ADVANCED DATABASES
Course Code:AIT505
Regulation: IARE-R16
IT
V SEMESTER

Prepared by:
Mr. D Rahul, Assistant Professor
Mr. N Bhaswanth, Assistant Professor

UNIT - I

ACTIVE DATABASES

Active Databases: Topics

- Introduction
- Representative Systems and Prototypes
- Applications of Active Rules

Active-Database Technology

Passive DBMS: all actions on data result from explicit invocation in application programs (they only do what application programs tell them to do)

Active DBMS: execution of actions can be automatically triggered in response to monitored events, including database updates (upon deletion of the data about a customer) points in time (on January 1, every hour) events external to the database (whenever paper jams in the printer)

Event - Condition - Action Rules

- When an event occurs, if a condition holds, then an action is performed

Event a customer has not paid 3 invoices at the due date

Condition if the credit limit of the customer is less than 20 000 Euros

Action cancel all current orders of the customer

- ECA rules are part of the database (⇒ “rule base”), available to all applications

Rules May Express Various Aspects of Application Semantics

Static constraints (e.g., referential integrity, cardinality, value restrictions) only regular students can register at the library students can register in no more than 20 courses the salary of employees cannot exceed the salary of their manager

Semantics Modeled by Rules (cont'd)

Implementation of generic relationships (e.g., generalization)

a person is a student or a lecturer, but not both

• Derived data: materialized attributes, materialized views, replicated data

the number of students registered in a course must be part of the course data

orders received are summarized daily in the planning database

Benefits of Active Technology

Simplification of application programs: part of the functionality can be programmed with rules that belong to the database

• Increased automation: actions are triggered without direct user intervention

• Higher reliability of data thru more elaborate checks and repair actions ⇒ better computer-aided decisions for operational management

Starburst

Relational prototype by IBM Almaden Research Center

• Event-Condition-Action rules in Starburst:

event: data-manipulation operations (INSERT, DELETE, UPDATE) in SQL

condition: Boolean predicate in SQL on the current state of the database

action: SQL statements, rule-manipulation statements, rollback

Starburst: Syntax of Rule Definition

```
<Starburst-rule> ::= CREATE RULE <rule-name> ON
<relation-name>
WHEN <list of trigger-events>
[ IF <condition> ]
THEN <list of SQL-statements>
[ PRECEDES <list of rule-names> ]
[ FOLLOWS <list of rule-names> ]
<trigger-event> ::= INSERTED | DELETED | UPDATED [ (
<attributes> )
```

Starburst: Syntax of Rule Definition

<Starburst-rule> ::= CREATE RULE <rule-name> ON

<relation-name>

WHEN <list of trigger-events>

[IF <condition>]

THEN <list of SQL-statements>

[PRECEDES <list of rule-names>]

[FOLLOWS <list of rule-names>]

<trigger-event> ::= INSERTED | DELETED | UPDATED [(

<attributes>)

Starburst: Semantics

Rules are triggered by the execution of operations in statements that are part of transactions

- Rules are statement-level: they are executed once per statement even for statements that trigger events on several tuples
- Execution mode is deferred: 3 all rules triggered during transaction execution are placed in a conflict set 3 rules are not considered until the end of the transaction (transaction commit) unless an explicit PROCESS RULES is executed in the transaction

Rule Processing

Algorithm for rule selection and execution

```
While the conflict set is not empty
(1) Select a rule R in the conflict set among those rules at
highest priority; take R out of the conflict set
(2) Evaluate the condition of R
(3) If the condition of R is true, then execute the action of R
```

Precise Definition of Rule Triggering

A rule is triggered if any of the transition relations corresponding to its triggering operations is not empty

- *Rule can reference transition relations (this can be more efficient than referring* to database relations)

Oracle: Triggers

Respond to modification operations (insert, delete, update) to a relation *Granularities for rules*

- ***tuple-level (or row-level):*** *a rule is triggered once for each tuple concerned* by the triggering event
- ***statement-level:*** *a rule is triggered only once even if several tuples are* involved
- ***Immediate execution mode:*** *rules are considered immediately after the event* has been requested (Starburst rules are deferred)*Rules can be considered and executed before, after, or instead of the operation* of the triggering event is executed 23

UNIT - II

TEMPORIAL AND OBJECT DATABASES

Temporal Databases

- Temporal Data Models: extension of relational model by adding temporal attributes to each relation
- Temporal Query Languages: TQUEL, SQL3
- Temporal Indexing Methods and Query Processing

Temporal databases

- Some data may be inherently historical
  - e.g., medical or judicial records
- Temporal databases provide a uniform and systematic way of dealing with historical data
- Considerable effort has been expended on the development of temporal databases and query languages
  - TQuel [Snodgrass87], TSQL2 [Snodgrass95], SQL/Temporal [Snodgrass96]

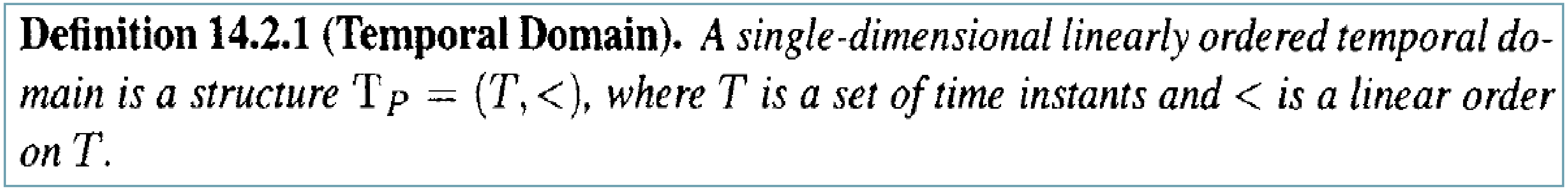
—But none of them has been adopted as **the standard language of temporal databases in practice**
—No established **the theoretical foundations** for management of time-dependent data
—No universal consensus on **how *temporal features* should be added to the standard relational model**

Outline

- The fundamental notions of *temporal databases*
  - A formal foundation for temporal data models
  - How to introduce time into the relational model
- Query languages for temporal databases
  - Temporal extensions of SQL
- Limitations of simple linearly-ordered, first-order temporal data models
  - More complex models of time

Structure of time

- They used a very simple notion of time in this chapter:
  - a linear ordering of time instants



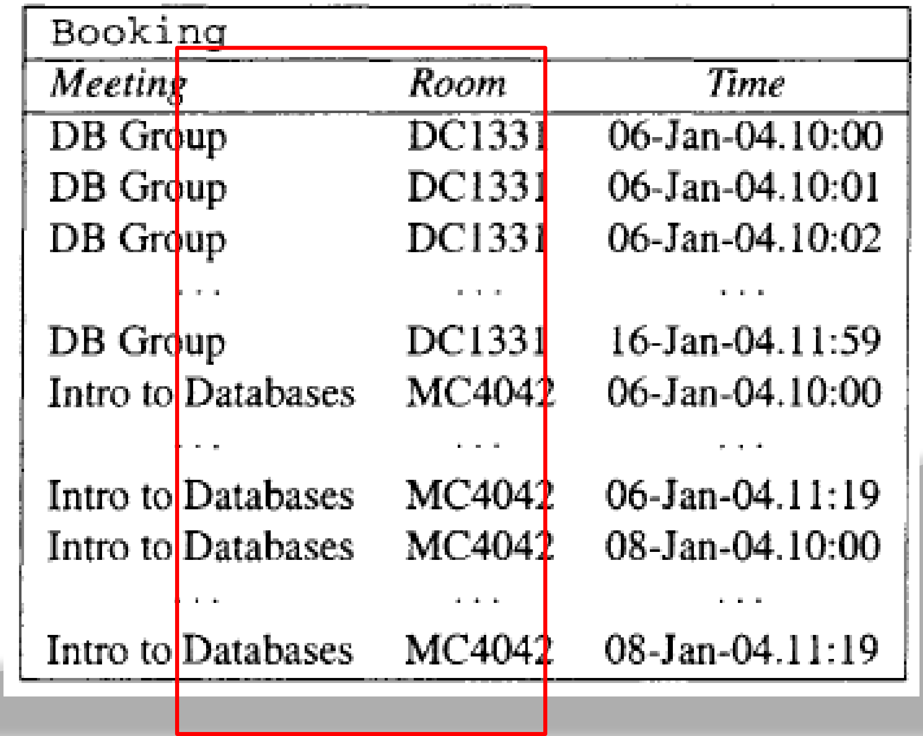
Definition 14.2.1 (Temporal Domain). *A single-dimensional linearly ordered temporal domain is a structure* $\mathrm{T}_P = (T, <)$, *where* T *is a set of time instants and* $<$ *is a linear order on* T.

- In addition to linear ordering, we may consider:
  - Discrete or dense
  - Bounded or unbounded
  - Single dimensional or multi-dimensional
  - Linear or non-linear

The time stamp model

- All the tuples in a relation have an additional temporal attribute
- Example: Booking (meeting, room, time)
  - A tuple (m,r,t) denotes the fact that:

    meeting m is in room r at time t

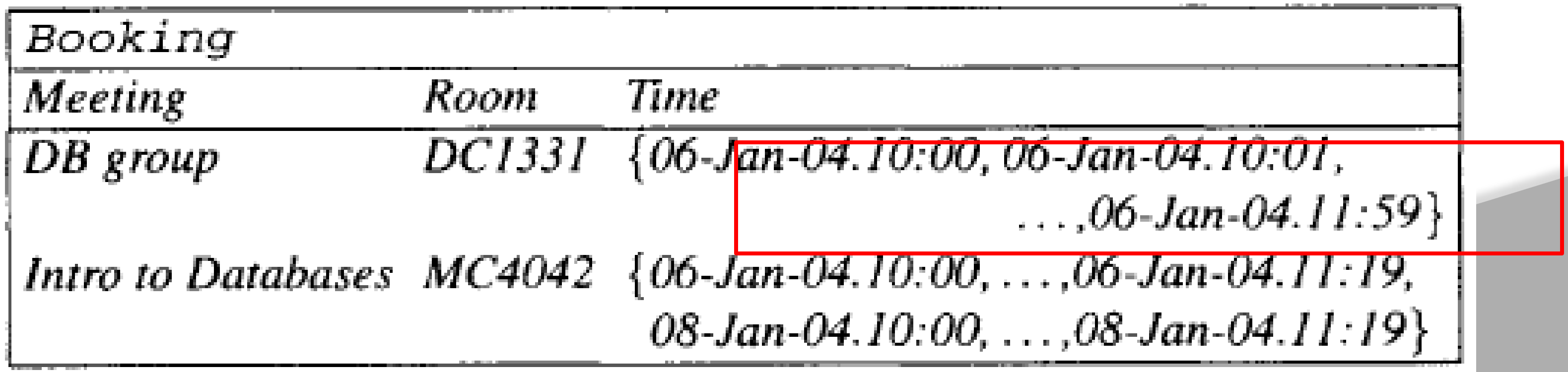


Booking

Meeting	Room	Time
DB Group	DC1331	06-Jan-04.10:00
DB Group	DC1331	06-Jan-04.10:01
DB Group	DC1331	06-Jan-04.10:02
...	...	...
DB Group	DC1331	16-Jan-04.11:59
Intro to Databases	MC4042	06-Jan-04.10:00
...	...	...
Intro to Databases	MC4042	06-Jan-04.11:19
Intro to Databases	MC4042	08-Jan-04.10:00
...	...	...
Intro to Databases	MC4042	08-Jan-04.11:19

Temporal attribute

- Single-dimensional: temporal relations were allowed only a single temporal attribute
- Multiple dimensional: with each tuple in a relation there can be more than one temporal attribute
    - Example: two kinds of time are stored: the valid time (when a particular tuple is true) and the transaction time (when the particular tuple was inserted/deleted in the database)
- Non-1NF: can be flattened to obtain the 1NF

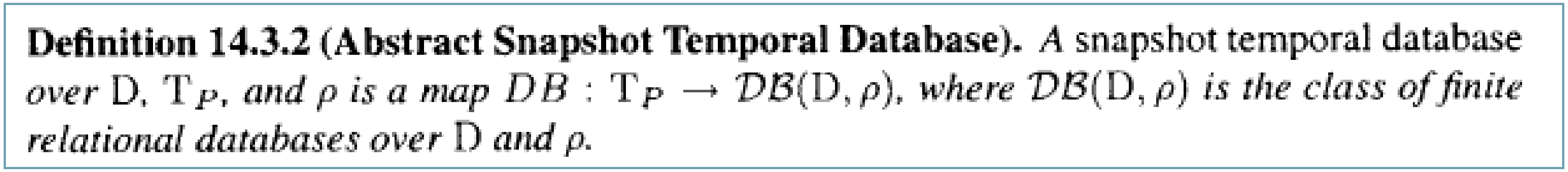


Booking

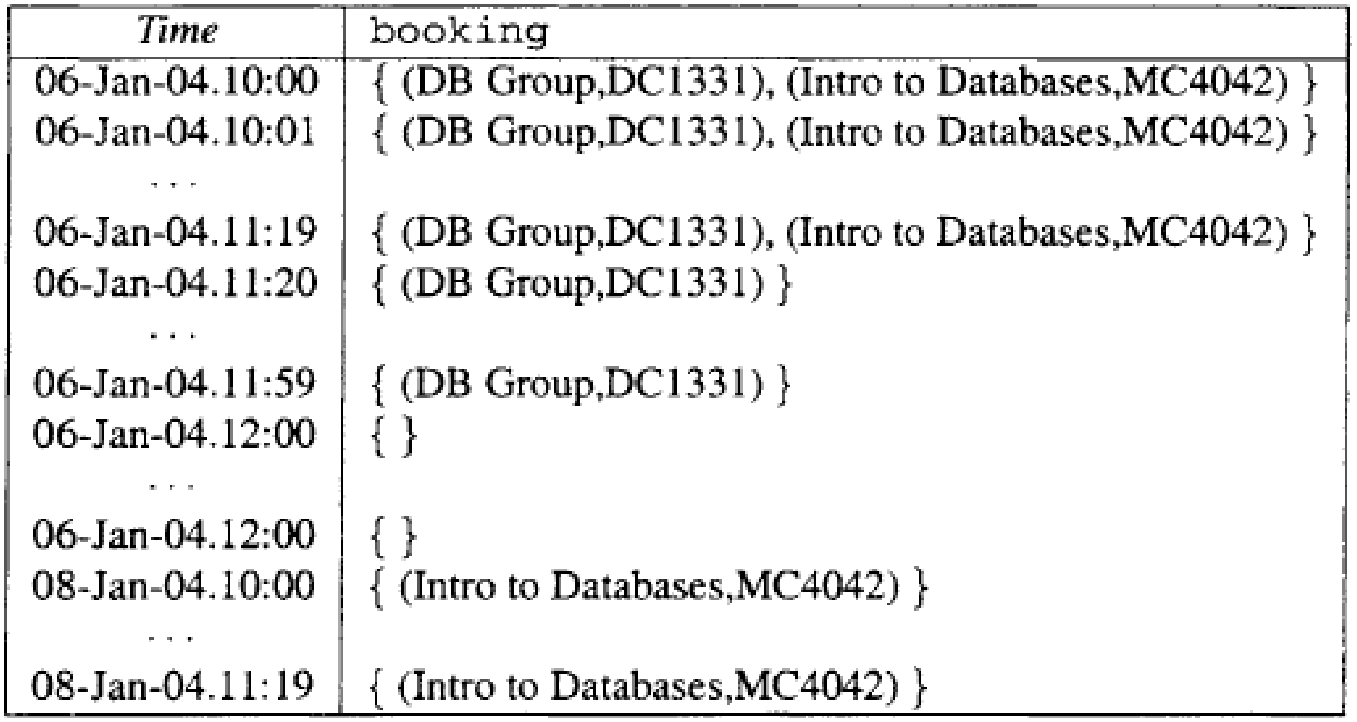
Meeting	Room	Time
DB group	DC1331	{06-Jan-04.10:00, 06-Jan-04.10:01, ...,06-Jan-04.11:59}
Intro to Databases	MC4042	{06-Jan-04.10:00, ...,06-Jan-04.11:19, 08-Jan-04.10:00, ...,08-Jan-04.11:19}

The snapshot model

- Different view from the time stamp model (of the same data)



Definition 14.3.2 (Abstract Snapshot Temporal Database). *A* snapshot temporal database *over* D, T_P, *and* ρ *is a map* $DB : \mathrm{T}_P \to \mathcal{DB}(\mathrm{D}, \rho)$, *where* $\mathcal{DB}(\mathrm{D}, \rho)$ *is the class of finite relational databases over* D *and* ρ.



Time	booking
06-Jan-04.10:00	{ (DB Group,DC1331), (Intro to Databases,MC4042) }
06-Jan-04.10:01	{ (DB Group,DC1331), (Intro to Databases,MC4042) }
...	
06-Jan-04.11:19	{ (DB Group,DC1331), (Intro to Databases,MC4042) }
06-Jan-04.11:20	{ (DB Group,DC1331) }
...	
06-Jan-04.11:59	{ (DB Group,DC1331) }
06-Jan-04.12:00	{ }
...	
06-Jan-04.12:00	{ }
08-Jan-04.10:00	{ (Intro to Databases,MC4042) }
...	
08-Jan-04.11:19	{ (Intro to Databases,MC4042) }

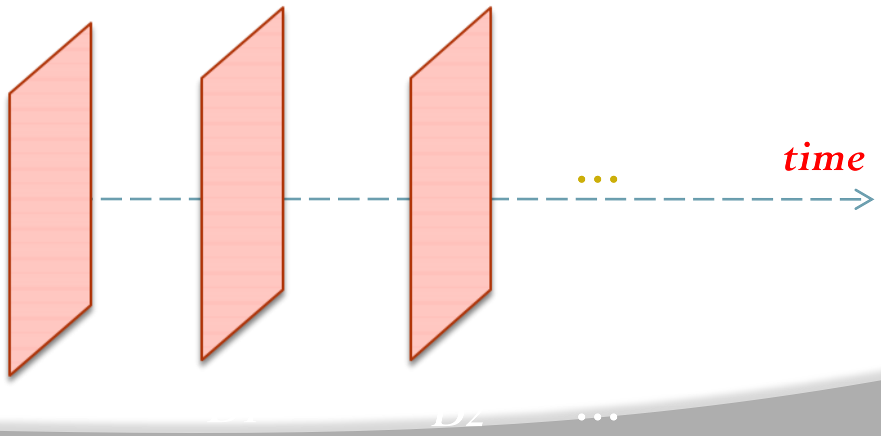
Relational database histories

- A history over a database schema *p* and a data domain *D* is a sequence *H : (Do,..., Dn)* of database instances such that:

1. all the states *Do , . . . , Dn* share the same schema *p* and the same data domain *D*

2. Do is the initial instance of the database

3. Di results from applying an update to *Di-1,* for i > 1



Temporal database design

- Reconstruction of Jensen’s formal framework [Jensen96]
- Based on the notion of temporal functional dependency:

$$\boxed{X \xrightarrow{T} Y}$$

A temporal FD $X \xrightarrow{T} Y$ lds in a snapshot temporal relation DB if the (classical) FD $X \rightarrow Y$ olds in every snapshot of DB

- Example: the temporal FD $\boxed{Meeting \xrightarrow{T} Room}$

  means every meeting is held in a single room at any given time
- Several advantages: can use the classical notions of FD inference, dependency closure, normal forms, mix temporal and non-temporal FDs

Multiple dimensions

- How to express two temporal dimensions using temporal FD:
  - *valid time (VT)*
  - *transaction time (TT)*
- 3 kind of temporal FDs:
  - *Transaction time:* $\boxed{X \; TT \rightarrow Y}$
  - *Valid time:* $\boxed{X \; VT \rightarrow Y}$
  - *Bitemporal:* $\boxed{X \; TT \, VT \rightarrow Y}$
- Example: $\boxed{Meeting \; TT \, VT \rightarrow Room}$ means the record at any time of the room booked for a meeting at any time is uniquely determined
- Disadvantage:
  - Can no longer talk about, e.g., temporal keys, but only about valid-time, transaction-time or bitemporal keys
  - The framework becomes so complicated that it is unlikely to be of any use

Temporal queries

- Databases are inherently first-order structures
- Temporal extensions first-order logic
- Query: using a natural first-order query language
- The answer: the set of tuple that make the query true in the given relational database
- Examples:
  - *find all meetings that always meet in the same room*
  - *find all rooms in which the last meeting was 'DB group'*

Temporal logic

- Historically, many different variants of temporal logic based on different sets of connectives have been developed [Gabbay*94*]
- Some connectives are well-known and have been universally accepted:
  - *sometime in the future*
  - *always in the future*
- In general any appropriate first-order formula in the language of the temporal domain can be used to define a temporal connective

First order temporal logic

- First they define the first order language of *Tp* extended with propositional variables *Xi* :

$$O ::= t_i < t_j \mid O \wedge O \mid \neg O \mid \exists t_i . O \mid X_i$$

- Then use it to define a (*k-ary*) temporal connective:
  - an O-formula with exactly one free variable *t0* and *k* propositional variables *X1,.., Xk*
  - They assume *ti* is the only temporal variable in the formula to be substituted for *Xi*
- Example: common binary temporal connectives:

$$X_1 \ \textbf{until}\ X_2 \ \triangleq\ \exists t_2 . t_0 < t_2 \wedge X_2 \wedge \forall t_1 (t_0 < t_1 < t_2 \rightarrow X_1)$$
$$X_1 \ \textbf{since}\ X_2 \ \triangleq\ \exists t_2 . t_0 > t_2 \wedge X_2 \wedge \forall t_1 (t_0 > t_1 > t_2 \rightarrow X_1)$$

Temporal connectives

- Other temporal connectives:
  - Sometime in the future:
  - Sometime in the past:
  - Always in the future:
  - Always in the past:
  - Next:
  - Previous:

$$\Diamond X_1 \triangleq \text{true } \textbf{until } X_1$$

$$\blacklozenge X_1 \triangleq \text{true } \textbf{since } X_1$$

$$\Box X_1 \triangleq \neg\Diamond\neg X_1$$

$$\blacksquare X_1 \triangleq \neg\Diamond\neg X_1$$

$$\bigcirc X_1 \triangleq \exists t_1 . t_1 = t_0 + 1 \wedge X_1$$

$$\bullet X_1 \triangleq \exists t_1 . t_1 + 1 = t_0 \wedge X_1$$

First order temporal logic

- Ω : A set of temporal connectives , e.g. {since, until}
- L^{Ω} : First order temporal logic (FOTL) over a schema ρ

$$\boxed{F ::= r(x_{i_1}, \ldots, x_{i_k}) \mid x_i = x_j \mid F \wedge F \mid \neg F \mid \omega(F_1, \ldots, F_k) \mid \exists x.F}$$

$$r \in \rho \textit{ and } \omega \in \Omega.$$

- *standard FOTL language* $L^{\{\mathbf{since},\mathbf{until}\}}$

$$\boxed{F ::= r(x_{i_1}, \ldots, x_{i_k}) \mid x_i = x_j \mid F \wedge F \mid \neg F \mid F_1 \textbf{ since } F_2 \mid F_1 \textbf{ until } F_2 \mid \exists x.F}$$

Examples

- How to use temporal connectives to formulate queries:
- Find all rooms in which the last meeting was 'DB group‘:

$$(\neg\exists y.\mathtt{booking}(y,x))\ \mathbf{since}\ \mathtt{booking}(\mathtt{DB\ group},x)$$

- Find all meetings with a scheduled break:

$$\blacklozenge\exists y.\mathtt{booking}(x,y) \wedge \neg\exists y.\mathtt{booking}(x,y) \wedge \lozenge\exists y.\mathtt{booking}(x,y)$$

$$\lozenge X_1 \triangleq \mathrm{true}\ \mathbf{until}\ X_1$$
$$\blacklozenge X_1 \triangleq \mathrm{true}\ \mathbf{since}\ X_1$$

$$\square X_1 \triangleq \neg\lozenge\neg X_1$$
$$\blacksquare X_1 \triangleq \neg\lozenge\neg X_1$$

Temporal extensions of SQL

- A point based extension of SQL: SQL/TP [Toman97]
- The syntax and semantics of SQL/TP are defined as a natural extension of SQL
  - An additional data type based on the point-based temporal domain *Tp* (i.e., a linearly ordered set of time instants)

SQL/TP Example

- List all meetings with a scheduled break :

$\blacklozenge \exists y.\text{booking}(x, y) \land \neg \exists y.\text{booking}(x, y) \land \lozenge \exists y.\text{booking}(x, y)$

```
select r1.Meeting
from   Booking r1, Booking r2
where  r1.Meeting = r2.Meeting
  and  r1.time < r2.time
  and  not exists ( select *
                    from   Booking r3
                    where  r3.Meeting = r1.Meeting
                      and  r1.time < r3.time
                      and  r3.time < r2.time )
```

Extensions of SQL based on interval based language

- TSQL2 or SQL/Temporal [Snodgrass95]

```
select  r1.Meeting
from    Booking r1, Booking r2
where   r1.Meeting = r2.Meeting
  and   r1.time before r2.time
```

- Time attributes range over intervals and the before relationship denotes the before relationship between two intervals

Updating temporal databases

- **Insertion: a new booking for a room for a meeting**

```
INSERT into Booking (
   SELECT 'DBgroup', 'DC1331', t
   FROM   unit
   WHERE  '23-Jan-04.14.00' <= t <= '23-Jan-04.16.00' )
```

  - **Unit is an auxiliary table that contains a single tuple**
  - **The inner query produces:**

$$\{(DB\ group, DC1331, t) : 23\text{-}Jan\text{-}04.14.00 \leq t \leq 23\text{-}Jan\text{-}04.16.00\}$$

- **Deletion: Creating 20 minute break in the middle of meeting**

```
DELETE from Booking
   WHERE  Meeting = 'DBgroup'
     AND  Room = 'DC1331'
     AND  '23-Jan-04.14.50' <= t <= '23-Jan-04.15.10'
```

Complex structure of time

- Complex structure of time: more complex than linearly ordered sets of time instants
  - Natural numbers, integers, reals
  - Additional structures: durations, temporal distances, periodic sets
- Impact on integrity constraints : more complex constraint dependencies
- Impact on query languages (use new predicate symbols in the same way the linear order < symbol has been used so far)

Time Ontology

- Several different structures of time
  - Linear is simplest and most common
- 5 fundamental temporal data types
- Several dimensions of time
  - TSQL2 supports transaction and valid time

Boundedness of Time

- Assume a linear time structure
- Boundedness
  - Unbounded
  - Time origin exists (bounded from the left)
  - Bounded time (bounds on two ends)
- Nature of bound
  - Unspecified
  - Specified
- Physicists believe that the universe is bounded by the “Big Bang” (12-18 billions years ago) and by the “Big Crunch” (? billion years in the future)

Time Density

- Discrete
  - Time line is isomorphic to the integers
  - Time line is composed of a sequence of non-decomposable time periods, of some fixed minimal duration, termed chronons
  - Between each pair of chronons is a finite number of other chronons
- Dense
  - Time line is isomorphic to the rational numbers
  - Infinite number of instants between each pair of chronons
- Continuous
  - Time line is isomorphic to the real numbers
  - Infinite number of instants between each pair of chronons
- Distance may optionally be defined

TSQL2: Time Ontology

- Structure
  - TSQL2 uses a linear time structure
- Boundedness
  - TSQL2 time line is bounded on both ends, from the start of time to a point far in the future
- Density
  - TSQL2 do not differentiate between discrete, dense, and continuous time ontologies
  - No questions can be asked that give different answers
    - ∗ E.g., instant *a* precedes instant *b* at some specified granularity. Different granularities give different answers
  - Distance is defined in terms of numbers of chronons

Ontological Temporal Types

- Instant: chronon in the time line
  - Event: instantaneous fact, something occurring at an instant
  - Event occurrence time: valid-time instant at which the event occurs in the real world
- Instant Set: set of instants
- Time period: time between two instants
  - Also called interval, but conflicts with SQL data type INTERVAL
- Time interval: a directed duration of time
- Duration: amount of time with a known length, but no specific starting or ending instants
  - positive interval: forward motion time
  - negative interval: backward motion time
- Temporal element: finite union of periods

Temporal Data Types in SQL-92 and TSQL2

- SQL92
  - DATE (YYYY-MM-DD)
  - TIME (HH:MM:SS)
  - DATETIME (YYYY-MM-DD HH:MM:SS)
  - INTERVAL (no default granularity)
- TSQL2
  - PERIOD: DATETIME - DATETIME

UNIT - III

COMPLEX QUERIES AND REASONING

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

sid	bid	day
22	101	10/10/96
58	103	11/12/96

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Relational Algebra

- Basic operations:
  - *Selection* : Selects a subset of rows from relation.
  - *Projection* : Deletes unwanted columns from relation.
  - *Cross-product* : Allows us to combine two relations.
  - *Set-difference* : Tuples in reln. 1, but not in reln. 2.
  - *Union* : Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming

Projection

- Schema of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate duplicates! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it.

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname,rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

Selection

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating>8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Set operations

- Union(U), Intersection(∩), Set-Difference(-) are set operations available in in relational algebra
- Union(RUS):
- Two relational instances are said to be union compatible if the following conditions hold—
- they have same number of the fields and corresponding fields
- taken in order from left to right,have the same domains
- Intersection(R ∩ S):returns a relational instance containing all tuples that occur in both R and S.
- Set-difference(R-S): returns a relational instance containing all tuples that occur in R but not in S.
- Cross product(RXS): returns a relational instance whose schema contains all fields of R followed by all fields of S

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Cross-Product

- Each row of S1 is paired with each row of R1
- *Result schema* has one field per field of S1 and R1, with field names `inherited’ if possible.

Conflict: Both S1 and R1 have a field called *sid*.

S1 X R1

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- *Renaming operator(* ρ (old name -> new name) or ρ (position -> new name)

$$\rho\,(C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$$

Joins

- ***Condition Join***: $R \bowtie_{c} S = \sigma_{c}(R \times S)$

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

- ***Result schema*** **same as that of cross-product.**
- **Fewer tuples than cross-product, might be able to compute more efficiently**
- **Sometimes called a *theta-join*.**

- ***Equi-Join***: A special case of condition join where the condition *c* contains only *equalities*.

$$S1 \bowtie_{sid} R1$$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

- ***Result schema*** similar to cross-product, but only one copy of fields for which equality is specified.
- ***Natural Join***: Equijoin on *all* common fields.
- If two relations have no attributes in common,natural join is simply cross product.

Division

- Not supported as a primitive operator, but useful for expressing queries like:

$$\left\{\langle x\rangle \mid \exists \langle x, y\rangle \in A \ \forall \langle y\rangle \in B\right\}$$

- *Find sailors who have reserved all boats*.
- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* =
  - i.e., *A/B* contains all *x* tuples (sailors) such that for <u>*every*</u> *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.

    $\cup$
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and *x* *y* is the list of fields of *A*.

Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

pno
p2

pno
p2
p4

pno
p1
p2
p4

sno
s1
s2
s3
s4

sno
s1
s4

sno
s1

A/B3

Relational Calculus

- Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).
- Calculus has *variables, constants, comparison ops*, *logical connectives* and *quantifiers*.
  - *TRC*: Variables range over (i.e., get bound to) *tuples*.
  - *DRC*: Variables range over *domain elements* (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called *formulas*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

Tuple relational calculus

- A tuple rc query has the form {T|P(T)} where T is a tuple variable and P(T) denotes a formula that describes T.
- Find all sailors with rating above 7
- {S|S € Sailors Л s.rating>7}
- Let Rel be a relation name, R & S be tuple variables,'a' be an attribute of R and 'b' be attribute of S. Let op denote operator.
- An atomic formula is one of the following
- R € Rel, R.a € S.b, R.a op constant or constant op R.a

Tuple relational calculus

- A formula is recursively defined to be one of the following

  any atomic formula

  -- ┐P,PЛQ,P V Q or P=>Q

  -- эR(P(R)) where R is tuple variable

  -- forall R(P(R)) where R is tuple variable

- A variable is said to be **free** in formula if it does not contain an occurence of quantifiers that bind it.
- Find the names and ages of sailors with rating above 7
- {P| эS є Sailors(S.Rating >7 Л P.name=S.Sname Л P.age=S.age)

Queries

- Find the sailor name,boat id and reservation date for each reservation
- {P|эR є Reserves эS є Sailors (R.Sid=S.sid Л P.bid=R.bid Л P.day=R.day Л P.sname=S.sname)
- Find the names of sailors who have reserved boat 103
- {P|эR є Reserves эS є Sailors (R.Sid=S.sid Л R.bid=103 Л P.sname=S.sname)
- Find the names of sailors who have reserved boat 103
- {P|эR є Reserves эS є Sailors (R.Sid=S.sid Л P.sname=S.sname Л эB є Boats(B.bid=R.bid Л B.color='red'))}

Find sailors rated > 7 who've reserved a red boat

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T>7 \wedge$$

$$\exists\ Ir,Br,D\ (\langle Ir,Br,D\rangle \in Reserves \wedge Ir=I \wedge$$

$$\exists\ B,BN,C\ (\langle B,BN,C\rangle \in Boats \wedge B=Br \wedge C='red'))\}$$

- Observe how the parentheses control the scope of each quantifier's binding.
- Find names of sailors who've reserved a red boat

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge$$

$$\exists\langle I,Br,D\rangle \in Reserves \wedge \langle Br,BN,'red'\rangle \in Boats)\}$$

Find sailors who've reserved all boats

$$\left\{ \langle I,N,T,A \rangle \mid \langle I,N,T,A \rangle \in Sailors \wedge \right.$$

$$\forall\ B,BN,C \left(\neg \left(\langle B,BN,C \rangle \in Boats \right) \vee \right.$$

$$\left. \left. \left(\exists\ Ir,Br,D \left(\langle Ir,Br,D \rangle \in \text{Re}serves \wedge I = Ir \wedge Br = B \right) \right) \right) \right\}$$

Find sailors who've reserved all boats (again!)

$$\left\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge\right.$$

$$\forall\ \langle B,BN,C\rangle \in Boats$$

$$\left.\left(\exists\ \langle Ir,Br,D\rangle \in \mathrm{Reserves}\left(I=Ir \wedge Br=B\right)\right)\right\}$$

- **To find sailors who've reserved all red boats:**

$$\left.\left(C\neq'red' \vee \exists \langle Ir,Br,D\rangle \in \mathrm{Reserves}\left(I=Ir \wedge Br=B\right)\right)\right\}$$

Data Definition Language

Allows the specification of:

- The schema for each relation, including attribute types.
- Integrity constraints
- Authorization information for each relation.
- Non-standard SQL extensions also allow specification of
    - The set of indices to be maintained for each relations.
    - The physical storage structure of each relation on disk.

Create Table Construct

- An SQL relation is defined using the create table command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of attribute A_i

  Example:

```
create table branch
    (branch_name     char(15),
    branch_city      char(30),
    assets      integer)
```

Domain Types in SQL

- char(n). Fixed length character string, with user-specified length *n.*
- varchar(n). Variable length character strings, with user-specified maximum length *n.*
- int. Integer (a finite subset of the integers that is machine-dependent).
- smallint. Small integer (a machine-dependent subset of the integer domain type).
- numeric(p,d). Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- float(n). Floating point number, with user-specified precision of at least *n* digits.

Integrity Constraints on Tables

- **not null**
- **primary key (A_1, ..., A_n)**

Example: Declare *branch_name* as the primary key for *branch*

.

```
                create table branch
                        (branch_name        char(15),
                         branch_city        char(30) not
null,
                         assets      integer,
                         primary key (branch_name))
```

primary key declaration on an attribute automatically ensures not null in SQL-92 onwards, needs to be explicitly stated in SQL-89

Basic Insertion and Deletion of Tuples

- Newly created table is empty
- Add a new tuple to *account*

```
insert into account
    values ('A-9732', 'Perryridge', 1200)
```

Insertion fails if any integrity constraint is violated

- Delete *all* tuples from *account*

```
delete from account
```

Drop and Alter Table Constructs

- The drop table command deletes all information about the dropped relation from the database.
- The alter table command is used to add attributes to an existing relation:

  alter table *r* add *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned *null* as the value for the new attribute.
- The alter table command can also be used to drop attributes of a relation: alter table *r* drop *A*

  where *A* is the name of an attribute of relation *r*
  - Dropping of attributes not supported by many databases

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.

- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

- **The select clause list the attributes desired in the result of a query**
  - **corresponds to the projection operation of the relational algebra**
- **Example: find the names of all branches in the *loan* relation:**

  select *branch_name*
  from *loan*
- **In the relational algebra, the query would be:**

  ⍰$_{branch\_name}$ (*loan*)
- **NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)**
  - **E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name***
  - **Some people use upper case wherever we use bold font.**

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword distinct after select.
- Find the names of all branches in the *loan* relations, and remove duplicates

  select distinct *branch_name*
  from *loan*

The keyword all specifies that duplicates not be removed.

  select all *branch_name*
  from *loan*

The select Clause (Cont.)

- An asterisk in the select clause denotes "all attributes

  select * from *loan*

- The select clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
- E.g.:

  select *loan_number, branch_name, amount* ∗ 100 from *loan*

“All” Construct

- Find the names of all branches that have greater assets than all branches located in Brooklyn

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

“Exists” Construct

- Find all customers who have an account at all branches located in Brooklyn.

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

Absence of Duplicate Tuples

- The unique construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
  select T.customer_name

   from depositor as T
   where unique (
       select R.customer_name
       from account, depositor as R
       where T.customer_name = R.customer_name and
               R.account_number = account.account_number
and
               account.branch_name = 'Perryridge')
```

Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

```
select distinct T.customer_name
from depositor  as T
where not unique (
      select R.customer_name
      from account, depositor as R
      where T.customer_name = R.customer_name and
              R.account_number = account.account_number
and
              account.branch_name = 'Perryridge')
```

Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

```
delete from account
where branch_name = 'Perryridge'
```

- Delete all accounts at every branch located in the city 'Needham'.

```
delete from account
where branch_name in (select branch_name
                      from branch
                      where branch_city = 'Needham')
```

Modification of the Database – Insertion

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

```
insert into account
   select loan_number, branch_name, 200
   from loan
   where branch_name = 'Perryridge'
insert into depositor
   select customer_name, loan_number
   from loan, borrower
   where branch_name = 'Perryridge'
            and loan.account_number = borrower.account_number
```

- The select from where statement is evaluated fully before any of its results are inserted into the relation
  - Motivation: insert into *table*1 select * from *table*1

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two update statements:

```
update account
set balance = balance ∗ 1.06
where balance > 10000

update account
set balance = balance ∗ 1.05
where balance ≤ 10000
```

  - The order is important
  - Can be done better using the case statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
        update account
    set balance =  case
                       when balance <= 10000 then balance
*1.05
                       else   balance * 1.06
                     end
```

Joined Relations – Examples

- ***loan* inner join *borrower* on *loan.loan_number = borrower.loan_number***

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan* **left outer join** *borrower* **on** *loan.loan_number = borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

- ***loan* natural inner join *borrower***

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- *loan* **natural right outer join** *borrower*

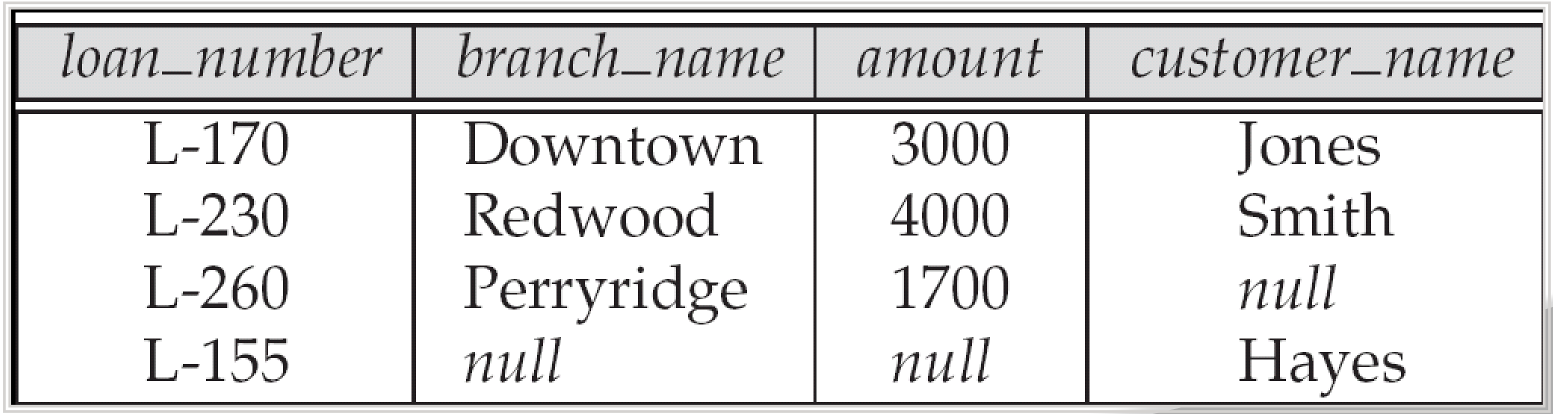
loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

ind all customers who have either an account or a loan (but not both) at the bank.

select *customer_name*
from (*depositor* natural full outer join *borrower*)
where *account_number* is null or *loan_number* is null

Joined Relations – Examples

- Natural join can get into trouble if two relations have an attribute with same name that should not affect the join condition
  - e.g. an attribute such as *remarks* may be present in many tables
- *Solution:*
  - *loan* full outer join *borrower* using (*loan_number*)



loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Derived Relations

- SQL allows a subquery expression to be used in the from clause
- Find the average account balance of those branches where the average account balance is greater than $1200.

```
select branch_name, avg_balance
from (select branch_name, avg (balance)
        from account
        group by branch_name )
       as branch_avg ( branch_name, avg_balance )
where avg_balance > 1200
```

Note that we do not need to use the having clause, since we compute the temporary (view) relation *branch_avg* in the from clause, and the attributes of *branch_avg* can be used directly in the where clause.

Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.
  - EX:Create domain ratingval integer default 1 check(value>=1 and value<=10)
  - Rating ratingval

UNIT - IV

SPATIAL, TEXT AND MULTIMEDIA DATABASES

Why indexing?

- Speed up retrieval
  - Non-key attributes
  - Feature based

Applications

- Image databases (2-D, 3-D)
  - Shapes, colors, textures
- Financial analysis
  - Sales patterns, stock market prediction, consumer behavior
- Scientific databases
  - Sensor data/Simulation results:
    - Scalar/vector fields
- Scientific databases

Traditional indexing methods

A record with *k* attributes

$\Leftrightarrow$

A point in *k*-dimensional space

Name	Salary	Age	Dept
Smith	40000	45	3
Dilbert	35000	35	4
Wally	35000	37	4
Dogbert	45000	30	5
…			

4 attributes: Name, salary, age, dept.

Spatial query complexity

- Exact match

  name = ’Smith’ and salary=40000 and age=45
- Partial match

  salary=40000 and age=45
- Range

  35000 ≤ salary ≤ 45000 and age=45
- Boolean

  ((not name = ’Smith’) and salary ≥ 40000) or age ≥ 50
- Nearest-neighbor (similarity)

  Salary ≈ 40000 and age ≈ 45

Inverted files

Given an attribute,

Name	Salary	Age	Dept

- For each attribute value, store
  1. A list of pointers to records having this attribute value
  2. (Optionally) The length of this list
- Organize the attribute values using
  - B-trees, B+-trees, B*-trees
  - Hash tables

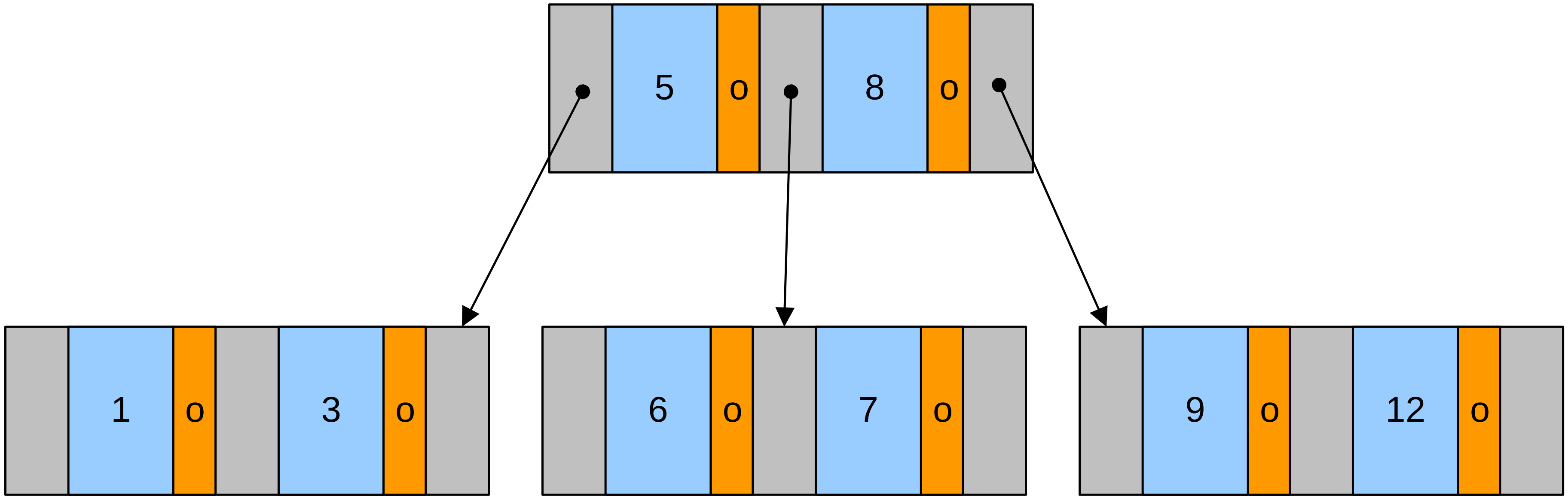
B-tree

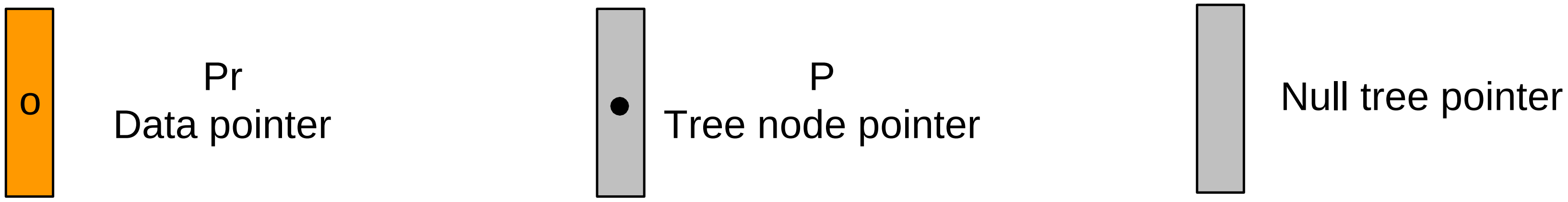
- B = Bayer or "Balanced"
  - Bayer: *Binary B-Trees for Virtual Memory*, ACM-SIGFIDET Workshop 1971
- Data structure
  - Balanced tree of order *p*
  - Node: <P_1, <K_1,Pr_1>, P_2, <K_2, Pr_3>, … P_q>

    $q \leq p$

    For all search key fields X in subtree P_i: $K_{i-1} < X < K_i$
- Algorithm
  - Guarantees logarithmic insert/delete time
  - Keeps tree balanced

B-tree


5
o
8
o
1
o
3
o
6
o
7
o
9
o
12
o


o
Pr
Data pointer
P
Tree node pointer
Null tree pointer

B-tree variants

- **B^+-tree**

  (More commonly used than B-tree)

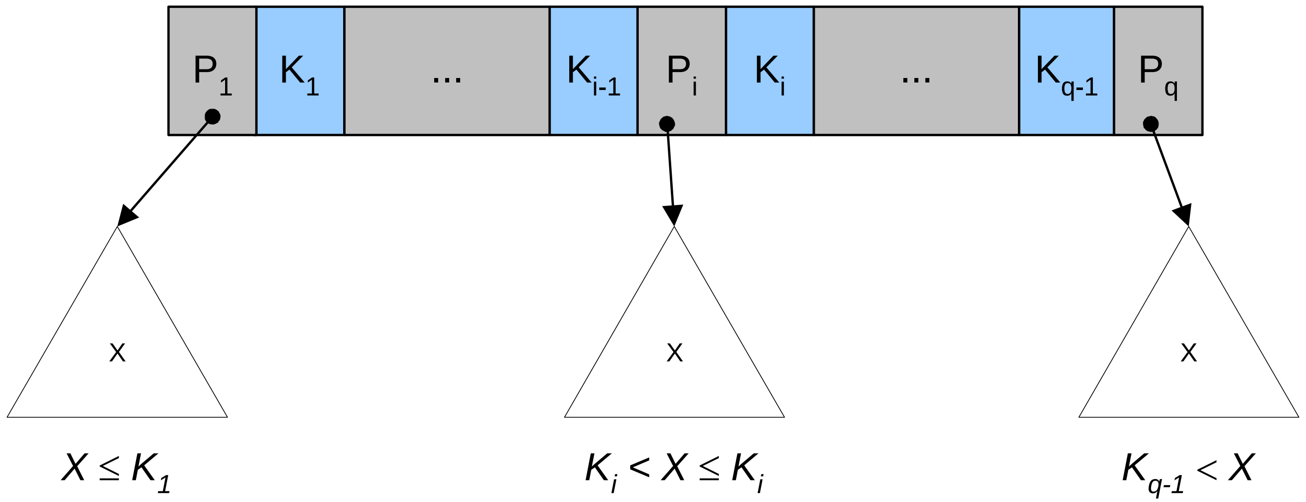
- **Data pointers *only* at the leaf nodes**
- **All leaf nodes linked together**

  ⇒ Allows ordered access

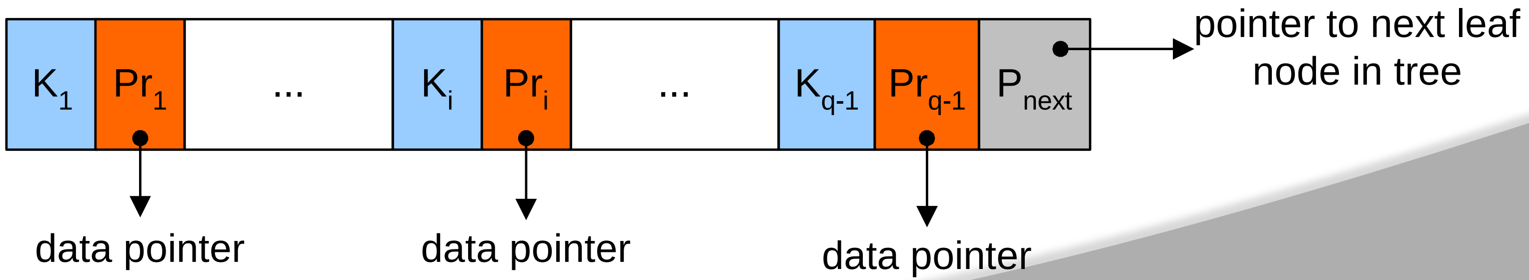
Internal node: $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$

Leaf node: $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$

B⁺-tree



P_1
K_1
...
K_{i-1}
P_i
K_i
...
K_{q-1}
P_q
X
X
X
$X \leq K_1$
$K_i < X \leq K_i$
$K_{q-1} < X$



K_1
Pr_1
...
K_i
Pr_i
...
K_{q-1}
Pr_{q-1}
P_{next}
pointer to next leaf node in tree
data pointer
data pointer
data pointer

B($^{+}$)-tree index SQL syntax

```
CREATE TABLE emp (
   ssn int(11) NOT NULL default
'0',
   name text,
   PRIMARY KEY  (ssn));

CREATE INDEX
part_of_name_index on emp
(name(10));
```

Multi dimensional index methods

- Point Access Methods
  - Grid files
  - *k*-D trees
- Spatial Access Methods
  - Space filling curves
  - R-trees
- Nearest (similarity)

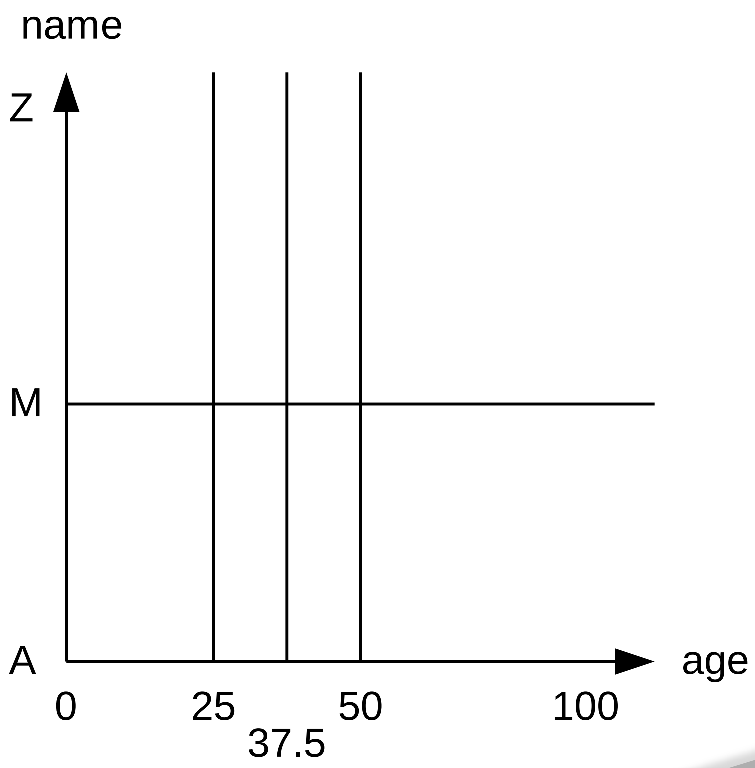
Applications

- GIS
- CAD
- Image analysis, computer vision
- Rule indexing
- Information Retrieval
- Multimedia databases

Grid files

”multi dimensional hashing”

- Partition address space:
  - Each cell corresponds to one disk page
  - Cuts allowed on predefined points only (¼, ½, ¾, …) on each axis
  - Cut all the way ⇒ a grid is formed



Grid files

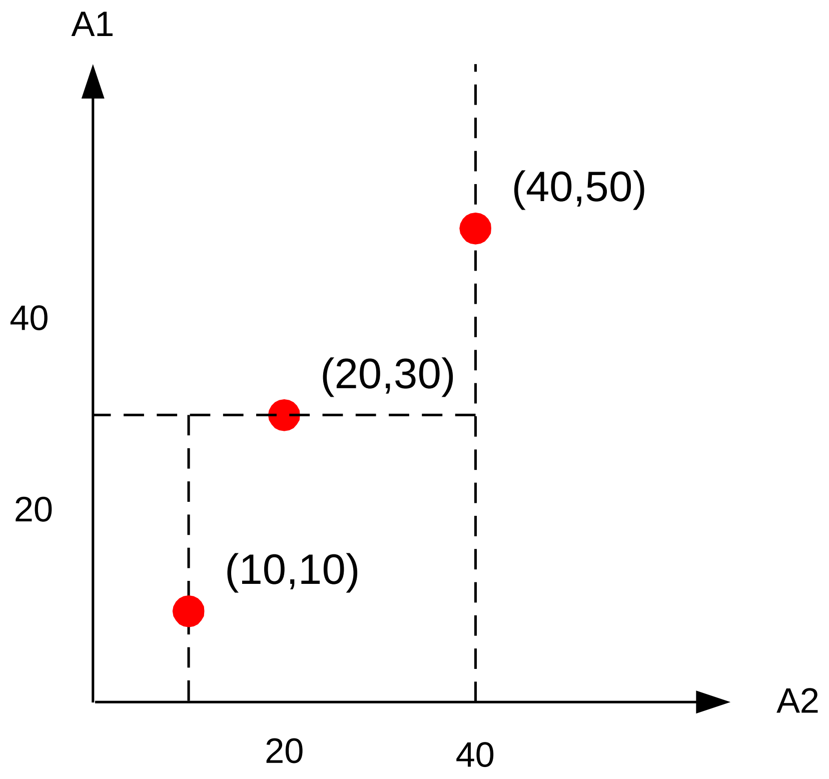
- Shortcomings
  - Correlated values:
  - Large directory is needed for high dimensionality
- OTOH:
  - Fast
  - Simple

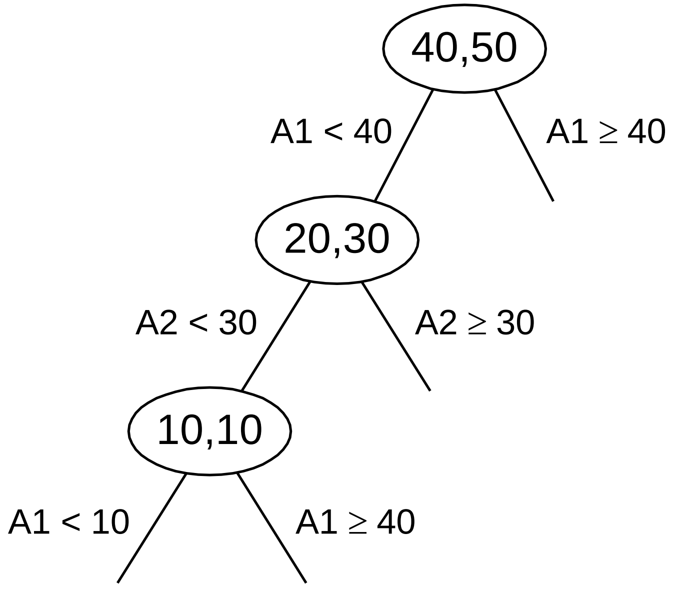
k-D trees

- Binary search tree
  - Each level splits in one dimension
    - dimension 0 at level 0,
    - dimension 1 at level 1
    - … (round robin)

Each internal node:
  - left pointer
  - right pointer
  - split value
  - data pointer

k-D trees


A1
(40,50)
40
(20,30)
20
(10,10)
A2
20
40


40,50
A1 < 40
A1 ≥ 40
20,30
A2 < 30
A2 ≥ 30
10,10
A1 < 10
A1 ≥ 40

k-D trees

- **Shortcomings**
    - **Incremental inserts/deletes can unbalance the tree**
        - **Re-balancing is difficult**
    - **Re-constructing the tree from scratch**

Space filling curves

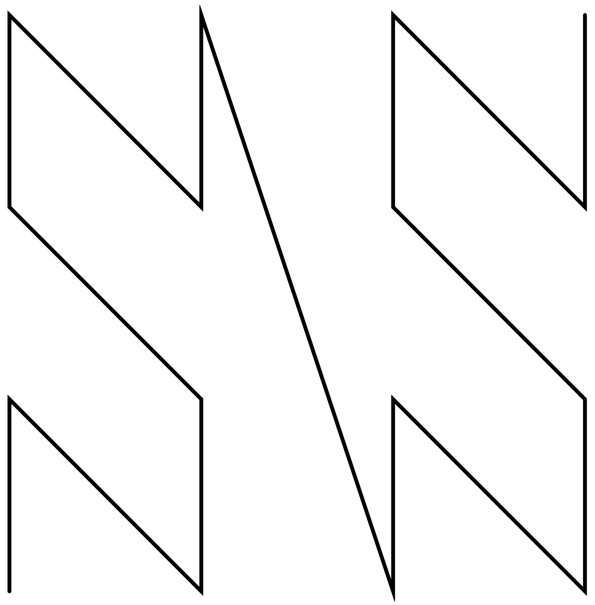
Idea: Impose a linear ordering on multidimensional data

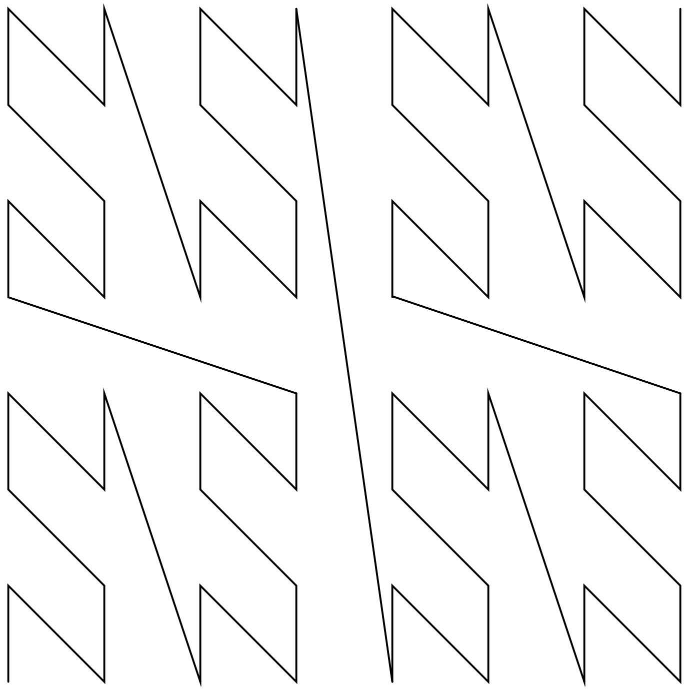
$\Rightarrow$

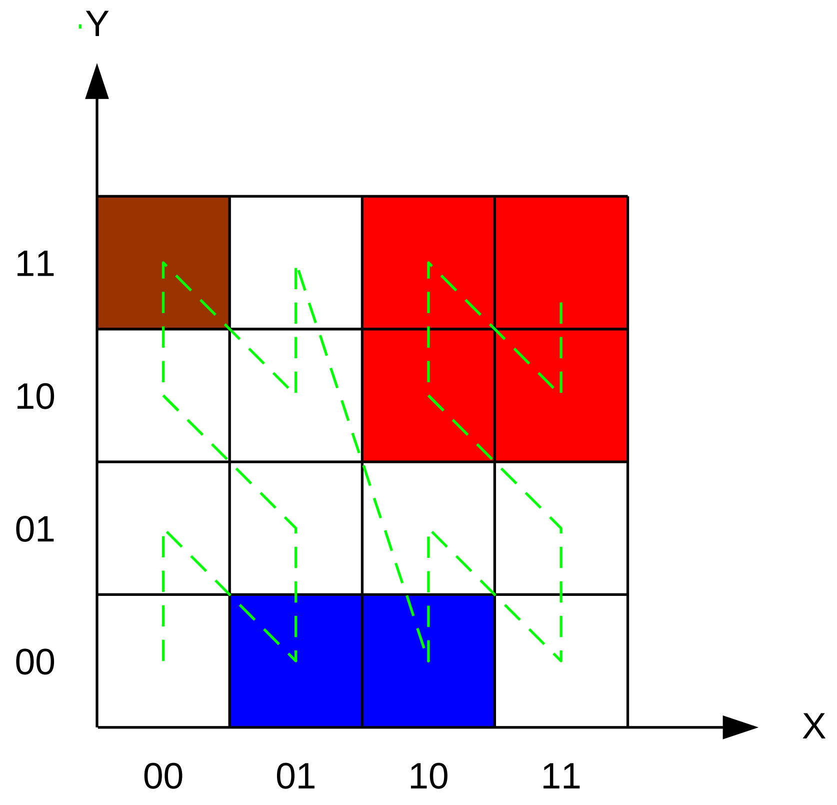
Allows for one-dimensional index and search on multidimensional data

- Z-ordering

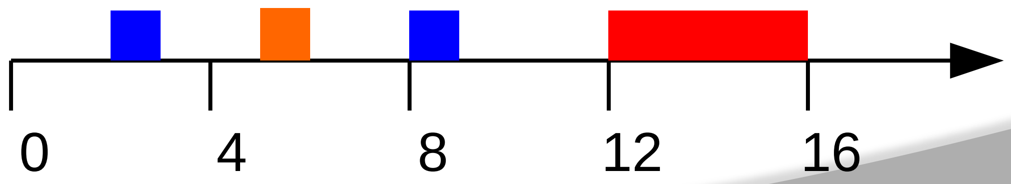
N





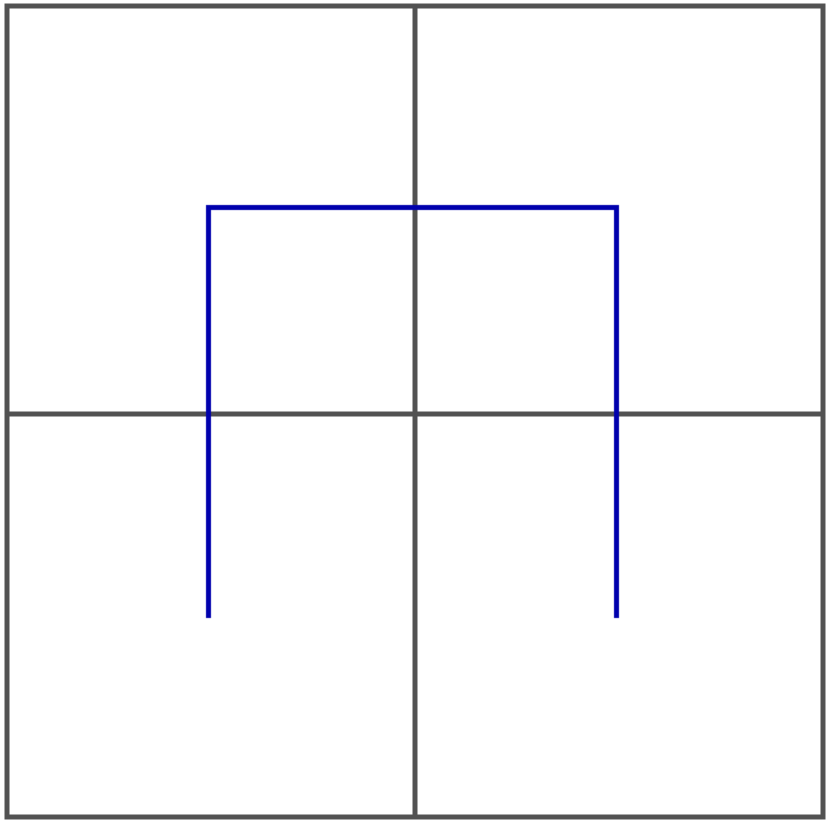


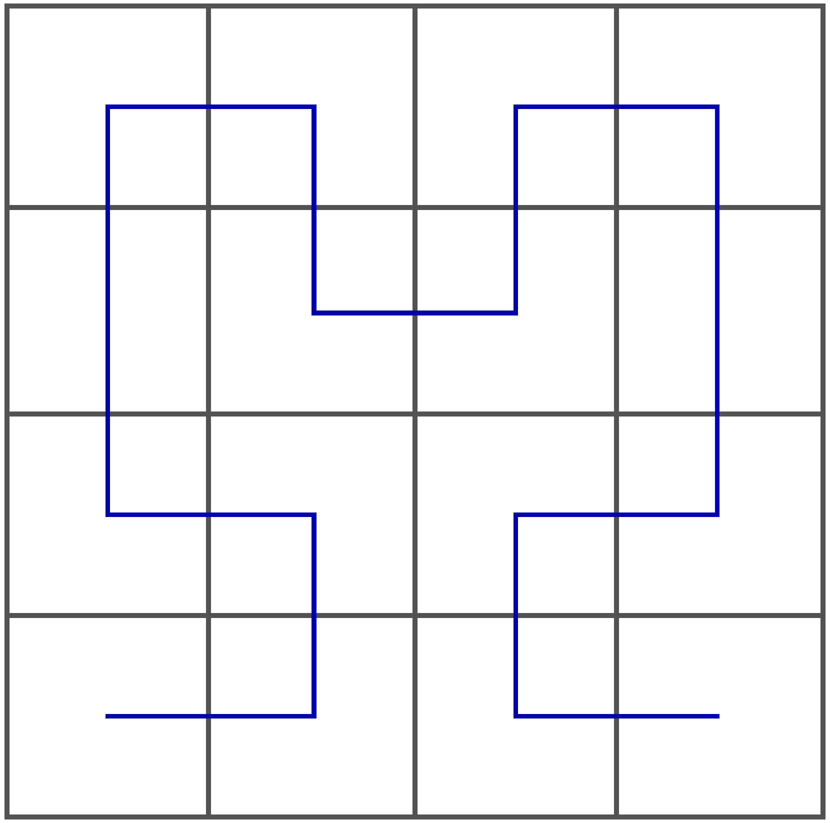
z_O= shuffle("1,2,1,2",x_O,y_O)
= shuffle("1,2,1,2",00,11)
= 0101 = $(5)_{10}$

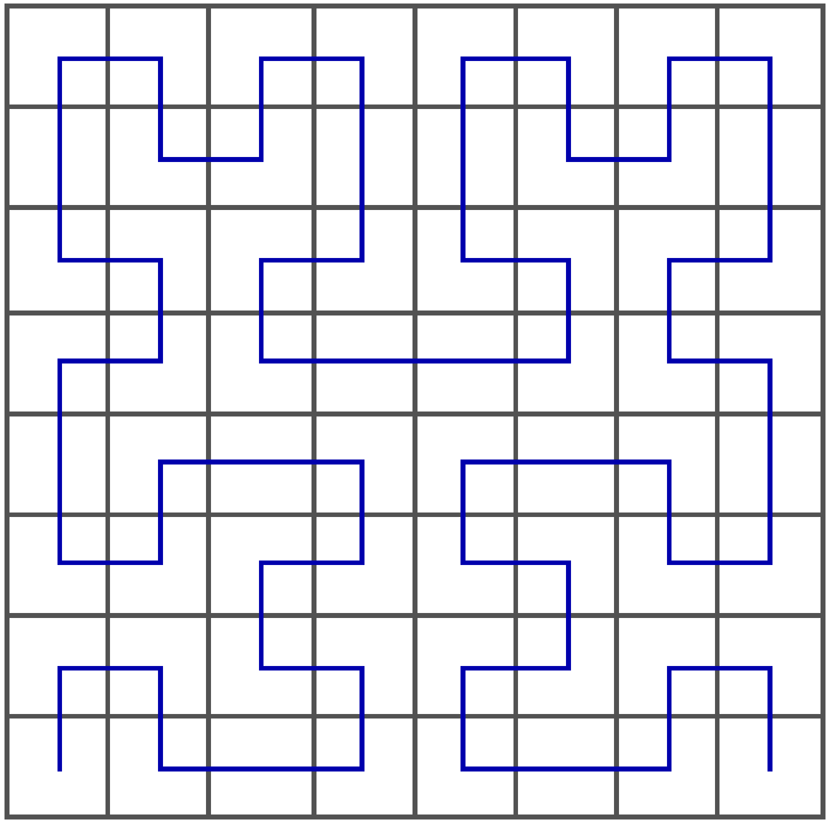


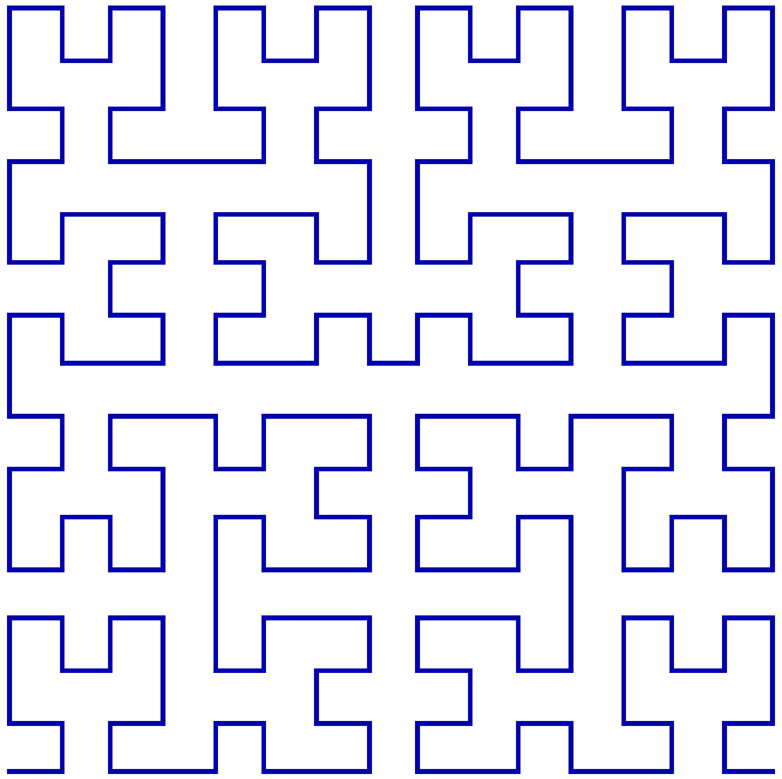
Hilbert curves

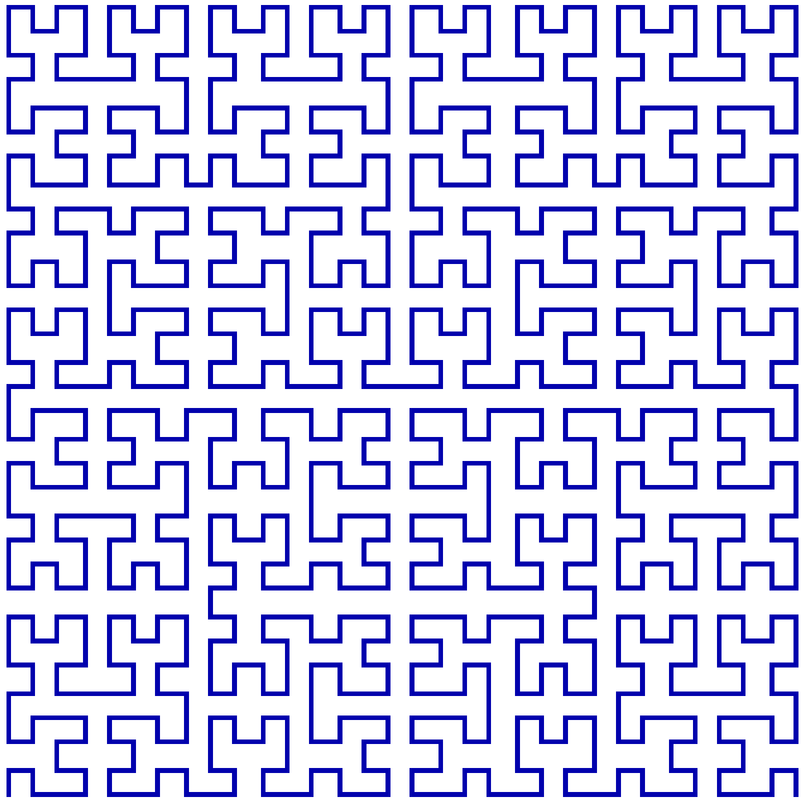
- Z-ordering has long diagonal jumps in space $\Rightarrow$
  - Connected objects split and separate far
  - Distances are not preserved
- Hilbert curves preserve distances better

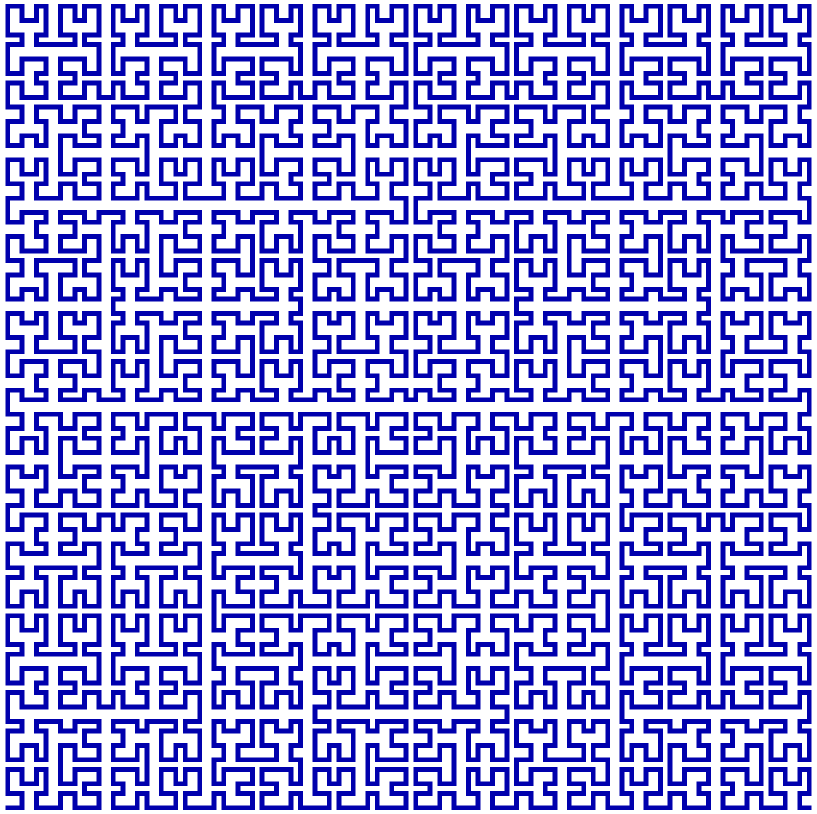












Space filling curves

- ”Quick” algorithm:

  O(b) for calculcating values

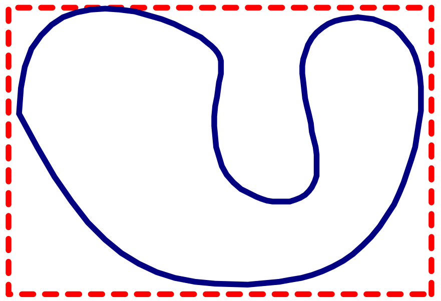
  - *b* – number of bits of the z/Hilbert value

    - typically, $b = x^D$

  - *x* – size of one dimension

R-trees

- **B-trees in multiple dimensions**
- **Spatial object represented by its MBR**



Rectangle

R-trees

- Nonleaf nodes
  - <*ptr*, *R*>
    - *ptr* – pointer to a child node
    - *R* – MBR covering all rectangles in the child node
- Leaf nodes
  - <*obj-id*, *R*>
    - *obj-id* – pointer to object
    - *R* – MBR of the object

R-trees

- Algorithms
  - Insert
    - Find the most suitable leaf node
    - Possibly, extend MBRs in parent nodes to enclose the new object
    - Leaf node overflow $\Rightarrow$ split
  - Split
    - Heuristics based

(Possible propagation upwards)

R-trees

- Range queries
  - Traverse the tree
    - Compare query MBR with the current node’s MBR
- Nearest neighbor
  - Branch and bound:
    - Traverse the most promising sub-tree
      - find neighbors
      - Estimate best- and worstcase
    - Traverse the other sub-trees
      - Prune according to obtained thresholds

R-trees

- Spatial joins

  ”find intersecting objects”

  - Naïve method:
    - Build a list of pairs of intersecting MBRs
    - Examine each pair, down to leaf level

  (Faster methods exist)

Variants

- R$^+$-tree

  (Sellis et al 1987)

  Avoids overlapping rectangles in internal nodes

- R*-tree

  (Beckmann et al 1990)

Applications

- Spatial databases
- Text retrieval
- Multimedia retrieval

Text retrieval

- Full text scanning
  - Somewhat like sequence analysis in bioinformatics
- Inversion
  - Build an index using keywords
- Signature files
  - A hash-like structure $\Rightarrow$ quick filtering of non-relevant material
- Vector space model
  - document clustering
- Performance measures
  - Precision, recall, average precision

Vector space model

- Hypothesis:

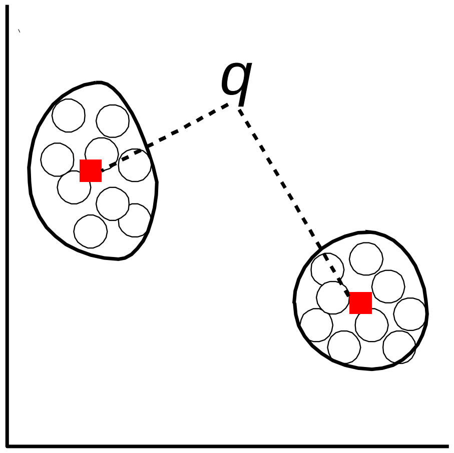
  Closely associated documents are relevant to the same requests
- Method:
  - For each document

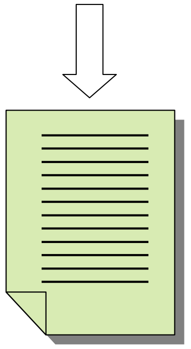
    Generate a histogram vector containing word counts, each bin counts one word
  - Group documents together in clusters, based on histogram vector similarity.
    - Popular metric: *Cosine similarity*

$$\cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\|\|\vec{y}\|}$$

Vector space model

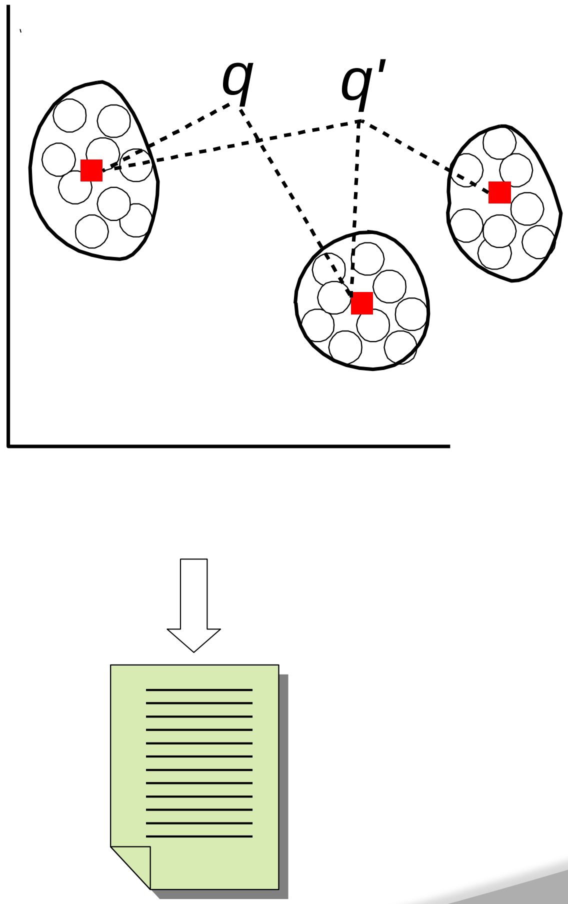
- Given a query phrase *q*
  - Generate a histogram vector of *q*
  - Compute similarity between *q* and all document cluster centroids
  - Compute similarity between *q* and all documents in the relevant clusters
  - Return a list of documents in descending similarity





Relevance feedback

- User pinpoints the most relevant documents
- These documents are added to the original query vector histogram $\Rightarrow q'$
- Similarity computations based on q'
- A new improved retrieval list is presented to the user



Retrieval performance

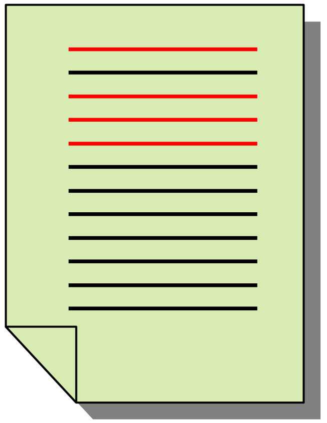
Precision *p*

The proportion of retrieved material that is relevant.

Given a retrieval list of *n* items,

$$p = \frac{g(n)}{n}$$

, where *g(n)* is the number of items in the list relevant to the query.



Retrieval performance

Average precision p_{avg}

How the relevant items are *distributed* in the retrieval list.

- R – the number of relevant items in the retrieval list
- n_i – the rank of each relevant item, $1 \leq i \leq R$
- For each n_i, calculate p_{ni} – the average precision of the partial list of top n_i items
- The average precision is the average of all p_{ni}:

$$p_{avg} = \frac{1}{R} \sum_{i=1}^{R} p_{n_i}$$

Multimedia databases

- Data structures
  - Bitmap image: 2D (3D) array of pixels
  - Sound clip/song: Sequence of samples
  - Video: Sequence of images
- User requirements
  - Music written by a particular artist
  - Texture similarity
  - ”Fuzzy” requirements, e.g. Musical preference

Multimedia databases

- Meta data queries
  - Images and video described by text
    - Figure captions
    - Keywords
    - Associated paragraphs
  - Retrieval based on text
    - Keywords
    - Textual features

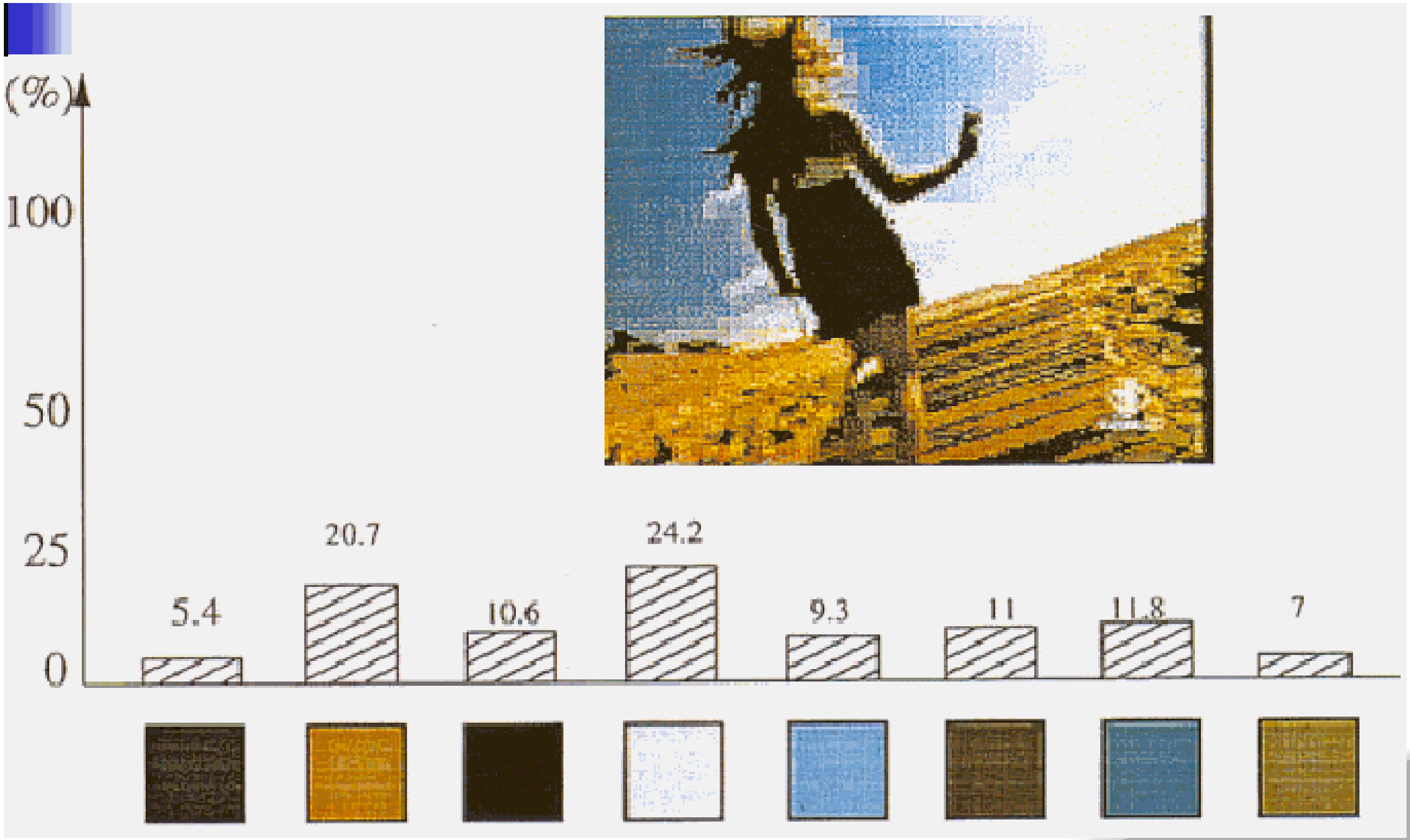
Features

- Images
  - Color of pixels
  - Line segments and edges
  - Texture
  - Shape
- Sound
  - Spectral content
  - Rhythm (music)
- Video
  - Motion

Color

- Perception-based models:
    - CIE chromaticity (X,Y,Z)
    - Opponent color model: Luv
    - Hue, saturation, value or brightness
- Hardware-oriented models: RGB, CMY
- Color histograms
    - Relative frequency distribution of each color dimension
    - Compute similarity between corresponding histograms of each color dimension

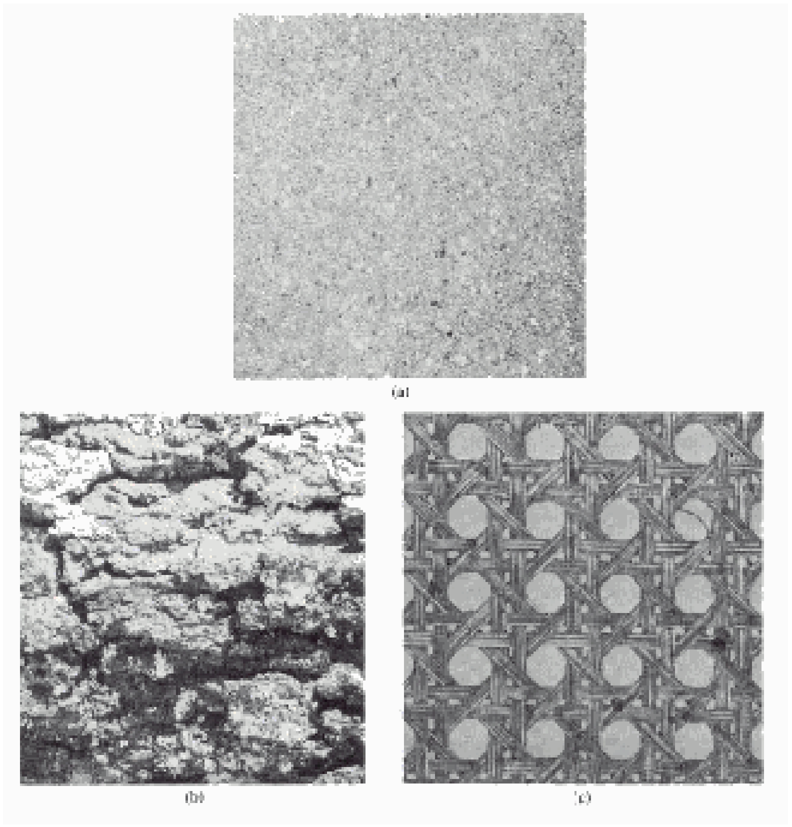
Histogram



Texture representation

- Pixel based
  - Co-occurrence matrix
  - Markov models
  - Auto-regressive models
- Pattern properites
  - Contrast
  - Orientation
  - PCA

Textures



Shapes, regions

- Image analysis methods
  - Description of regions
    - Moments or normalized moments
    - 2 D transforms
  - Description of boundaries
    - Chain encoding
    - Fourier descriptors
    - Skeletons
  - Regions
    - Edge detection
    - Corners detection
    - Edge Linking
    - Region segmentation
    - Region description

Video

- Segments, scenes, and basic frames
- Transitions
- Motion
  - Motion of objects
  - Camera
- Compression standards
  - MPEG 2 – Region coding and motion compensation
  - MPEG 4 – Content-based compression and synthetic data representation
  - MPEG 7 – Standardization of structures and arbitrary description schemes

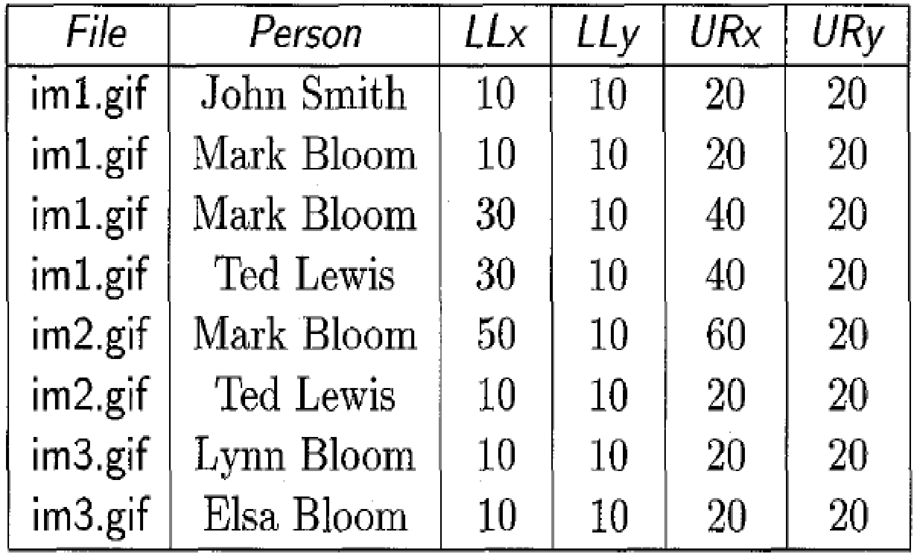
UNIT - V

UNCERTAINITY IN DATABASES AND KNOWLEDGE BASES

Uncertainty in Image Database:

For example, consider the problem of representing image content UNIT-II in a relational database. Consider a very simple relation called face that specifies which persons' faces are contained in which image files. Such a relation may have the schema

(File, Person, LLx, LLy, Ugx, URy)



File	Person	LLx	LLy	URx	URy
im1.gif	John Smith	10	10	20	20
im1.gif	Mark Bloom	10	10	20	20
im1.gif	Mark Bloom	30	10	40	20
im1.gif	Ted Lewis	30	10	40	20
im2.gif	Mark Bloom	50	10	60	20
im2.gif	Ted Lewis	10	10	20	20
im3.gif	Lynn Bloom	10	10	20	20
im3.gif	Elsa Bloom	10	10	20	20

Uncertainty in Image Database:

The attribute names may be interpreted as follows:

File is the name of an image file (e.g., iml .gif).

(LLx, LLy) and (URx, URy) specify the lower-left corner and the upperright corner of a rectangle (with sides parallel to the x- and y-axes) that bounds a particular person's face. Thus, in the above example, the first tuple indicates that there is a face (in iml. gif) in the rectangular region whose lower-left corner is at (10,10) and whose upper-right corner is at (20,20). Thus, the (LLx,LLy) and (URx,URy) components of any tuple uniquely capture a rectangle within the specified image.

Person specifies the name of the person whose face occurs in the rectangle specified by a tuple in this relation. Thus, for instance, the first tuple in the face relation states that the person in the rectangular region whose lower-left corner is at (10,10) and whose upper-right corner is at (20,20) is John Smith.

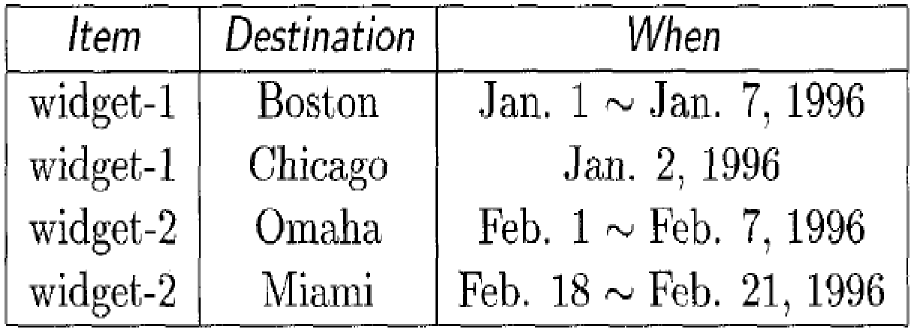
Uncertainty in Temporal Database

Often, a tuple in a relational database is time stamped with an interval of time. This often denotes the fact that the tuple was true at some time instant in that interval. For example, we may have a temporal relation called shipping that is maintained by a factory. This relation may have the schema

(Item, Destination).

When extended to handle temporal information, we may have a new additional attribute called ShipDate that denotes the date on which the item was shipped. The expanded shipping relation may contain the following tuples:

Uncertainty in Temporal Database



Item	*Destination*	*When*
widget-1	Boston	Jan. 1 ~ Jan. 7, 1996
widget-1	Chicago	Jan. 2, 1996
widget-2	Omaha	Feb. 1 ~ Feb. 7, 1996
widget-2	Miami	Feb. 18 ~ Feb. 21, 1996

The first tuple above says that the factory shipped an order of widget-1 to Boston sometime between January 1 and January 7 (inclusive). However, the precise date is unknown. Consider now the query "find all places to which widget-1 was shipped on or before January 5, 1996." As we will see below, some different answers are possible:

Uncertainty in DBs: A Null-Value Example

As you are probably aware, it is not always possible to associate a value with each and every column of each and every tuple in a given relation. For example, because of some unforeseen conditions (e.g., a coffee spill), the destination of a particular shipment may not be deductible from a given shipping invoice. However, the name of the intended recipient may be visible, leading the database administrator to conclude that the shipment was intended for one of the two factories of that company, located in New York and Denver.

Uncertainty in DBs: A Null-Value Example

The database administrator, after speaking to the shipping department, may conclude that most likely the shipment was intended for Denver (with 90% certainty). In this case, the following data may be entered into the database.

Fuzzy logic

In classical logic, there is a close correspondence between sets and logic. If F is a formula in such a logical language, then F denotes the set of all interpretations that satisfy it, where satisfaction. Formulas in fuzzy logic have exactly the same syntax as those of classical logic. However, they differ from classical logic in the following ways:

Fuzzy logic

An interpretation of a fuzzy language is a function, I, that maps ground atoms in the language to real numbers in the unit interval [0, 11].

The notion of satisfaction is fuzzy-if Sat(F) denotes the set of interpretations that satisfy F, then each interpretation I of the language has a degree of membership in Sat(F).

Fuzzy logic

$$I(\neg A) = 1 - I(A)$$
$$I(A \wedge B) = \min(I(A), I(B))$$
$$I(A \vee B) = \max(I(A), I(B))$$
$$I(\forall x.F) = \inf\{I(F[x/a]) \mid a \text{ is a ground term}\}$$
$$I(\exists x.F) = \sup\{I(F[x/a]) \mid a \text{ is a ground term}\}$$

Fuzzy Sets

We are all familiar with standard set theory (usually called naive set theory). Given a set S, we may associate with S a characteristic function Xs, defined as

Fuzzy Sets

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

Thus, all elements x are either in the set S or not.

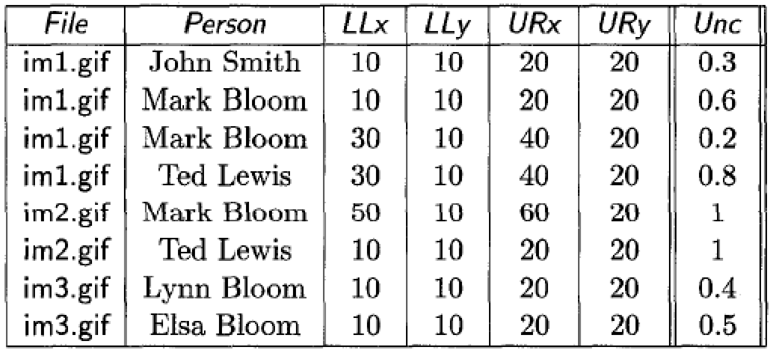
In contrast to the above behavior, in fuzzy sets, the function χ_S may assign any real number in the unit interval $[0, 1]$ to element x. Thus, a fuzzy set S has an associated characteristic function, χ_S, that assigns *grades* or *levels* of membership to elements x. Intuitively, if $\chi_S(x) = 0$, then this means that x is definitely not in set S; if $\chi_S(x) = 1$, then this means that x is definitely in set S and if $\chi_S(x_1) = 0.3$ while $\chi_S(x_2) = 0.4$, then the degree of membership of x_1 in S is somewhat less than the degree of membership of x_2 in S.

Uncertainty in Relational Databases

The relational model of data may be extended to incorporate uncertainty either at the tuple level or at the attribute level. In the tuple-level approach, we extend each tuple to have one or more uncertainty attributes. Typically, the uncertainty attribute would either be "a single real number r **G** [0, 1] or an interval [ri, r 2] of real numbers, or "a lattice element - drawn from the complete lattice of truth values being considered.

Lattice based relational databases

Suppose (L) is a complete lattice of truth values. Suppose R is a relation over schema (A_1,..., An). The tuple-based lattice extension, R^l, of relation R is a relation over schema (A_1 ,... , A_n, Unc) where dom(Unc) = L. A $_1$,... , A_n, are called the data attributes of R. Notice that R^l handles uncertainty at the tuple level, not the attribute level. If, for example, L = [0, 1],then the following table shows a tuple-level table that extends the face table



File	Person	LLx	LLy	URx	URy	Unc
im1.gif	John Smith	10	10	20	20	0.3
im1.gif	Mark Bloom	10	10	20	20	0.6
im1.gif	Mark Bloom	30	10	40	20	0.2
im1.gif	Ted Lewis	30	10	40	20	0.8
im2.gif	Mark Bloom	50	10	60	20	1
im2.gif	Ted Lewis	10	10	20	20	1
im3.gif	Lynn Bloom	10	10	20	20	0.4
im3.gif	Elsa Bloom	10	10	20	20	0.5