Ranking in SQL

Přemysl Čech
Martin Stránský
Outline

- Introduction
- Motivation
- RankSQL
- OPT* framework
Introduction

- **Usage of queries ranking**
  - similarity queries in multimedia applications
  - searching Web databases
  - middleware
  - data mining
Introduction

- What are (top-k) queries?
  - queries providing only the top $k$ query results
  - queries are order according to user-specified ranking function
  - ranking function may aggregate multiple criteria

- in SQL: ORDER BY ... LIMIT $k$
Introduction

- Example:

```
SELECT *
FROM Hotel h, Restaurant r, Museum m
WHERE c_1 AND c_2 AND c_3
ORDER BY p_1 + p_2 + p_3 LIMIT k
```

- $c_1$: r.cousine = Italian; $c_2$: h.price + r.price < 100; $c_3$: r.area = h.area
- $p_1$: cheap(h.price); $p_2$: close(h.addr, r.addr); $p_3$: related(m.collection, "dinosaur")
Motivation

- Most RDBMSs support ranking only as an add-on, not primary function
- Evaluation in current RDBMSs
  - exhaust the input tables and materialize the whole joint result
  - evaluate all predicates $p_1, \ldots, p_n$ on each join result
  - sort the join results according to predicates $(p_1 + \ldots + p_n)$
  - report the top $k$ results
Motivation

● Problems of a such approach
  ○ inputs may be huge and don't have to be local
  ○ full materialization and join can be overkill and cause unnecessary overhead and is expensive
  ○ every ranking predicate has to be evaluated against every result
    ■ ranking predicates could be also expensive
    ■ e.g. evaluating $p_1$ by a Web hotel database, comparing geographical data, etc.
RankSQL

- Provides top-k queries as a first class query type
  - integrates such queries in the existing SQL query engines (PostgreSQL)
  - achieves performance improvements
  - proposes rank-relation algebra, optimization techniques and query operators

Ranking principle:
RankSQL

● Rank-Relational Algebra - requirements
  ○ Splitting
    ■ ranking should be evaluated in stages, predicate by predicate – instead of monolithic
  ○ Interleaving
    ■ ranking should be interleaved with other operators – instead of always after filtering (selections and joins)
RankSQL

- **Rank-Relational Algebra**
  - extension of existing operators with rank awareness
  
  \[ (\pi, \sigma, \cup, \cap, -, \otimes) \]

  - for example - rank-join performs the normal boolean join operation and, at the same time, outputs tuples in the “aggregate” order of the operands
    - aggregate order use all the evaluated predicates

  - algebra allows optimization of rank queries as they define equivalent plans in the search space
RankSQL

- Query optimization
  - plan for previous example

(a) A traditional plan.  
(b) A ranking plan.
RankSQL

- The architecture of RankSQL
RankSQL

- Plan builder
  - constructs a physical execution plan
  - saves plan to XML
  - developers can produce their own plans
(1) User connects database and insert query
RankSQL

- Demonstration

(2) Execution plan with operations and products
(3) Summary information about the selected (sub-)plan from (2)
(4) A table with query results after plan execution
Any questions about RankSQL?
Opt* framework introduction

- Target answers are described by:
  - qualifying constraint $B$ which specifies what tuples should be considered valid
  - qualifying function $O$ which measures their degree of matching
  - number $k$ representing how many top matches we want

- Query is then $Q=(B, O, k)$
  - $k$-constrained optimization query
  - specifies goal function $G$
**OPT* framework introduction**

- **Goal function** $G$ consist of
  - boolean constraint expression $B$
  - numeric optimization expression $O$
  - $G = B \times O$

- **Example 1**
  - $Q = (B: h\text{.price} \leq 200k \lor h\text{.price} \geq 400k, O:h\text{.size}/(\lvert h\text{.price}-300k \rvert), k: 1)$
  - select h\text{.address} from House h where h\text{.price} \leq 200k \lor h\text{.price} \geq 400k order by h\text{.size}/(\lvert h\text{.price}-300k \rvert) limit 1
  - not fundamentally supported in RDBMS
OPT* framework motivation

- Seamless optimization of both constraint expressions leads to unified goal function $G$
  - if tuple satisfies constraint $B$ then its score is simply its optimization expression $O$
  - in contrast if tuple fails to satisfy $B$ it is assigned with low score (0) so it can't make it to top-k.
  - $O: \text{dom}(A_1) \times \ldots \times \text{dom}(A_m) \rightarrow \mathbb{R}^+$
  - $B: \text{dom}(A_1) \times \ldots \times \text{dom}(A_m) \rightarrow \{0, 1\}$
    - where $A_i$ are query attributes
    - and $\text{dom}(A_i)$ the domain values of $A_i$
OPT* framework motivation

- Example 1
  - \( Q = (B: \text{h.price} \leq 200k \lor \text{h.price} \geq 400k, O: \text{h.size/} \ (|\text{h.price}−300k|), k: 1) \)
  - price and size are query attributes
OPT* framework motivation

- Left image shows example database instance
- Right image plots landscape of $G$ over price and size
- $Q=\langle G, 1 \rangle$ retrieves top tuple in database maximizing $G$, i.e. (600k, 4500) which is the highest point in landscape

<table>
<thead>
<tr>
<th></th>
<th>A1:Price</th>
<th>A2:Size</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>600K</td>
<td>4500</td>
<td>15</td>
</tr>
<tr>
<td>2.</td>
<td>350K</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>3.</td>
<td>150K</td>
<td>1000</td>
<td>6.67</td>
</tr>
<tr>
<td>4.</td>
<td>250K</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>5.</td>
<td>300K</td>
<td>3500</td>
<td>0</td>
</tr>
<tr>
<td>6.</td>
<td>80K</td>
<td>500</td>
<td>2.27</td>
</tr>
</tbody>
</table>
Goal is to search for database tuples that maximizes $G$ with minimal cost

1. Searching over database suggests use of access methods, e.g. table scan or index scan
   ○ table scan always requires exhaustive search, thus is not optimizable
   ○ index scan allows a focused search by organizing tuples into discrete states preserving value locality

2. Minimizing the cost suggests to effectively guide the search toward maximizing $G$
OPT* framework query mechanism

- Two perspectives:

1. Discrete state search perspective: From the view of using indices (search over a discrete set of index nodes) to find the satisfying data tuples.

2. Continuous function optimization perspective: From the view of optimizing G - optimize the goal function G over the domain of a database.
OPT* framework query mechanism

- From discrete state search perspective we will use B+ trees
  - commonly available in DBMS
  - provide efficient traversal among node clusters by node pointers

<table>
<thead>
<tr>
<th></th>
<th>A1:Price</th>
<th>A2:Size</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>600K</td>
<td>4500</td>
<td>15</td>
</tr>
<tr>
<td>2.</td>
<td>350K</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>3.</td>
<td>150K</td>
<td>1000</td>
<td>6.67</td>
</tr>
<tr>
<td>4.</td>
<td>250K</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>5.</td>
<td>300K</td>
<td>3500</td>
<td>0</td>
</tr>
<tr>
<td>6.</td>
<td>80K</td>
<td>500</td>
<td>2.27</td>
</tr>
</tbody>
</table>

(c) Indices $I$. 
OPT* framework query mechanism

- From continuous function optimization perspective
  - existing optimizing function schemes identify the values optimizing the given function over continuous value space
  - in contrast, a k-constrained optimization query optimizes over database with arbitrary “membership”
we have a "map" of discrete set of regions and have efficient traversals among them

to enable efficient search we let regions to reflect the landscape of $G$ to pursue an informed search guided by such a landscape

such an informed search should be guided properly with some heuristic

A* is a well-known search algorithm that finds shortest path given an initial and a designated goal state

A* has been proven to be complete and optimal with proper heuristics
OPT* What is A* search

- shortest path search algorithm using heuristic to optimize number of visited nodes
- finds shortest path(s) from initial state(s) to goal state
- heuristic function must be admissible and descending
  - admissible = never overestimate
  - descending never increase
Why to encode searching as a general problem, while existing algorithms do not?
- existing methods build upon their problem specific assumptions on goal functions or node traversal, e.g. monotonicity of $G$
- we generalize our search to support:
  1. arbitrary $G$
  2. over multiple indices
  3. hierarchical and interleaf traversals
to enable general support for $k$-constrained queries
OPT* - How A*-search

- How to encode searching as a general problem?
  - From discrete state search perspective we need to define our map for A* search.
  - From continuous function optimization perspective we need to develop the "landscape" to determine starting point and how to proceed.
OPT* - Creating state space

- We construct state space induced by indices - lay out static map that reflects available index structures in DB.
  - Locations on map are states and routes are transitions
  - We construct joined space composed from individual search graphs induced from indices
  - Vertices are database tuples and set of index nodes
  - Edges are set of parent-child links between index nodes, a set of sibling links between leaf nodes and a set of links from leaf nodes to the containing tuples.
  - In principle, such a composite graph, as a cartesian product of those individual index graphs
OPT* - Creating state space

(c) Indices $\mathcal{I}$.

(d) Value space.
OPT* - States

- States represent "localities" of value at different granularity.

1. Region State: While an individual index node defines a range of values along an attribute, a composition of multiple index nodes thus defines a region.

2. Tuple State: Correspond to potential query answers. The goal of search is to find the tuple states that maximize the goal function G.
OPT* - Transitions

- Transitions further capture possible paths followed to reach our destination of query answers.

1. Internal state–Branch in: For internal state \( r \), the reachable states are parent-child links in the index nodes of \( r \).
   a. branches from a parent state to subsuming children states
   b. enable a top-down search approach, starts from root region and gradually zooms into answers

2. Leaf state–Branch out and materialize
OPT* - Transitions

1. Internal state–Branch in
2. Leaf state–Branch out and materialize: consist of two parts – neighbors of r and tuples contained in r.
   a. Expansion to the neighbor states effectively branches out from a leaf region to its surrounding leaf regions
   b. Expansion to the tuple states materializes tuples

Such branch-out transitions enable bottom-up search, which starts from specific leaf states and gradually spreads out to reach answers.
**OPT* - Example 2**

- May follow the path $M_{11} \rightarrow M_{33} \rightarrow M_{77} \rightarrow 1$ to reach the target tuple state - top-down search strategy.
- Bottom-up search, suppose we start from $M_{67}$, $M_{67} \rightarrow M_{77} \rightarrow 1$. 
OPT* - Goal state

- We have state space - our map, but for A* search we need a goal state - a destination, so we add new pseudo state and convert all tuple states to transitions to new pseudo goal state.
- For shortest path to find our goal state and use tuple transition with highest score we set cost of this transition to -score of tuple.
- All other transitions have cost 0.
- Now when algorithm finds top k shortest paths to goal state, we have top k tuples of our query.
OPT* - Heuristic

- A* search needs a heuristic function that is descending and admissible
  - descending means that it never increases
  - admissible means its never overestimate

- we will use function that gives maximum score achievable within the region - even in non existing tuples and again its negative value because we want to highest score be shortest path
OPT* - Heuristic

- this function is not admissible and ascending

$$(M_{67}, 0) -> (M_{77}, 45)$$
OPT* - Heuristic

- this function is not admissible and ascending
- it is caused by some "problematic" links, as we aim to support all access paths, so we remove them
- all links that are uphill are marked as blacklinks and removed from space state
OPT* - Heuristic

- heuristic function is now admissible and descending but now we can't reach every state from every other state
- can't reach $M_{71}$ from $M_{77}$
- $M_{77}$ has the top answer
OPT* - Initial states

- to fix this problem we must pick correct initial states
- because we can travel only downhill we pick all local optimas as initial states, this way we can reach all states
- \( h.\text{price} \leq 200k \lor h.p \)
- \( h.\text{size}/(|h.\text{price} - 300k| \)
- \( M_{57}, M_{77} \)
**OPT*** - Initial states

- \( M_{57}, M_{77} \)
- \( h.\text{size}/(|h.\text{price} - 300k|) \)
- \( h.\text{price} \leq 200k \lor h.\text{price} \geq 400k \)
OPT* - Compare to other

- $Q_1: \frac{\text{size} \times \text{bedrooms}}{|\text{price} - 450k|}$
  
  $[40k \leq \text{price} \leq 50k]$
OPT* - Compare to other

- $Q_2: (\text{size} \times e^{(\text{bedrooms})}) / |\text{price} - 450k|$
  - $\text{price} < 400k \land \text{size} > 4000$
OPT* - Compare to other

- \( Q_3 \): size/price
  \[ \text{bedrooms} = 3 \lor \text{bedrooms} = 4 \]
References

- Chengkai Li, Mohamed A. Soliman, Kevin Chen-Chuan Chang, Ihab F. Ilyas, RankSQL: Supporting Ranking Queries in Relational Database Management Systems, 2005

- Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A. Lang, Yuan-chi Chang, Boolean + Ranking: Querying a Database by K-Constrained Optimization.