Top-k Queries: basics

Information Systems M

Prof. Paolo Ciaccia

http://www-db.deis.unibo.it/courses/SI-M/

Top-k queries: what and why

Top-k queries aim to retrieve,
from a potentially (very) large result set,
only the k ($k \geq 1$) best answers

- Best = most important/interesting/relevant/…
- The need for suck kind of queries arises in all the scenarios we have seen, as well as in many others
- The definition of top-k queries requires a system able to "**rank**" objects (the 1st best result, the 2nd one, …)

Ranking = ordering the DB objects based on their "relevance" to the query

Top-k queries: the naïve approach (1)

- There is a straightforward way to compute the result of any top-k query
- Assume that, given a query q, there is a *scoring function* S that assigns to each tuple t a numerical score, according to which tuples can be ranked
  - E.g. S(t) = t.Points + t.Rebounds

Name	Points	Rebounds	...
Shaquille O'Neal	1669	760	...
Tracy McGrady	2003	484	...
Kobe Bryant	1819	392	...
Yao Ming	1465	669	...
Dwyane Wade	1854	397	...
Steve Nash	1165	249	...
...	...	...	...

ALGORITHM Top-k-naïve
Input: a query q, a dataset R
Output: the k highest-scored tuples with respect to S

1. for all tuples t in R: compute S(t); // S(t) is the "score" of t
2. sort tuples based on their scores;
3. return the first k highest-scored tuples;
4. end.

Top-k queries: the naïve approach (2)

- Processing top-k queries using the naïve algorithm is very expensive for large databases, as it requires sorting a large amount of data
- The problem is even worse if the input consists of more than one relation

S(t) = t.Points + t.Rebounds

Name	Points	...
Shaquille O'Neal	1669	...
Tracy McGrady	2003	...
Kobe Bryant	1819	...
Yao Ming	1465	...
Dwyane Wade	1854	...
Steve Nash	1165	...
...	...	...

Name	Rebounds	...
Shaquille O'Neal	760	...
Tracy McGrady	484	...
Kobe Bryant	392	...
Yao Ming	669	...
Dwyane Wade	397	...
Steve Nash	249	...
...	...	...

- Now we have first to join all tuples, which is also a costly operation
- Note that in the above example the join is 1-1, but in general it can be M-N (each tuple can join with an arbitrary number of tuples)

Top-k queries in SQL (1)

- Expressing a top-k query in SQL requires the capability of:

  1) **Ordering** the tuples according to their scores
  2) **Limiting** the output cardinality to k tuples

- We first consider the case in which the following template query written in *standard SQL* is used, in which only point 1) above is present:

```
SELECT    <some attributes>
FROM      R
WHERE     <Boolean conditions>
ORDER BY S(…) [DESC]
```

- Possibly, we have to spend some effort in specifying our preferences this way, however the troubles are others…

Limits of the ORDER BY solution

- Consider the following queries:

```
A) SELECT     *
   FROM       UsedCarsTable
   WHERE      Vehicle = 'Audi/A4' AND Price <= 21000
   ORDER BY   0.8*Price + 0.2*Mileage
B) SELECT     *
   FROM       UsedCarsTable
   WHERE      Vehicle = 'Audi/A4'
   ORDER BY   0.8*Price + 0.2*Mileage
```

  - The values 0.8 and 0.2, also called "weights", are a way to normalize our preferences on Price and Mileage

- Query A will likely lose some relevant answers! (*near-miss*)
  - e.g., a car with a price of $21,500 but very low mileage
- Query B will return as result all Audi/A4 in the DB! (*information overload*)
  - …and the situation is horrible if we don't specify a vehicle type!!

ORDER BY solution & C/S architecture (1)

- Before considering other solutions, let's take a closer look at how the DBMS server sends the result of a query to the client application
- On the client side we work "1 tuple at a time" by using, e.g., **`rs.next()`**
  - However this does not mean that a result set is shipped (transmitted) 1 tuple at a time from the server to the client!
- Most (all?) DBMSs implement a feature known as *row blocking*, aiming at reducing the transmission overhead
- **Row blocking**:
  1. The DBMS allocates some buffers (a "block") on the server side
  2. It fills the buffers with tuples of the query result
  3. It ships the whole block of tuples to the client
  4. The client consumes (reads) the tuples in the block
  5. Repeat from 2 until no more tuples (rows) are in the result set

block of tuples

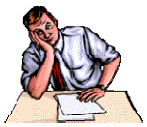
OBJ	Price
t07	10
t24	20
t16	32

block

t14	38
t21	40
t06	46
...	...

ORDER BY solution & C/S architecture (2)

- Why row blocking is not enough? I.e.: why do we need "k"?
- In DB2 the block size is established when the application connects to the DB (default size: 32 KB)
- If the buffers can hold, say, 1000 tuples but the application just looks at the first, say, 10, we waste resources:
  - We fetch from disk and process too many (1000) tuples
  - We transmit too many data (1000 tuples) over the network



- If we reduce the block size, then we might incur a large transmission overhead for queries with large result sets
  - Bear in mind that we don't have "just one query": our application might consist of a mix of queries, each one with its own requirements
- Also observe that the DBMS "knows nothing" about the client's intention, i.e., it will optimize and evaluate the query so as to deliver the whole result set (more on this later)

Top-k queries in SQL (2)

- The first step to support top-k queries is simple: *extend SQL with a new clause that explicitly limits the cardinality of the result*:

```
 SELECT     <some attributes>
 FROM       <some relation(s)>
 WHERE      <Boolean conditions>
[GROUP BY <some grouping attributes>]
 ORDER BY    S(…) [DESC]
 STOP AFTER k
```

  where k is a positive integer
- This is the syntax proposed in [CK97] (see references on the Web site), most DBMSs have proprietary (equivalent) extensions, e.g.:
  - **FETCH FIRST k ROWS ONLY (**DB2 UDB), **LIMIT TO k ROWS** (ORACLE), **LIMIT k**, …
  - [CK97] also allows a numerical expression, uncorrelated with the rest of the query, in place of k

Semantics of top-k queries

- Consider a top-k query with the clause **STOP AFTER k**
- Conceptually, the rest of the query is evaluated as usual, leading to a table T
- Then, only the first k tuples of T become part of the result
- If T contains at most k tuples, **STOP AFTER k** has no effect
- If more than one set of tuples satisfies the ORDER BY directive, any of such sets is a valid answer (non-deterministic semantics)

```
SELECT *
FROM   R
ORDER BY Price
STOP AFTER 3
```

T

OBJ	Price
t15	50
t24	40
t26	30
t14	30
t21	40

OBJ	Price
t26	30
t14	30
t21	**40**

OBJ	Price
t26	30
t14	30
t24	**40**

Both are valid results

- If no ORDER BY clause is present, then any set of k tuples from T is a valid (correct) answer (useful for sampling DB contents)

Top-k queries: examples (1)

- The best NBA player (considering points and rebounds):

```
SELECT *
FROM NBA
ORDER BY Points + Rebounds DESC
STOP AFTER 1
```

- The 2 cheapest chinese restaurants

```
SELECT *
FROM RESTAURANTS
WHERE Cuisine = `chinese'
ORDER BY Price
STOP AFTER 2
```

- The top-5% highest paid employees

```
SELECT E.*          -- a top-k query with a numerical expression
FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER (SELECT COUNT(*)/20 FROM EMP)
```


Top-k queries: examples (2)

- The top-5 Audi/A4 (based on price and mileage)

```
SELECT *
FROM USEDCARS
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5
```

- The 2 hotels closest to the Bologna airport

```
SELECT H.*                  -- a top-k distance join query
FROM HOTELS H, AIRPORTS A
WHERE A.Code = 'BLQ'
ORDER BY distance(H.Location,A.Location)
STOP AFTER 2
```

`Location` is a "point" UDT (User-defined Data Type)
`distance` is a UDF (User-Defined Function)

UDT's and UDF's

- Modern DBMSs allow their users to define (with some restrictions) new data types and new functions and operators for such types

```
CREATE TYPE Point AS (Float,Float) ...

CREATE FUNCTION distance(Point,Point)
     RETURNS Float
     EXTERNAL NAME 'twodpkg.TwoDimPoints!euclideandistance'
     LANGUAGE JAVA
     ...
```

package class method

☺ UDT's and UDF's are two basic ingredients to extend a DBMS so as it can support novel data types (e.g., multimedia data)

- Although we will not see details of UDT's and UDF's definitions, we will freely use them as needed

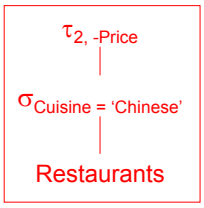
Evaluation of top-k queries

- Concerning evaluation, there are two basic aspects to consider:
  - query type: 1 relation, many relations, aggregate results, …
  - access paths: no index, indexes on all/some ranking attributes
- The simplest case to analyze is the top-k selection query, where only 1 relation is involved:

```
SELECT     <some attributes>
FROM       R
WHERE      <Boolean conditions>
ORDER BY   S(…) [DESC]
STOP AFTER k
```

Top-k queries: algebraic representation

- In order to concisely reason on alternative evaluation strategies, we have first to extend the relational algebra (RA)
- To this end, we introduce a logical Top operator, denoted $\tau_{k,S}$, which returns the k top-ranked tuples according to S
  - Unless otherwise specified, we assume that S has to be maximized



  - Later we will introduce a more powerful representation, due to [LCI+05], in which ranking (not just "limiting") is a "first-class citizen"

Implementing Top: physical operators

- How can the Top operator be evaluated?

2 relevant cases:

Top-Scan: the stream of tuples entering the Top operator is already sorted according to S: in this case it is sufficient to just read (consume) the first k tuples from the input

> Scan-Stop can work in pipeline:
> it can return a tuple as soon as it reads it!

Top-Sort: the input stream is not S-ordered; if k is not too large (which is the typical case), rather than sorting the whole input we can perform an in-memory sort

> Sort-Stop cannot work in pipeline:
> it has to read the whole input before returning the first tuple!

Physical operators: the iterator interface

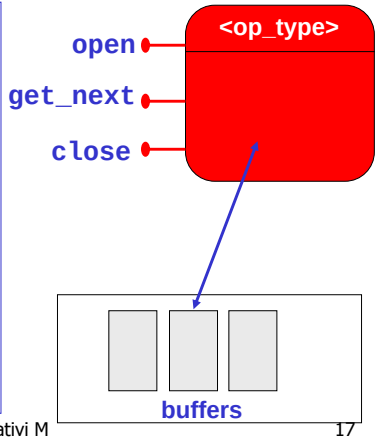
- In a DBMS, each operator is implemented as an iterator over one or more input streams
- Each physical operator must implement (at least) the methods:

open: initializes the state of the operator by allocating buffers for its inputs and output; it recursively invokes **open** on the children operators (input nodes); it is also used to pass input parameters (e.g., selection conditions)

get_next: requests another tuple to the operator; it includes the invocation of **get_next** on the input nodes and specific code that respects the operator semantics

has_next: just checks if get_next succeeds

close: terminates the execution and releases allocated system resources



The Top-Sort physical operator: open

- The idea of the Top-Sort method is to maintain in a main-memory buffer B only the best k tuples seen so far

RATIONALE: if tuple t is not among the top-k tuples seen so far, then t cannot be part of the result

- A crucial issue is how to organize B so that the operations of lookup, insertion and removal can be performed efficiently
- Since B should act as a *priority queue* (the priority is given by the score), it can be implemented using a *heap* (not described here)

Method open
Input: k, S
1. create a priority queue B of size k; // B can hold at most k tuples
   // B[i] is the current i-th best tuple, and B[i].score is its score
2. invoke open on the child node;
3. return.

The Top-Sort physical operator: get_next

- The get_next method first fills B with the first k tuples
  - For simplicity, the pseudocode does not consider the case when the input has less than k tuples
- Then, for each new read tuple t, it compares t with B[k], the worst tuple currently in B
  - If S(t) > B[k].score, then B[k] is dropped and t is inserted into B
  - If S(t) < B[k].score, t cannot be one of the top-k tuples
  - If S(t) = B[k].score, it is safe to discard t since Top has a non-deterministic semantics!

Method get_next

```
1.  for i=1 to k:                          // fills B with the first k tuples
2.      t := input_node.get_next(); ENQUEUE(B,t);  // inserts t in B
3.  while (input_node.has_next()) do:
4.      t := input_node.get_next();
4.      if S(t) > B[k].score then: DELETE(B,B[k]); ENQUEUE(B,t);
5.  return DEQUEUE(B).                     // returns the best tuple in B
```


Top-Sort: a simple example

- Let k = 2

EMP

ENO	Salary
E1	1000
E2	1200
E3	1400
E4	1100
E5	1500

B

compare

Order	ENO	Salary
2	E1	1000
1	E2	1200

insert E3

Order	ENO	Salary
1	E3	1400
2	E2	1200

compare

Order	ENO	Salary
1	E3	1400
2	E2	1200

compare

insert E5

Order	ENO	Salary
2	E3	1400
1	E5	1500

Experimental results from [CK97] (1)

```
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER N;
```

No Index available

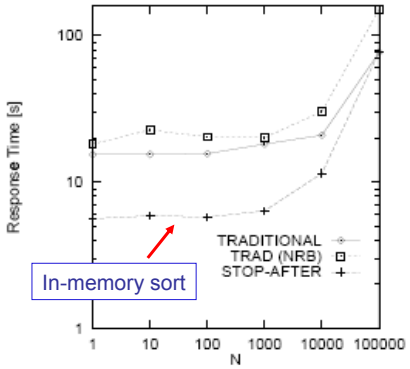


Figure 6: Resp. Time (log), Q1
No Index on Emp.salary.

- TRADITIONAL = row blocking (about 500 tuples)
- TRAD(NRB) = no row blocking

➡ The naïve method sorts ALL the tuples!

Results from [CK97] (2)

```
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER N;
```

Unclustered Index on Emp.Salary

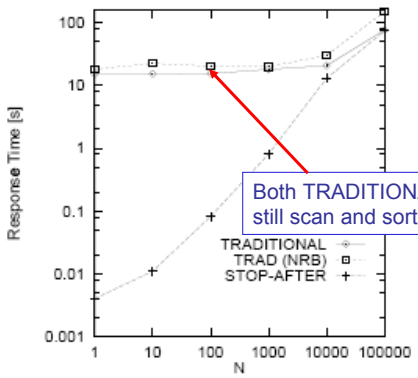


Figure 5: Resp. Time (log), Q1
Unclustered Index on Emp.salary.

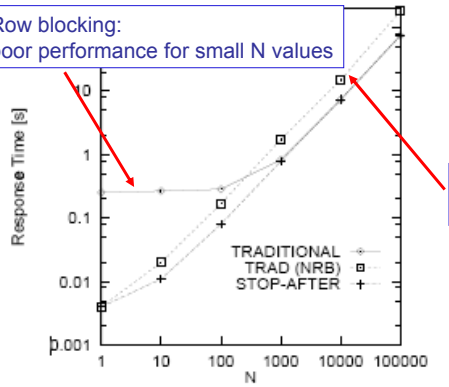
➡ If the DBMS ignores that we just need k tuples, it will not use the index: it will scan the EMP table and then sort ALL the N tuples!

Results from [CK97] (3)

```
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER N;
```

Clustered Index on Emp.Salary

Row blocking:
poor performance for small N values



No row blocking:
poor performance for large N values

Figure 4: Resp. Time (log), Q1
Clustered Index on Emp.salary.

Multi-dimensional top-k queries

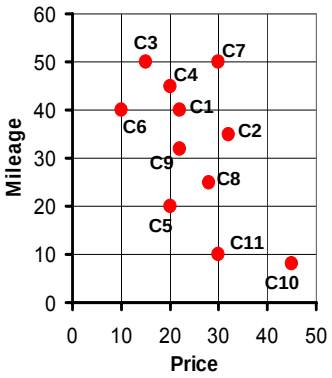
- In the general case, the scoring function S involves more than one attribute:

```
SELECT *
FROM USEDCARS
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

- If no index is available, we cannot do better than apply a Top-Sort operator by sequentially reading ALL the tuples ☹
- If an index is available on Vehicle the situation is better, yet it depends on how many Audi/A4 are in the DB 😐
  - Back to the 1st case if the WHERE clause is not present at all ☹
- Assume we have an *index* on the ranking attributes (i.e., Price and Mileage)
  - How can we use it to solve a top-k query?
  - What kind of index should we use?
- We first need to better understand the underlying geometry of the problem…

The "attribute space": a geometric view

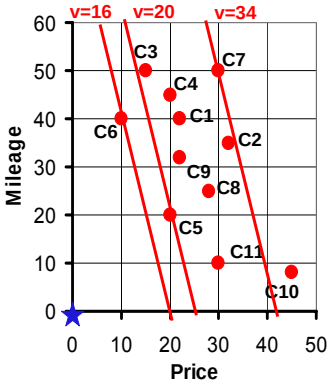
- Consider the 2-dimensional (2-dim) *attribute space* (Price,Mileage)



- Each tuple is represented by a 2-dim point (p,m):
  - p is the Price value
  - m is the Mileage value
- Intuitively, minimizing
  `0.8*Price + 0.2*Mileage`
  is equivalent to look for points "close" to (0,0)
- (0,0) is our (ideal) "target value" (i.e., a free car with 0 km's!)

The role of weights (preferences)

- Our preferences (e.g., 0.8 and 0.2) are essential to determine the result



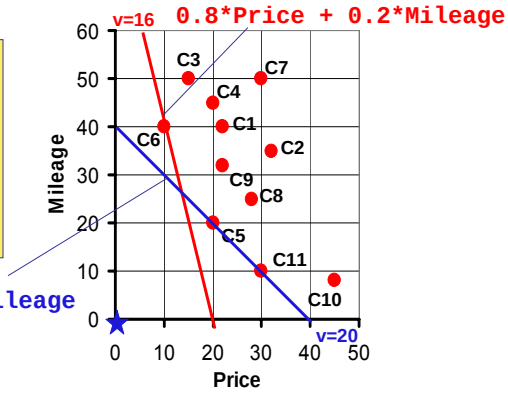
- Consider the line l(v) of equation
  `0.8*Price + 0.2*Mileage = v`
  where v is a constant
- This can also be written as
  `Mileage = -4*Price + 5*v`
  from which we see that all the lines l(v) have a slope = -4
- By definition, all the points of l(v) are "equally good" to us

- With preferences (0.8,0.2) the best car is C6, then C5, etc.
- In general, preferences are a way to determine, given points (p1,m1) and (p2,m2), which of them is "closer" to the target point (0,0)

Changing the weights

- Clearly, changing the weight values will likely lead to a different result

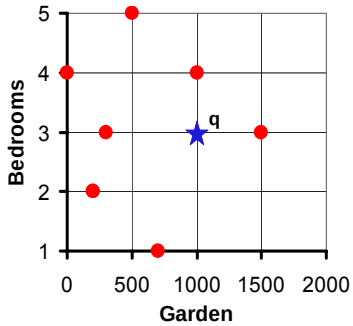
- With
  `0.8*Price + 0.2*Mileage`
  the best car is C6
- With
  `0.5*Price + 0.5*Mileage`
  the best cars are C5 and C11



- On the other hand, if weights do not change too much, the results of two top-k queries will likely have a high degree of overlap

Changing the target

- The target of a query is not necessarily (0,0), rather it can be any point q=(q1,q2) (qi = query value for the i-th attribute)
- Example: assume you are looking for a house with a 1000 m^2 garden and 3 bedrooms; then (1000,3) is the target for your query



- In general, in order to determine the "goodness" of a tuple t, we compute its "distance" from the target point q:
  The lower the distance from q, the better t is

Note that distance values can always be converted into goodness "scores", so that a higher score means a better match
Just change the sign and possibly add a constant,…

Top-k tuples = k-nearest neighbors

- In order to provide a homogeneous management of the problem when using an index, it is useful to consider distances rather than scores
  - since most indexes are "distance-based"
- Therefore, the model is now:
  - A D-dimensional (D≥1) attribute space **A** = (A1,A2,…,AD) of ranking attributes
  - A relation R(A1,A2,…,AD,B1,B2,…), where B1,B2,… are other attributes
  - A target (query) point q = (q1,q2,…,qD), q ∈ **A**
  - A function d: **A** x **A** → ℜ, that measures the distance between points of **A** (e.g., d(t,q) is the distance between t and q)
- Under this model, a top-k query is transformed into a so-called

k-Nearest Neighbors (k-NN) Query
- Given a point q, a relation R, an integer k ≥ 1, and a distance function d
- Determine the k tuples in R that are closest to q according to d

Some common distance functions

- The most commonly used distance functions are Lp-norms:

$$L_p(t,q) = \left(\sum_{i=1}^{D} |t_i - q_i|^p \right)^{1/p}$$

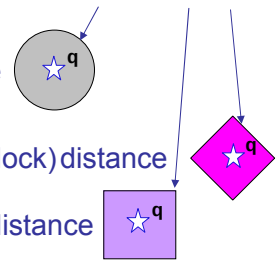
- Relevant cases are:

Iso-distance (hyper-)surfaces

$$L_2(t,q) = \sqrt{\sum_{i=1}^{D} |t_i - q_i|^2}$$ Euclidean distance

$$L_1(t,q) = \sum_{i=1}^{D} |t_i - q_i|$$ Manhattan (city – block) distance

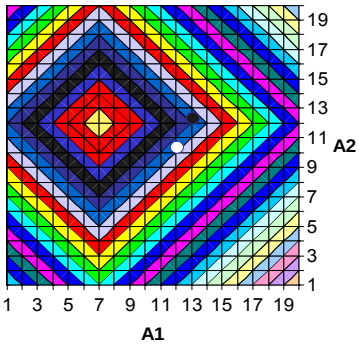
$$L_\infty(t,q) = \max_i \{|t_i - q_i|\}$$ Chebyshev (max) distance



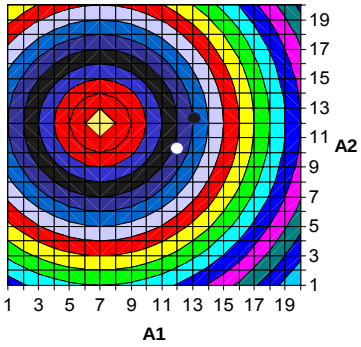
Shaping the attribute space

- Changing the distance function leads to a different shaping of the attribute space (each colored "stripe" in the figures corresponds to points with distance values between v and v+1, v integer)

L1; q=(7,12)



L2; q=(7,12)



Note that, for 2 tuples t1 and t2, it is possible to have
L1(t1,q) < L1(t2,q) and L2(t2,q) < L2(t1,q)

E.g.: t1=(13,12) ●
t2=(12,10) ○

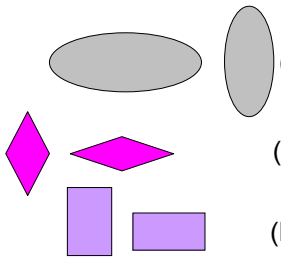
Distance functions with weights

- The use of weights just leads to "stretch" some of the coordinates:

$$L_2(t,q;W) = \sqrt{\sum_{i=1}^{D} w_i |t_i - q_i|^2}$$

$$L_1(t,q;W) = \sum_{i=1}^{D} w_i |t_i - q_i|$$

$$L_\infty(t,q;W) = \max_i \{w_i |t_i - q_i|\}$$



(hyper-)ellipsoids

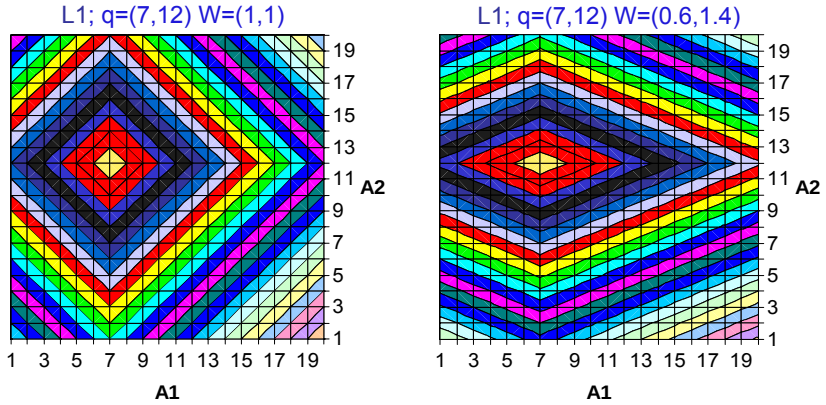
(hyper-)romboids

(hyper-)rectangles

- Thus, the scoring function
  `0.8*Price + 0.2*Mileage`
  is just a particular case of weighted L1distance

Shaping with weights the attribute space

- The figures show the effects of using L1 with different weights



- Note that, if w2 > w1, then the hyper-romboids are more elongated along A1 (i.e., difference on A1 values is less important than an equal difference on A2 values)

… and?

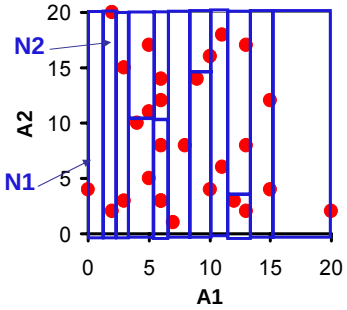
- Going back to our original problem:

How can we exploit indexes to solve multi-dimensional Top-k queries?

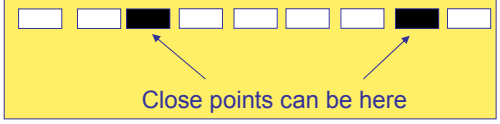
- As a first step we consider B+-trees, assuming that we have one multi-attribute index that organizes (sorts) the tuples according to the order A1,A2,…,AD (e.g., first on Price, then on Mileage)
- Again, we must understand what this organization implies from a geometrical point of view…

The geometry of B+-trees

- Consider the list of leaf nodes of the B+-tree: N1→N2→N3 →…
- The 1st leaf, N1, contains the smallest value(s) of A1, the number of which depends on the maximum leaf capacity C (=2*B+-tree order) and on data distribution
- The 2nd leaf starts with subsequent values, and so on
- The "big picture" is that the attribute space **A** is partitioned as in the figure

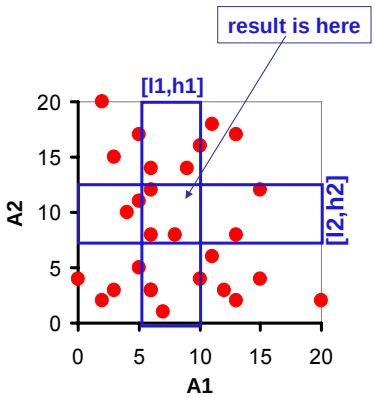


- No matter how we sort the attributes, searching for the k-NN of a point q will need to access too many nodes
- The basic reason is that "close" points of **A** are quite far apart in the list of leaves, thus moving along a coordinate (e.g., A1) will "cross" too many nodes



Another approach based on B+-trees

- Assume that we somehow know, e.g., using DB statistics (see [CG99]), that the k-NN of q are in the (hyper-)rectangle with sides [l1,h1]x [l2,h2]x…
- Then we can issue D independent range queries `Ai BETWEEN li AND hi` on the D indexes on A1,A2,…,AD, and then intersect the results



- Besides the need to know the ranges, with this strategy we waste a lot of work
- This is roughly proportional to the union of the results minus their intersection

➡ We will come back to the D-indexes scenario with smarter algorithms!

Multi-dimensional (spatial) indexes

- The multi-attribute B+-tree maps points of $\mathbf{A} \subseteq \Re^D$ into points of $\Re$
- This "linearization"necessarily favors, depending on how attributes are ordered in the B+-tree, one attribute with respect to others
  - A B+-tree on (X,Y) favors queries on X, it cannot be used for queries that do not specify a restriction on X
- Therefore, what we need is a way to organize points so as to preserve, as much as possible, their "spatial proximity"

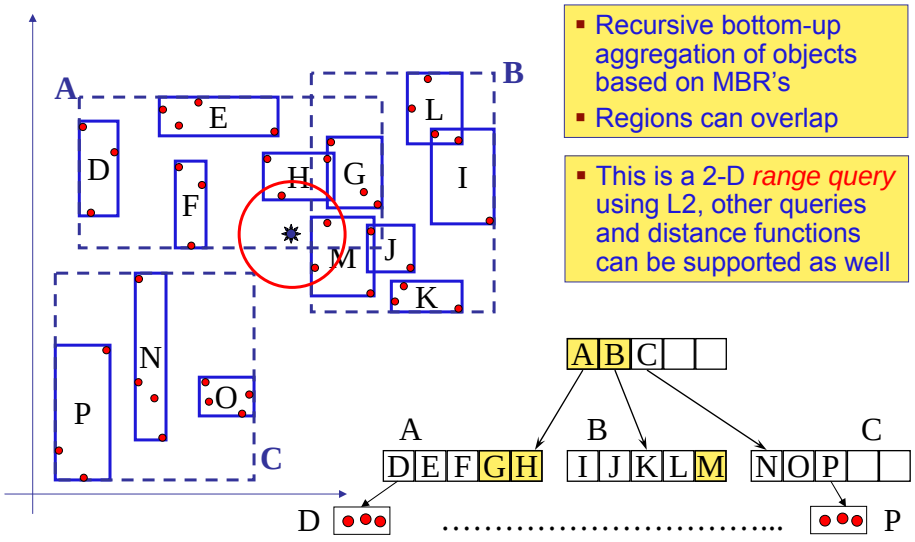
- The issue of "spatial indexing" has been under investigation since the 70's, because of the requirements of applications dealing with "spatial data" (e.g., cartography, geographic information systems, VLSI, CAD)
- More recently (starting from the 90's), there has been a resurrection of interest in the problem due to the new challenges posed by several other application scenarios, such as multimedia and data mining
- We will now just consider one (indeed very relevant!) spatial index…

The R-tree (Guttman, 1984)

- The R-tree [Gut84] is (somewhat) an extension of the B+-tree to multi-dimensional spaces, in that:
- The B+-tree organizes objects into
  - a set of (non-overlapping) 1-D intervals,
  - and then applies recursively this basic principle up to the root,
- the R-tree does the same but now using
  - a set of (possibly overlapping) m-D intervals, i.e., (hyper-)rectangles,
  - and then applies recursively this basic principle up to the root

- The R-tree is also available in some commercial DBMS's, such as Oracle
- In the following we just present the aspects relevant to query processing

R-tree: The intuition



- Recursive bottom-up aggregation of objects based on MBR's
- Regions can overlap

- This is a 2-D *range query* using L2, other queries and distance functions can be supported as well

R-tree basic properties (1)

- The R-tree is a dynamic, height-balanced, and paged tree
- Each node stores a variable number of *entries*

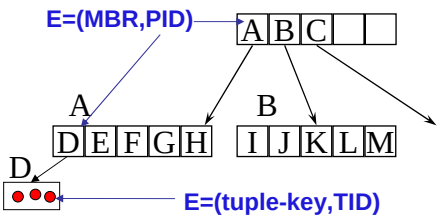
Leaf node:

- An entry E has the form E=(tuple-key,TID), where tuple-key is the "spatial key" (position) of the tuple whose address is TID (remind: TID is a pointer)

Internal node:

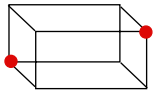
- An entry E has the form E=(MBR,PID), where MBR is the "Minimum Bounding Rectangle" (with sides parallel to the coordinate axes) of all the points reachable from ("under") the child node whose address is PID (PID = page identifier)

- We can uniform things by saying that each entry has the format
  E=(key,ptr)
- If N is the node pointed by E.ptr, then E.key is the "spatial key" of N



R-tree basic properties (2)

- The number of entries varies between c and C, with $c \leq 0.5*C$ being a design parameter of the R-tree and C being determined by the node size and the size of an entry (in turn this depends on the space dimensionality)
- The root (if not a leaf) makes an exception, since it can have as low as 2 children
- Note that a (hyper-)rectangle of $\Re^D$ with sides parallel to the coordinate axes can be represented using only 2*D floats that encode the coordinate values of 2 opposite vertices



Search: range query (1)

- We start with a query type simpler than k-NN queries, namely the

Range Query
- Given a point q, a relation R, a search radius $r \geq 0$, and a distance function d,
- Determine all the objects t in R such that $d(t,q) \leq r$

- The region of $\Re^D$ defined as $Reg(q) = \{p: p \in \Re^D, d(p,q) \leq r\}$ is also called the query region (thus, the result is always contained in the query region)
  - For simplicity, both d and r are understood in the notation Reg(q)
- In the literature there are several variants of range queries, such as:
  - Point query: when r = 0 (looking for a perfect (exact) match)
  - Window query: the query region is a (hyper-)rectangle (a window)
    - Window queries are just a special case of range queries obtained when the distance function is a weighted $L\infty$

Search: range query (2)

- The algorithm for processing a range query is extremely simple:
  - We start from the root and, for each entry E in the root node, we check if E.key intersects Reg(q):

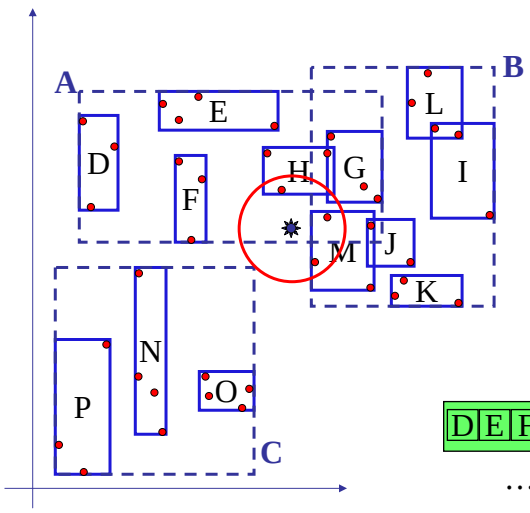
    Reg(q) ∩ E.key ≠ ∅: we access the child node N referenced by E.ptr

    Reg(q) ∩ E.key = ∅: we can discard node N from the search
  - When we arrive at a leaf node we just check for each entry E if E.key ∈ Reg(q), that is, if d(E.key,q) ≤ r.
    - If this is the case we can add E to the result of the index search

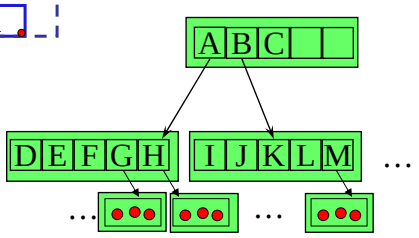
```
RangeQuery(q,r,N)
{ if N is a leaf   then: for each E in N:
                         if d(E.key,q) ≤ r then add E to the result
                   else: for each E in N:
                         if Req(q) ∩ E.key ≠ ∅ then RangeQuery(q,r,*(E.ptr) }
```

- The recursion starts from the root of the R-tree
  - The notation N = *(E.ptr) means "N is the node pointed by E.ptr"
  - Sometimes we also write ptr(N) in place of E.ptr

Range queries in action



- The navigation follows a depth-first pattern
- This ensures that, at each time step, the maximum number of nodes in memory is h=height of the R-tree
- Such nodes are managed using a stack



Search: k-NN query (1)

- With the aim to better understand the logic of k-NN search, let us define for a node N = *(E.ptr) of the R-tree its region as

  $\mathrm{Reg}(*(E.ptr)) = \mathrm{Reg}(N) = \{p: p \in \Re^D, p \in E.key=E.MBR\}$

- Thus, we access node N if and only if (iff) $\mathrm{Req}(q) \cap \mathrm{Reg}(N) \neq \emptyset$
- Let us now define $\mathbf{d_{MIN}(q,Reg(N)) = inf_p\{d(q,p) \mid p \in Reg(N)\}}$,

  that is, the minimum possible distance between q and a point in Reg(N)

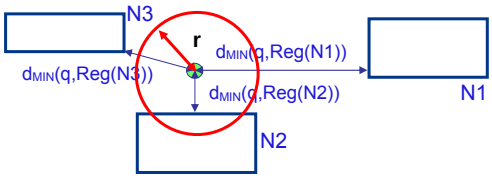
  - The "MinDist" $d_{MIN}(q,Reg(N))$ is a lower bound on the distances from q to any indexed point reachable from N

- We can make the following basic observation:

  $\mathbf{Reg(q) \cap Reg(N) \neq \emptyset}$

  $\mathbf{\Leftrightarrow}$

  $\mathbf{d_{MIN}(q,Reg(N)) \leq r}$



Search: k-NN query (2)

- We now present an algorithm, called kNNOptimal [BBK+97], for solving k-NN queries with an R-tree that is I/O-optimal
  - The algorithm also applies to other index structures (e.g., the M-tree) that we will see in this course
- We start with the basic case k=1
- For a given query point q, let $t_{NN}(q)$ be the 1st nearest neighbor (1-NN = NN) of q in R, and denote with $r_{NN} = d(q, t_{NN}(q))$ its distance from q
  - Clearly, r_{NN} is only known when the algorithm terminates

Theorem:

Any correct algorithm for 1-NN queries must visit at least all the nodes N whose MinDist is strictly less than r_{NN}, i.e., $d_{MIN}(q,Reg(N)) < r_{NN}$

Proof: Assume that an algorithm A stops by reporting as NN of q a point t, and that A does not read a node N such that (s.t.) $d_{MIN}(q,Reg(N)) < d(q,t)$; then Reg(N) might contain a point t' s.t. $d(q,t') < d(q,t)$, thus contradicting the hypothesis that t is the NN of q ■

The logic of the kNNOptimal Algorithm

- The kNNOptimal algorithm uses a priority queue PQ, whose elements are pairs [ptr(N), d_{MIN}(q,Reg(N))]
- PQ is ordered by *increasing values of* d_{MIN}(q,Reg(N))
  - DEQUEUE(PQ) extracts from PQ the pair with minimal MinDist
  - ENQUEUE(PQ, [ptr(N), d_{MIN}(q,Reg(N))]) performs an ordered insertion of the pair in the queue
- Pruning of the nodes is based on the following observation:

> - If, at a certain point of the execution of the algorithm, we have found a point t s.t. d(q,t) = r,
> - Then, all the nodes N with $d_{MIN}(q,Reg(N)) \geq r$ can be excluded from the search, since they cannot lead to an improvement of the result

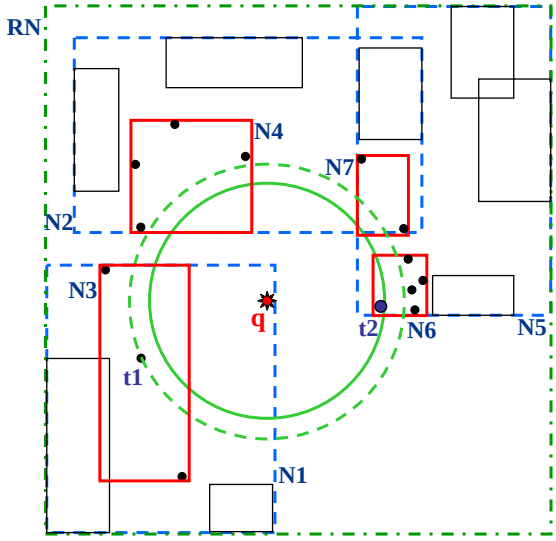
  - In the description of the algorithm, the pruning of pairs of PQ based on the above criterion is concisely denoted as UPDATE(PQ)
  - With a slight abuse of terminology, we also say that "the node N is in PQ" meaning that the corresponding pair [ptr(N), d_{MIN}(q,Reg(N))] is in PQ
- Intuitively, kNNOptimal performs a "*range search with a variable (shrinking) search radius*" until no improvement is possible anymore

The kNNOptimal Algorithm (case k=1)

Input: query point q, index tree with root node RN
Output: t_{NN}(q), the nearest neighbor of q, and r_{NN} = d(q, t_{NN}(q))

```
Initialize PQ with [ptr(RN),0];          // starts from the root node
r_NN := ∞;                               // this is the initial "search radius"
while PQ ≠ ∅:                                          // until the queue is not empty…
    [ptr(N), d_MIN(q,Reg(N))] := DEQUEUE(PQ);          // … get the closest pair…
    Read(N);                                           // … and reads the node
    if N is a leaf then: for each point t in N:
                    if d(q,t) < r_NN then: {t_NN(q) := t; r_NN := d(q,t); UPDATE(PQ)}
                                         // reduces the search radius and prunes nodes
                else: for each child node Nc of N:
                    if d_MIN(q,Reg(Nc)) < r_NN then:
                                         ENQUEUE(PQ,[ptr(Nc), d_MIN(q,Reg(Nc))]);
return t_NN(q) and r_NN;
end.
```

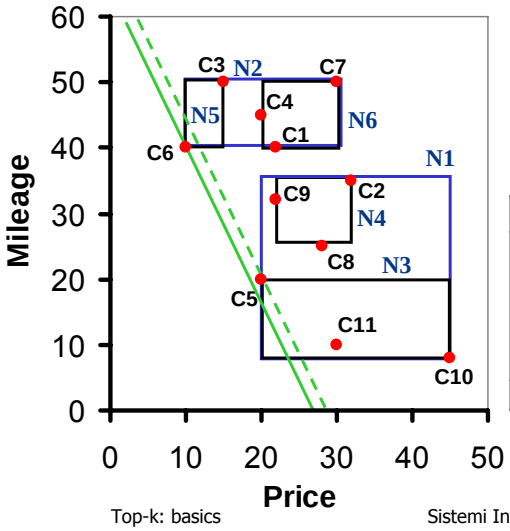
kNNOptimal in action



- Nodes are numbered following the order in which they are accessed
- Objects are numbered as they are found to improve (reduce) the search radius
- The accessed leaf nodes are shown in red

kNNOptimal: The best used car

d = 0.7*Price + 0.3*Mileage



Node	d_{MIN}
N1	16.4
N2	19
N3	16.4
N4	22.9
N5	19
N6	26

tuple	d
C5	20
C6	19
C11	24
...	...

Action	PQ		
Read(RN)	(N1,16.4)	(N2,19)	
Read(N1)	(N3,16.4)	(N2,19)	(N4,22.9)
Read(N3)	(N2,19)		
Read(N2)	(N5,19)		
Read(N5)			
Return(C6,19)			

Correctness and optimality of kNNOptimal

- The kNNOptimal algorithm is clearly correct
- To show that it is also I/O-optimal, that is, it reads the minimum number of nodes, it is sufficient to prove the following

Theorem:
The kNNOptimal algorithm for 1-NN queries never reads a node N whose MinDist is strictly larger than r_{NN}, i.e., $d_{MIN}(q,Reg(N)) > r_{NN}$

Proof:

- Node N is read only if, at some execution step, it becomes the 1st element in PQ
- Let N1 be the node containing $t_{NN}(q)$, N2 its parent node, N3 the parent node of N2, and so on, up to Nh = RN (h = height of the tree)
- Now observe that, by definition of MinDist, it is:

$$r_{NN} \geq d_{MIN}(q,Reg(N1)) \geq d_{MIN}(q,Reg(N2)) \geq \ldots \geq d_{MIN}(q,Reg(Nh))$$

- At each time step before we find $t_{NN}(q)$, one (and only one) of the nodes N1,N2,…,Nh is in the priority queue
- It follows that N can never become the 1st element of PQ ■

What if $d_{MIN}(q,Reg(N)) = r_{NN}$?

Q: The optimality theorem says nothing about regions whose MinDist equals the NN distance. Why?

A: Because it cannot!

Note that all such regions tie, thus their relative ordering in PQ is undetermined. The possible cases are:

1. The NN is in a node whose region has MinDist < r_{NN}.
   In this case no node with $d_{MIN}(q,Reg(N)) = r_{NN}$ will be read
2. The NN is in a node whose region has exactly MinDist = r_{NN}.
   Now everything depends on how ties are managed in PQ.
   In the worst case, all nodes with $d_{MIN}(q,Reg(N)) = r_{NN}$ will be read

The general case ($k \geq 1$)

- The algorithm is easily extended to the case $k \geq 1$ by using:
  - a data structure, which we call Res, where we maintain the k closest objects found so far, together with their distances from q
  - as "current search radius" the distance, $r_{k\text{-}NN}$, of the current k-th NN of q, that is, the k-th element of Res

Res

ObjectID	distance
t15	4
t24	8
t18	9
t4	12
t2	**15**

k = 5
- No node with distance ≥ 15 needs to be read

- The rest of the algorithm remains unchanged

The kNNOptimal Algorithm (case $k \geq 1$)

Input: query point q, integer $k \geq 1$, index tree with root node RN
Output: the k nearest neighbors of q, together with their distances

```
Initialize PQ with [ptr(RN),0];
for i=1 to k: Res[i] := [null,∞]; r_k-NN := Res[k].dist;
while PQ ≠ ∅:
    [ptr(N), d_MIN(q,Reg(N))] := DEQUEUE(PQ);
    Read(N);
    if N is a leaf then: for each point t in N:
                    if d(q,t) < r_k-NN then: {  remove the element in ResultList[k];
                                                insert [t,d(q,t)] in ResultList;
                                                r_k-NN := Res[k].dist; UPDATE(PQ)}
            else: for each child node Nc of N:
                    if d_MIN(q,Reg(Nc)) < r_k-NN then:
                                                ENQUEUE(PQ,[ptr(Nc), d_MIN(q,Reg(Nc))]);
return Res;
end.
```

Distance browsing

- Now we know how to solve top-k selection queries using a multi-dimensional index; but, what if our query is

```
SELECT *
FROM USEDCARS
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

  and we have an R-tree on (Price,Mileage) built over ALL the cars?
  - The k = 5 best matches returned by the index will not necessarily be Audi/A4
- In this case we can use a variant of kNNOptimal, which supports the so-called "distance browsing" [HS99] or "incremental NN queries"
- For the case k = 1 the overall logic for using the index is:
  - get from the index the 1st NN
  - if it satisfies the query conditions (e.g., AUDI/A4) then stop, otherwise get the next (2nd) NN and do the same
  - until 1 object is found that satisfies the query conditions

The get_next_NN algorithm

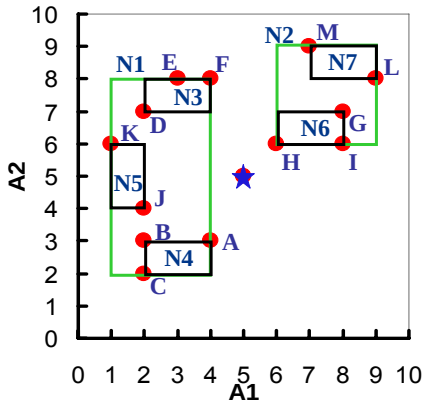
- In the queue PQ now we keep both tuples and nodes
  - If an entry of PQ is tuple t then its distance d(q,t) is written d_{MIN}(q,Reg(t))
- Note that no pruning is possible (since we don't know how many objects have to be returned before stopping)
- Before making the first call to the algorithm we initialize PQ with [ptr(RN),0]
- When a tuple t becomes the 1st element of the queue the algorithm returns

Input: query point q, index tree with root node RN
Output: the next nearest neighbor of q, together with its distance

```
while PQ ≠ ∅:
    [ptr(Elem), d_MIN(q,Reg(Elem))] := DEQUEUE(PQ);
    if Elem is a tuple t then: return t and its distance // no tuple can be better than t
            else: if N is a leaf then: for each point t in N: ENQUEUE(PQ,[t,d(q,t)])
                        else: for each child node Nc of N:
                                    ENQUEUE(PQ,[ptr(Nc), d_MIN(q,Reg(Nc))]);
end.
```

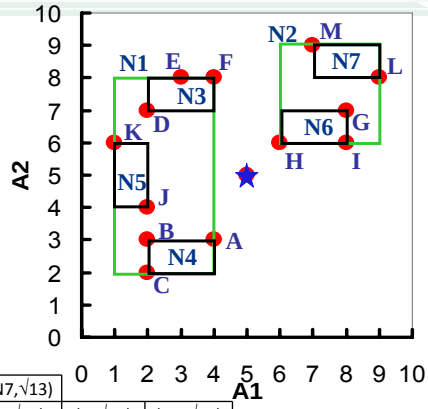
Distance browsing: An example (1/2)

- q=(5,5), distance: L2



Elem	d_{MIN}
A	√5
B	√13
C	√18
D	√13
E	√13
F	√10
G	√13
H	√2
I	√10
J	√10
K	√17
L	√25
M	√20
N1	√1
N2	√2
N3	√5
N4	√5
N5	√9
N6	√2
N7	√13

Distance browsing: An example (2/2)



Action	PQ									
Read(RN)	(N1,√1)	(N2,√2)								
Read(N1)	(N2,√2)	(N3,√5)	(N4,√5)	(N5,√9)						
Read(N2)	(N6,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(N7,√13)					
Read(N6)	(H,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(I,√10)	(G,√13)	(N7,√13)			
Return(H,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(I,√10)	(G,√13)	(N7,√13)				
Read(N3)	(N4,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(N7,√13)		
Read(N4)	(A,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(B,√13)	(N7,√13)	(C,√18)
Return(A,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(B,√13)	(C,√18)		
Read(N5)	…	…	…							
…										

Indexes as iterators

- get_next_NN is just an implementation of the general get_next method for indexes that support incremental k-NN queries
- In practice, the specific query type (range, k-NN, incremental k-NN, etc.), is a parameter passed to the index with the **`open`** method, after that a simple **`get_next()`** suffices

Recap

- The basic way to process a top-k selection query is to insert a Top-Sort operator in the access plan of the query, which saves the overhead of sorting the whole input stream
- If the scoring function S is 1-dimensional, a B+-tree index on the ranking attribute can be used to efficiently retrieve only the best k tuples
- On the other hand, for D-dim scoring functions an R-tree based solution is the only way to avoid unnecessary work. In this case the problem amounts to solve a k-NN query, where the query point q is the "target" of the search (the ideal case) and the distance function d is derived from the scoring function S