Supporting Top-k Join Queries in Relational Databases

Ihab F. Ilyas

Joint work with:

Walid G. Aref and Ahmed K. Elmagarmid

Purdue University, West Lafayette

Motivation

- Many applications have requirements that can only be matched by a *combination* of IR/DB
- IR on text and multimedia require techniques involving uncertainty and *ranking* for *effective retrieval*
- IR needs the DB advanced handling of data
- DB does not handle *effective* retrieval (*Boolean logic*)
- Supporting new data types is certainly **not enough**
- True integration requires significant changes in the standard database techniques for **indexing** and **query optimization** and may require new **query languages**
- We focus on supporting **Ranking** in DB query processors

Why Ranking Queries?

- **Applications**
  - Multimedia search by contents (multi-features, multiple examples)
  - Middleware
  - Information retrieval (search engines)
  - Data mining
- **New requirements**
  - Multi-criteria ranking
  - Rank aggregation from external sources
  - Joining ranked "infinite" streams
- **Applications are interested in the top-k results**

Outline

- **Query Model**
- **Related Work**
- **The New Rank-join Algorithm and Query Operators**
- **Performance Results**
- **Conclusion**

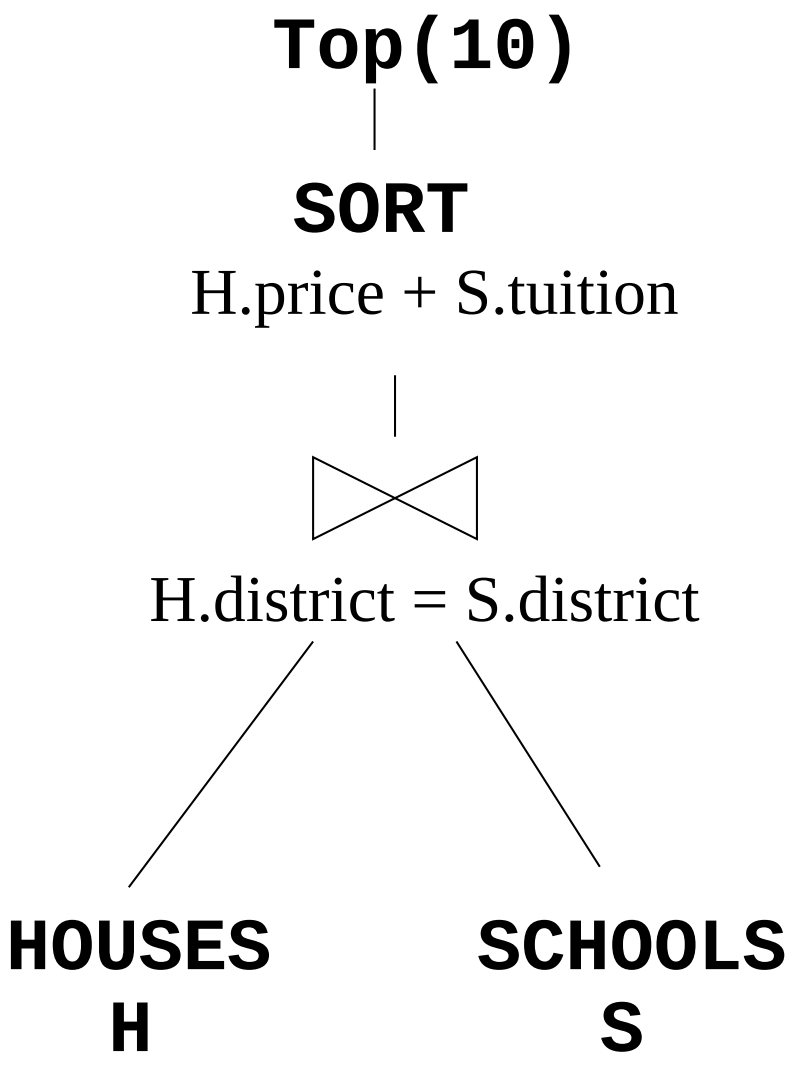
Definition: Top-k Join Query Model

- m Inputs $R_1, \ldots, R_m$. R_i has:
  - n attributes
  - score attribute, s_i (can be an expression over other attributes)
- Input R_i is *sorted* in a descending order on the score attribute s_i
- A global score for a join result is computed as $f(s_1, \ldots, s_m)$
- f is *monotone*

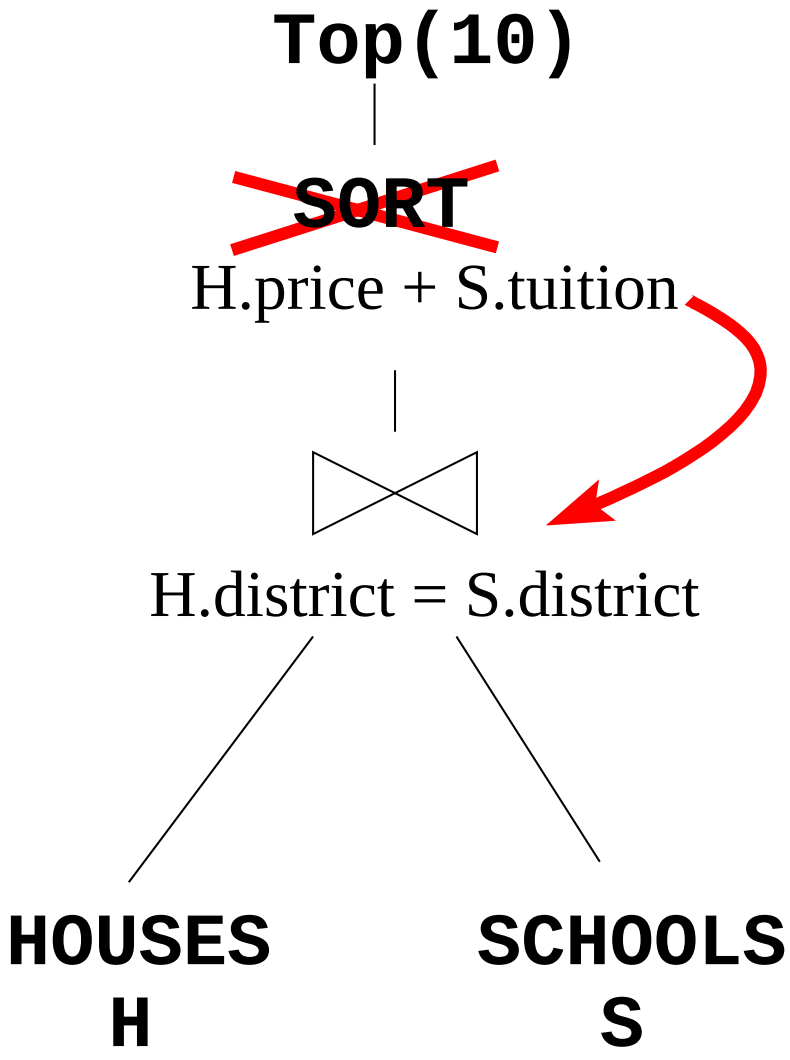
Example

```
SELECT H.id , S.name
FROM HOUSES H , SCHOOLS S
WHERE H.district = S.district
ORDER BY H.price + S.tuition
STOP AFTER 10;
```

Ranking in a Query Plan - Existing Techniques



Ranking in a Query Plan - Our Solution



How to Integrate Ranking into an RDBMS

- **Approach 1: Using table functions**

  (+) Easy to implement and ready-to-go solution

  (-) Implementation outside the SQL engine ⇒ lose efforts of the query optimizer

- **Approach 2: Using a query operator**

  (+) Under the optimizer's control

  (+) Can be shuffled with other operators in a query evaluation plan for better performance

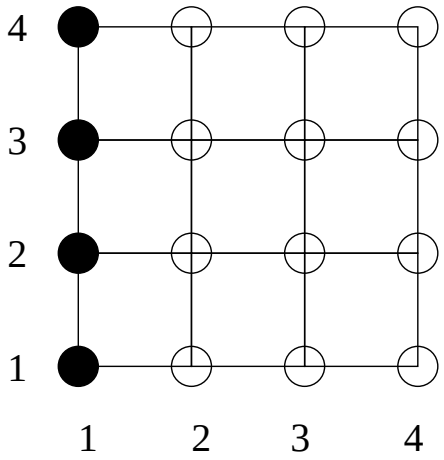
  (+) General enough and highly applicable

  (-) Changes to query engine

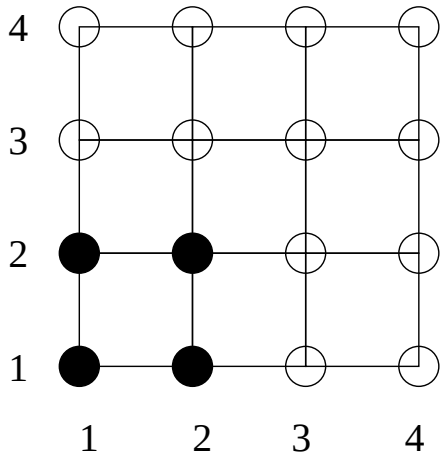
Related Work: Rank Aggregation Algorithms

Algorithm	No Random Access	Pipe-linable	Join Condition
TA [PODS'01]	No	No	key
Multi-step [ICDE'99]	No	No	key
Quick-combine [VLDB'00]	No	No	key
Stream-combine [ITCC'01]	Yes	Yes	key
NRA [PODS'01]	Yes	No	key
J^* [VLDB'01]	Yes	Yes	General
Minimal Probing [SIGMOD'02]	No	No	Fuzzy
Rank-Join this paper	Yes	Yes	General

Top-k Join



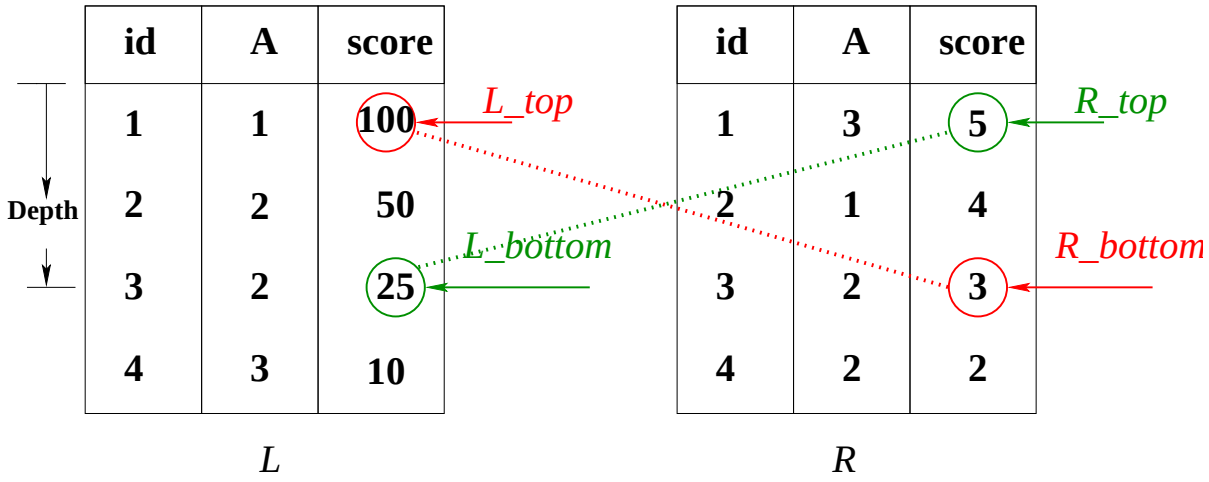
Strategy (a)
Nested−Loops



Strategy (b)
Symmetric Hash Join

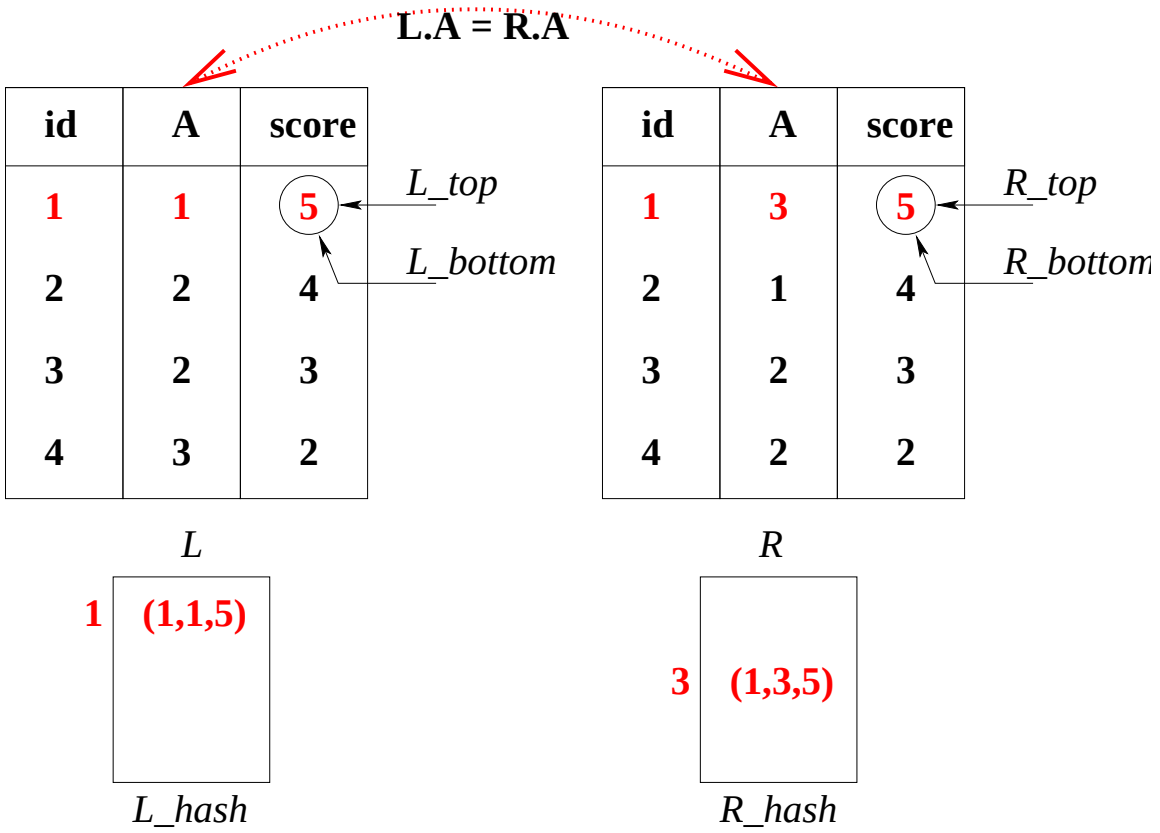
If the inputs are ordered individually, only part of the Cartesian space needs to be visited to get the top-k join results

The New Rank-join Algorithm (Binary Case)



- $LeftThreshold = f(L\_top, R\_bottom) : 100 + 3 = 103$
- $RightThreshold = f(R\_top, L\_bottom) : 5 + 25 = 30$
- $T = MAX(LeftThreshold, RightThreshold) : 103$

Hash Rank-join Operator ($HRJN$) - Example

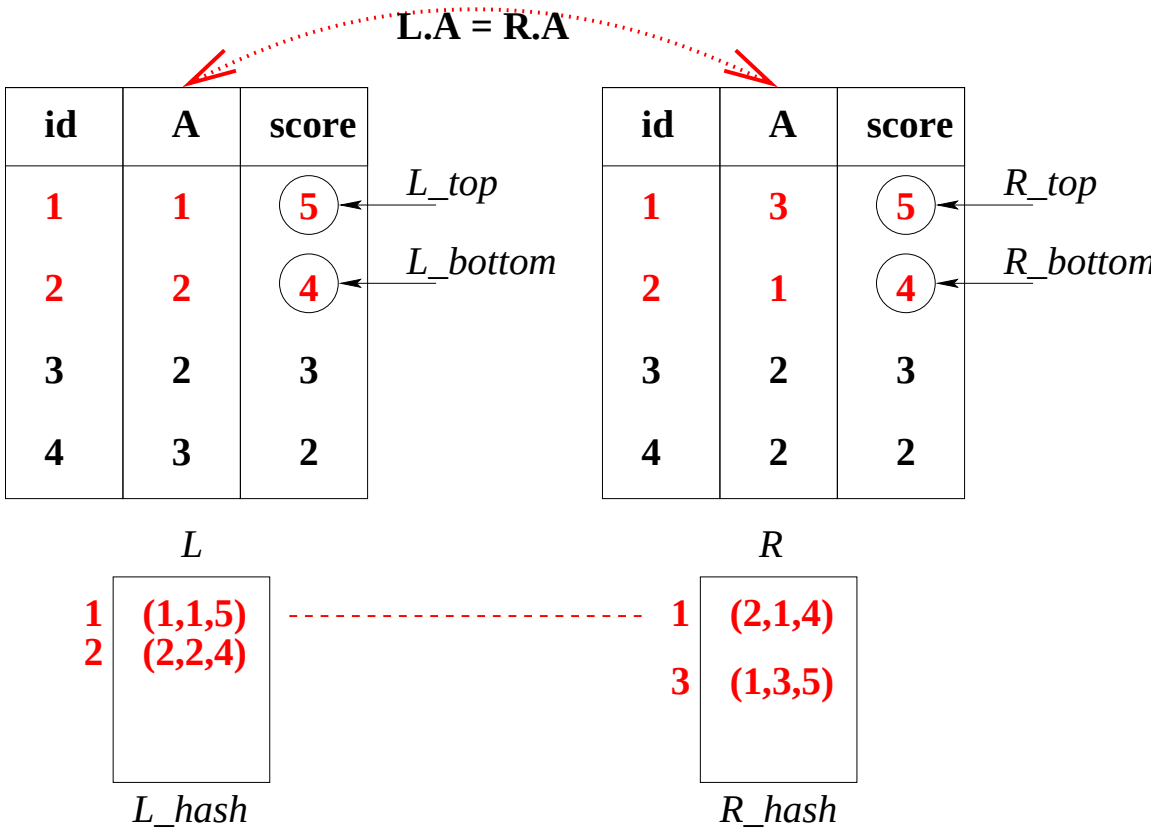


f(L_top, R_bottom) = 5 + 5 = 10

f(L_bottom, R_top) = 5 + 5 = 10

T=Max(10, 10) = 10

Hash Rank-join Operator ($HRJN$) - Example



f(L_top, R_bottom) = 5 + 4 = 9

f(L_bottom, R_top) = 4 + 5 = 9

T=Max(9, 9) = 9

The Effect of Join Strategy

id	A	score
1	1	100
2	2	50
3	2	25
4	3	10

L_top → 100

L_bottom → 50

L

id	A	score
1	3	5
2	1	4
3	2	3
4	2	2

R_top → 5

R_bottom → 2

R

T = Max (2 + 100, 5 + 50) = 102

The Effect of Join Strategy

id	A	score
1	1	100
2	2	50
3	2	25
4	3	10

L_top → 100

L_bottom → 25

L

id	A	score
1	3	5
2	1	4
3	2	3
4	2	2

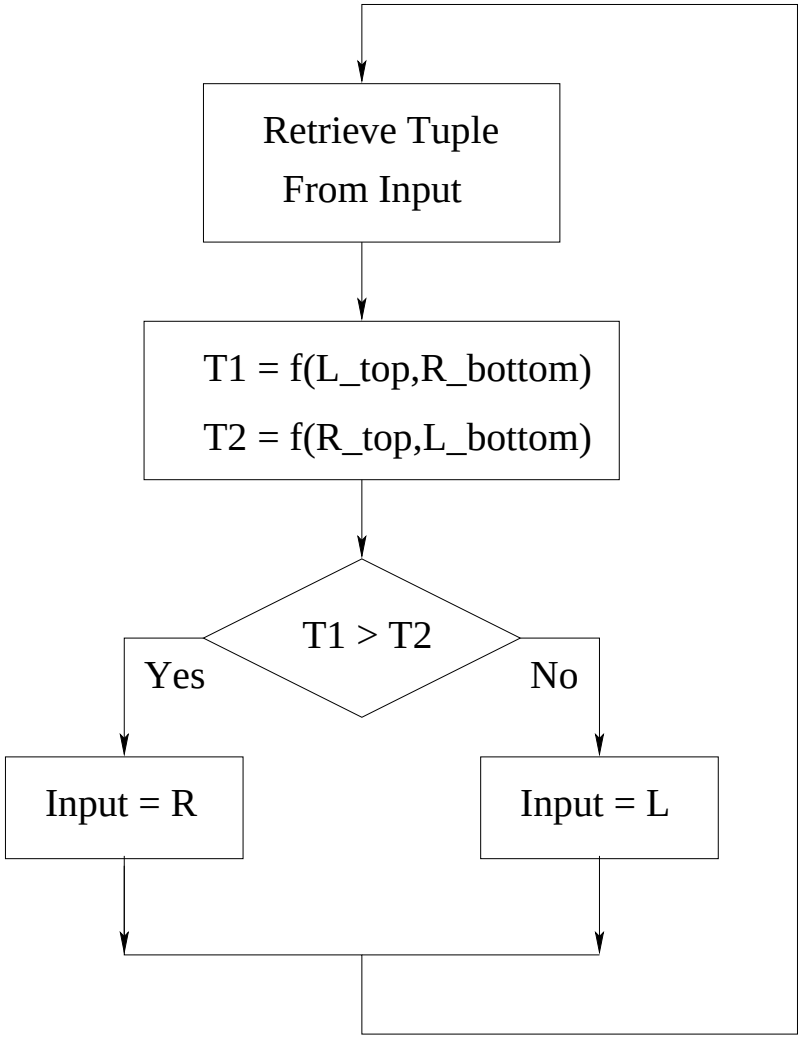
R_top → 5

R_bottom → 3

R

T = Max (3 + 100, 5 + 25) = 103

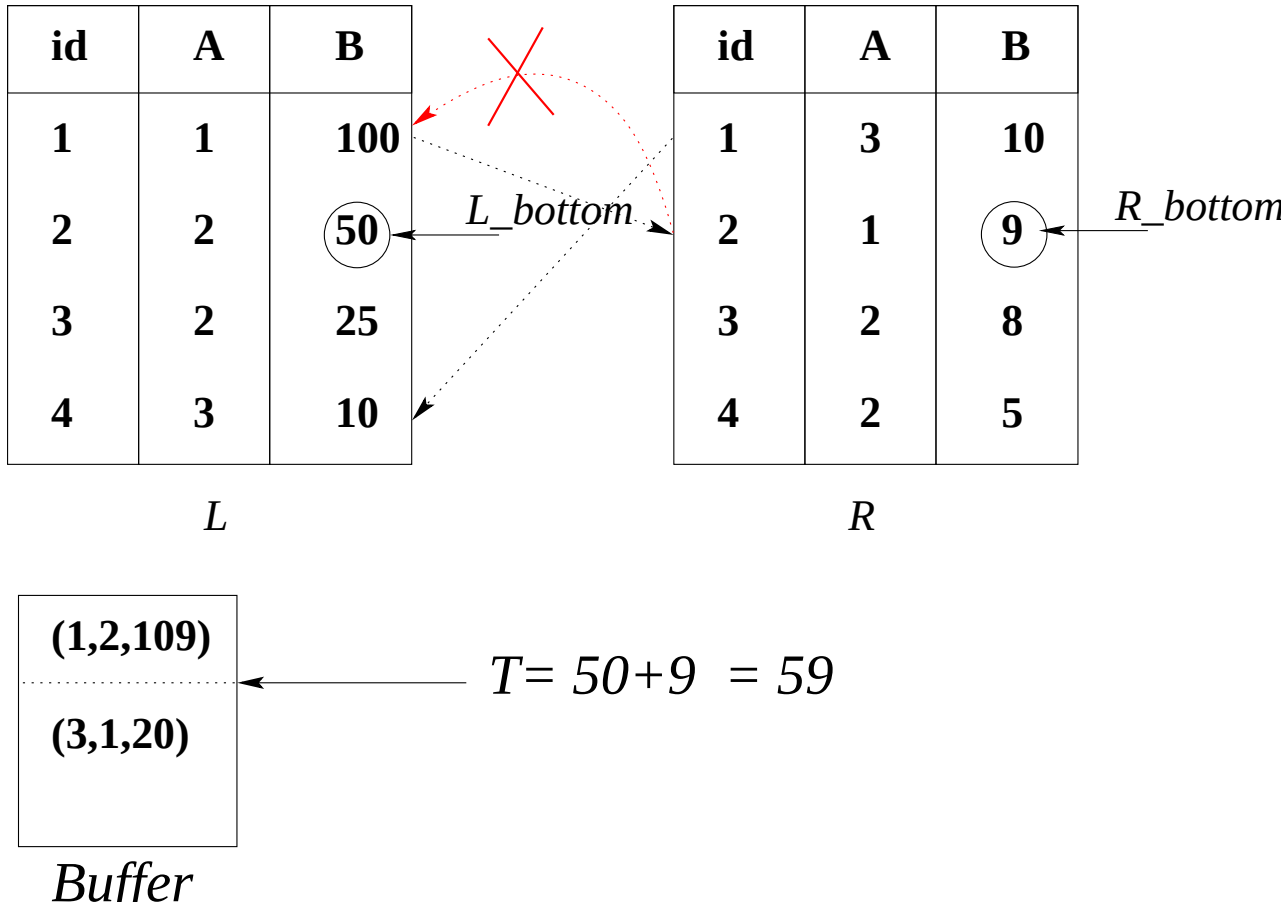
$HRJN^*$: Score-guided Join Strategy



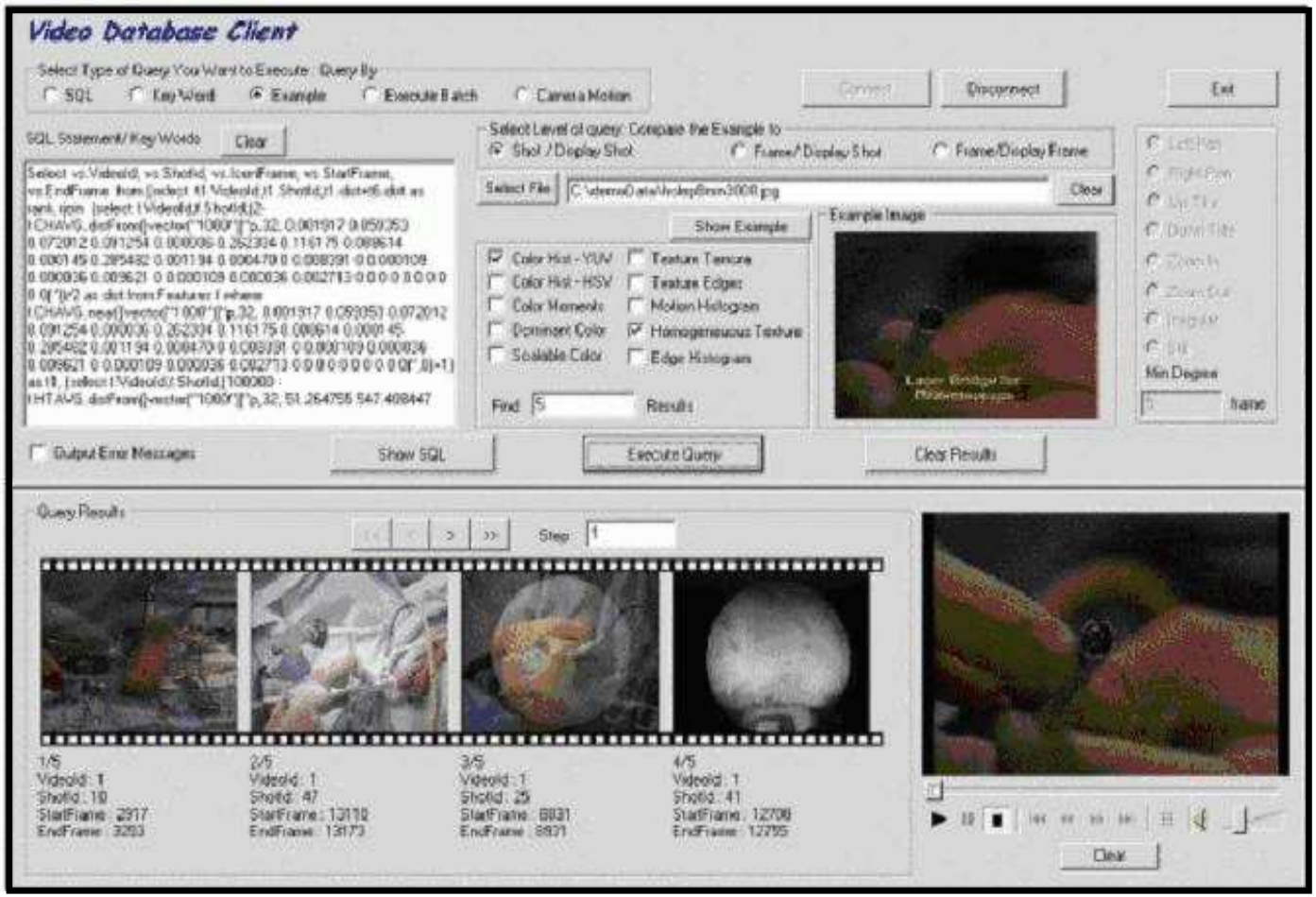
Exploiting Available Indexes

- Generalize Rank-join to use **random access** if available
- Two cases: Two relations L and R
  - **An index on the join attribute(s) of one input R:** Upon receiving a tuple from L, the tuple is first inserted in L's hash table and is used to probe the R index
  - **An index on the join attribute(s) for each input:** Upon receiving a tuple from $L(R)$, the tuple is used to probe the index of $R(L)$. No need to build hash tables
- **Problem:** *Duplicates can be produced* because indexes contain all the data seen and not yet seen

Exploiting Indexes: on-the-fly Duplicate Elimination



Evaluation - Purdue VDBMS



Evaluating $HRJN$ and $HRJN^*$

Q: SELECT T1.id, T2.id, T3.id, T4.id

FROM T1, T2, T3, T4

WHERE T1.JC=T2.JC and

T2.JC=T3.JC and

T3.JC=T4.JC

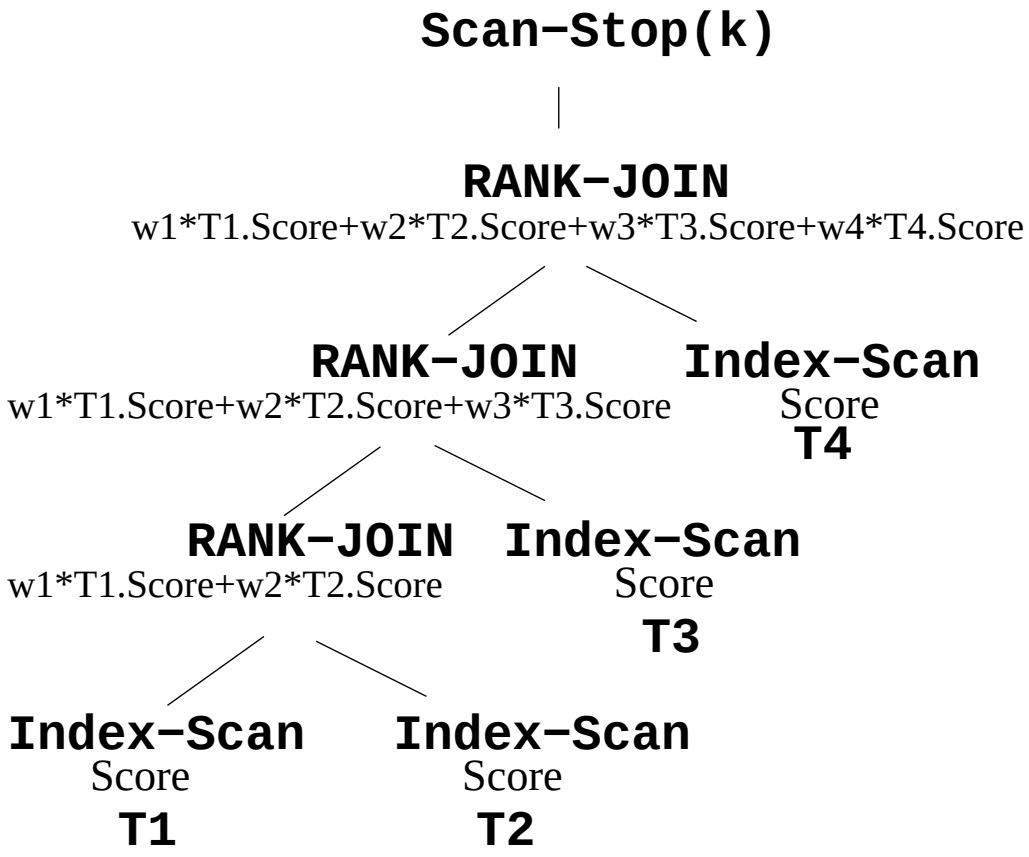
ORDER BY w_1*T1.Score + w_2*T2.Score +

w_3*T3.Score + w_4*T4.Score

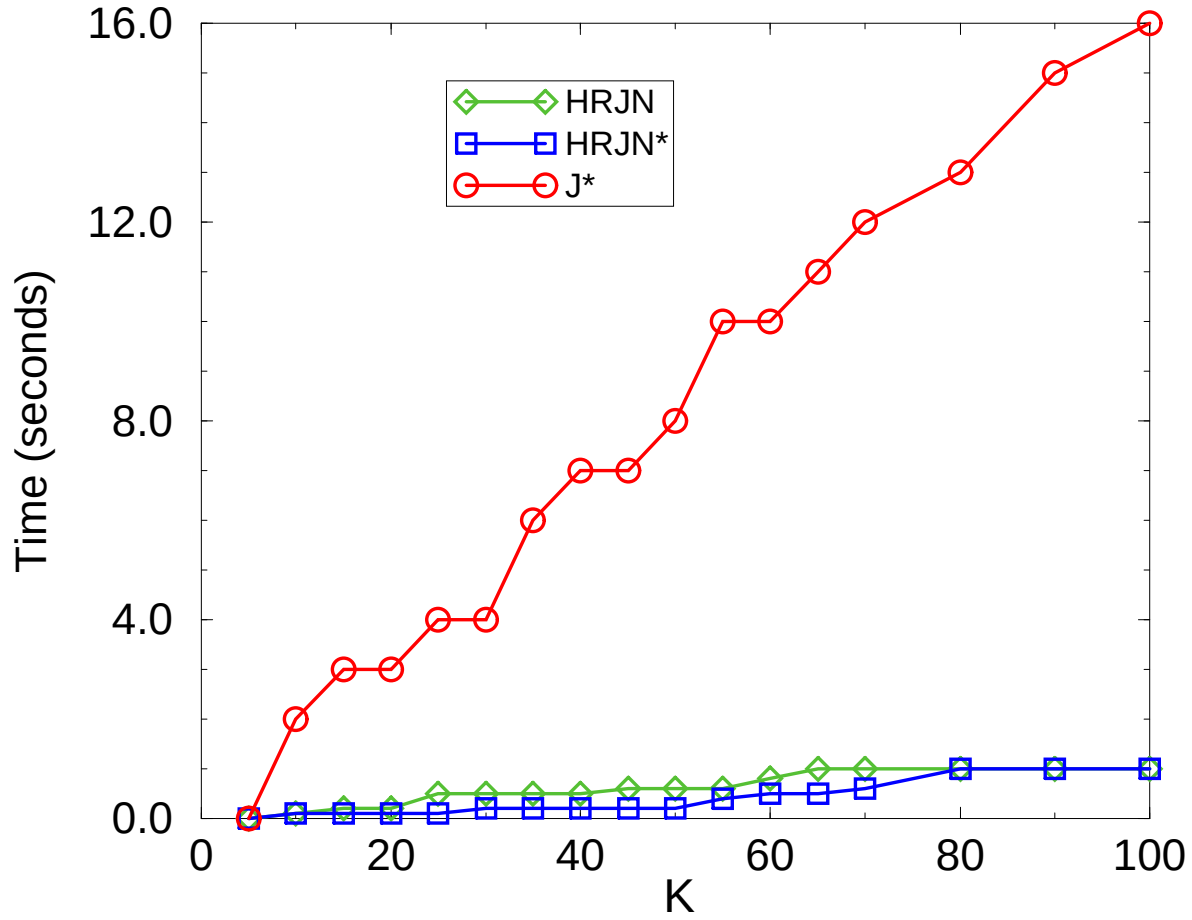
STOP AFTER k;

- Each table has 100,000 tuples

Possible Execution Plan

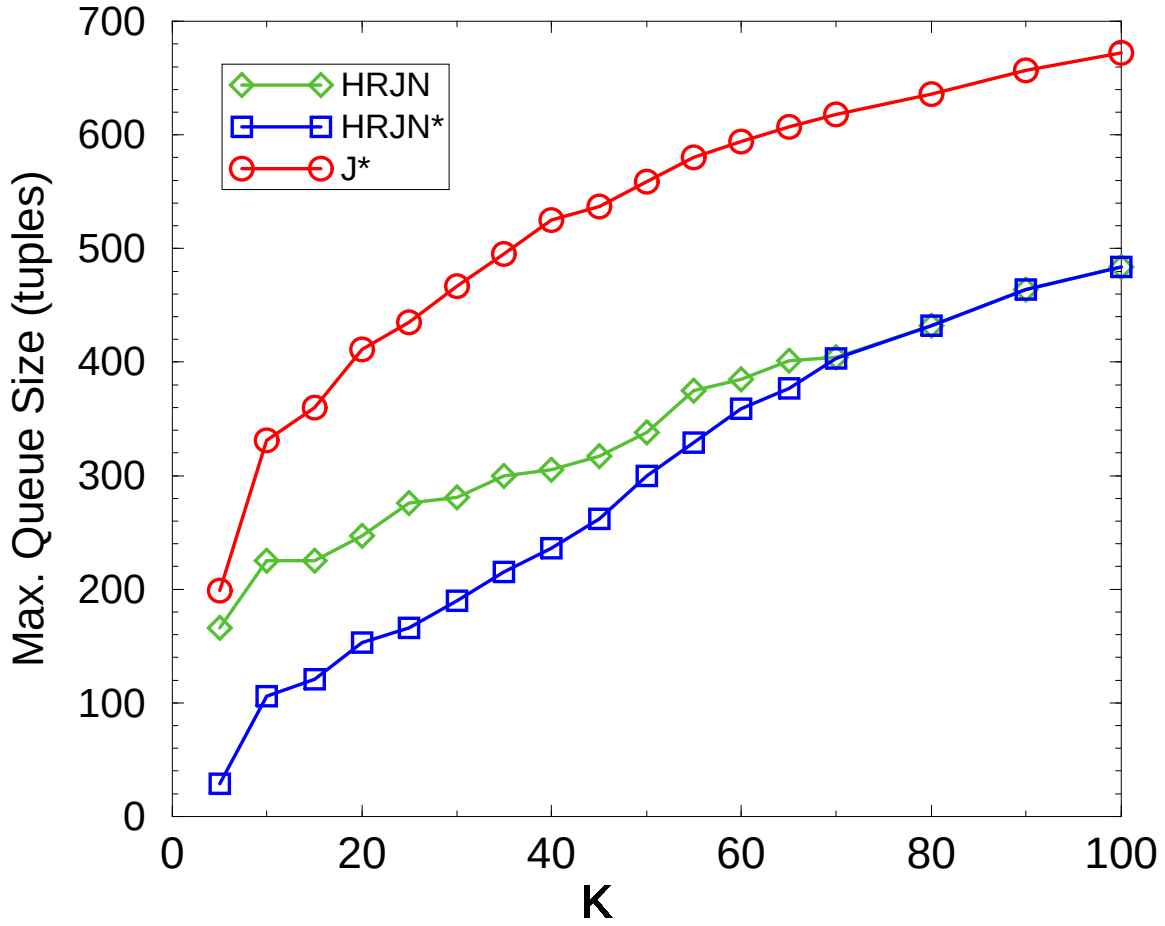


Performance Evaluation of Top-k Join Operators



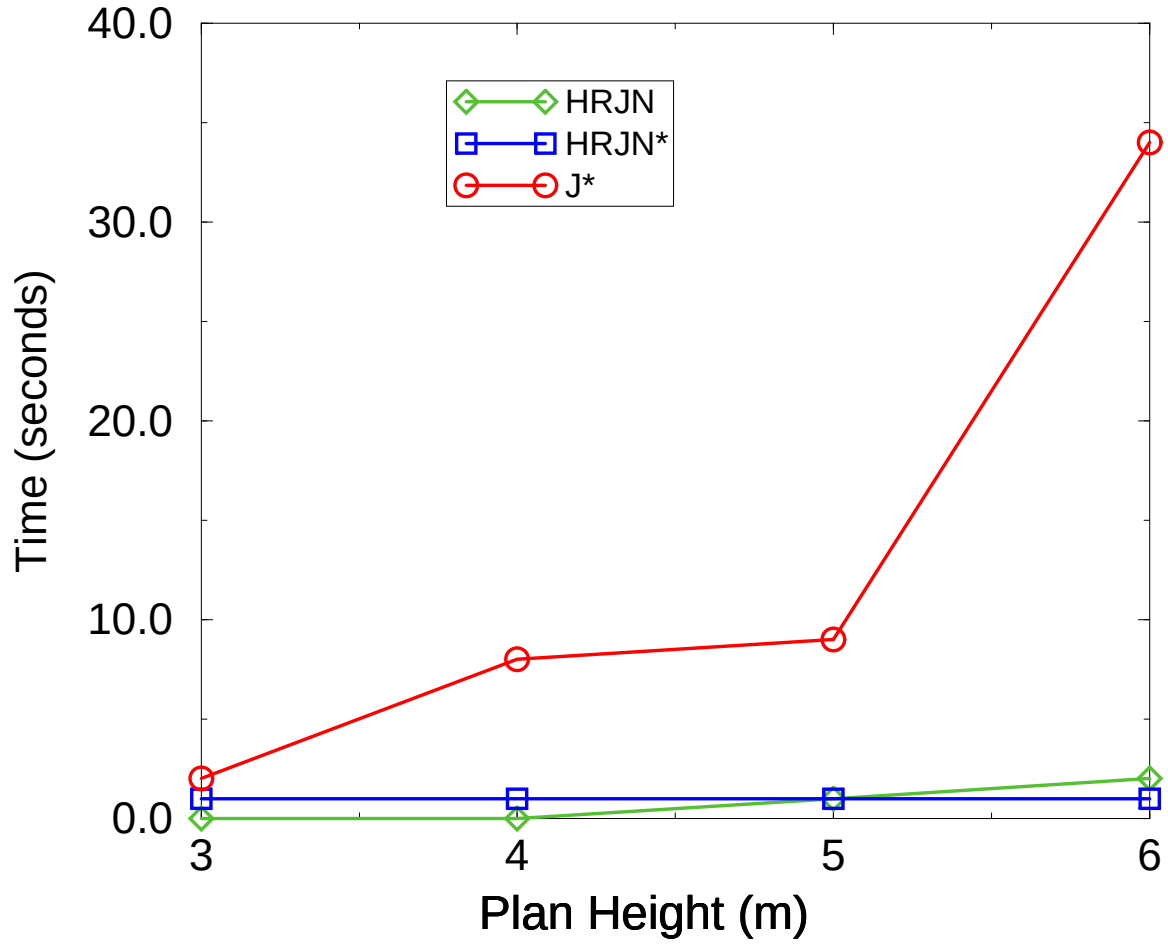
$m = 4$

Performance Evaluation of Top-k Join Operators



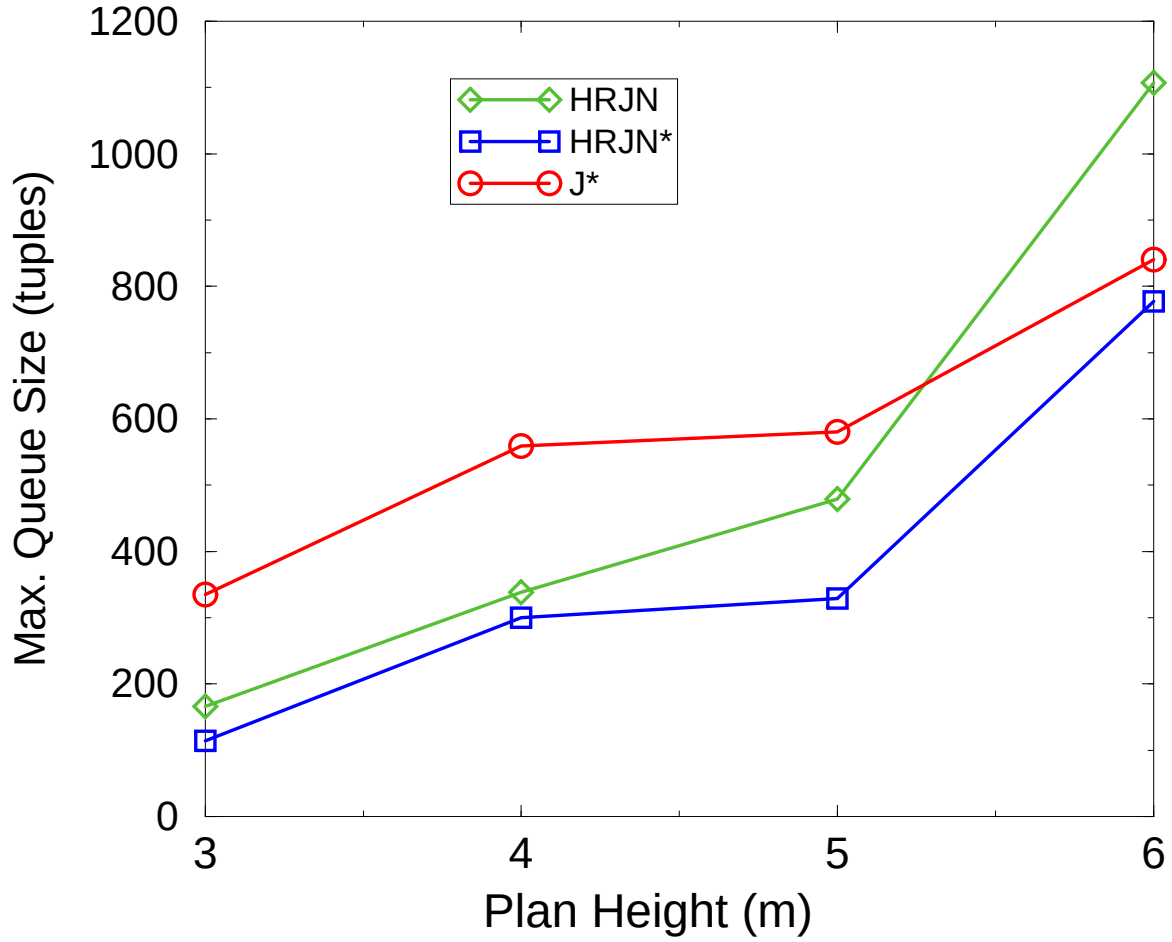
$m = 4$

Performance Evaluation of Top-k Join Operators



$k = 50$

Performance Evaluation of Top-k Join Operators



$k = 50$

Conclusion

- **Goal:** Efficient support for top-k join queries in relational query engines
- **Solution:** new rank-aware query operators that progressively rank the join results and stop after producing the top-k answers
- **Contribution:**
  - New rank-join algorithm
  - Efficient implementation in terms of a pipelined physical query operators: $HRJN$ and $HRJN^*$ are examples
  - Making use of available indexes (random-access)
  - The introduced operators perform better that existing solution (up to orders of magnitude faster)