Skyline Queries

by: Victor Mier

30.11.2010

Seminar Location-based Services (19562)

Prof: Dr. Agnès Voisard

Index (1)

Index (2)

1. Introduction (1)

What are Skyline Queries and why do we need them?

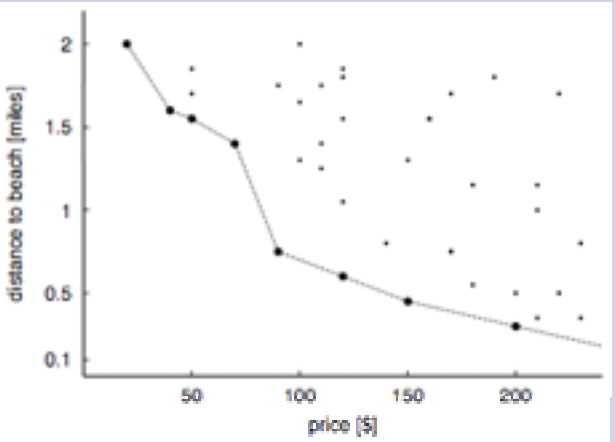
- In a database, a **Skyline** is a set of tuples of information (points) which stand out among the others because are of special interest to us.

More formally:

- A **Skyline** is defined as those points which are not dominated by any other point.

- A point dominates another point if it is as good or better in all dimensions and better in at least one dimension

Example1: Find cheap hotels near to the beach.

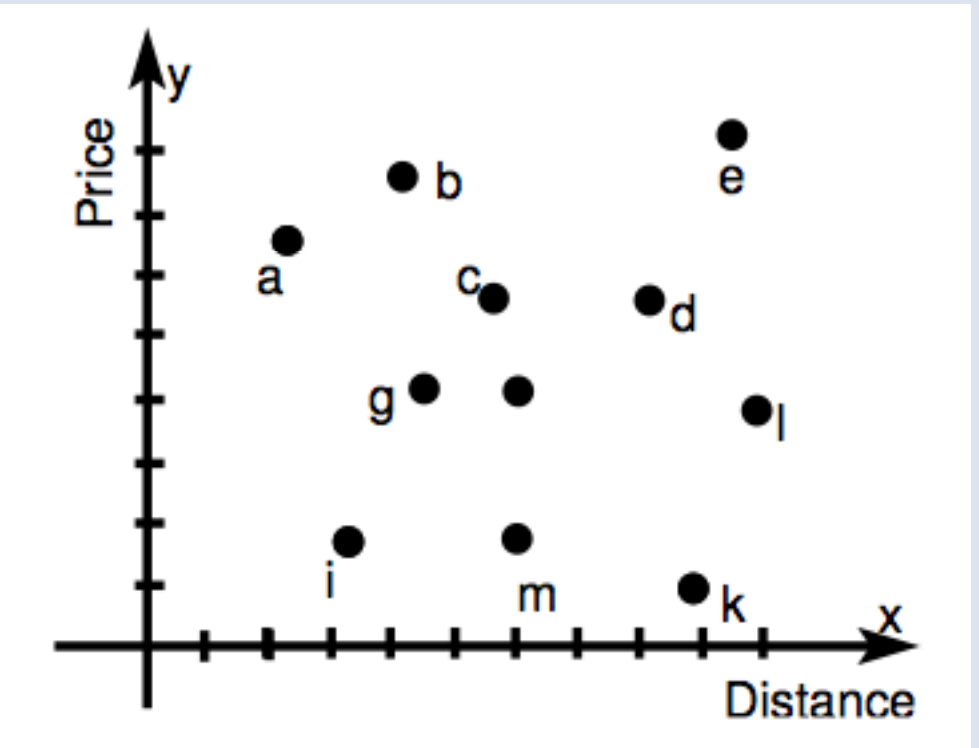


Hotel [price $50 and dist. 1.5] dominates Hotel [price $50 and dist 1.7]

Example2: Skyline of Manhattan is the set of buildings which are high and close to the Hudson river and same x position.

1. Introduction (2)

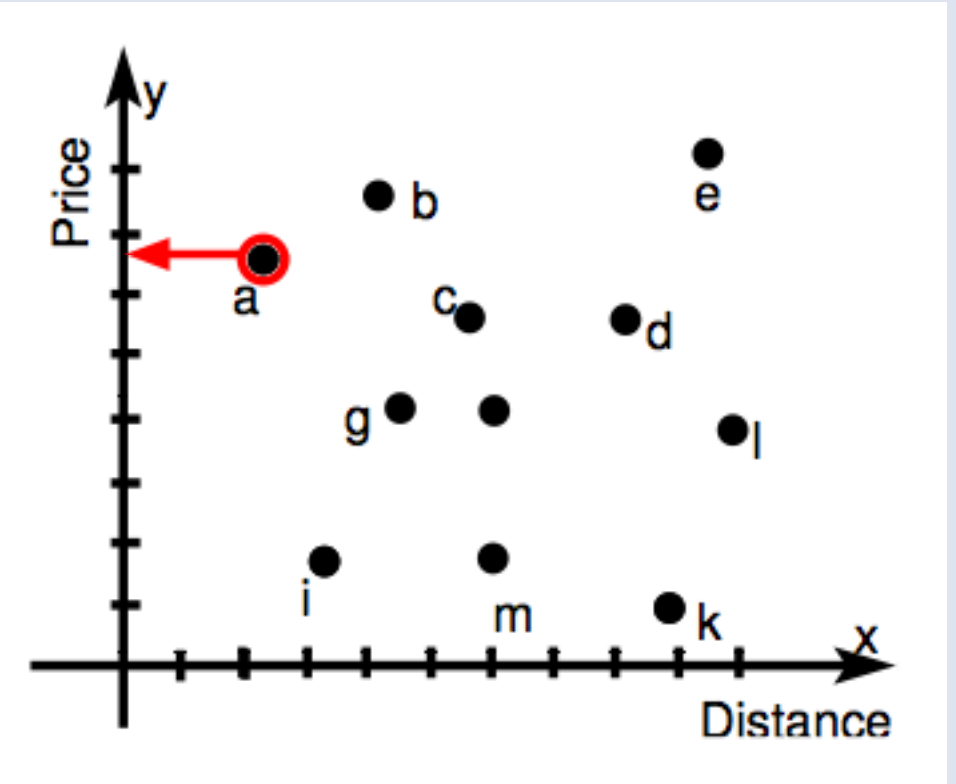
- Example: Hotels near to the beach with a low price.



- The interesting points in the dataset are {a, i, k}.

1. Introduction (3)

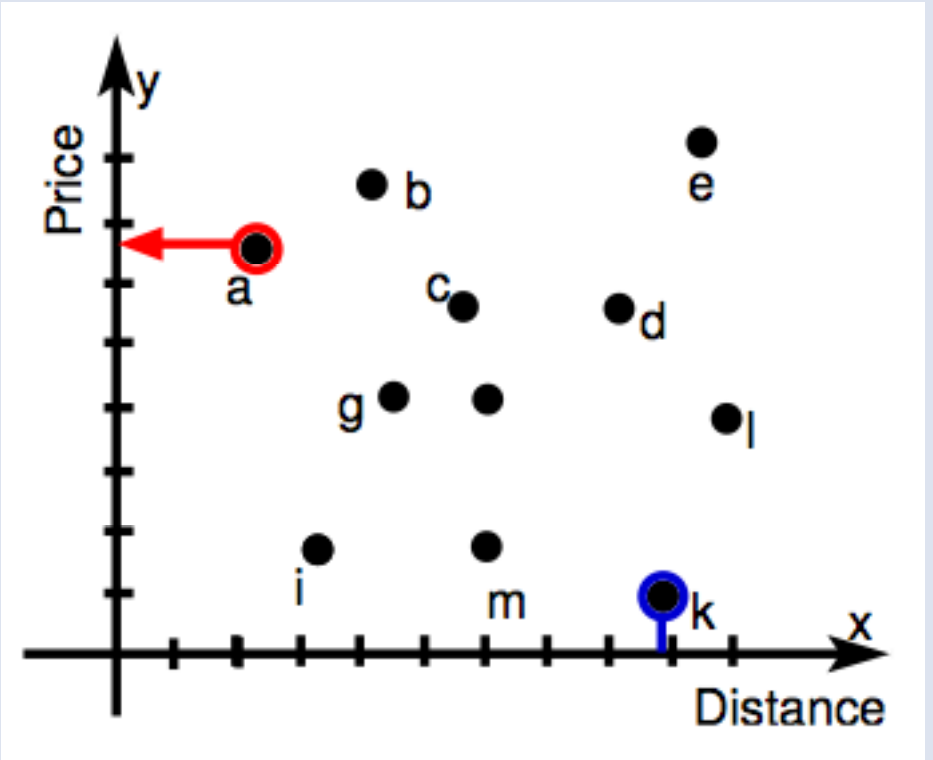
- Example: Hotels near to the beach with a low price.



- The interesting points in the dataset are {a, i, k}. a has the least distance to the beach.

1. Introduction (4)

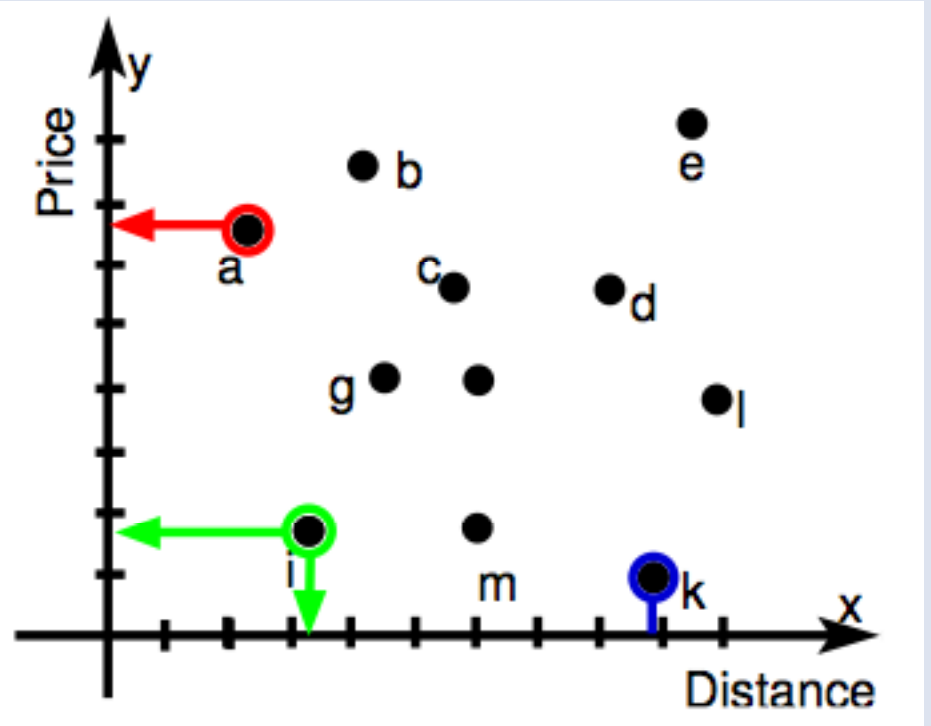
- Example: Hotels near to the beach with a low price.



- The interesting points in the dataset are {a, i, k}. a has the least distance to the beach, k has the lowest price,

1. Introduction (5)

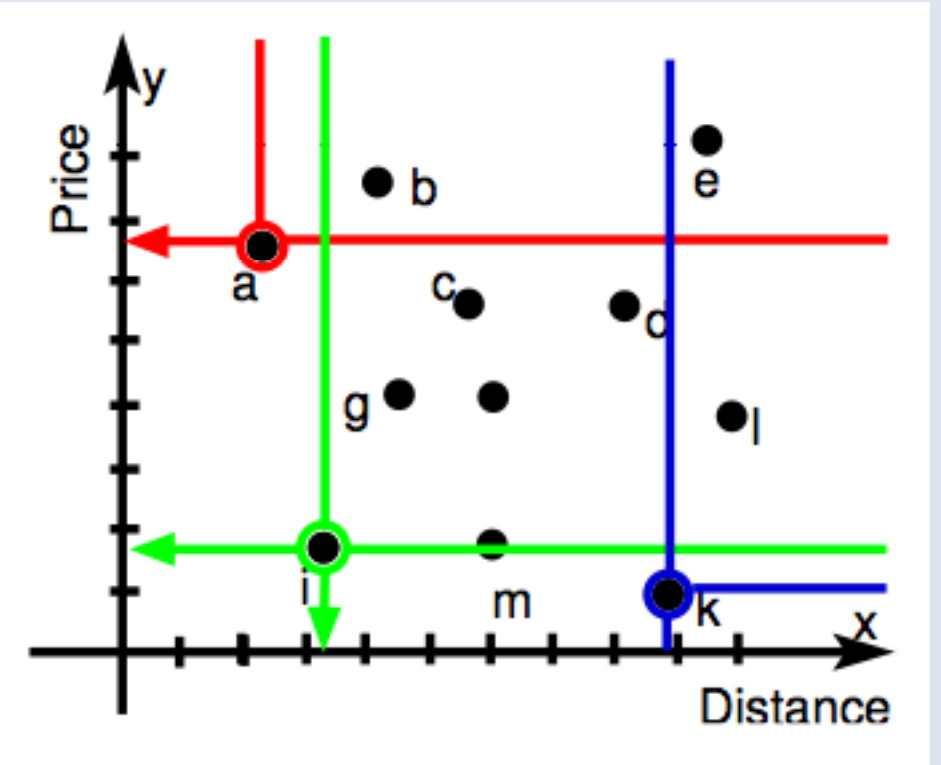
- Example: Hotels near to the beach with a low price.



- The interesting points in the dataset are {a, i, k}. a has the least distance to the beach, k has the lowest price, i has neither the shortest distance nor the minimum price. i has a less distance value than k and a lower price value than a. i is hence not dominated by either one of them.

1. Introduction (6)

- Example: Hotels near to the beach with a low price.



- All other data-points in the dataset are dominated by the set of points {a, i, k}

2. Specification Options(1)

- A one-dimensional Skyline is trivial because it is equivalent to computing min, max or distinct.

There are 2 Options:

1 - Build it on top of relational database system: using existing SQL Queries (poor performance, explained on section 3.1)

2 - Extend SQL with a new "Skyline Operator": SKYLINE OF (which selects all the tuples that are not dominated by any other tuple)

SELECT ... FROM ... WHERE ...

GROUPBY ... HAVING...

SKYLINE OF [DISTINCT] d1 [MIN| MAX| DIFF], ..., dm [MIN| MAX|DIFF]

ORDER BY ...

- d1 ... dm denote de dimensions of the skyline, e.g.: price, distance to the beach, etc.

- MIN, MAX and DIFF specify if the attribute should be MINimized, MAXimized, or be DIFFerent.

- DISTINCT specifies how to deal with duplicates (all the properties in MIN|MAX|DIFF

2. Specification Options(2)

- Extending the definition, a

tuple p=(p1,...,pk,pk+1,...,pl,pl+1, ...,pm,pm+1,...,pn)

dominates

tuple q =(q1,...,qk,qk+1,...,ql,ql+1,...,qm,qm+1,...,qn)

for a Skyline query

SKYLINE OF d1 MIN, ..., dk MIN, dk + 1 MAX, ..., dl MAX, dl + 1 DIFF, ..., dm DIFF

if the following three conditions hold:

$$p_i \leq q_i \text{ for all } i=1,...,k$$

$$p_i \geq q_i \text{ forall } i=(k+1),...,l$$

$$p_i = q_i \text{ forall } i=(l+1),...,m$$

2. Specification Options (3)

- Examples:

```
SELECT *
FROM Hotels
WHERE city = 'Nassau'
SKYLINE OF price MIN, distance MIN;
```

Query 1

```
SELECT *
FROM Buildings
WHERE city = 'New York'
SKYLINE OF distance MIN, height MAX,
          x DIFF;
```

Query 2

```
SELECT e.name, e.salary, sum(s.volume) as volume
FROM Emp e, Sales s
WHERE e.id = s.repr AND s.year = 1999
GROUP BY e.name, e.salary
SKYLINE OF e.salary MIN, volume MAX;
```

Query 3

```
SELECT name, distance,
       (CASE WHEN price ≤ 50 THEN 'cheap'
             WHEN price > 50 THEN 'exp') AS pcat
FROM Hotels
WHERE city = 'Nassau'
SKYLINE OF pcat MIN, distance MIN;
```

Query 4

Query1: Cheap hotels near to the beach

Query2: High buildings close to the river (NY Skyline)

Query3: Employees that sell a lot and have a low salary

Query4: Cheap hotels near to the beach (only 2: one for category [cheap|expensive])

3. Implementation of the Skyline Operator (1)

3.1 Nested SQL Queries

- Build on top of a relational database system.

- It is done by translating the Skyline query into a nested SQL query.

```
SELECT *
FROM   Hotels h
WHERE  h.city = 'Nassau' AND NOT EXISTS(
       SELECT *
       FROM   Hotels h1
       WHERE  h1.city = 'Nassau' AND h1.distance <= h.distance AND
              h1.price <= h.price AND
              (h1.distance < h.distance OR h1.price < h.price));
```

Cons: very poor performance

- Query cannot be unnested

- If the query involves a join or group-by, this would have to be executed as part of the outer and inner query.

- We will see later that the Skyline query can be combined with other operations (join/ top-N) with little additional cost to compute the skyline

3. Implementation of the Skyline Operator (2)

3.2 Two-dimensional Skyline Operator

- A two-dimensional Skyline can be computed by sorting the data.
- The data has to be sorted according to one of the two attributes.
    1. We only need to compare a tuple with its predecessor.
    2. One of the tuples is dominated by the other, we eliminate it. Therefore, we only compare a tuple with the last tuple which is part of the Skyline.

$$\begin{array}{llll} \langle h_1, & \$50, & 3.0 \text{ miles} \rangle \\ \langle h_2, & \$51, & 5.0 \text{ miles} \rangle \\ \langle h_3, & \$52, & 4.0 \text{ miles} \rangle \\ \langle h_4, & \$53, & 2.0 \text{ miles} \rangle \end{array}$$

- Hotel2 (h2) is dominated by Hotel1 (h1) so we eliminate it.
- Hotel3 (h3) is dominated by its predecesor (Hotel1), so we eliminate it.

3. Implementation of the Skyline Operator (3)

3.3 Block-nested-loops Algorithm

3.3.1 Basic Algorithm

- This is an algorithm that is significantly faster than the nested-loops naive way of computing the Skyline (3.1): we produce a block of Skyline tuples in every iteration.

- We keep a **window** of incomparable tuples in main memory.

- When a tuple p is read from the input, p is compared to all tuples of the window.

- Based on this comparison, p is either eliminated, placed into the window or into a temporary file (which will be considered in the next iteration). Three cases can occur:

1. p is dominated by a tuple within the window --> p is eliminated
2. p dominates one or more tuples in the window --> these tuples are eliminated and
   p is placed in the window
3. p is incomparable with all the tuples in the window --> p is inserted in the window
   if there is not space, p is written into atemporary file

- At the end of each iteration, we can output the tuples of the window which have been compared to all the tuples in the temporary file (it's part of the Skyline).

- We assign a timestamp to each tuple in the window and temporary file. When we read a tuple from the temporary file with timestamp t, we can output all the tuples in the window with timestamp < t.

3. Implementation of the Skyline Operator (4)

3.3.2 Variants

1. Maintaining the window as a self-organizing list:
   - A lot of time is spent comparing a tuple with the tuples in the window
   - When a tuple of the window dominates another tuple, it is moved to the top of the window, so it's compared first with new tuples.
   - It works great if we have a "killer" tuple that dominates a lot of other ones.

2. Replacing tuples in the window
   - We try to keep the **most dominant set** of tuples in the window.
   - Therefore, dominated tuples are rapidly eliminated.

3. Implementation of the Skyline Operator (5)

3.4 Divide and Conquer

- This is theoretically the best known algorithm for the worst case, so it will outperform the block-nested-loops algorithm in bad cases (and worse in good cases).

- Basic algorithm:

1. Compute the median of the input for some dimension ***d1*** (e.g.: price). Divide the input into 2 partitions (***P1***(tuples with price < median(***d1***)) and ***P2*** (the rest))

2. Compute the Skylines ***S1*** and ***S2*** of ***P1*** and ***P2***. This is done by recursively applying the whole algorithm to ***P1*** and ***P2***. The recursive partitioning stops if a partition contains few tuples (then, computing the Skyline is trivial).

3. Merge ***S1*** and ***S2*** to compute the overall Skyline. This means eliminating all the tuples of ***S2*** which are dominated by tuples of ***S1***. (No tuples of ***S1*** can be dominated by tuples of ***S2*** as tuples in ***S1*** have a better ***d1*** value).

3. Implementation of the Skyline Operator (6)

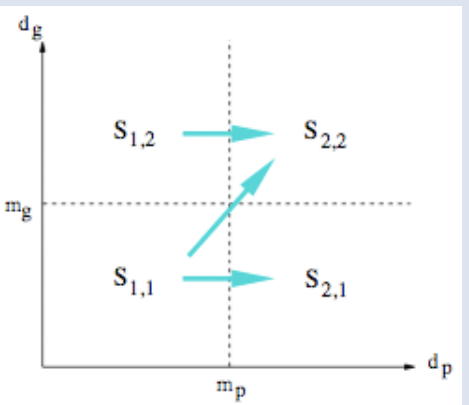
For merging ***S1*** and ***S2*** we:

1. We choose another unconsidered dimension ***d2***(!= ***d1***) and divide both ***S1*** and ***S2*** into ***S1.1, S1.2, S2.1*** and ***S2.2***, using the median of dimension ***d2***.

S1.i is better than ***S2.i*** in dimension ***d1***, and ***Si.1*** is better than ***Si.2*** in dimension ***d2***.

2. Now, we need to merge S1.1 and S2.1, S1.2 and S2.2, S1.1 and S2.2. The beauty is that S2.1 and S1.2 don't need to be compared as it are guaranteed to be incomparable.

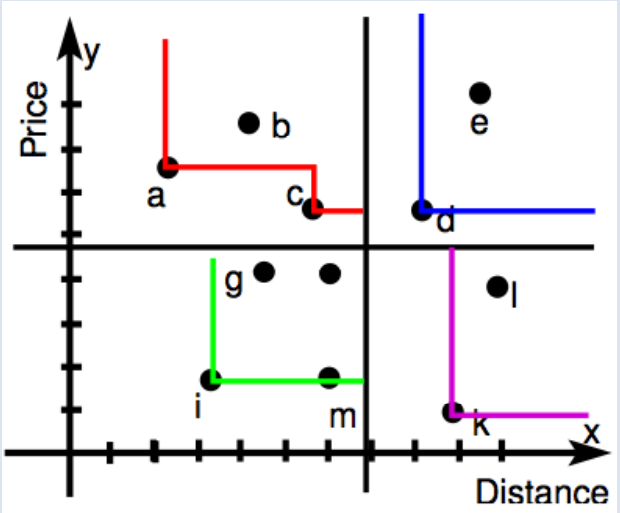
3. The merging of Si.i and Sj.j is done by recursively applying the same merging algorithm. The recursion ends when all the dimenions have been considered or if one of the partitions is empty (in both cases the merge function is trivial).



3. Implementation of the Skyline Operator (7)

3.4.1 Extension: M-way partitioning

- If the input does not fit in main memory, it shows terrible performance (input is read, partitioned, written to disk, reread to be partitioned again and so on until a partition fits into main memory).



- *Solution*: Partitioning the initial set of tuples into m partitions so each partition is small enough to fit in memory. Instead of a median, we calculate α-quantiles.

- This can be applied to the first and third (merging) steps of the algorithm. In the case of merging, the m-partitioning is applied so all the sub-partitions can be merged in memory.

3. Implementation of the Skyline Operator (8)

3.4.2 Extension: Early Skyline

- This is an extension for situations in which the available main memory is limited.

In the first step (m-partition the data) we do this:

1. Load as much tuples as fit into the available memory.

2. Apply the basic divide-and-conquer algorithm to this block of tuples in order to inmediately eliminate tuples which are dominated by others.

3. Partition the remaining tuples into m partitions.

- It saves I/O because less tuples need to be written and reread in the partitioning steps. It is attractive if the Skyline is *selective* (the result of the whole Skyline operation is small)

3. Implementation of the Skyline Operator (9)

3.4 Using B-Trees

- We can use ordered indexes to compute Skylines withous scanning through the whole index (example):

- We have 2 ordered indexes: hotel.price, hotel.distance. We use this indexes to find a superset of the Skyline.

- We scan simultaneously through both indices and stop as soon as we have found the first match. (In the figure, h2 is the first match). We conclude the following:

* h2 is defenetely part of the Skyline. Any hotel which has not been yet inspected is not part of the Skyline because it is dominated by h2. All the other hotels may be or not part of the Skyline, so we apply a Skyline algorithm to compute it. The algorithm is very attractive if the Skyline is very small.

hotel	*price*
h_1	\$25
h_3	\$27
h_{25}	\$30
h_2	\$35
h_{35}	\$40
h_{17}	\$70
...	

hotel	*distance*
h_{17}	0.1 miles
h_2	0.2 miles
h_{35}	0.3 miles
h_{25}	0.3 miles
h_1	0.7 miles
h_3	1.0 miles
...	

3. Implementation of the Skyline Operator (10)

3.4 Using R-Trees

- We can use R-Trees to compute a Skyline.

- We need an R-Tree that considers all the Skyline dimensions.

- Given a point p, we need not search in any branches of the R-tree which are guaranteed to contain only points that are dominated by p.

Example:

- In the cheap and close to the beach hotel example, we don't need to search in branches with hotels in the price-range $40-$60 and distance range (2.0 - 3.5 miles) if we already have a hotel h with price $30 and distance 1.0 miles, because h dominates all of them.

4. Skyline and Joins (1)

4.1 Pushing the Skyline Operator *Through* a Join

- As seen in section 2, the Skyline Operator is applied after Joins (WHERE) and Group-Bys.

- If there's a ***Non-Reductive Join*** (>= number of tuples as the result of the join) it's cheaper to compute the skyline before the join.

Example:

- We want to know young employees with the highest salary:

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno = d.dno
SKYLINE OF e.age MIN, e.salary MAX;
```

- As every employee works in a department, the result of a Join would have sizeof(Employees) tuples.

- It would be cheaper to compute first the Skyline for all the employees and then compute the join.

4. Skyline and Joins (2)

4.2 Pushing the Skyline Operator *Into* a Join

- We consider the following query which asks for Emps with high salary that work in a Dept with low budget:

```
SELECT *
FROM Emp e, Dept d
WHERE e.dno = d.dno
SKYLINE OF d.budget MIN, e.salary MAX;
```

- Now assume that we have three Emps that work in Dept N: A (200K salary), B (400K) and C (100K). Without knowing the budget of Dept N, we can immediately eliminate A and C.

1. Sort the Emp table by dno, salary DESC, thereby eliminating Emps that have a lower salary than another Emp with the same dno.
2. Sort the Dept table by dno.
3. Apply a merge join.
4. Compute the Skyline using any approach of Section 3.

- What we've done here is applying an ***Early Skyline***.

5. Skyline and Top-N

- Usually, a Top-N operation (selecting first N tuples for a given attribute/s) would be computed after the Skyline. However, ***if the Top-N operation is based on the same or is a subset of the columns used in the Skyline***, Skyline and Top-N operations can be combined:

Example: compute the N cheapest hotels which are interesting in terms of price, distance:

1. Sort all hotels by price (i.e., by the criterion of the Top N operation).
2. Scan through the sorted list of hotels. For each hotel, test whether it is interesting by comparing that hotel to all hotels that have the same or lower price.
3. Stop as soon as N hotels have been produced.

- We can see that each Hotel needs only to be compared with those with a lower price, and we will stop when we reach N hotels.

- If no index exists, to save computing time and memory, we can just apply the operations to the hotels with the lowest range of price. If we get N hotels, we're done. Otherways, we take a set of hotels with the following ascending range of price.

6. Performance experience and results (1)

- For experimenting, we consider 3 kinds of databases:

  - indep: The dimensions (attributes) are independent to each other.

  - corr: The dimensions are correlative (points which are good in one dimension, are probably good in the other dimensions).

  - anti: Anti-correlated dimensions (good in one dimension, bad in another).

6.1 2-d Skylines

	corr		indep		anti	
	Time	I/O	Time	I/O	Time	I/O
BNL-basic	1.1	10	1.1	10	1.3	10
D&C-basic	88.0	90	87.0	90	89.0	90
D&C-mpt	25.0	30	27.0	30	26.0	30
D&C-mptesk	2.6	10	2.7	10	3.1	10
Sort	2.0	10	1.9	10	2.0	10
RDBMS	28.0	–	37.0	–	92.0	–

6. Performance experience and results (2)

6.2 Multidimensional Skylines

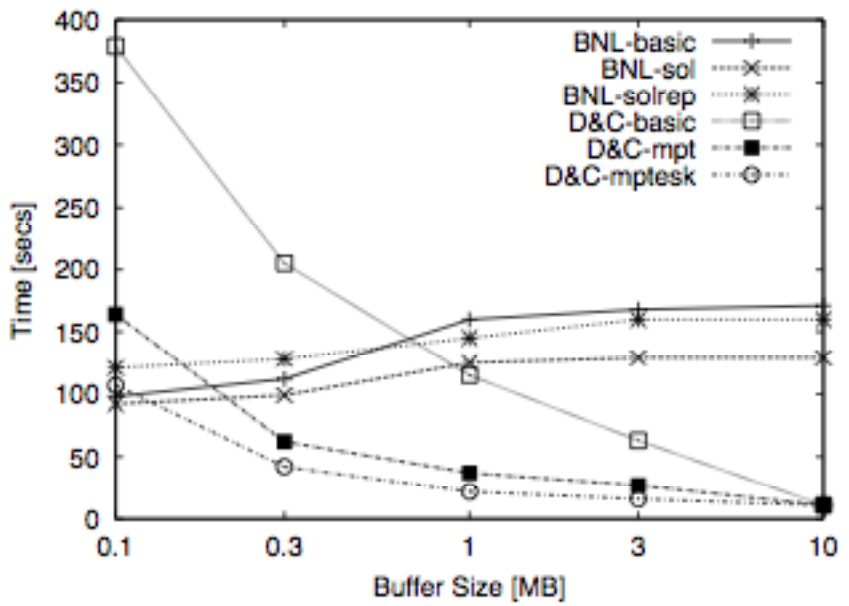


Figure 18: Time (secs), Anti, $d = 5$

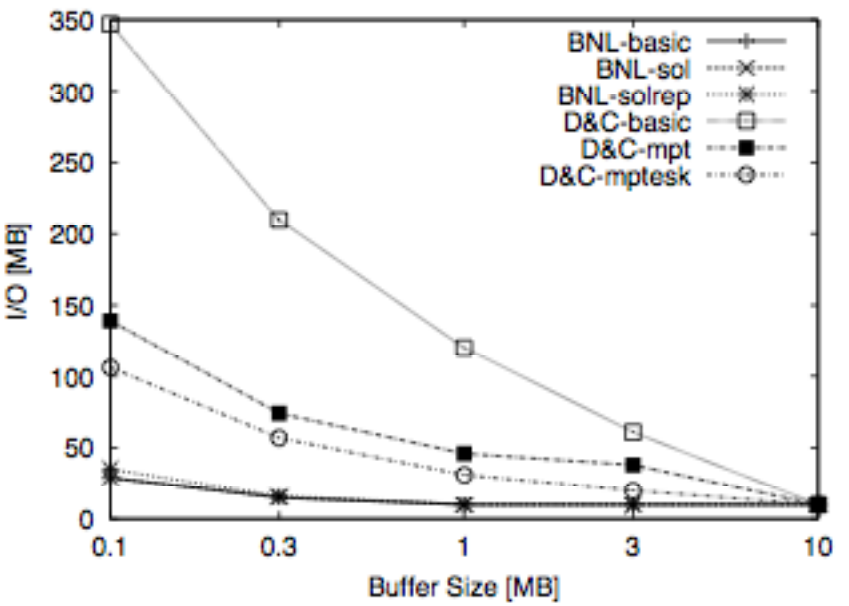


Figure 19: I/O (MB), Anti, $d = 5$

- BNL variants are good if the size of the Skyline is small. BNL-sol is the best of the 3 variants.

- D&C variants less sensitive than BNL to number of dimensions and correlation in the database. The winner of the variants is D&Cmptearlyskylines.

- A system should implement the D&C-mptesk and BNL-sol algorithms

6. Performance experience and results (3)

6.3 Varying the size of the buffer

- Incrementing the size of the main-memory buffer --> performance of the D&C algorithm improves

- There's not much difference with BNL algorithms

6.4 Varying the size of the database

	10 MB Database	100 MB Database
BNL-basic	160	813
BNL-sol	125	605
BNL-solrep	145	924
D&C-basic	115	1768
D&C-mpt	37	388
D&C-mptesk	22	211

Table 2: Running Time (secs); 1 MB Buffer, $d = 5$, *Anti-correlated*, Vary Size of Database

- Results don't change much with a larger database; D&C still outperforms BNL.

- We can see that the BNL algorithms get more and more attractive as the database gets bigger.

7. Conclusion

- In order to compute a Skyline within a database environment, the best choice would be extending the SQL's SELECT statement with a SKYLINE OF operator.

- The right approach for a relational database system, would be implementing a block-nested loops algorithm with a window that is organized as a self-organizing list and a divide-and-conquer algorithm that carries out m-way partitioning and "Early Skyline" computation.

8. References

- *The Skyline Operator*, Stephan Börzönyi, Donald Kossmann, Konrad Stocker

- *Skyline Queries and it variations*, Jagan Sankaranarayanan, [CMSC828S]

- *An Optimal and Progressive Algorithm for Skyline Queries*, D.Papadias, Y.Tao, G.Fu, B. Seeger, [SIGMOD 2003]

THE END

?

Any questions?

Thanks for your attention