Top-k Queries on SQL Databases

Prof. Paolo Ciaccia
http://www-db.deis.unibo.it/courses/SI-LS/

02_TopK-SQL.pdf

Which queries for our scenarios?

- In all the scenarios we have seen (as well as many others):
  1. We have some "requirements/preferences", which are expressed in some way depending on the system interface
     - I would prefer leaving in December 2003, I would like to pay no more than $21,000 for a used car, I'm interested in images of St. Peter's Dome, …
  2. Order of results matters
     - Who will look at all the 41,074 results of Altavista for the "Pareto set" query?
- If we want to take into account both aspects we have to revise our notion of what the "result of a query" is
- Traditional (relational) view: given a database schema DB, a query Q is a function that computes, for each instance db of DB, a relation res with schema RES

  $Q: DB \rightarrow RES, \qquad res = Q(db)$
- Thus, the result is a relation, i.e., a set of tuples

Using ORDER BY

- SQL semantics differs in 2 key aspects from the pure relational view
  - We deal with tables, i.e., bags (multisets) of tuples, thus duplicates are allowed
  - Tables can be ordered using the ORDER BY clause
- Then, we might tentatively try to capture both aspects 1) and 2) in this way:

```
SELECT   <what we want to see>
FROM     <relevant tables>
WHERE    <query constraints>
ORDER BY <preference criteria> [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 sample queries:

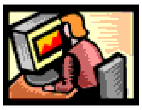
```
A)SELECT   *
  FROM     UsedCarsTable
  WHERE    Vehicle = 'Audi/A4' AND Price <= 21000
  ORDER BY 0.8*Price + 0.2*Mileage

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

- The values 0.8 and 0.2 are also called "weights": they are a way to express our preferences and to "normalize" Price and Mileage values
- 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 terrible 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 DBMS’s implement a feature known as *row blocking*, aiming at reducing the transmission overhead
- **Row blocking**:
  1. the DBMS allocates a certain amount of buffers 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

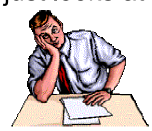
OBJ	Price
t07	10
t24	20
t16	32

buffers

t14	38
t21	40
t06	46
…	…

ORDER BY solution & C/S architecture (2)

- Why row blocking is not enough?
- In DB2 UDB 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) objects
  - 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

- The first part of our solution is indeed simple: *extend SQL with a new clause that explicitly limits the cardinality of the result*:

```
SELECT     <what we want to see>
FROM       <relevant tables>
WHERE      <query constraints>
ORDER BY   <preference criteria> [DESC]
STOP AFTER <value expression>
```

where `<value expression>` is any expression that evaluates to an integer value, and is uncorrelated with the rest of the query

- We refer to queries of this kind as **Top-k queries**
- We use the syntax proposed in [CK97] (see references on the Web site), some commercial DBMS's have proprietary (equivalent) extensions, e.g.:
  - DB2 UDB: **FETCH FIRST K ROWS ONLY**
  - ORACLE: **LIMIT TO K ROWS**

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

Top-k queries: examples (1)

- The 50 highest paid employees, and the name of their department

```
SELECT E.*, D.Dname
FROM EMP E, DEPT D
WHERE E.DNO = D.DNO
ORDER BY E.Salary DESC
STOP AFTER 50;
```

- The top 5% highest paid employees

```
SELECT E.*
FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER (SELECT COUNT(*)/20 FROM EMP);
```

- The 2 cheapest chinese restaurants

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


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.*
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 DBMS's 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

- As seen, it is not difficult to extend a DBMS so as to be able to specify a Top-k query (just extend SQL with the clause **`STOP AFTER K`**!)
- Concerning evaluation, there are two basic approaches to consider:

Naïve: compute the result without **`STOP AFTER`**, then discard tuples in excess

Integrated: extend the DBMS engine with a new (optimizable!) operator (i.e., the DBMS knows that we only want k tuples)

- In general, the naïve approach performs (very) poorly, since it wastes a lot of work:

  - ⇨ Fetches too many tuples
  - ⇨ Sorts too many tuples
  - ⇨ The optimizer may miss useful access paths

Why the naïve approach doesn't work (1)

- Consider the query asking for the 100 best paid employees:

```
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER 100;
```

  and assume that EMP contains 10,000 tuples and no index is available

- Carey and Kossmann [CK97a] experimentally show that the time to answer the above query is 15.633 secs, whereas their method (wait some minutes to see it!) requires 5.775 secs (i.e., 3 times faster)

➡ The naïve method needs to sort ALL the 10,000 tuples!

Why the naïve approach doesn't work (2)

- Consider again the query :

```
SELECT E.* FROM EMP E
ORDER BY E.Salary DESC
STOP AFTER 100;
```

  but now assume that an unclustered index on Salary is available

- If the DBMS ignores that we want just 100 tuples it will not use the index: it will sequentially scan the EMP table and then sort ALL the 10,000 tuples (the cost is the same as before: 15.633 secs)

  remind: *retrieving all the N tuples of a relation with an unclustered index can require a lot (up to N) random I/O's*

- On the other hand, if we use the index and retrieve only the first 100 tuples, the response time drops to 0.076 secs (i.e., 200 times faster)

➡ The naïve method cannot exploit available access methods!

Why the naïve approach doesn't work (3)

- Consider now the query :

```
SELECT E.*, D.Dname FROM EMP E, DEPT D
WHERE E.DNO = D.DNO
ORDER BY E.Salary DESC
STOP AFTER 100;
```

- Without knowledge of k, the optimizer will:
  - first join ALL the EMP tuples with the corresponding DEPT tuple (FK-PK join),
  - then, it will sort the join result on Salary
- Alternatively, we could FIRST determine the top 100 employees, then perform the join only for such tuples!

➡ The naïve method cannot discover good access plans!

The (S)top operator

- As a first step let's introduce a Stop (or Top, T) logical operator:

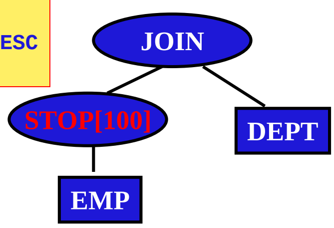
STOP[k;sort-directive](E)

Given the input expression E, the Stop operator produces the first k tuples according to the sort-directive. E.g.: STOP[100;Salary DESC](EMP)

➢ The naïve method always places the Stop "on-top" of an access plan

➢ Extending the DBMS can lead to more efficient access plans

```
SELECT E.*, D.Dname
FROM EMP E, DEPT D
WHERE E.DNO = D.DNO
ORDER BY E.Salary DESC
STOP AFTER 100;
```



Implementing the Stop: physical operators

- How can the Stop operator be evaluated?

2 relevant cases:

Scan-Stop: the input stream is already sorted according to the Sort directive: 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!

Sort-Stop: when the input is not sorted, if k is not too large (which is the typical case) 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!

The Sort-Stop physical operator

- Assume we want to determine the k best paid employees (i.e. we look at the k highest values of Salary)
- The idea is as follows:

1. **We allocate in main memory a buffer B of size k;**
2. **We insert the first k tuples in the buffer;**
3. **For each other tuple t in the input stream**
   1. **Compare t.Salary with the lowest salary in the buffer (B[k].Salary)**
   2. **If t.Salary > B[k].Salary then remove the tuple in B[k] and insert t in B else discard t**

- If t.Salary = B[k].Salary it is safe to discard t since Stop has a non-deterministic semantics!
- A crucial issue is how to organize B so that the operations of lookup, insertion and removal can be performed efficiently
- In practice, B is implemented as a *priority heap* (not described here)

Sort-Stop: a simple example

- Assume 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

compare

Order	ENO	Salary
1	E3	1400
2	E2	1200

compare

Order	ENO	Salary
1	E3	1400
2	E2	1200

insert E5

Order	ENO	Salary
2	E3	1400
1	E5	1500

Optimization of Top-k queries

- In order to determine an "optimal" plan for a Top-k query, the optimizer needs to understand

  where a Stop operator can be placed in the query tree

- We have already seen a query (EMP-DEPT join) where the Stop operator has been "pushed-down" a Join; is it always possible? **NO!**

- Consider the following:

```
SELECT E.*, D.Dname FROM EMP E, DEPT D
WHERE E.DNO = D.DNO AND D.Type = 'Research'
ORDER BY E.Salary DESC
STOP AFTER 100;
```

- If we perform the Stop BEFORE the Join we could generate a wrong answer, as the following example demonstrates…

Wrong Stop placement: an example

EMP

ENO	Name	Salary	DNO
E1	Tom	1000	D1
E2	Alice	1400	D2
E3	John	1200	D3
E4	Jane	1100	D2
E5	Mary	1500	D1

DEPT

DNO	Dname	Type
D1	TestTeam	Research
D2	Planning	Management
D3	DesignTeam	Research

- Assuming k = 2 the correct result is:

ENO	Name	Salary	DNO	Dname
E5	Mary	1500	D1	TestTeam
E3	John	1200	D3	DesignTeam

- If we first compute the Stop on EMP

ENO	Name	Salary	DNO
E5	Mary	1500	D1
E2	Alice	1400	D2

and then take the Join with DEPT:

ENO	Name	Salary	DNO	Dname
E5	Mary	1500	D1	TestTeam

The result is wrong!

Safe Stop placement rule

- If we anticipate the evaluation of Stop, we must be sure that none of its output tuples is subsequently discarded by other operators, that is:

if t is a tuple produced by the Stop operator,
then t must contribute to generate
at least one tuple in the query result

- This can be verified by looking at:
  - DB integrity constraints (FK,PK,NOT NULL,…), and
  - The query predicates that remain to be evaluated after the Stop is executed

Unsafe placement strategies

- In order to improve performance, [CK97] also considers "unsafe" (*aggressive*) placement strategies
- The idea is to insert in the query plan another Stop operator (in an unsafe place) with a value $k_{STOP} > k$, where k_{STOP} is estimated using DB statistics
- The estimation of k_{STOP} is based on the following: if t is a tuple in the input stream of the Stop operator, which probability p has t to "survive" after the application of the other operators? Then

$$k_{STOP} = k / p$$

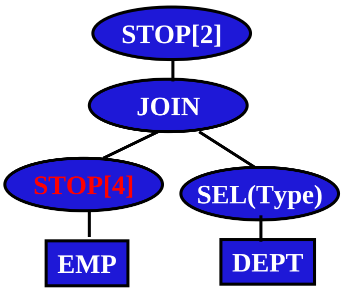
- If the result of the query contains at least k tuples we are done, otherwise it is necessary to *restart* the query with a larger value of k_{STOP}
- If the statistics are not accurate this strategy can lead to a poor performance, because many restarts might be needed

Unsafe placement strategies: an example

Consider the query

```
SELECT E.*, D.Dname FROM EMP E, DEPT D
WHERE E.DNO = D.DNO AND D.TYPE = 'Research'
ORDER BY E.Salary DESC
STOP AFTER 2;
```

- If we know that only about 1/2 of the Emp's work in a research Dept, then we can set k_{STOP} = k*2/1 = 4 and execute the following plan:



- The Stop[4] operator now returns

ENO	Name	Salary	DNO
E5	Mary	1500	D1
E2	Alice	1400	D2
E3	John	1200	D3
E4	Jane	1100	D2

from which the final result can be correctly computed

Multi-dimensional Top-k queries

- What if our preferences involve more than one attribute?

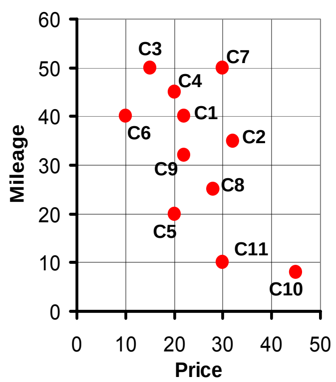
```
SELECT *
FROM USEDCARS
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

- If no index is available, we cannot do better than apply a Sort-Stop 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 😐
  - What if no predicate on Vehicle is specified? ☹
- If we have an index on Price and another on Mileage, we can "integrate" their results (we'll see later techniques for this case ☺)

- What if we have *just one index* (either on Price or on Mileage)?
- What if we have one *combined index* on Price and Mileage?
- In general, we first need to better understand the "geometry" of the problem

The "attribute space"

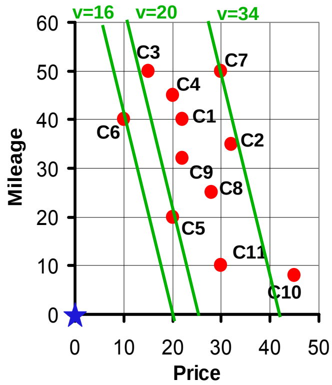
- Consider the 2-dimensional (2-D) attribute space (Price,Mileage)



- Each tuple is represented by a 2-D 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 preferences

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



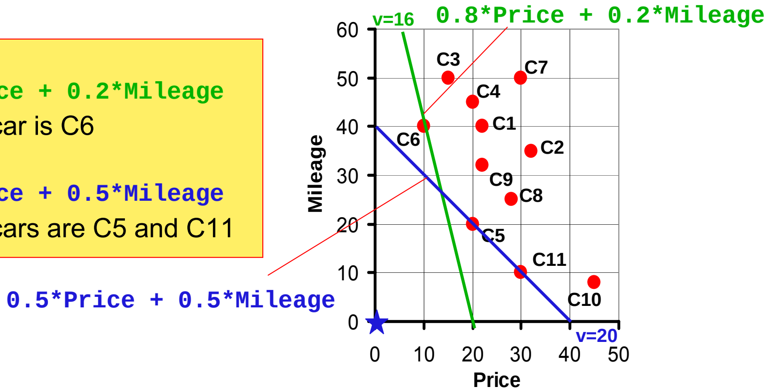
- Consider the line l(v) of equation
  `0.8*Price + 0.2*Mileage = v`
  where v is a constant
- This can also be written as
  `Mileage = -4*Price + 5*v`
  from which we see that all the lines l(v) have a slope = -4
- By definition, all the points of l(v) are “equally good” to us

- 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 preferences

- Clearly, changing preference will likely lead to a different result

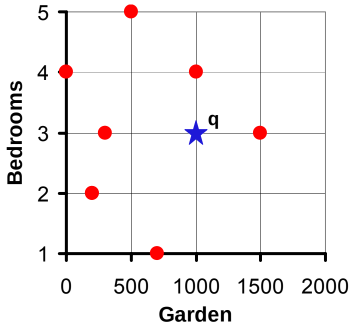
- With
  `0.8*Price + 0.2*Mileage`
  the best car is C6
- With
  `0.5*Price + 0.5*Mileage`
  the best cars are C5 and C11



- On the other hand, if preferences 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

Ranking the tuples

- Whenever we have
  - a “target” query value, and
  - a way to assert how well an object matches the query condition

  objects in the DB get “ranked”

  (i.e., each object has a “rank” = “position” in the result)
- Thus, the result of a query is not anymore a set of tuples, rather it is more properly interpreted as a ranked list of tuples
- We use the term “ranking” in place of “sorting” to make clear that the output order is dependent on a “goodness of match”
  - Alternatively: sorting is just needed for presentation purposes, ranking aims to present better results first and, with Top-k queries, to discard tuples whose rank is higher than k
- Going the other way, we can also say that, sometimes, sorting (according to the ranking criterion) is a way to produce a ranked result

Note that it is not always the case that the result exhibits the ranking criterion, which might remain “hidden” in the system (e.g., Web search engines)

A model with distance functions

- In order to provide a homogeneous management of the problem, we consider:
  - A D-dimensional (D≥1) attribute space **A** = (A1,A2,…,AD)
  - A relation R(A1,A2,…,AD,B1,B2,…), where B1,B2,… are other attributes
  - A target (query) point q = (q1,q2,…,qD), $q \in$ **A**
  - A function d: A x A $\rightarrow \Re$, that measures the distance between points of **A** (e.g., d(t,q) is the distance between t and q)
  - When preferences are specified, we represent them as a set of "weights" W, and write d(t,q;W) or $d_W(t,q)$ to make explicit that d(t,q) depends on W
- Our Top-k query is then transformed into a so-called

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

A closer look at distance functions

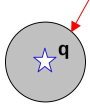
- The most commonly used distance functions are Lp-norms:

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

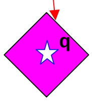
- Relevant cases are:

Iso-distance (hyper-)surfaces

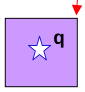
$$\mathbf{L_2(t,q)} = \sqrt{\sum_{i=1}^{D} |\mathbf{t_i} - \mathbf{q_i}|^2} \quad \text{Euclidean distance}$$



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



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

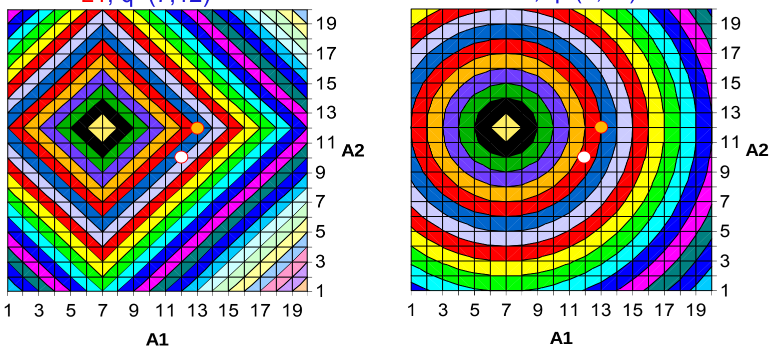


- …we will see other *metrics* later in this course

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)

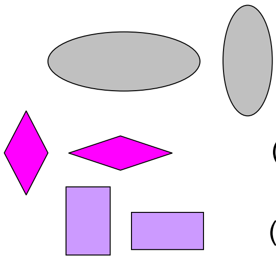
Distance functions with weights

- 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

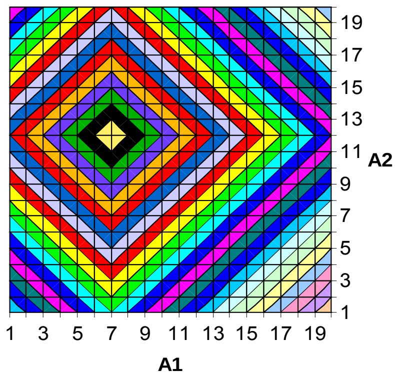
(hyper-)rectangles

- Thus, the preference
  0.8*Price + 0.2*Mileage
  is just a particular case of weighted L1distance!

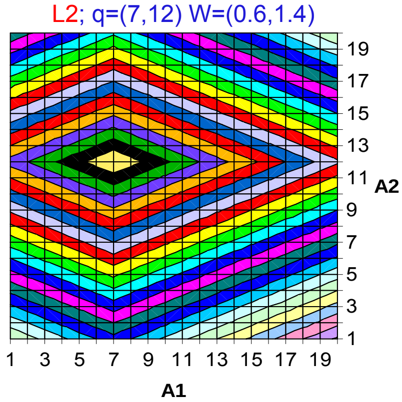
Shaping with weights the attribute space

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

L1; q=(7,12) W=(1,1)



L2; q=(7,12) W=(0.6,1.4)



- 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?

- Now that we have some understanding on the underlying geometry, it is time to go back to our original problem, that is:

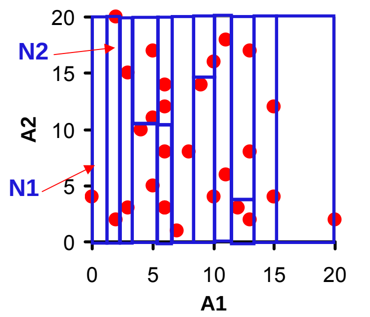
Can we exploit indexes to solve multi-dimensional Top-k queries?

- 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)

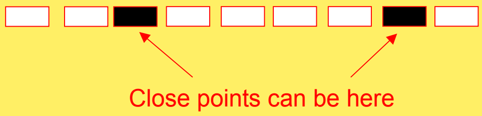
- Again, we must understand what this organization implies from a geometrical point of view…

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

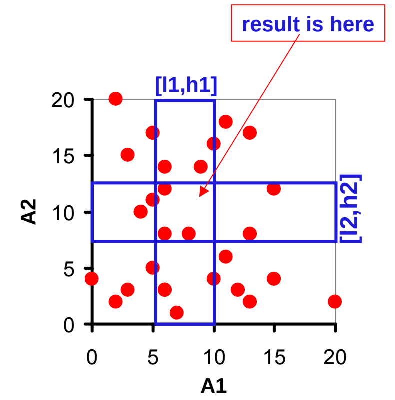


- No matter how we sort the attributes, searching for the k-NN of a point q will need to access too many nodes
- The basic reason is that "close" points of **A** are quite far apart in the list of leaves, thus moving along a coordinate (e.g., A1) will "cross" too many nodes



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 a smarter algorithm!

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
- We will now just consider one (indeed very relevant!) spatial index…

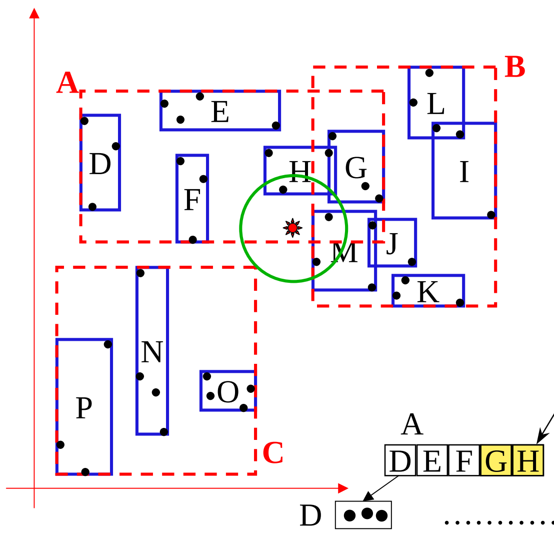
The R-tree (Guttman, 1984)

- 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 Oracle9i
- In the following we just present the aspects relevant to query processing, and postpone the discussion on R-tree management (insertion and split)

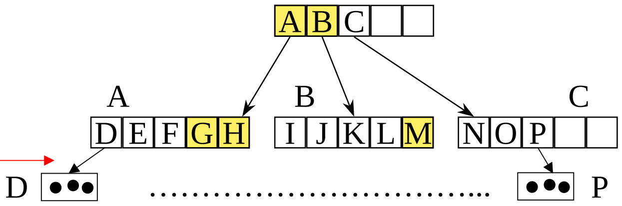
➡ Be sure to understand what the index looks like and how it is used to answer queries; for the moment don't be concerned on how an R-tree with a given structure can be built!

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*

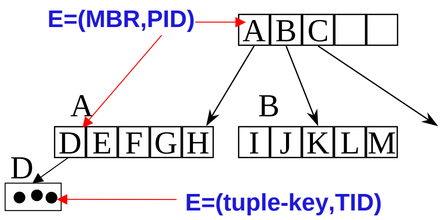
Leaf node:

- 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

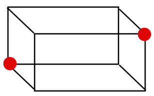


R-tree basic properties (2)

- 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 (i.e., it looks for a perfect (exact) match)
  - Window query: when the query region is a (hyper-)rectangle (i.e., a window)
    - We view window queries as a special case of range queries arising when the distance function is the weighted L_∞

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):

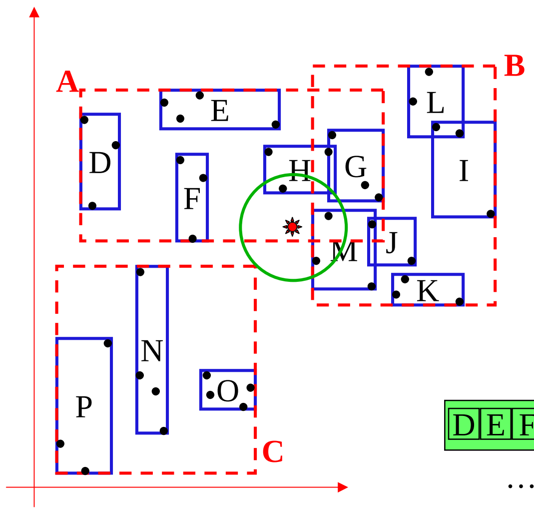
    Req(q) ∩ E.key ≠ ∅:   we access the child node N referenced by E.ptr

    Req(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

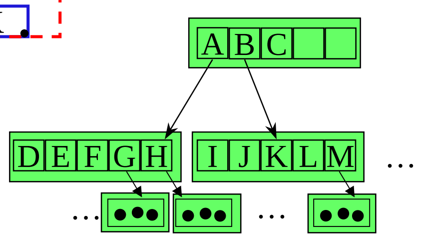
```
RangeQuery(q,r,N)
{ if N is a leaf   then: for each E in N:
                              if d(E.key,q) ≤ r then add E to the result
                   else: for each E in N:
                              if Req(q) ∩ E.key ≠ ∅ then RangeQuery(q,r,*(E.ptr) }
```

- 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)

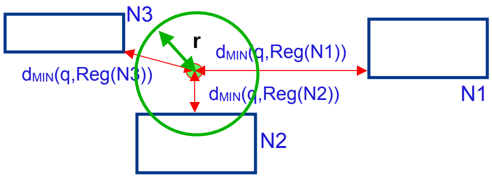
- With the aim to better understand the logic of k-NN search, let us define for a node N = *(E.ptr) of the R-tree its region as

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

- Thus, we access node N if and only if (iff) $Req(q) \cap Reg(N) \neq \emptyset$
- 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:

  $$\mathbf{Req(q) \cap Reg(N) \neq \emptyset}$$
  $$\Leftrightarrow$$
  $$\mathbf{d_{MIN}(q,Reg(N)) \leq r}$$



Search: k-NN query (2)

- We now present an algorithm, called kNNOptimal [BBK+97], for solving k-NN queries with an R-tree
  - The algorithm also applies to other index structures (e.g., the M-tree) that we will see in this course
- For simplicity, consider 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 algorithm for 1-NN queries must visit at least all the nodes N whose MinDist is less than r_{NN}

Proof: Assume that an algorithm ALG stops by reporting as NN of q a point t and that ALG 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

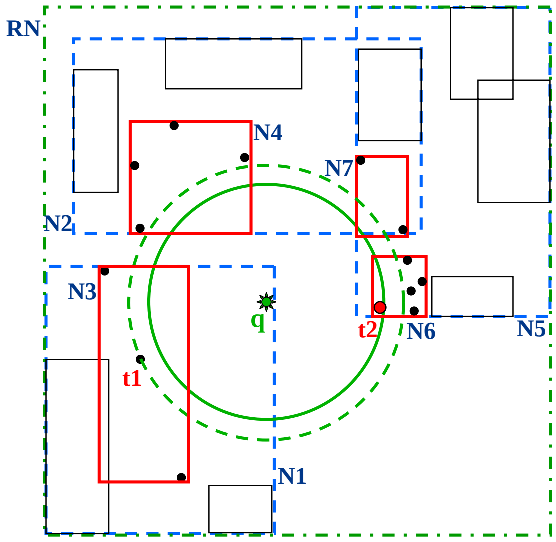
- The kNNOptimal algorithm uses a priority queue PQ, whose elements are pairs [ptr(N), d_{MIN}(q,Reg(N))]
- PQ is ordered by *increasing values of* d_{MIN}(q,Reg(N))
  - DEQUEUE(PQ) extracts from PQ the pair with minimal MinDist
  - ENQUEUE(PQ, [ptr(N), d_{MIN}(q,Reg(N))]) performs an ordered insertion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

Correctness and Optimality of kNNOptimal

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

it never reads a node N s.t. $d_{MIN}(q,Reg(N)) > r_{NN}$

Proof:

- Indeed, N is read only if, at a certain execution step, it becomes the 1st element in the priority queue 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 ■

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 ResultList, 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 ResultList

ResultList

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: ResultList(i) := [null,∞]; r_k-NN := ResultList(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 := ResultList(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 ResultList;
end.
```

Distance browsing

- Ok, now we know how to solve Top-k query 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) but 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 next-NN algorithm

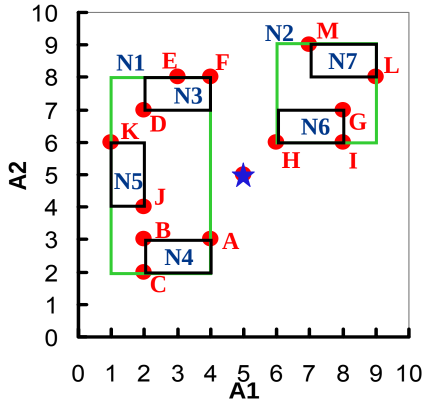
- In the queue PQ now we keep both objects and nodes!
  - If an entry of PQ is an object t then its distance d(q,t) is written d_{MIN}(q,Reg(t))
- Note that no pruning is possible (since we don't know how many objects have to be returned before stopping)
- Before making the first call to the algorithm we initialize PQ with [ptr(RN),0]
- When an object 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), dMIN(q,Reg(Elem))] := DEQUEUE(PQ);
    if Elem is a tuple t then: return t and its distance // no object 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), dMIN(q,Reg(Nc))]);
end.
```

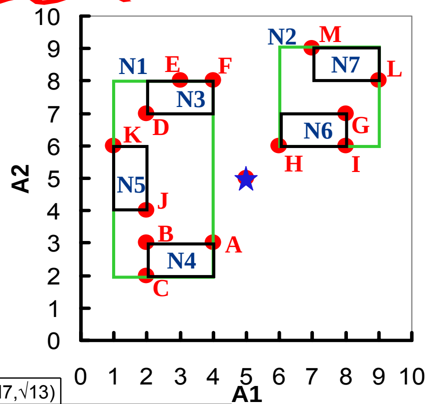
Distance browsing: An example (1/2)

- 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)	…	…	…							
…										