course:

Searching the Web and Multimedia Databases (BI-VWM)

SS2011/12

lecture 9:

Similarity queries and aggregation operators

doc. RNDr. Tomáš Skopal, Ph.D.
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague
Department of Software Engineering, Faculty of Information Technology, Czech Technical University in Prague

https://edux.fit.cvut.cz/courses/BI-VWM/

Today’s lecture outline

- fundamentals
- similarity queries
  - similarity ordering, range, kNN, kRNN
  - possible integration into SQL
- operators
  - similarity joins, self-joins
  - (k) closest pairs
  - skyline operator
  - top-k operator
    - Fagin algorithm
    - Threshold algorithm

Fundamentals

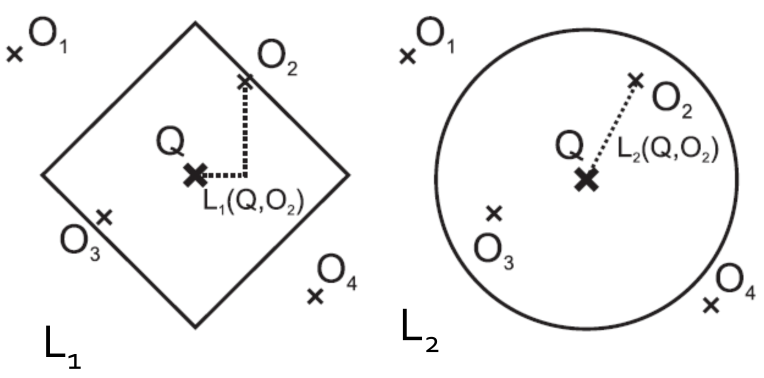
- similarity search = a content-based retrieval concept
- formalized model
  - **feature extraction** procedure $\mathbf{e: X \rightarrow U}$
    - transforming a multimedia object from database universe **X** into a descriptor of descriptor universe **U**
      - the original database $\mathbf{D \subset X}$,
      - the descriptor database $\mathbf{S \subset U}$
  - **distance (dissimilarity) function** $\boldsymbol{\delta}\mathbf{: U \times U \rightarrow R}$
    - i.e., close means similar
  - similarity queries: query-by-example paradigm
    - similarity query defined by its **type**, an **example** object **q** and an **extent** of the query
    - returns **ranked query result**, consisting of descriptors based on their closeness to **q**

Similarity queries

- similarity ordering
  - given a query object $\mathbf{q} \in \mathbf{U}$ and the descriptor universe $\mathbf{U}$,
  - SimOrder: $\mathbf{U} \rightarrow \mathbf{S}^{|S|}$,
    where $\forall \mathbf{i} \in (1,|\mathbf{S}|)$: $\delta(\mathbf{q}, \text{SimOrder}(\mathbf{q})[i]) \leq \delta(\mathbf{q}, \text{SimOrder}(\mathbf{q})[i+1])$
    - informally, SimOrder is the database **S** ordered desc. by distance of their elements to **q**
  - SimOrder is the basic concept when defining a similarity query
  - just an abstraction (i.e., SimOrder is not fully materialized when querying)

Similarity queries

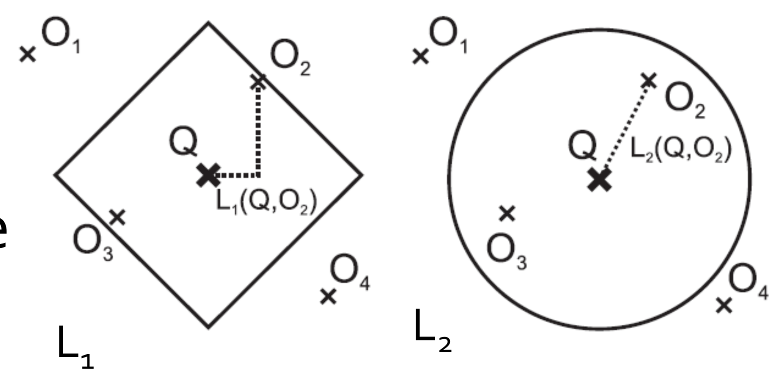
- range query
  - given a distance radius **r** (dissimilarity threshold), a range query returns all database descriptors the distances of which to **q** is no more than **r**
    - i.e., a prefix **P** ⊂ SimOrder(**q**), such that **x**∈**P**, δ(**q**, **x**) ≤ **r**
    - shortly, (**q**, **r**) = {**x**∈**S** | δ(**q**, x) ≤ **r**}
  - a "ball "in the distance space
    - just visualization, as geometry is meaningful only in vector spaces
    - delimits a region (subset) in **S**



$$L_p(v_1, v_2) = (\sum_{i=1}^{D} |v_1[i] - v_2[i]|^p)^{\frac{1}{p}} \qquad (p \geq 1)$$

Similarity queries

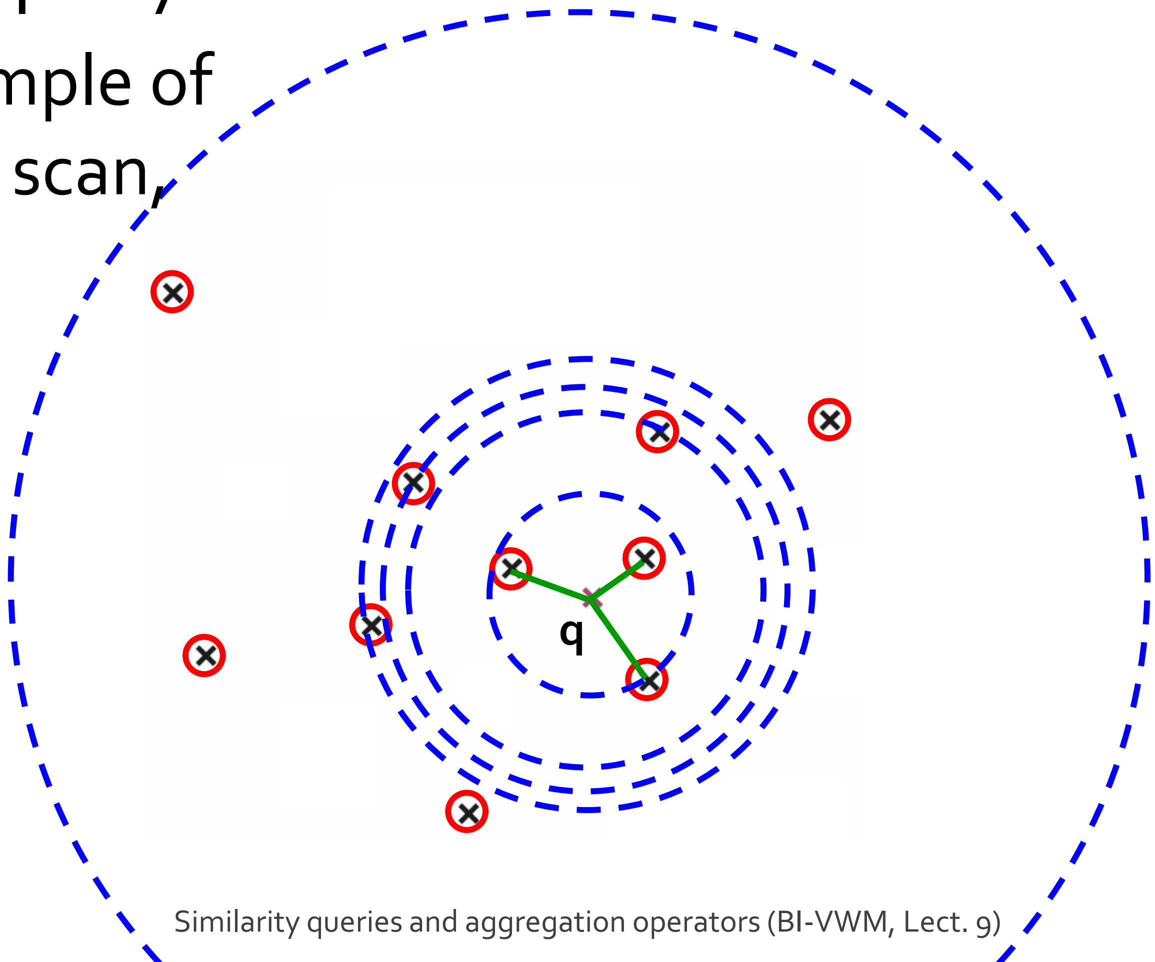
- k nearest neighbor (kNN) query
  - given a number of desired descriptors k,
    a kNN query returns k database descriptors closest to **q**
    - i.e., a prefix **P** $\subset$ SimOrder(**q**), such that **|P| = k**
    - shortly, (**q**, **k**) = {**C** | **C**$\subseteq$**S**, |**C**|=**k**, $\forall$**x**$\in$**C**, **y**$\in$**S**-**C**$\Rightarrow$ δ(**q**,**x**) ≤ δ(**q**,**y**) }
    - ties are solved arbitrarily
      (**k**th and (**k**+1)th NN could be
      the same close to **q**)
  - also "ball "in the distance space
    - but the query radius
      is not known in advance
    - delimits a region (subset) in **S**



$$L_p(v_1, v_2) = (\sum_{i=1}^{D} |v_1[i] - v_2[i]|^p)^{\frac{1}{p}} \qquad (p \geq 1)$$

Similarity queries

- kNN query
  - example of seq. scan, k=3



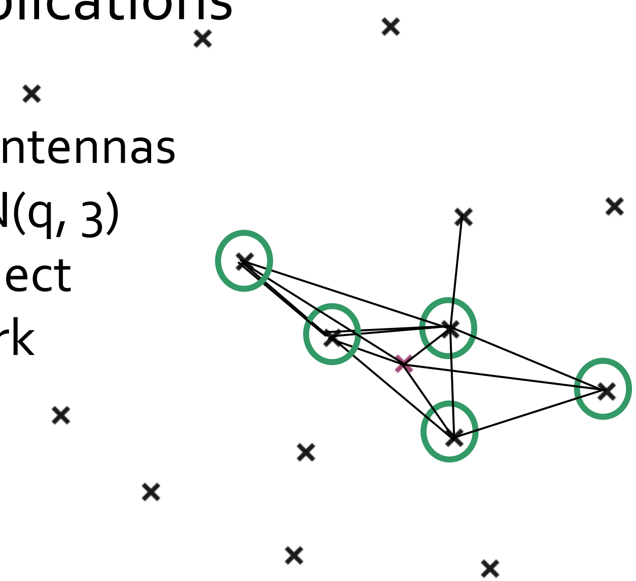
Similarity queries

- range vs. kNN queries
  - **range query** appropriate when
    - end-user is able to specify **r**, i.e., knows the semantics of the model
    - e.g., edit distance on strings,
      counting the smallest number of character edits to transform s_1 into s_2
      range query ('driver', 2) = {drivers, d_iver, _river, _rivers, drive_}
    - 100% recall is guaranteed (because of user's confidence on **r**)
  - **kNN query** appropriate when
    - user cannot specify **r**, i.e., does not know the semantics of the model
    - majority of cases

4NN, q=  ⇨  0.2  0.3  0.5  0.6  0.7

Similarity queries

- *k* reverse nearest neighbors (kRNN)
  - not very frequent, but interesting query type
  - for a query descriptor **q**, *k*RNN returns all descriptors x_i from the database for which **q** $\in$ kNN(x_i, *k*)
  - identifies the closest distinct neighborhood around **q**
- mostly used in spatial databases applications
  - e.g., in a GIS application,
    let the descriptors be positions of GSM antennas
    (**q** is a planned one), if the result of kRNN(q, 3)
    is large enough, **q** is needed to interconnect
    the other antennas into a reliable network
- could be used as similarity query
  - e.g., for identification of redundancy



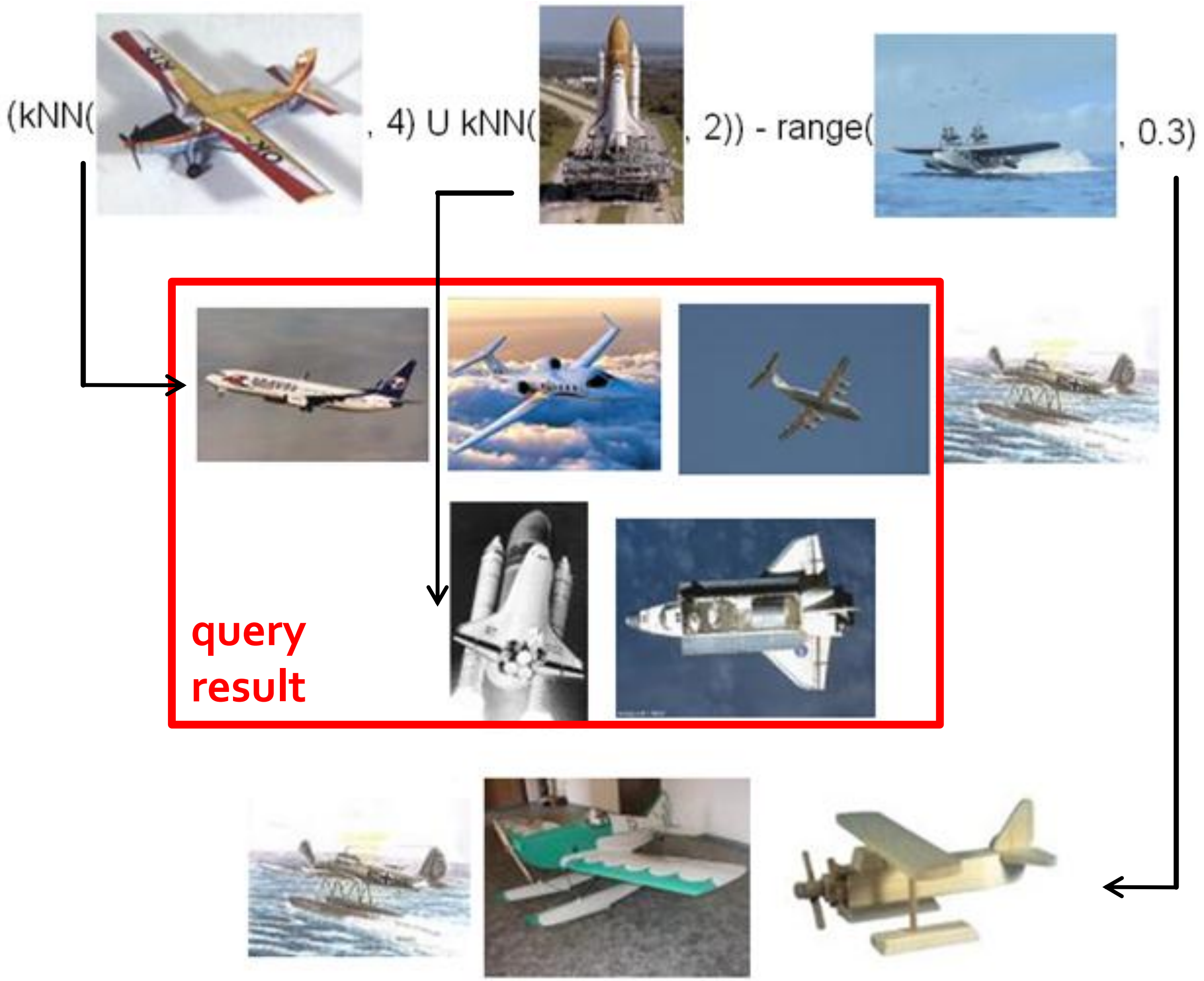
Query languages for similarity queries

- similarity queries could serve
  as a basis for higher-level query models

- query languages for similarity search
  - ad-hoc set-based expressions
  - extension of SQL

Query languages for similarity queries

- set operations with similarity queries



Query languages for similarity queries

- let a database of descriptors is stored in a BLOB-type attribute in a table of relational database
- new SQL predicates could enable relational databases to execute similarity queries
  - general SQL predicate
    - given an expression, the predicate condition is evaluated for all rows of a given table (or a join)
    - if the row passes the expression, the predicate is true
    - classic SQL predicate is, e.g., LIKE, ANY, IN, etc. in WHERE or HAVING
  - similarity predicates
    - range(example.MMattribute, table.MMattribute, r)
    - kNN(example.MMattribute, table.MMattribute, k)

Query languages for similarity queries

- examples

```
SELECT Id FROM BioData WHERE
    range(JohnSmith.Fingerprint, BioData.FingerPrint, 0.01)

SELECT Id FROM DNAData WHERE
    kNN(MickeyMouse.DNA, DNAData.DNA, 1)
```

Operators

- database operator = an operation on the database with result
- query vs. operator
  - query – the query expression/example is a parameter
    - small subset of the database is expected as a result
    - repeated queries of the same type on a static database makes sense (query expression is changing)
  - operator – mostly defined as non-parameterized operation
    - repeated processing of non-parameterized operator on a static database leads to the same result
    - only dynamic databases expected to run non-parameterized operators repeatedly
    - the answer is often large (more a database transformation than query)

Operators

- similarity join
  - joining descriptors of database A with descriptors of database B, based on a similarity-query predicate
    - range or kNN query predicate
  - if A=B, we talk about similarity self-join
  - joins can pair traditional structured elements (business data), but based on content-based similarity of some descriptors stored in attributes
  - self-joins are suitable for near-duplicates detection

Operators

- join based on range query
  - range(A's descriptor, B's descriptor, query radius)
  - example in SQL

    SELECT Criminal.Id, Citizen.Id FROM Criminal
    SIMILARITY JOIN Citizen ON range(Criminal.FingerPrint, Citizen.FingerPrint, 0.01)
- join based on kNN query
  - kNN(A's descriptor, B's descriptor, k)
  - example in SQL

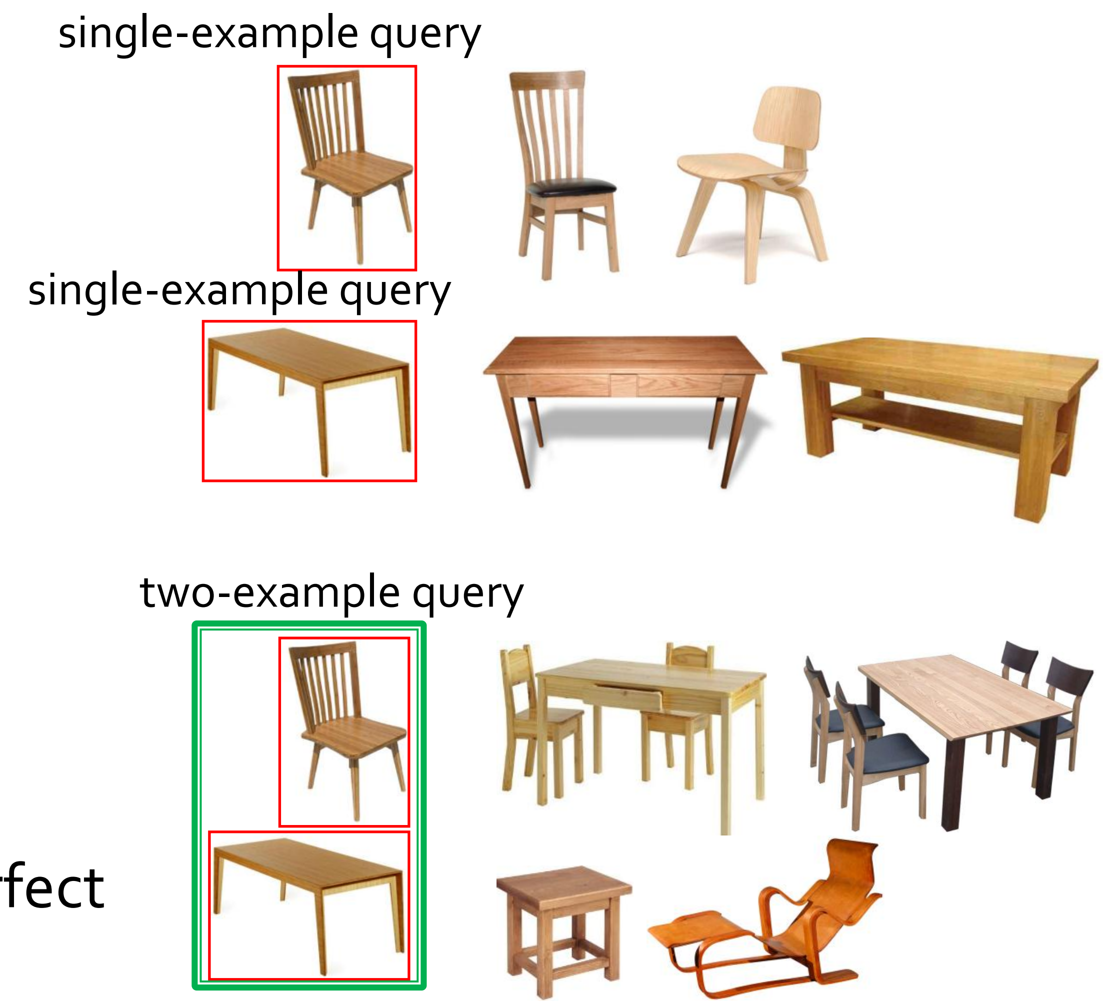
    SELECT Mammal.Id, Insect.Id FROM Mammal
    SIMILARITY JOIN Insect ON kNN(Mammal.DNA, Insect.DNA, 2)

Operators

- *k* closest pairs
  - based on the distance function δ, select the k pairs $\langle x, y\rangle \in A \times B$, that have the smallest distance $\delta(x, y)$
  - repeated usage makes sense for different combinations of A and B, dynamic or streamed databases, where the closest pairs have to be continuously updated

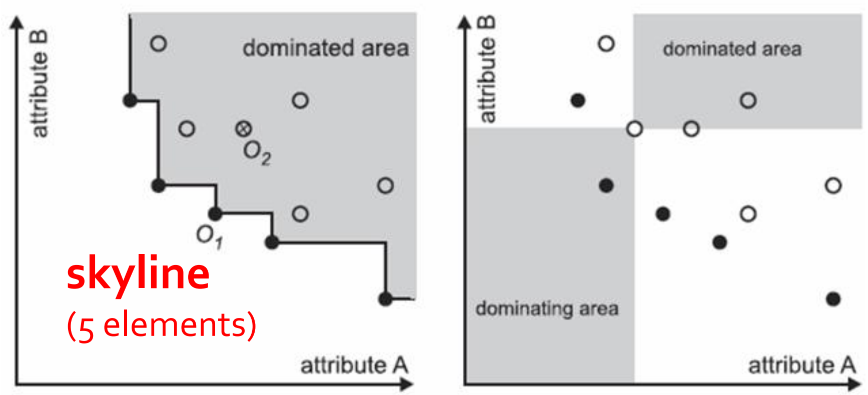
Skyline operator – motivation

- single-example queries often not sufficient
  - user is not able to find perfect example
- multi-example queries
  - aggregating queries/operators
  - multiple not-so-perfect examples



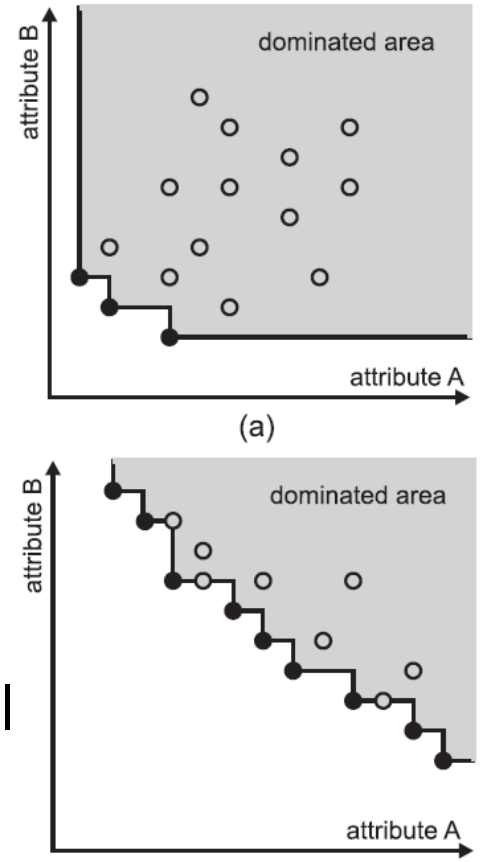
Skyline operator

- traditional skyline operator
  - database **S** modeled in several ordered domains (attributes)
  - skyline = subset of elements from **S** that are not dominated by other elements
    - element is not dominated if there does not exist another element that is better in all the attributes (let's better = lower value)
  - why the term "skyline"?
    - when connected by vertical and horizontal lines, the set looks like skyline of a city
  - application
    - e.g., market basket,
      consider database of hotels
      with attributes
      price and
      distance to airport



Skyline operator

- problems
  - skyline is not limited in size
    - correlated data lead to very small skyline (a)
    - anti-correlated data lead to very large skyline (b)
  - there is not ranking/ordering inside
  - i.e., problems similar to the Boolean model
- often too large skyline
  - manufacturers/distributors create additional unique attributes to put their products on the skyline
    - e.g., my hotel is the closest one to a winery ☺



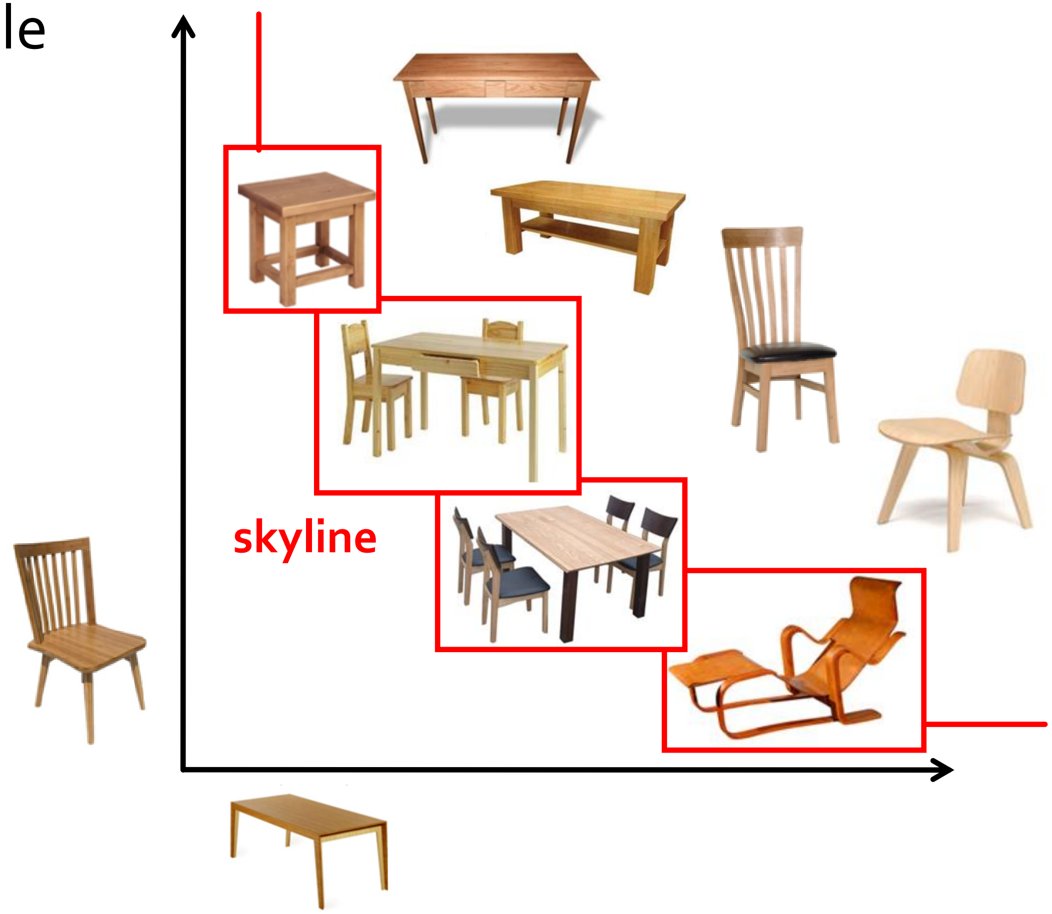
(a)
(b)

Skyline query in similarity search

- let the “attributes” be interpreted as similarity orderings with respect to multiple query examples
- the skyline operator then becomes a **multiple-example similarity query**
  - a particular skyline is a set of descriptors that are compromises with respect to the query examples
  - dynamic schema (attributes)
    - the coordinate system is established for each query separately
    - cannot be implemented by traditional skyline operator

Skyline query in similarity search

- example of two-example similarity skyline query



top-k operator

- motivation
  - consider a single database of multimedia objects (or web pages)
  - consider several (similarity) models that can be used to rank the database
    - similarity query, the similarity ordering, respectively
    - other rankings, e.g., PageRank
  - for example, database of web pages including images
    - ranking #1 = vector model of inf. retrieval (cosine sim. of web page text)
    - ranking #2 = similarity of images of the web pages (e.g., MPEG7 and L1)
    - ranking #3 = the PageRank of the web pages
  - thus, we need an aggregation procedure to create one final ranking based on the partial rankings – the **top-k operator**

top-k operator

- top-k operator
  - real-valued ordered attributes $A_1,\ldots,A_m$
    - in our case, let the attributes be **different rankings of the same objects**
    - for now let's better value = higher value (but could be defined inversely)
  - aggregation function $f: A_1 \times \ldots \times A_m \rightarrow R$
    - let f be monotonic, i.e., if $x > y$, then $f(\ldots, x, \ldots) > f(\ldots, y, \ldots)$
    - e.g., Min, Max, Avg
  - the top-k operator evaluates the aggregation function on all the objects' partial ranks and returns *k* objects with the highest aggregate ranks
    - could be done sequentially, but how to do that efficiently?
    - Fagin and Threshold algorithm (the rest of the slides)

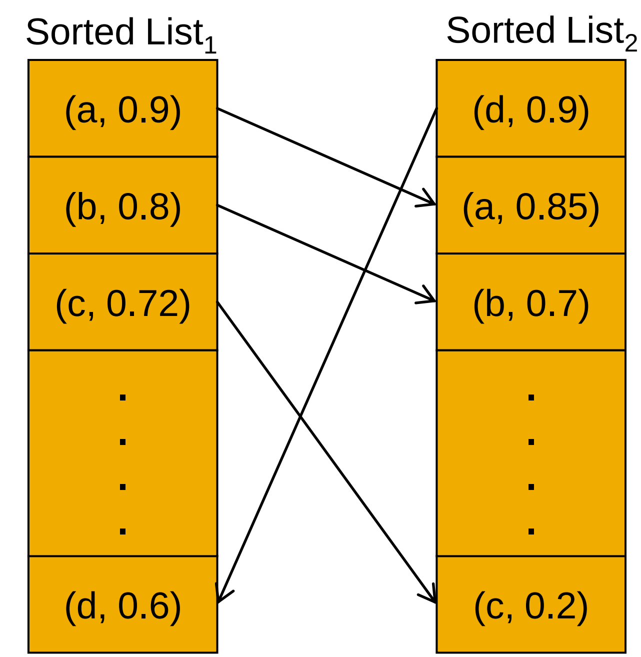
top-k operator

- example
  - consider a table of objects with two attributes (higher value is better), processed into two sorted lists with cross links, let f(a,b) = Min(a,b)

m

Obj. ID	Attribute 1	Attribute 2
a	0.9	0.85
b	0.8	0.7
c	0.72	0.2
d	0.6	0.9
. . . .	. . . .	. . . .

n



(examples taken from a presentation by Fagin et al)

top-k operator, Fagin algorithm

- Fagin algorithm
  1) read (and remember) entries (o_i, rank) from the sorted lists in parallel, until for some *k* objects all *m* ranks have been read (i.e., each rank from one list)
     - note that for some additional *l* objects there could be only some ranks (<*m*) read
       (i.e., in total, there were some attributes read for *l*+*k* objects)
  2) for the other objects that were read determine also their remaining ranks in the lists by random access (i.e., follow the cross links)
  3) compute the aggregations on the all objects already read (i.e., at least *k*)
  4) return the *k* objects with highest aggregated ranks

Fagin algorithm, example

STEP 1

loop(Read attributes from every sorted list)

stop when *k* objects have been seen in common from all lists (let's *k* = 2)

$List_1$	$List_2$
(a, 0.9)	(d, 0.9)
(b, 0.8)	(a, 0.85)
(c, 0.72)	(b, 0.7)
. . . .	. . . .
(d, 0.6)	(c, 0.2)

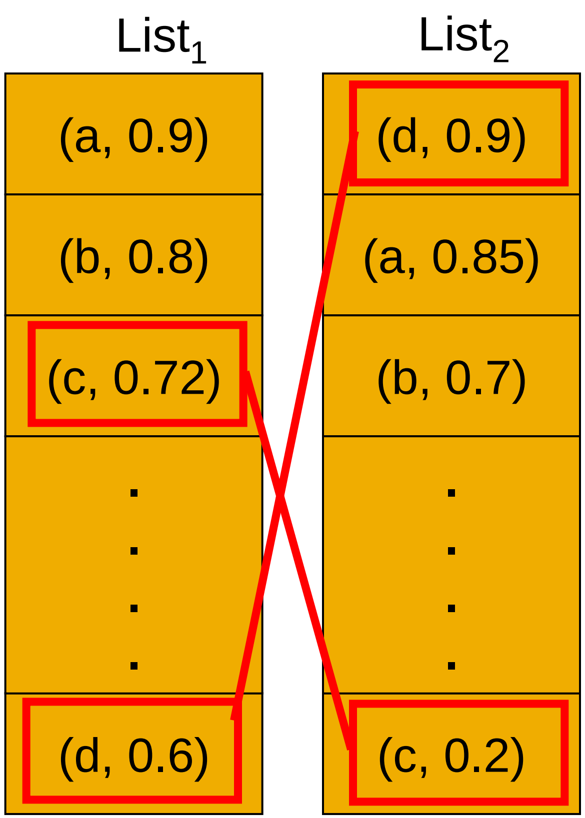
ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	
d		0.9	
b	0.8	0.7	
c	0.72		

(examples taken from a presentation by Fagin et al)

Fagin algorithm, example

STEP 2

Determine missing ranks by random access (follow the cross links)



ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	
d	0.6	0.9	
b	0.8	0.7	
c	0.72	0.2	

(examples taken from a presentation by Fagin et al)

Fagin algorithm, example

STEP 3

Compute the aggregate ranks of all objects already accessed
Return the *k* highest aggregate-ranked objects (in our example *k* = 2)

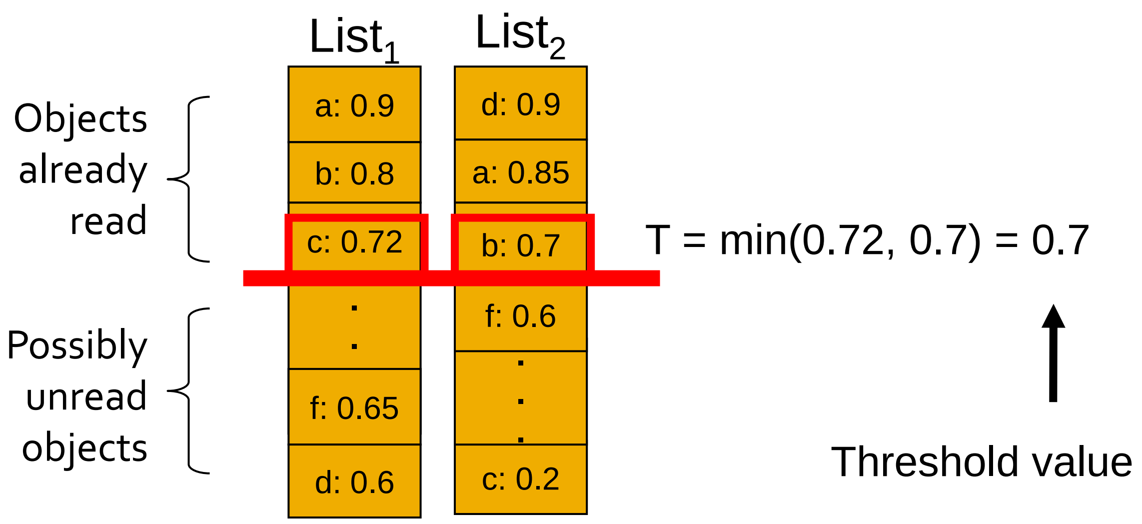
$List_1$	$List_2$
(a, 0.9)	(d, 0.9)
(b, 0.8)	(a, 0.85)
(c, 0.72)	(b, 0.7)
. . . .	. . . .
(d, 0.6)	(c, 0.2)

ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	$\boxed{0.85}$
d	0.6	0.9	0.6
b	0.8	0.7	$\boxed{0.7}$
c	0.72	0.2	0.2

(examples taken from a presentation by Fagin et al)

top-k operator, Threshold algorithm

- motivation
  - Fagin's algorithm is not optimal with respect to the parallel scan – there is not needed to see all *m* ranks for some *k* objects
  - instead, do parallel search + immediately random accesses, after you obtain top-k object candidates
  - predict the maximal possible aggregate rank of unread objects
  - repeat the parallel + random search until the threshold gets lower than the last of the top-k objects



Threshold algorithm - example

STEP 1:

1) Read k objects in parallel, such that all their ranks are determined immediately (either by parallel search or random access)
2) Determine the aggregate ranks for the read objects
3) Keep in buffer the top-k candidate objects (and their ranks)

$List_1$	$List_2$
$\boxed{(a, 0.9)}$	$\boxed{(d, 0.9)}$
(b, 0.8)	$\boxed{(a, 0.85)}$
(c, 0.72)	(b, 0.7)
. . . .	. . . .
$\boxed{(d, 0.6)}$	(c, 0.2)

ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	0.85
d	0.6	0.9	0.6

(examples taken from a presentation by Fagin et al)

Threshold algorithm - example

STEP 2:

1) Determine the maximal possible aggregate rank of unread objects (threshold T)
2) if there are *k* objects in the buffer that have aggregate rank greater or equal to the threshold, stop the algorithm, otherwise goto STEP 1

$List_1$	$List_2$
(a, 0.9)	(d, 0.9)
(b, 0.8)	(a, 0.85)
(c, 0.72)	(b, 0.7)
. . . .	. . . .
(d, 0.6)	(c, 0.2)

$T = \min(0.9, 0.9) = 0.9$

ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	0.85
d	0.6	0.9	0.6

(examples taken from a presentation by Fagin et al)

Threshold algorithm - example

STEP 1:

1) Read k objects in parallel, such that all their ranks are determined immediately (either by parallel search or random access)
2) Determine the aggregate ranks for the read objects
3) Keep in buffer the top-k candidate objects (and their ranks)

$List_1$	$List_2$
(a, 0.9)	(d, 0.9)
(b, 0.8)	(a, 0.85)
(c, 0.72)	(b, 0.7)
. . . .	. . . .
(d, 0.6)	(c, 0.2)

ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	0.85
d	0.6	0.9	0.6
b	0.8	0.7	0.7

(examples taken from a presentation by Fagin et al)

Threshold algorithm - example

STEP 2:

1) Determine the maximal possible aggregate rank of unread objects (threshold T)
2) if there are *k* objects in the buffer that have aggregate rank greater or equal to the threshold, stop the algorithm, otherwise goto STEP 1

L_1	L_2
(a, 0.9)	(d, 0.9)
(b, 0.8)	(a, 0.85)
(c, 0.72)	(b, 0.7)
. . . .	. . . .
(d, 0.6)	(c, 0.2)

$T = \min(0.8, 0.85) = 0.8$

ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	0.85
b	0.8	0.7	0.7

(examples taken from a presentation by Fagin et al)

Threshold algorithm - example

The algorithm stops, because the threshold T is below or equal to the aggregate rank of all the candidate top-k objects (these become the final top-k objects).

$List_1$	$List_2$
(a, 0.9)	(d, 0.9)
(b, 0.8)	(a, 0.85)
(c, 0.72)	(b, 0.7)
⋮	⋮
(d, 0.6)	(c, 0.2)

$T = \min(0.72, 0.7) = 0.7$

ID	A_1	A_2	$Min(A_1,A_2)$
a	0.9	0.85	0.85
b	0.8	0.7	0.7

(examples taken from a presentation by Fagin et al)

Fagin vs. Threshold algorithm

- correctness of the algorithms
  - mainly due to the monotonicity of the aggregation function
- Threshold algorithm (TA) reads less objects than Fagin algorithm (FA)
  - TA stops at least as early as FA
- TA may perform more random accesses than FA
- TA requires only bounded buffer space (*k*)
  - at the expense of random accesses
  - FA makes use of unbounded buffer