함수의 형태로 표현하는 것. (x, y 가 연속적인 형태일때 많이 사용) 아래 그림서 (c)

2) **Object based** : area 하나를 object(entity)로 보고 object화 시키는 것. 아래 그림서 (b)



Object Model:

- Object : 뚜렷하고 식별가능한 사물 (attribute 와 operation 을 가짐)
 - o **attribute** : simple property of object
 - o **operations** : object attribute 와 다른 object 를 mapping 하는 function
- ex) Roadmap 에서 objects : road, landmarks...
 - o Attribute :
 - spatial : location (polygon boundary) 좌표 정보 필요한 attribute
 - non-spatial : name (Route 66), type (interstate), ...
 - o Operation : determine center line, determine intersection with other roads...

Spatial Object 의 type

- **Simple** : 0 - dimensional (points), 1 - dimensional (curves/lines), 2 - dimensional (surfaces/polygon)
- **Collections** : polygon collection (Multisurface/Multicurve/Multipoint)
- **OGIS Data Model** 의 Spatial Object Type (관계)

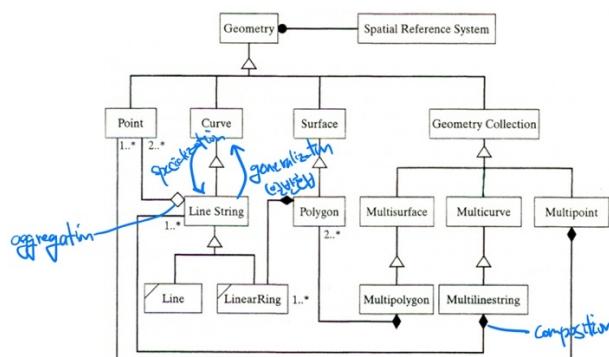


Figure 2.2. Each rectangle shows a distinct spatial object type

위의 OGIS Data Model 관계도에서 relationship:

점 : Geometry 는 Spatial Reference System 으로 **Defined**.

삼각형 :

- Geometry 는 Point, Curve, Surface, Geometry Collection 으로 **Specialization**.
- Point, Curve, Surface, Geometry Collection 의 **Generalization** : Geometry

사각형 : Point 의 **aggregation** 으로 Line String

검정마름모 : 여러 LineString 의 **Composition** : MultiLineString

Aggregation 과 Composition 의 차이?

Aggregation : 동일한 생명주기는 아님. (Person – Address)

Composition : 동일한 생명주기 (Car – Engine)

Spatial Object 의 다양한 Classifying Operations

- **Set based** : Set Operation (집합 관계)
- **Topological Operation**
- **Directional Operation** : 두 point 간 방향성
- **Metric Operation** : 가장 대표적인 metric 인 유클리디언 dist.

Set theory based	Union, Intersection, Containment,
Topological	Touches, <u>Disjoint</u> , Overlap, etc.
Directional	East, North-West, etc.
Metric	<u>Distance</u>

Topological Operation (위상 관련해서 T/F 를 return 하는)

- ** 공간을 비틀어도 (변형을 가해도) 변화 x
 - o ex) 평면 지도에서 touch 관계를 갖는 두 나라는 지구본 형태의 곡면상에서도 같은 관계

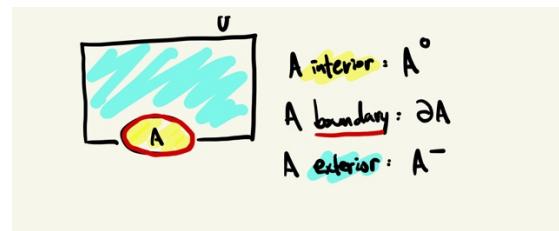
Topological

endpoint(point, arc)	A point at the end of an arc.
simple-nonself-intersection (arc)	A nonself-intersecting arc does not cross itself.
on-boundary (point, region)	Vancouver is on the boundary of Canada and the United States.
inside (point, region)	Minneapolis is inside of Minnesota state.
outside (point, region)	Madison is outside of Minnesota state.
open (region)	Interior of Canada is an open region (excludes all its boundaries).
close (region)	Carleton County is a closed region (includes all its boundaries).
connected (region)	Switzerland is a connected region, whereas Japan is not (given any two points in the area, it is possible to follow a path from one point to the other such that the path is entirely within the area).
inside (point, loop)	A point is within the loop.
crosses (arc, region)	A road (arc) passes through a forest (region).
touches (region, region)	Minnesota (region) is a neighboring state of Wisconsin (region).
touches (arc, region)	Interstate freeway 90 (arc) passes by Lake Michigan (region).
overlap (region, region)	Land cover (region) overlaps with land use (region).

Nontopological

Euclidean-distance (point, point)	Distance between two points.
direction (point, point)	Madison city is east of Minneapolis city.
length (arc)	Length of a unit vector is one unit.
perimeter (area)	Perimeter of a unit square is 4 units.
area (region)	Area of unit square is one square unit.

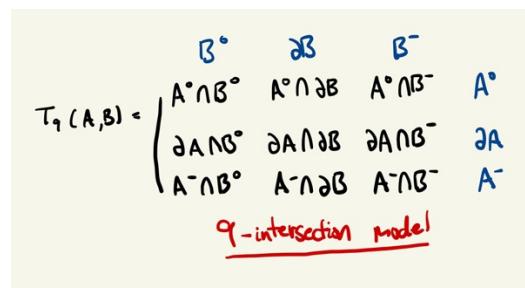
Topological Concepts



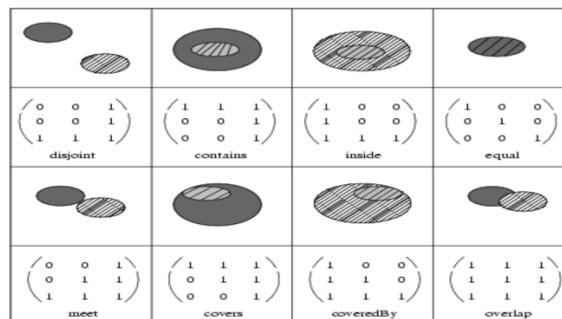
Define Interior, Boundary, Exterior on curves and points

- Line: interior : 정의 X, boundary : line, exterior : Universe - line
- Point : interior : 정의 X, boundary : 정의 X, exterior : 여매함 (open question)

9 – intersection Model



9 – intersection Model 을 통해 Topological Operation 정의 가능..



DB 설계 3 단계

- 1) **Conceptual–datatypes**, relationships, constraints (ER Model) 개념 설계
- 2) **Logical–mapping** to a relational model & associated QL (Relational Algebra)
- 3) **Physical–file Structures**

ER – Model :

- **Entities** : independent conceptual / physical existence (ex) Forest, Road, Manager)
- **Attributes** : Entities are characterized by attributes (ex) Forest has attributes of name, elevation..)
- **Relationships** : Entities interact with each other through relationships (ex) Roads access forest interiors)

ER – model does not permit **general user defined operation**. (object 내부 operation 허용하지 않는다)

→ Operation 을 통해 비슷하게 simulate

Concept	Symbol
Entities	□
Attributes	○
Multi-valued Attributes	○○
Relationships	◇
Cardinality of Relationship	1:1, M:1, M:N

Multivalued attributes : 대부분 spatial data (적어도 2 차원 이상)

Primary Key 속성 : underlined

Relationship Types :

Cardinality Constraints :

- One-One (1:1), Many-One (M:1), Many-Many (M:N)

관계 참여 entity 갯수 : unary, binary, ternary ..

Relational Model (Logical Data Model)

- Domains : set of values for simple attribute
- Relations : cross-product of set of domains (테이블 형태)

Conceptual Data Model 과 비교했을 때,

- Relations := entity
- Domains := attributes

Integrity Constraints :

- Key : Every relation has PK
- Entity Integrity : Value of PK 반드시 defined.
- Referential Integrity : Value of attribute of FK must be NULL or 유효값 (옳지 않은 값 불가).
- R 의 FK 값은 반드시 S relation row 의 값과 match

Normal Forms (NF)

: 중복성 (data redundancy)을 줄이고, 일관성을 유지하기 위해..

PK 가 아닌 것과 dependency 가 있으면 안된다!

ER -> Relational (Transition)

- Entity -> Relation
- Attribute -> Column in the relation
- Multi-valued attribute -> new relation
- Relationships (1:1, 1:N) -> Foreign Keys (별도의 relation 을 만드는 것이 아니다!)
- M:N 이면 별도의 relation 을 생성

Spatial Data (Point, Line, Polygon, Elevation) 에 대한 Relational Schema : 추가적인 table 이 필요함!

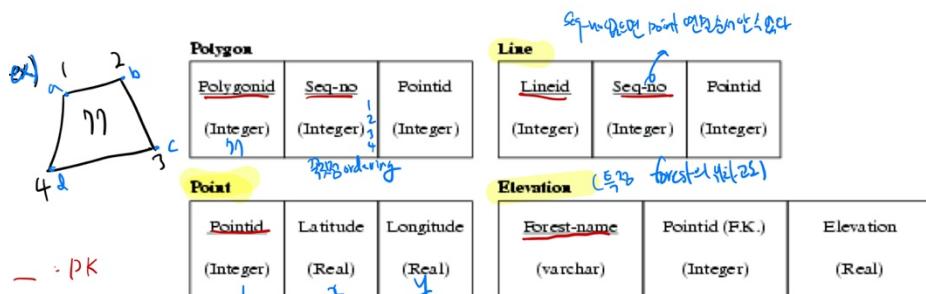


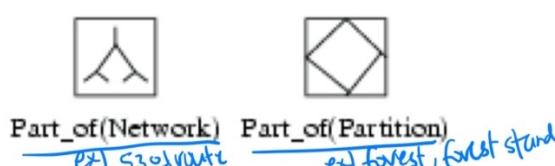
Figure 2.6. Schema for point, line, polygon, and elevation.

Meanwhile, 기존의 ER-Model 은 복잡하다..

- 암묵적인 관계는 포함 X
- Spatial data 는 implicit 관계들이 굉장히 많기 때문에, 기존의 ER 에는 문제점들이 많다..
 - o ex) spatial entity 간의 distance, direction 같은 relationship
 - o ER diagram 이 너무 난잡해지고,
 - o relational schema 에서 추가적인 table 이 필요하다.
 - o spatial relationship 의 implicit constraint 를 놓칠 수 있음.

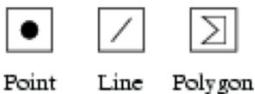
그렇기 때문에 이를 Pictogram 을 통해 확장.

- spatial relationship 제거 가능 (ex) cross within 삭제 가능), topological relationship 은 따로 표시 필요 X (explicit하게 존재)
- 1) relationship pictogram



2) Basic shapes

사각형 안에 다음 기호들로 표현



3) Cardinality : 0,1 1 1,n 0,n n

이런식으로 shape 과 같이 쓰임



4) Derived shapes : ex) city center point from boundary polygon

마름모 안에 Basic Shape 를 넣어서 표현



5) Alternate shapes : 한 가지 이상의 shape 로 표현할 수 있을 때

<basic shape> <derived shape> 혹은 <basic shape> <basic shape>으로 표현



Ch03. SQL

Query? DB 에게 하는 질문.

관계형 DB 에서

- SQL (Declarative Language) : what I want 형태
- Relational Algebra (Procedural Language) : How? 형태

Query Language :

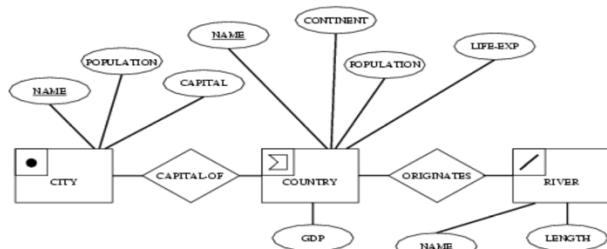
- Natural Language
- PL
- **Structured Query Language (SQL)**
 - o data intensive query 표현 좋지만,
 - o **SQL1,2: 관계형 DB 에 대한 SQL, recursive query** 에는 한계.
 - o **SQL3 : recursive OK, spatial 도 지원**

SQL 의 세가지 속성

1) **DDL** : Data Definition Language

2) **DML** : Data Manipulation Language

3) **DCL** : Data Control Language



위의 Extended ER 의 conceptual model 분석:

- 3 entity : country, city, river
- 2 relationship : captial-of (N:1(1:1 0| 아니다!!)) , originates-in (N:1)

Logical Model

3 Relations with PK, FK (1:N 은 별도의 relation 을 만들지 않고 FK(추가 속성)으로 표현)

- Country (Name, Cont, Pop, GDP, Life-Exp, Shape)
- City(Name, Country, Pop, Capital, Shape)
- River (Name, Origin, Length, Shape)

Many-to-one relation 에서 one 쪽의 PK 가 다른쪽의 FK 로.

SQL 연습

쉼표로 구분, ends with “;”

1) Creating Table

“CREATE TABLE” : table 생성 (no rows).

Table name, attribute names, data types 명시!!

ex)

```
CREATE TABLE RIVER (
    Name varchar(30),
    Origin varchar(30),
    Length number,
    Shape LineString );
```

cf) Related statements : ALTER TABLE (table schema 변경), DROP TABLE (empty table 삭제)

2) Table 에 data 삽입/수정/삭제

“INSERT INTO” : table 에 행 추가하기

Table name, attribute name, 추가할 value 명시해주기.

ex)

```
INSERT INTO River(Name, Origin, Length) VALUES ('Newyork', 'USA', 6000)
```

cf) Related statements: SELECT ~ INTO (table 에 여러 row insert 가능), DELETE (row 제거), UPDATE (선택한 row 의 값 변경)

3) Table Querying

“SELECT” : 검색

Returns a relation (table)

can refer to many operators, functions, allows nested queries

SELECT desired columns

- **FROM** : relevant tables
- **WHERE** : qualifying conditions for row
- **ORDER BY** : sorting columns for result
- **GROUP BY, (HAVING)** : aggregation and statistics (HAVING 은 GROUP BY 필수)

WHERE clause : Operators/functions \nmid condition 으로 올 수 있음

- 산술 연산자 ex) +, -, ...
- 비교 연산자 ex) =, >, <, BETWEEN, LIKE, ...
- 논리 연산자 ex) AND, OR, NOT, EXISTS, ...
- 집합 연산자 ex) UNION, IN, ALL, ANY, ...
- 통계 함수 ex) SUM, COUNT, ...
- 문자열, 날짜, 환율 관련 여러 연산 등..

ex)

a) Query : List all cities and the country they belong to
ans) SELECT Name, Country FROM CITY

b) Query : List the names of the capital cities in the CITY table
ans) SELECT * FROM CITY WHERE Capital = "Y"

c) Query : List the attributes of countries in the Country Relation where the life-expectancy is less than 70 years (Using alias)
ans) SELECT Co.Name, Co.Life-Exp FROM COUNTRY Co WHERE Co.Life-Exp < 70

d) Query : List the capital cities and populations of countries whose GDP exceeds one trillion dollars
ans) SELECT Ci.Name, Co.Pop FROM COUNTRY Co, CITY Ci WHERE Co.GDP > 1000 AND Ci.Capital = 'Y'

e) Query : What is the name and population of the capital city in the country where the St. Lawrence River originates?
ans) SELECT Ci.Name, Co.Pop FROM COUNTRY Co, CITY Ci, RIVER Ri WHERE Ri.Origin = Co.Name AND Co.Name = Ci.Country AND Ri.Name = "St.Lawrence" AND Ci.Capital = "Y"

f) Query : What is the average population of the noncapital cities listed in the City table?
ans) SELECT AVG(Ci.Pop) FROM CITY Ci WHERE Ci.Capital = "N"

g) Query: For each continent, find the average GDP (show average GDP in column-name : “Continent-GDP”)
ans) SELECT Co.Cont, AVG(Co.GDP) AS Continent-GDP FROM COUNTRY Co GROUP BY Co.Cont

h) Query : For each country in which at least two rivers originate, find the length of the smallest river. (show length in column-name : “Min-length”)

ans) `SELECT R.Origin, MIN(R.Length) as Min-length FROM River R GROUP BY R.Origin HAVING COUNT(*) > 1`

i) Query : List the countries whose GDP is greater than that of Canada.

ans) `SELECT C.Name FROM Country Co WHERE Co.GDP > ANY(SELECT Co1.GDP FROM Country Co1 WHERE Co1.Name = “Canada”)`

SQL3 (Spatial Data 확장)

1. Spatial Data Type 지원

ex) `CREATE TABLE Country(`
 `Name varchar(30),`
 `....`
 `Shape Polygon);`

2. Spatial Operation 지원

TABLE 3.9: A Sample of Operations Listed in the OGIS Standard for SQL [OGIS, 1999]

Basic Functions	<code>SpatialReference()</code>	Returns the underlying coordinate system of the geometry
	<code>Envelope()</code>	Returns the minimum orthogonal bounding rectangle of the geometry
	<code>Export()</code>	Returns the geometry in a different representation
	<code>IsEmpty()</code>	Returns true if the geometry is a null set
	<code>IsSimple()</code>	Returns true if the geometry is simple (no self-intersection)
	<code>Boundary()</code>	Returns the boundary of the geometry
Topological/ Set Operators	<code>Equal</code>	Returns true if the interior and boundary of the two geometries are spatially equal
	<code>Disjoint</code>	Returns true if the boundaries and interior do not intersect
	<code>Intersect</code>	Returns true if the geometries are not disjoint
	<code>Touch</code>	Returns true if the boundaries of two surfaces intersect but the interiors do not
	<code>Cross</code>	Returns true if the interior of a surface intersects with a curve
	<code>Within</code>	Returns true if the interior of the given geometry does not intersect with the exterior of another geometry
	<code>Contains</code>	Tests if the given geometry contains another given geometry
	<code>Overlap</code>	Returns true if the interiors of two geometries have nonempty intersection
Spatial Analysis	<code>Distance</code>	Returns the shortest distance between two geometries
	<code>Buffer</code>	Returns a geometry that consists of all points whose distance from the given geometry is less than or equal to the specified distance
	<code>ConvexHull</code>	Returns the smallest convex geometric set enclosing the geometry
	<code>Intersection</code>	Returns the geometric intersection of two geometries
	<code>Union</code>	Returns the geometric union of two geometries
	<code>Difference</code>	Returns the portion of a geometry that does not intersect with another given geometry
	<code>SymmDiff</code>	Returns the portions of two geometries that do not intersect with each other

Simple examples)

a) Query : List the name, population, and area of each country listed in the Country table

ans) `SELECT Co.Name, Co.Pop, AREA(C.Shape) AS “AREA” FROM Country Co`

b) Query : List the GDP and the distance of a county’s capital city to the equator for all countries

ans) `SELECT Co.GDP, Distance(Point(Ci.Shape.x, 0), Ci.Shape) AS “Distance” FROM COUNTRY Co, CITY Ci WHERE Co.Name = Ci.Name AND Ci.Capital = “Y”`

c) Query : Find the names of all countries which are neighbors of the United States in the Country table

ans) `SELECT C1.Name FROM Country C1, Country C2 WHERE Touch (C1.Shape, C2.Shape) = 1 AND C2.Name = “USA”`

d) Query : For all the rivers listed in the River table, find the countries through which they pass.

ans) `SELECT R.Name, C.Name FROM River R, Country C WHERE Cross (C.Shape, R.Shape) = 1`

e) Query : The St. Lawrence River can supply water to cities that are within 300km. List the cities that can use water from the St. Lawrence.

ans) `SELECT C.Name FROM City C, River R WHERE Overlap(C.Shape, Buffer(R.Shape, 300)) = 1 AND C.Name = “St.Lawrence”`

Complex SQL examples) Aggregate queries, Nested Queries

a) Query : List all the countries, ordered by number of neighboring countries.

ans) `SELECT C1.Name, Count (C2.Name) FROM Country C1, Country C2 WHERE Touch(C1.Shape, C2.Shape) = 1 GROUP BY C1.Name ORDER BY Count(C2.Name)`

b) Query: For each river, identify the closest city.

ans) `SELECT C1.Name, R.Name FROM City C1, River R WHERE Distance(C1.Shape, R.Shape) <= ALL (SELECT Distance \(C2.Shape, R1.Shape\) FROM City C2 WHERE C1.Name <> C2.Name)`

-> 여기서 <> 은 “제외한”

c) Query: Find the smallest distance from each river to nearest city

ans) `SELECT R1.Name, MIN(Distance(R1.Shape, C1.Shape)) FROM River R1, City C1 GROUP BY R1.Name`

-> Nested Query 를 써야했던 b)에 비해 c)가 오히려 SQL 에서는 훨씬 간편함!

d) Query : List the countries with only one neighboring country. A country is a neighbor of another country if their land masses share a boundary. According to this definition, island countries like Iceland, have no neighbors.

ans) `SELECT Co.Name FROM Country Co WHERE Co.Name IN (SELECT C1.Name FROM Country C1, Country C2 WHERE Touch(C1.Shape, C2.Shape) GROUP BY C1.Name HAVING Count(*) = 1)`

ans2) `SELECT Co.Name FROM Country Co, Country C1 WHERE Touch(Co.Shape, C1.Shape) GROUP BY Co.Name HAVING Count(C1.name) = 1`

Nested Query 의 경우 Views 를 사용할 수 있음.

(Views : SQL 로 나온 결과, table 에 저장되는 것이 아니지만 이 결과를 access 할 수 있음)

ex) Query : Which country has the maximum number of neighbors?

ans)

“Neighbor” 뷰 생성

```
CREATE VIEW Neighbor AS
SELECT Co.Name, Count(Co1.Name) AS num_neighbors
FROM Country Co, Country Co1
WHERE Touch(Co.Shape, Co1.Shape)
GROUP BY Co.Name

SELECT Co.Name, num_neighbors
FROM Neighbor
WHERE num_neighbors = (SELECT Max(num_neighbors) FROM Neighbor)
```

Ch04. Spatial Access Methods 1

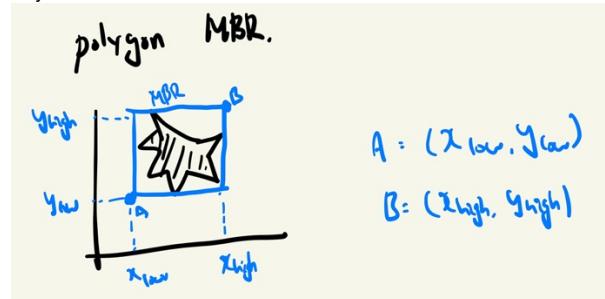
Spatial Query : 복잡하고 시간이 많이 소요되는 geometric operation 필요

SAMS (Spatial Access Methods)

spatial index 구조를 사용 (효율적)

직접 object geometry 의 indexing 대신 simple approximation 인 mbb (minimum bounding box) / mbr (minimum bounding rectangle)

Object MBB



- Range check 빠르게 (index traversal 의 cost 감소)
- constant-size entry 사용 가능

SAMS에서 Spatial Index 는 [mbb, oid]의 형태를 갖는 entry 의 collection 으로 built

- oid : object 의 id. oid 를 통해서 해당 object 의 physical page 접근 가능

mbb 로 indexing 된 collection of object 에 적용하는 operation 은 다음 두 가지 step 을 거쳐서 수행됨

1) filter step : spatial predicate 을 만족하는 mbb 를 갖는 object 를 선택함.

- index traverse, mbb 에 spatial test, 만족하는 oid set return

2) refinement step : mbb 는 만족하지만 실제 exact geometry 가 spatial predicate 에 만족하는지 확인 -> 걸러냄.

- filter step 을 통과한 object 들을 순차적으로 스캔하여 실제 object 의 geometry 에 대해 spatial testing.

- filter step 은 통과하지만 refinement step 을 통과하지 못하는 object 들 : false drops

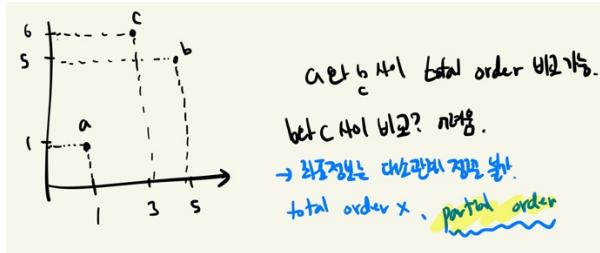
전통적인 DBMS 에서 B+ tree 와 hash table 을 사용하는 것이 일반적인 접근 방법.

cf) B+ - tree : Key 를 기준으로 작거나 같은 것 : 왼쪽 child node, 큰 것 : 오른쪽 child node 에 저장. [key, ptr] 형태의 entry 를 저장.

Time Complexity : logarithmic. Space Complexity : 실제 index 저장보다 절반은 절약 가능. Dynamic Update : 최적화된 구조 유지.

전통적인 DBMS

- I/O 횟수 logarithmic 으로 보장.
- B+ 트리 relies on key domain 의 **total order** (어떤 domain 영역에서 임의의 두 수를 꺼내면 순서가 있다/ 대소관계가 반드시 존재함..) BUT spatial data 의 경우 total order 가 보장되지 않음 : **partial order**



Geometric object 의 순서를 정할 때 편한 방법 : **Object Proximity** 를 유지할 수 있는 것.

- Object Proximity : 가까이 있는 object 는 index 구조에서도 가까이 있어야 한다.

Index Representation (simple, robust, efficient)

- 1) Grid
- 2) R - tree

둘 다 index construction, search operation 고려 가능해야 함.

SAM 기본 가정 :

- collection size >> 메인 메모리의 available space (disk 에 저장해야 함)
- disk access time > random access main memory
- Page 단위로 memory <-> disk 사이의 transfer

SAM 목표 :

- Time Complexity : sublinear time (ex $O(\log N)$) 예) exact(point) search, range(window) search 만족
- Space Complexity : n page \rightarrow index size : $O(n)$
- Dynamicity (역동성) : 삽입/삭제 시에도 index 구조에 큰 변화가 생기면 안됨.

Space – driven Structure (Grid) vs Data – Driven Structure (R-tree 계열)

1) Space-driven Structure

Fixed Grid, Grid File, linear structures methods. 공간 partitioning

2) Data-driven Structure

R-tree, R* tree, R+ tree. 공간 먼저 분할이 아닌 object 들을 partitioning, object 의 분포에 따라 변형.

위의 SAM 과 같은 다차원 접근 방법들은 둘 다 robustness 는 부족함.. (spatial data 특징 때문)

\Rightarrow worst case 성능을 보장하지 않는다. (B+ tree 는 worst case 에 대한 성능 보장함)

1) Space – driven Structure : The Grid

1-1) Fixed Grid

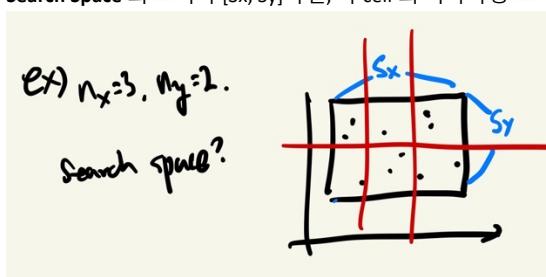
고정된 크기의 cell 에 search 공간이 분해됨

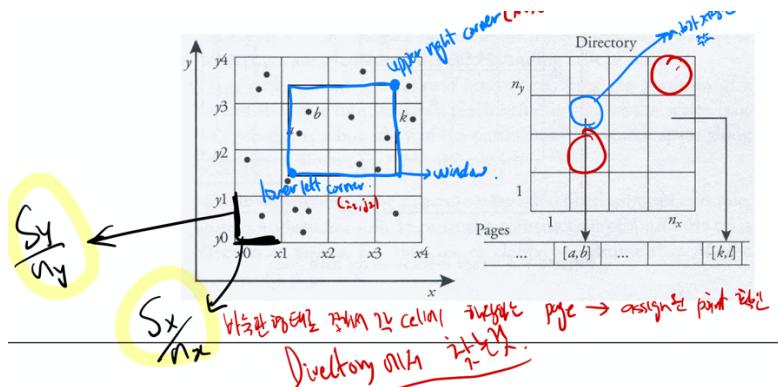
각 cell 은 disk page 와 연관되어 있고, 만약 해당 c.rect (cell rectangle) 에 점 P 가 포함되어 있다면, P assigned to c.rect.

Directory 로 2D array DIR [1:n_x, 1:n_y] 가 필요함.

- 각 DIR[i,j] 은 cell c_{ij} 에 할당된 점들이 저장된 page 의 PageID 주소가 저장되어 있음.

Search Space 의 크기가 [S_x, S_y]라면, 각 cell 의 직사각형 크기 : [S_x/n_x, S_y/n_y]

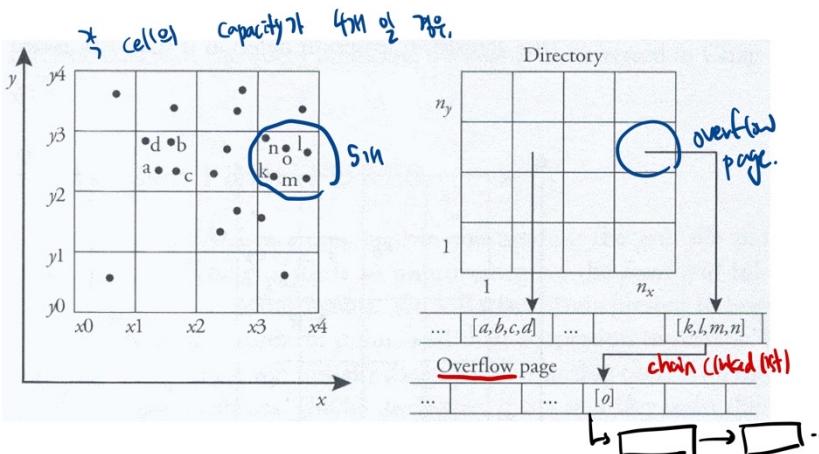




Fixed Grid에서,

- Inserting $P(a,b)$:
 - o $i = (a - x_0) / (S_x / n_x) + 1$, $j = (b - y_0) / (S_y / n_y) + 1$ 계산
 - o page $DIR[i,j].PageID$ 를 확인 (점 P 가 저장되어야 할 Directory 주소) \rightarrow P 삽입
- Point Query:
 - o Get page of Point $P(a,b)$ \rightarrow Read the page for $P(a,b)$ \rightarrow Scan all the entries \rightarrow check if one is P .
 - o Page 를 한 번 읽고, entry scan 후 확인하기 때문에 Single I/O => Efficient
- Window Query
 - o 위의 그림처럼 window 를 만들어 lowerleft, upper-right 좌표로 모든 disk page 의 데이터들을 가져와서 확인
 - o window query 의 I/O 횟수는 window 와 intersect 하는 cell 의 갯수에 비례함. => Efficient 하다고 보기 힘들다.
- Grid Resolution :
 - o indexing 이 필요한 점의 갯수 N 에 의존적.
 - o page capacity 가 M 일 경우, fixed grid 의 cell 갯수는 최소한 N/M .
 - o Page Capacity 가 정해져있을 경우, cell 에 M 개 이상 지정되는 경우 overflow 가 생긴다!
 - overflow page 들이 linked list 형태로 생성된다.

overflow 가 생기면? 다음과 같이 linked list 형태로 overflow page 들이 연결됨.



Fixed grid 의 search space에서 point 들이 균일하게 분포되어 있다면, overflow 가 적게 일어남 => overflow chain 짧음.

overflow chain 길어질 수록, I/O 발생 커짐 (효과적 x)

** 데이터가 균일하게 분포되면 fixed grid 효율적.

1-2) The Grid File

Fixed Grid 방식이 수식에 의해 균일하게 하는 것이었다면, Grid file은 cell이 다른 사이즈를 갖고, partition 분할이 분포 상태에 따라 달라짐. (Fixed Grid의 단점 보완)

Cell overflow 가 생기면, cell이 두 개로 분할되어 point 가 들어가야 할 cell로 할당됨.

다음 세가지 data structures:

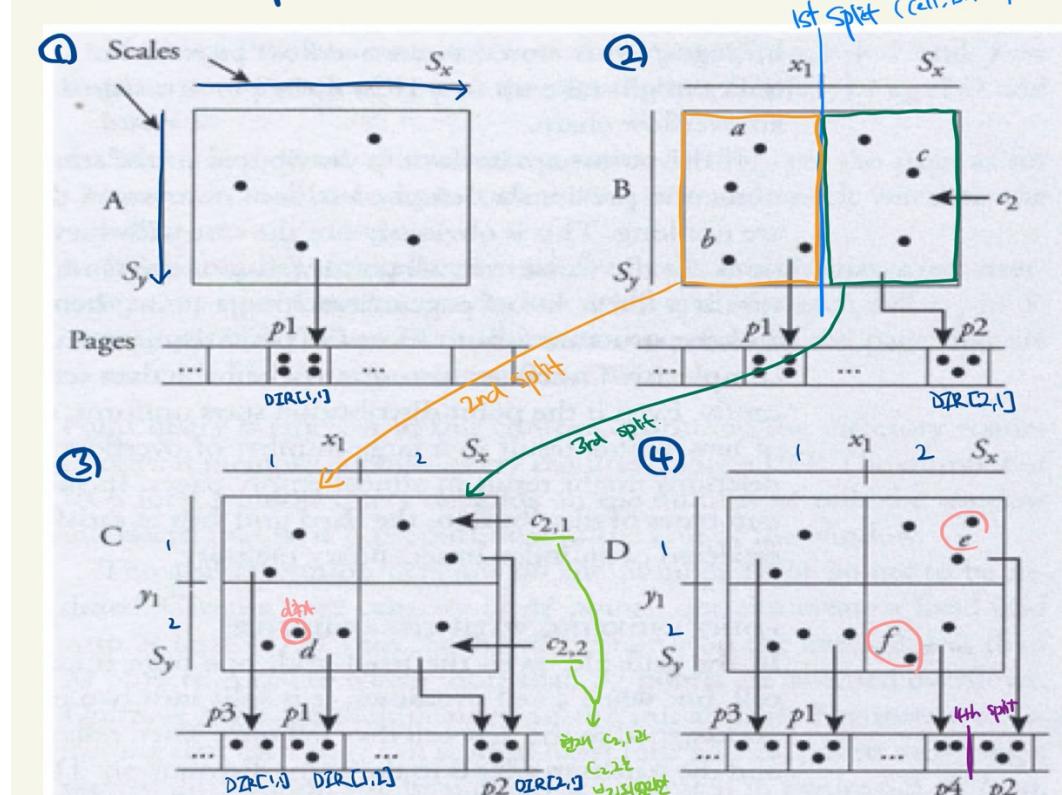
- Directory DIR (fixed grid 때와 동일)
- Scale Vectors S_x, S_y .
 - o 각 축 좌표의 구간별 partition 을 나타낼 수 있는 Linear Arrays (1 차원 배열)
 - o 각 scale의 각 값은 해당 차원의 탐색 구간의 partition boundary 를 의미함

Point Indexing with Grid File

- Insertion :
 - o No Cell Split (Directory Split, No Split 으로 또 나눌 수 있긴 함)
 - o Cell Split & No Directory Split
 - o Cell Split & Directory Split
- Point Search 는 반드시 2 번의 disk access 를 통해 수행됨
 - o directory 의 page p 를 접근하는데 한번, p 의 한 entry 와 연관된 data page 접근하는데 한 번
- 유동적인 구조(dynamic), 공간 활용성이 합리적이다.
- ⇒ fixed grid 의 한계를 보완함!!! (dynamic, disk access)
- 큰 dataset 에 대해서는 main memory 와 directory size 가 맞지 않을 수도 있음.

Fixed grid는 grid cell 크기로 overflow 체인.
Grid file은 scale을 쓰는.

if cell capacity가 4이면. (Grid File)



① 1st split: cell, directory split.
② 2nd split: cell, dir split
③ 3rd split: no cell split, dir split.
④ 4th split: cell split, no dir split (data overflow)
cf 추가. 더 이상 같은 p2에 저장 불가

①: $S_x: [1 2 3 4 \dots]$
 $S_y: [\infty \ \ \ \ \ \dots]$

②: $S_x: [x_1 \infty \ \ \ \ \dots]$
 $S_y: [\infty \ \ \ \ \ \dots]$

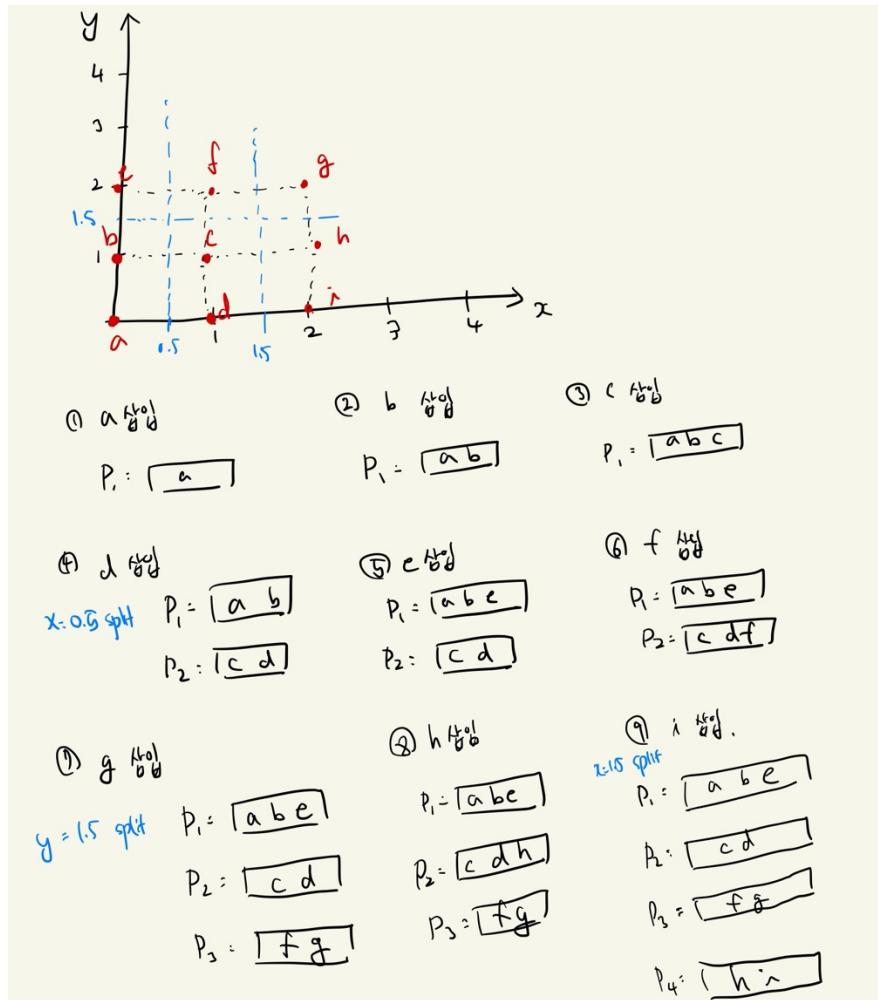
③: $S_x: [x_1 \infty \ \ \ \ \dots]$
 $S_y: [y_1 \infty \ \ \ \ \dots]$

위에서 S_x, S_y (Scale Vector)들을 통해 DIR 주소 찾는 법?

ex) (2)에서 c 를 포함하는 directory 를 찾을 때? c 좌표가 x_1 보다 큼 $\rightarrow 2$ 에 있다! \Rightarrow DIR [2,1]

Grid File 연습. $(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,2) \rightarrow (1,2) \rightarrow (2,2) \rightarrow (2,1) \rightarrow (2,0)$ 순차적으로 삽입. 한 개의 cell 은 capacity of 3.

각 점들을 a,b,c,d,e,f,g,h, i 를 표현.



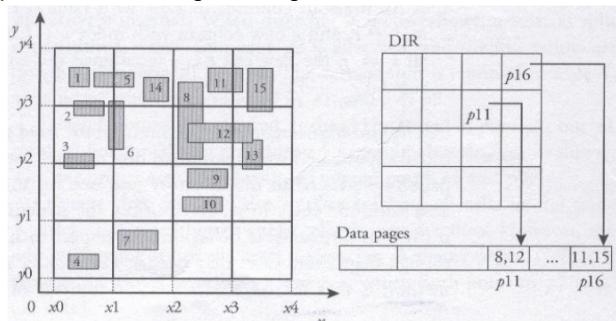
Split 최적화 알고리즘도 존재하지만, 지금 단계에서는 이 정도로..

지금까지 Grid File Point Indexing 을 확인했음.

Ch05. Spatial Access Methods 2 – Rectangle Indexing

Grid File : Rectangle Indexing

1) Fixed Grid for rectangle indexing



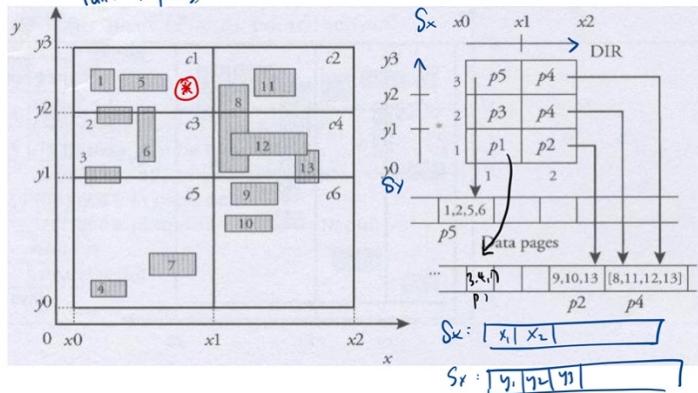
이때 위 사진에서 1,2,3,... 직사각형 : 안에 polygon (다각형) 을 둘러싸는 최소한의 직사각형 (MBR)

Disk Page p11, p12, p16 을 확인해보면 각각 [8,12], [12, 13, 15], [11,15] 가 저장되어 있음.

→ Point 저장과는 달리 MBR 을 저장하는 Rectangle Indexing 에서는 중복 저장 문제가 존재한다.

2) Grid File for rectangle indexing

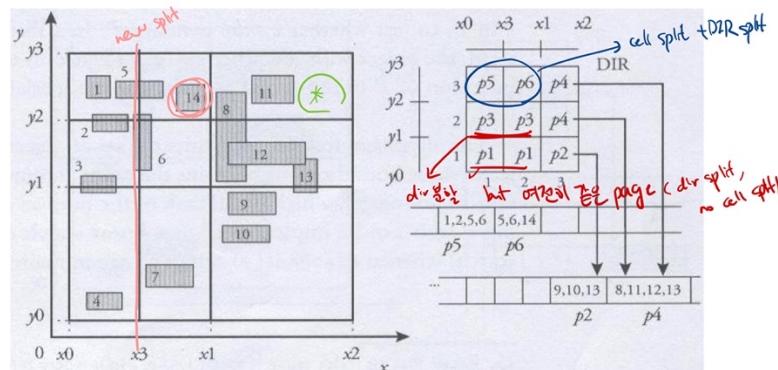
full: 4 points.



위 사진과 같이 삽입때마다 분할하는 방식 (Point Indexing 과 유사)

→ x 와 y 중 even하게 나눌 수 있는 쪽으로 분할 (not always)

만약 위 사진의 빨간색 (*) 부분에 새로운 object 14 가 삽입된다면?



만약 위 사진의 초록색 (*) 부분에 새로운 object 15 가 삽입된다면?

ans) 이미 DIR split은 되어있다. cell 만 추가로 split 됨. (p4, p4) -> (p4, p7)

1. Grid File Point Query Algorithm

GF-POINTQUERY ($P(a, b)$: point): set(oid)

```

begin
  result = ∅
  // Compute the cell containing P
  i = RANK(a, Sx); j = RANK(b, Sy)
  page = READPAGE (DIR[i,j].PageID)
  for each e in page do
    if ( $P \in e.mbb$ ) then result += {e.oid}
  end for
  return result
end

```

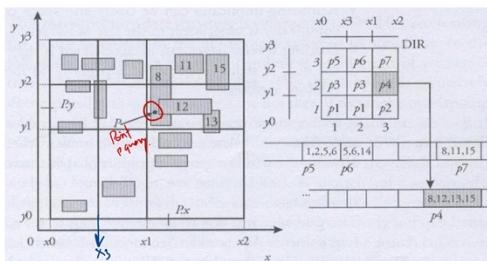
Point P가 들어온 때 정복 처리.

1) P 가 포함된 DIR 의 좌표 i,j 를 구한다. Rank(a, Sx), Rank(b, Sy)

2) 구한 i,j 를 통해 DIR[i,j].PageID 에 해당하는 Page 주소를 읽는다.

3) 읽는 page 의 entry 들을 모두 확인해서 해당 entry 의 mbb 에 Point P 가 포함된다면 결과에 추가

실제 예시) 다음 예시에서 빨간색 점 Point P Query



P의 좌표 $P(x_p, y_p)$ 존재

1) Scale 의 rank 와 좌표 비교

S_x	x_3	x_1	x_2	$ $	$ $
S_y	y_1	y_2	y_3	$ $	$ $

Scale:

$x_1 < x_p < x_2$: rank 3, $y_1 < y_p < y_2$: rank 2 $\Rightarrow (i = 3, j = 2)$

2) DIR[3,2] 에 해당하는 cell 주소 \rightarrow Page 접근하여 Read. Read p4

3) p4 안의 데이터를 하나씩 확인.

Point p 와 8 비교 8에 포함됨! result에 추가. 12 비교 : X, 13: X, 15: X

$\Rightarrow result = 8$

2. Grid File Window Query Algorithm

Point P 가 아니라 Window W에 대한 질의처리

```
GF-WINDOWQUERY ( $W(x_1, y_1, x_2, y_2)$ : rectangle): set( $oid$ )
begin
    result =  $\emptyset$ 
    // Compute the low-left cell intersecting W
     $i_1 = \text{RANK}(x_1, S_x)$ ;  $j_1 = \text{RANK}(y_1, S_y)$ 
    // Compute the top-right cell intersecting W
     $i_2 = \text{RANK}(x_2, S_x)$ ;  $j_2 = \text{RANK}(y_2, S_y)$ 
    // Scan the grid cells
    for ( $i = i_1$ ;  $i \leq i_2$ ;  $i++$ ) do
        for ( $j = j_1$ ;  $j \leq j_2$ ;  $j++$ ) do
            // Read the page, and test each entry
            page = READPAGE (DIR[i,j].PageID)
            for each (e in page) do
                if ( $W \cap e.mbr \neq \emptyset$ ) then result += {e.oid}
            end for
        end for
    end for
    // Sort the result, and remove duplicates
    SORT (result); REMOVEDUPL (result);
    return result
end
```

Window et mbr. 중복제거

↑ 찾은 rank cell을 정복 확인

↑ 정복하는 것 확인 \Rightarrow result

1) Window Query 와 overlap 되는 모든 cell 연산

- lower left point 의 i_1, j_1

- upper right point 의 i_2, j_2

2) 찾은 rank cell 들 내의 모든 page 들을 확인 \rightarrow 겹치는 mbr 들 모두 확인

3) 정렬 후 중복 제거 \Rightarrow result

I/O 횟수 : window area에 대해 비례함.

중복 제거가 costly, space consuming : Time complexity is superlinear to size of result

- 이웃 cell에서 **object duplication** : index의 entry 갯수가 증가.. \Rightarrow index size 증가
- result의 크기가 클수록 중복 제거 cost가 커진다..

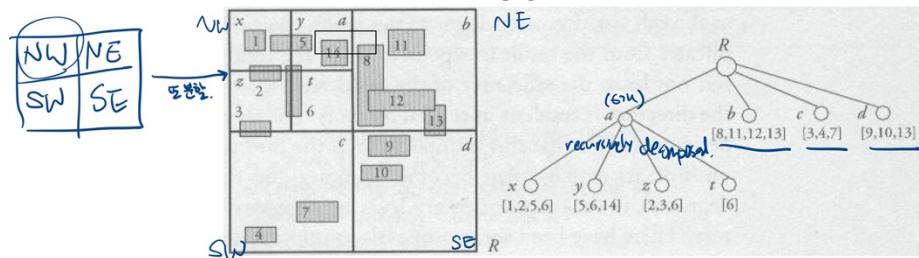
SAM의 효율성은 directory가 중앙메모리에 존재한다는 가정하에 이루어짐.

- 따라서 너무 큰 dataset은 SAM 불가.
- directory가 너무 커져서 디스크에 일부 존재해야한다면 굉장히 복잡해짐..

Ch06. Spatial Access Methods 3 – Linear Quadtree

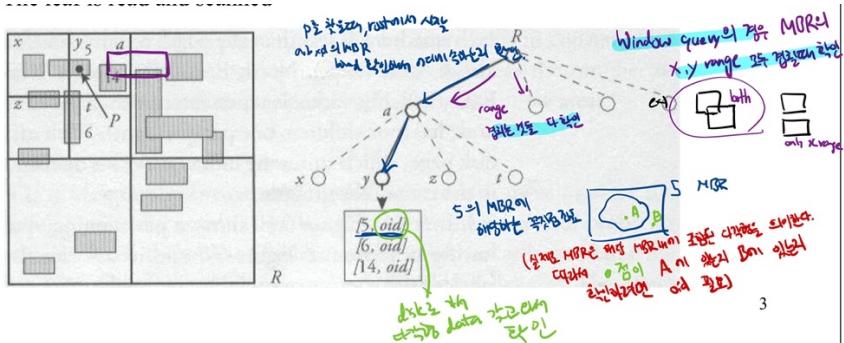
Quadtree : 탐색 구간을 네개의 사각형으로 page capacity 보다 각 rectangle 갯수가 작을때까지 recursively 분할. (until : # of rectangle overlapping quadrant <= page capacity)

이때, 분할된 quadrant는 NW, NE, SW, SE로 불린다.



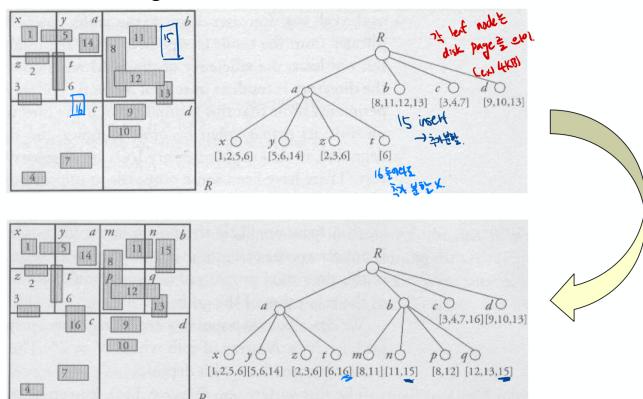
Quadtree에서 leaf들은 각각 disk page를 가리킨다.

Quadtree에서 Point Query / Window Query :



- Point/Window Query : Root에서 시작해서 각 level에서 four quadrant 중 MBR range를 확인해서 어느 child node에 속하는지 확인.
- Window의 경우 x, y range가 모두 겹쳐야 한다(형광색)
- Leaf node 까지 도착하면, 해당 leaf node가 가리키는 oid를 통해 해당하는 disk 주소를 접근해서 다각형 data들을 모두 갖고 와서 확인한다.

Quadtree rectangle 삽입



간단한 방식 : data를 넣으면서 capacity 인 4 개보다 더 들어간다면 추가로 분할.

Quadtree의 단점:

- children의 갯수가 4로 고정.
- Quadtree를 diskpage로 mapping하는 것이 쉽지 않음 (공간 활용도가 떨어진다..)
 - o B-tree나 R-tree의 경우 공간 활용도는 높다..
- Quadtree의 Query time은 tree depth와 연관되는데, 쉽게 커질수 있다..
 - o (Height balanced가 아니다 => 검색 속도 일정 X) node가 더 깊숙히 있을수록 오래걸림..
- 중복 저장 문제 (High Duplication Rate)

Space-filling curves:

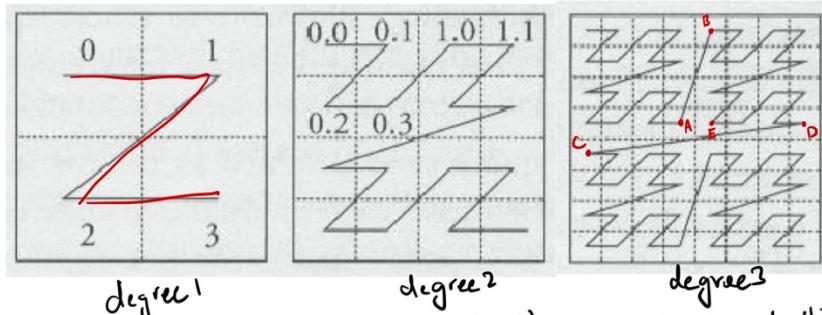
좌표가 있는 데이터는 total order 로 정렬이 불가능하다.

- ⇒ space filling curve 는 공간을 채우는 curve 로, 2 차원 좌표에서 total ordering 을 인위적으로 하는 것!
- ⇒ 이를 통해 Linear Quadtree 를 구성할 수 있다.

대표적인 방법으로

1) Z-order (z-ordering)

z 모양으로 lexicographical order 로 cell 정렬 가능.



- degree n : 4^n 개의 grid cell
- 각 cell segment 는 z 모양으로 0, 1, 2, 3 순서고, degree 가 증가할 수록 각 세그먼트상의 순서가 ‘.’ 뒤에 추가됨.
 - o ex) degree3 에서는 0.0.1, 0.0.2, ...

z-order 의 문제점?

위의 degree3 사진에서 A 점과 B 점의 z-order 순서는 붙어있지만, 실제로는 멀리 떨어져있음.

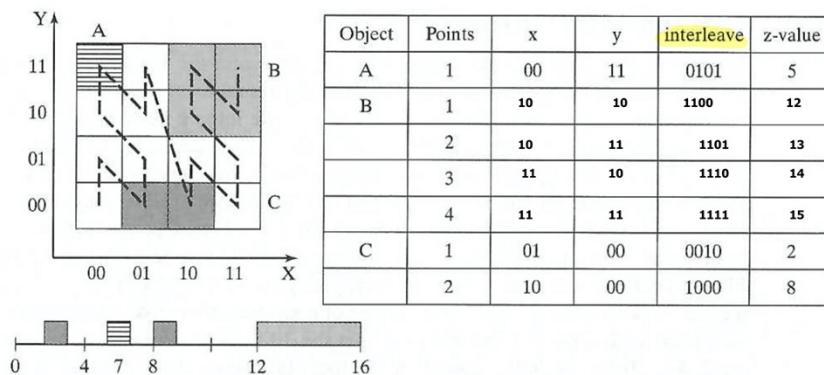
C, D 점 역시 z-order 상에서는 연속하지만, 실제 위치는 멀리 떨어져있다.

A 와 E 의 경우 실제로는 가깝지만, z-order 상에서는 멀리 있음.

- ⇒ 즉, z-order 순서와 공간적 거리상 순서가 다른 경우가 생긴다.

Object 의 z-value 찾기!

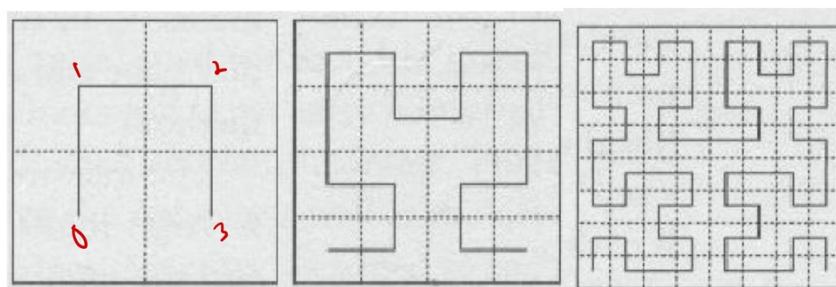
- 1) object 의 x,y 좌표를 binary number 로 표시
- 2) x,y 의 binary 값을 interleave 하여 bitstring 으로 표현 (interleave : x, y 한 비트씩 연결)
- 3) 이 bitstring 을 다시 decimal 로 변환



(Hilbert curve 보다 time cost 가 적어서 더 많이 쓰인다!)

2) Hilbert curve

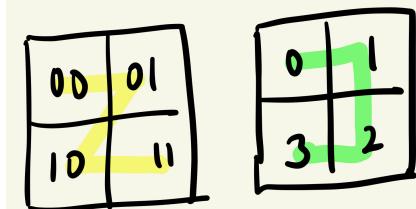
Z-order 처럼 다음과 같이 total ordering 을 하는 방식. (아래의 그림들처럼..degree 에 따라 변화..)



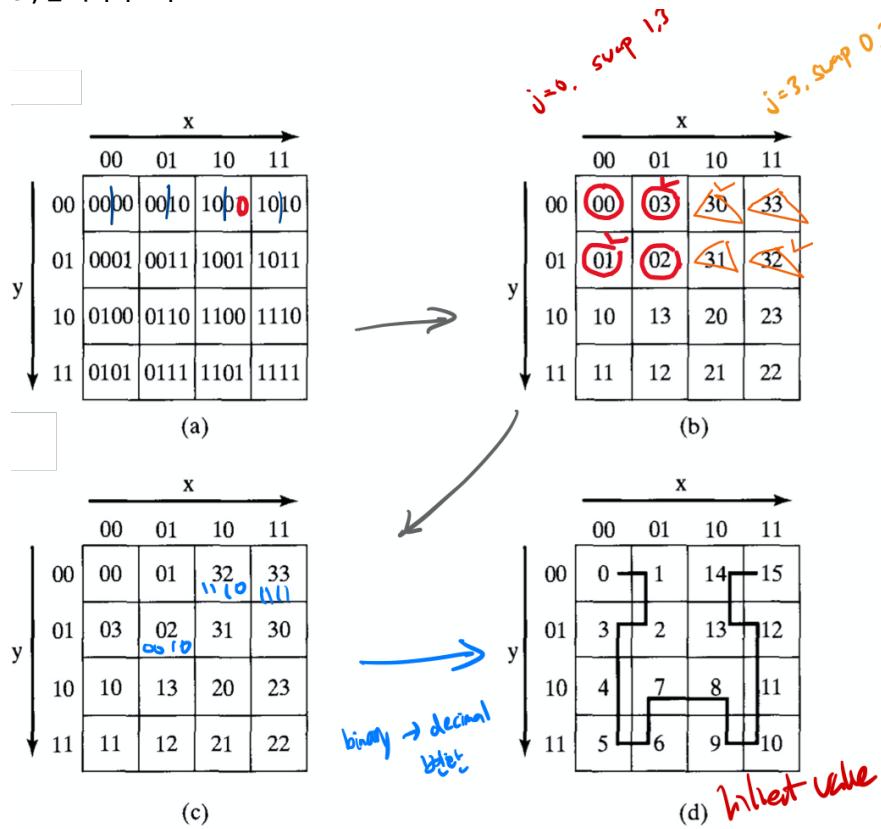
Z-order 와는 달리 Hilbert Curve 에서는 일단 Hilbert curve 로 연속된 점들에 대해서는 실제 공간적인 거리도 항상 가깝다.
 하지만, Hilbert Curve 또한 Z-order 와 같이 실제 공간상 거리는 가깝지만 Hilbert curve 상에서는 멀리 있는 경우가 존재함 (필연적)

Object 의 Hilbert-value 찾기!

- 1) x,y 좌표의 n-bit binary 표현 읽기
- 2) 역시나, x 와 y 를 의미하는 두 binary 값을 interleave 하여 bit string 으로 나타낸다.
- 3) bit string 을 왼쪽부터 2 bit 씩 나눈다.
- 4) 해당 2 bit string 에 대해 decimal 로 변환하는데, 일반적인 binary \rightarrow decimal 변환이 아니라, “00” \rightarrow 0, “01” \rightarrow 1, “10” \rightarrow 3, “11” \rightarrow 2
 -> why? 다음 그림과 관련 있음.. (Z order 상의 순서와 Hilbert Curve 상의 순서가 다르다..)



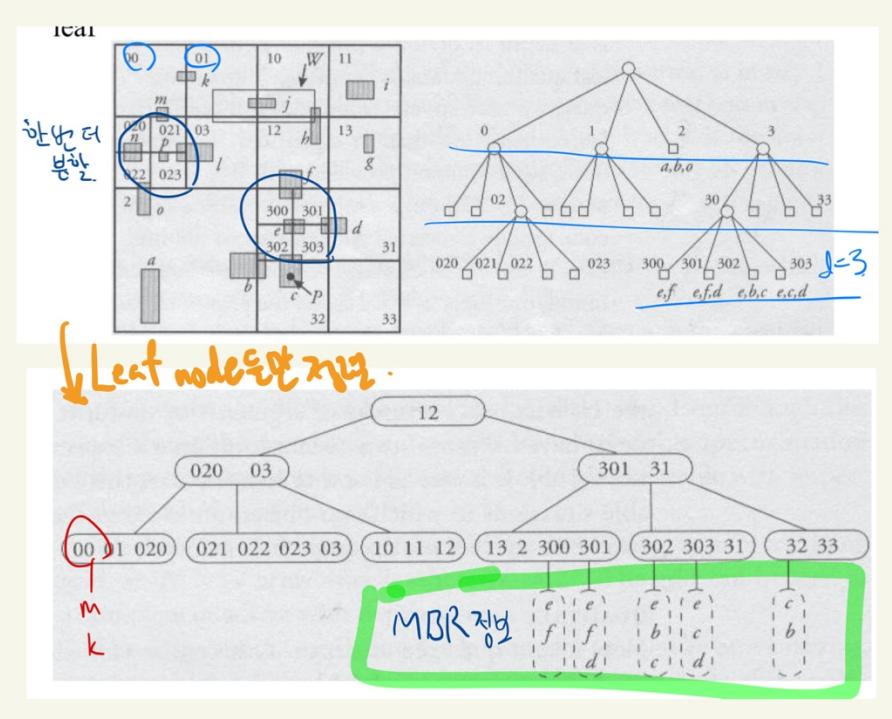
- 5) 2 bit string 에 대해 decimal 들을 변환한 것들을 좌표상에서 확인했을 때,
 - 앞자리가 0 이면 swap (1,3).
 - 앞자리가 3 이면, swap (0,2).
- 6) 이렇게 swap 과정이 완료된 숫자들을 그대로 각 숫자를 2bit binary 로 변환하여 bit string
- 7) 이 bit string 을 정상적인 binary \rightarrow decimal 변환 (0이 decimal 값들 0이 hilbert value)
- ex) 잘 따라가보자..



- (a) 각 x, y 좌표의 binary 표현들을 interleave 하여 표현 ($x_1y_1x_2y_2$)
- (b) 2 bit 씩 나누어서 확인해서 이진변환. (이때, 10 은 3, 11 은 2 로 바꾸는걸 반드시 기억!)
- (c) j 를 확인했을 때, j = 0 이면 3 을 1 로, 1 을 3 으로 바꿔준다. j = 3 인 것들은 0 을 2 로, 2 을 0 으로 바꿔준다.
- (d) 이 값들을 다시 2 bit 씩 2 진 변환을 해서, 이 이진수를 decimal 로 변환 (특별한 작업 필요 x)

Linear Quadtree

- d : depth of quadtree
- $N \times N$ grid 에 embedding 가능 $N = 2^d$



위 사진에서 [mbb, oid] entry 들이 quadtree leaf node 에 할당되고, 여기서 이 leaf node 들을 B+tree 에 indexing. B+ tree 에서 MBR 정보들이 leaf 에 할당됨.

- Redundancy (중복 저장) 문제 : MBR 들이 중복 저장된다.. (ex) e 가 4 번 저장, f 가 2 번 저장됨)

1. Point Query Algorithm with Linear Quadtree

LQ-POINTQUERY (P : point): set(oid)

```

begin
  result = ∅
  // Step 1: compute the label of the point
  l = POINTLABEL( $P$ )
  // Step 2: the entry [ $L, p$ ] is obtained by traversing the B+tree with key  $l$ .
  [ $L, p$ ] = MAXINF ( $l$ )
  // Step 3: get the page and retrieve the objects
  page = READPAGE ( $p$ )
  for each  $e$  in page do
    if ( $e.mbb$  contains  $P$ ) then result += { $e.oid$ }
  end for
  return result
end

```

(Point P 를 포함하는 object 찾기!)

- POINTLABEL(P) : Point p 를 포함하는 quadrant 의 z-order number 를 point 의 "label"
- MAXINF(l) : 이전에 찾은 label 을 key 로 하여 B+tree 에서 [label, page] entry 를 찾기.
- READPAGE(p) : 찾은 entry 의 page 를 읽고 가져옴 (MBR object 들이 있는 page)
- 반복문을 통해 page 의 entry 들을 하나씩 확인하여 $e.mbb$ 에 점 P 가 contain 되어있는지 확인

Point Query 에서 Disk I/O 횟수는? $d+1$ 번

- B+ tree 의 depth 가 d 라고 한다면, $d+1$

2. Window Query Algorithm with Linear Quadtree

```

LQ-WINDOWQUERY (W: rectangle): set(oid)

begin
    result = ∅
    // Step 1: From the vertices W.nw and W.se of the window, compute
    // the interval [L, L']. This necessitates two searches through the B+tree
    l = POINTLABEL (W.nw); [l, p] = MAXINF (l)
    l' = POINTLABEL (W.se); [l', p'] = MAXINF (l')
    // Step 2: The set Q of B+tree entries [l, p] with l ∈ [L, L'] is computed
    Q = RANGEQUERY ([l, l'])
    // Step 3: For each entry in Q whose quadrant overlaps W, access
    // the page
    for each q in Q do
        if (QUADRANT (q.l) overlaps W) then
            page = READPAGE (q.p)
            // Scan the quadtree page
            for each e in page do
                if (e.mbb overlaps W) then result += {e.oid}
            end for
        end if
    end for
    // Sort the result, and remove duplicates
    SORT (result); REMOVEDUPL (result);
    return result
end

```

- **POINTLABEL(W.nw), POINTLABEL(W.se)** : Window 의 좌표를 볼 때, Z-order 에서 가장 작은 NW 와 가장 큰 z-value 를 가지는 SE 의 Point 를 labeling 한다.
- **MAXINF (l)** : 해당 포인트들에 해당하는 key 인 label 로 B+tree 에서 [l, p] entry 를 찾고 page 주소를 찾는다.
- **RANGEQUERY([l, l'])** : label 해서 찾은 entry l 부터 l'까지 B+ tree 의 RangeQuery 진행. B+tree 를 타고 l 부터 l'까지 후보에 추가한다. B+트리는 leaf 들끼리 linked list 의 형태로 연결되어있다!!
- 반복문 : Q(후보군)에 있는 q(z-label)들을 하나씩 확인하면서 해당 q 의 quadrant 가 W 와 겹치는지 확인하며 후보군 선택/탈락.
 - o 선택된다면? **READPAGE** 를 통해 해당 entry 의 page 를 읽는다.
 - o 안쪽 반복문 : 해당 page 의 entry 들에 대해 mbr 를 확인하여 window W 와 overlap 되는지 확인 -> result 에 추가
- 결과 정렬 후 중복 제거하면 최종 결과.

Linear Quadtree 의 Window Query 에서 Tree 탐색은 몇번 일어나는가? 3 번

- W.nw 와 W.se 를 PointLabel 할때 tree 의 depth 만큼 각각 한 번씩 탐색
- RangeQuery 를 할 때 tree 의 depth 만큼 탐색 후, linked list 를 따라가면서 후보에 넣기.

그러면 Window Query 에서 Disk I/O 횟수는? **3d+k** 번

- step 1에서 **MAXINF** 를 실행할 때 각각 depth 만큼 확인. (2d)
- step2에서 **RANGEQUERY** 를 실행할 때, depth 만큼 확인 후, interval 만큼.
 - o 즉, linked list 를 따라서 확인하는 갯수를 k 라고 했을 때, k 만큼 추가로. (**d+k**)
 - o step2에서 I/O 횟수는 구간 크기에 **dependent**

Linear Quadtree 에서 Rectangle Insertion

두 가지 경우를 고려해야 한다.

1) No split of quadtreee

- quadrant page 에 접근하고 수정

2) Split of embedded quadtree

- B+tree 에서도 entry 가 하나 삭제되고 새 엔트리 네개 (one per new quadtree page)로 교체됨

Ch07. Spatial Access Methods 4 – Z-ordering Tree

Z-ordering Tree : MBR 을 끄집어내는 것이 아니라 overlap 되는 grid 형태의 집합으로!!

- mbb 사용하지 않는다!

Object Decomposition:

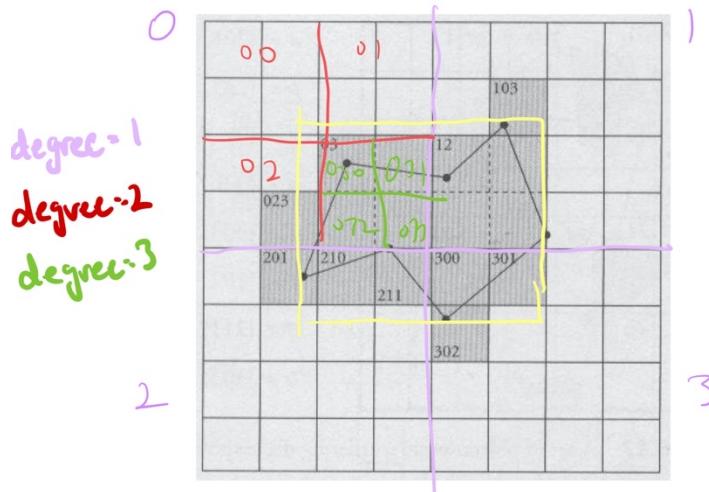
```

DECOMPOSE (o: geometry, q: quadrant): set(quadrant)
begin
  decompNW, decompNE, decompSW, decompSE: set(quadrant)
  result = ∅
  // Check that q overlaps o, else do nothing
  if (q overlaps o) then
    if (q is minimal) then
      result = {q}
    else
      // Decompose o into pieces, one per subquadrant
      for each sq in {NW(q), NE(q), SW(q), SE(q)} do
        decomp_sq = DECOMPOSE (o, sq)
      end for
      // If each decomposition results in the full subquadrant, return q
      if (decompNW ∪ decompNE ∪ decompSW ∪ decompSE = q) then
        result = {q}
      else
        // Take the set union of the four decompositions
        result = decompNW + decompNE + decompSW + decompSE
      end if
    end if
  end if
  return result
end

```

Z-ordering, object decomposition

ex) d (depth) = 3



Z-ordering depth 를 색깔 별로 나누어보았고, (보라, 빨강, 초록)

위의 object (점들이 이어진 부분) 에서 z-ordering object decomposition 을 해보면 다음과 같다.

d = 3 이기 때문에, 칠해진 부분들을 Z-order 순으로 각 부분들을 정리해보면,

{023, 030, 031, 032, 033, 103, 120, 121, 122, 123, 201, 210, 211, 300, 301, 302}

- 여기서 030, 031, 032, 033 은 03 으로, 120, 121, 122, 123 은 12 로 표현할 수 있다! (모을 수 있으면 모으기)

만약 Linear Quadtree 를 만드는 것이었다면, 위 사진의 노란색 부분처럼 MBR 을 먼저 구성하여 표현했을 것이다.

이런식으로 object 를 z-order 로 분해하는 것이 z-ordering decomposition.

위의 object 를 object a 라고 했을때, 이를 이름과 decomposition 된 각 값들과 pairing 해서 표현한다.

- {[023, a], [03, a], [103, a], [12, a], [201, a], [210, a], [211, a], [300, a], [301, a], [302, a]}

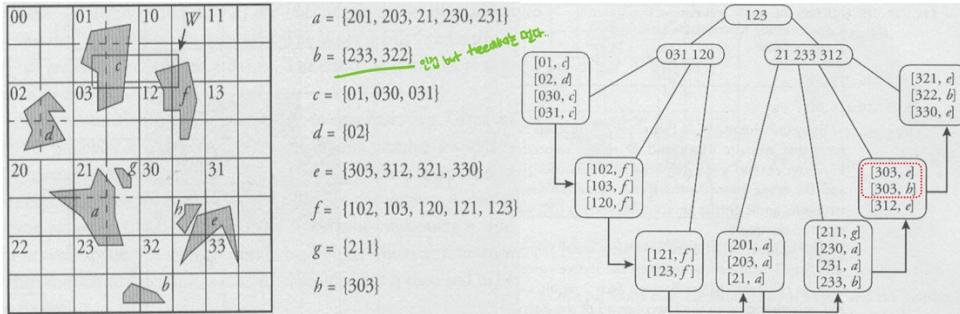
Z-Ordering Tree 만들기!!

1) Z-order decomposition

2) [l (label), oid] entry 의 형태로 표현 (ex) {[023, a], [03, a]}

3) OI entry 들로 B+ tree 구성!

ex)



위와 같이 object decomposition 후, 이를 pairing 해서 z-order 순서로 b+tree 구성

두가지 문제 : 1) 어떤 z-order 값들은 인접한 값이지만 tree 상에서는 멀다. 2) 중복 저장 문제 (z-value)

Z-Order Tree에서 Window Query

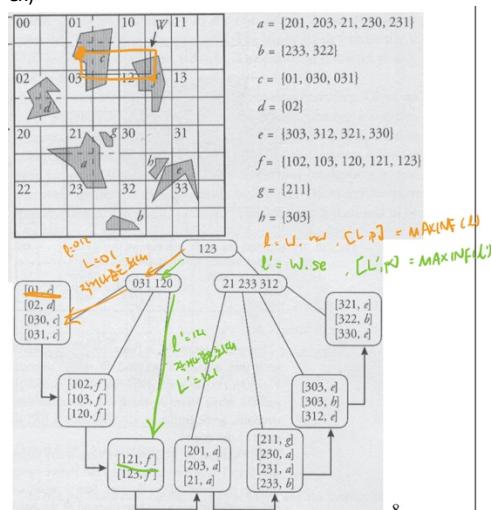
```
ZO-WINDOWQUERY ( $W$ : rectangle): set( $oid$ )
begin
    result =  $\emptyset$ 
    // Step 1: From the vertices  $W.nw$  and  $W.se$  of the window, compute
    // the interval  $[L, L']$ . This necessitates two searches through the B+tree.
     $l = \text{POINTLABEL}(W.nw); [l, p] = \text{MAXINF}(l)$ 
     $l' = \text{POINTLABEL}(W.se); [l', p'] = \text{MAXINF}(l')$ 
    // Step 2: Compute the set  $E$  of entries  $[l, oid]$  such that  $l \in [L, L']$ .
     $E = \text{RANGEQUERY}([l, l'])$ 
    // Step 3: For each entry in  $E$  that overlaps  $W$ , report the  $oid$ 
    for each  $e$  in  $E$  do
        if (QUADRANT( $e.l$ ) overlaps  $W$ ) then  $result += \{e.oid\}$ 
    end for
    // Sort the result, and remove duplicates
    SORT(result); REMOVEDUPL(result);
    return result
end
```

Linear Quadtree에서 Window Query 와 유사!

Z-order Tree 먼저 구성!

- NW, SE 의 Point Label 을 구하고 MAXINF 로 $[l, p]$ 를 구한다. (동일)
- $E = \text{RangeQuery}([l, l'])$ (동일)
 - o RangeQuery : l 보다 작거나 같은 최대값 ~ l' 보다 작거나 같은 최대값
- 반복문 : E 의 entry 들을 하나씩 확인해서 Window 와 겹치면 result 에 oid 를 추가
- 정렬 후, 중복 제거

ex)



nw 의 $l = 012$. \rightarrow B+ tree에서 작거나 같은 최대 $l = 01$

se 의 $l' = 121$. \rightarrow B+ tree에서 작거나 같은 최대값 $l' = 121$

$E = \text{WindowQuery}([l, l']) \Rightarrow E = \{01, 02, 030, 031, 102, 103, 120, 121, 122\}$

entry 확인하면서 Window 와 겹치는 것들의 oid result에 추가 $result = \{c, c, c, f, f, f, f\} \rightarrow$ 중복 제거, 정렬 : $\{c, f\}$

Z-ordering Tree 의 Window Query I/O 횟수 :

Linear Quadtree에서 window query I/O 횟수와 동일하다

- Step1 : **2d I/O** (d: depth)
 - Step2: **d I/O s + k** (k : # of chained B+ tree leaves to be scanned)

⇒ **3d + k**

Z-ordering Tree vs Linear Quad Tree

Z-ordering Tree :

- B+ tree 의 depth 가 더 클 가능성이 크다.. (object decomposition 을 한만큼 entry 가 많이 들어가게된다..)
 - grid resolution 좋을수록, object approximation 이 좋아진다. 하지만 B+tree 에 indexing 되어야할 entry 갯수는 증가
 - o = 잘게 쪼갤수록 더 비슷한 모양 (but indexing 되어야할 entry 증가 -> depth 는 증가한다)

Linear Quadtree :

- B+ tree entry 가 훨씬 적을 확률이 높다 . (quadtree 의 quadrant 갯수와 동일)
 - window 와 overlapping 하는 quadrant 하나당 분수적인 I/O 가 필요

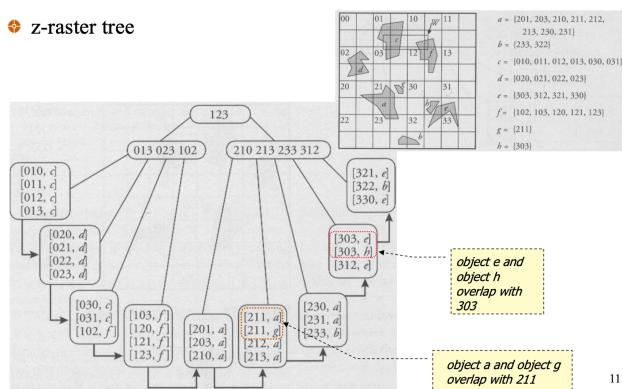
Raster Variant of Z-ordering

이전에 Z-ordering tree에서는 {210, 211, 212, 213}을 {21}로 통일 시켰다.

- 전부 동일한 degree로 통일하는 것! (이제는 다 표현한다..)
 - redundancy worse (중복성 증가..)
 - B+ tree entry 갯수 증가
 - 하지만 point/window query algorithm은 훨씬 간단해짐. (cell들이 모두 최소화이기 때문에 MAXINE가 필요 X)

ex)

• z-raster tree



다 표현. entry 갯수가 전에 비해 증가 + redundancy 도 증가 (object a,g 가 모두 211에 겹침, object e,h 가 모두 303에 겹침..)

알고리즘은 사실상 동일, MAXINF 만 필요없고, POINTLABEL 을 통해 nw, se 의 Label L, L' 을 바로 구해서 RANGEQUERY 진행하는 것.

I/O 횟수 : $d+k$ (MAXINF 2 회 : $2d$ 생략)

Ch08. KD-trees

2 차원 kd-tree construction 알고리즘 :

- 1) x,y 좌표를 선택
 - 2) x 축 median (vertical line)을 선택 or y 축 median (horizontal line)을 선택 (만약 선에 걸치면 left/right 선택해서 계속 진행)
 - 3) 양 축 번갈아가면서 recurse (recurse until 영역에 남은 data 1 개)

kd-tree에서 **Region(y)**: y 가 root인 subtree에 있는 모든 점들

Range Query in kd-tree :

v == root 부터 시작해서 recursive하게 Search(v, B)

Search(v, R);

- if v 가 leaf 면 v 에 저장된 점들 중 R 과 만나는 점 모두 report
 - else if $\text{Reg}(v)$ contained in R , subtree(v)에 있는 모든 점들 report
 - else if $\text{Reg}(\text{left}(v))$ 와 R 이 겹치면, Search ($\text{left}(v)$, R)
 - else if $\text{Reg}(\text{right}(v))$ 와 R 이 겹치면, Search ($\text{right}(v)$, R)

다차원 kd tree:

- Construction algorithm 은 2 차원 kd tree 와 동일
- root 부터 $x_1, x_2, x_3, \dots, x_n$ 축을 기준으로 차례대로 partition
- One point 남을 때 까지 recursion
- k-d tree 에서 점이 한 개만 남으면 그 점으로 leaf.

kd-tree division strategy :

- 절반씩 자르는 방식, average 로 자르는 방식, 다양한 방식으로 자를 수 있음.
- 어느 순서로 자르느냐? 1) Round Robin style (x, y 번갈아가면서)
- 2) widest spread 를 가지는 축에 수직으로 자르기 (RR 보다 조금 더 최적화, 각 노드에 따라 어떤 축으로 잘랐는지를 확인할 수 있음)

d 차원 kd-tree construction complexity :

- 1) Sort points in dimension : $O(dn \log n)$ 시간, 공간 : dn 시간
 - 2) widest spread 찾고 equally divide into two subset : $O(dn)$ 시간
- => $O(dn \log n)$ 시간, $O(dn)$ 공간**

median 을 split 하면서 전체를 check 하는 방식이 아닌,
data 를 하나씩 노드에 넣고 삽입하는 대로, kd-tree 를 construct 할 수도 있음.

Kd – tree 정리:

Construction 방식 세 가지 방식의 차이

- 1) 축은 alternative 하게 (x, y, x, y, \dots 순서대로)
 - 2) 축의 순서를 가장 even 하게 분할
- 1)과 2)의 공통점 : tree 에서 data point 는 leaf node 만.

둘 다, 전체를 보고 나눈다..

동적으로 삽입할 때 장점을 갖는,

- 3) internal node 가 routing node 가 아니라 data 를 표현하는 방식
- but skew 될 가능성 있음