Algorithm 2 Lexicographic Inverted-Access

```
1:  $k = 0$
2:  bucket[1] = root
3:  factor = weight(root)
4:  for i=1,...,f do
5:      factor = factor/weight(bucket[$i$])
6:      select $t \in$ bucket[$i$] agreeing with the answer
7:      if no such $t$ exists then
8:          return not-an-answer
9:      $k = k + \mathtt{start}(t) \cdot$ factor
10:     for child $V$ of layer $i$ do
11:         get the bucket $b \in V$ agreeing with the answer
12:         bucket[layer($V$)] = $b$
13:         factor = factor $\cdot$ weight($b$)
14: return $k$
```

total time $O(n \log n)$ for preprocessing and enables constant-time direct access to individual answers in ranked order.

A.7 Proof of Lemma 6.5

For the "if" direction, we can eliminate absorbed atoms from Q to obtain Q^m after making sure that the tuples in the database satisfy those atoms. Thus, to remove an atom $S(\vec{Y})$ which is absorbed by $R(\vec{X})$, we filter the relation R based on the tuples of S. Then, Q^m over the filtered database has the same answers as Q over the original one. For the "only if" direction, each atom $S(\vec{Y})$ that appears in Q but not Q^m is absorbed by some $R(\vec{X})$. We create the relation S by copying $\pi_{\vec{Y}}(R)$ into it, essentially making the atom $S(\vec{Y})$ obsolete. Note that we are allowed to create S without restrictions because Q has no self-joins, hence the database doesn't already contain the relation. Then, Q over the extended database has the same answers as Q^m over the original one. The above reductions take linear time, which is dominated by $T_S(n)$ since $T_S(n)$ is trivially in $\Omega(n)$ for the selection problem.

A.8 Proof of Lemma 6.6

By Lemma 6.5, it suffices to solve selection on the query $Q(\vec{X}) :- R(\vec{X})$, which is a maximal contraction of all queries with $\mathrm{mh}(Q) = 1$. Initially, we turn the attribute weights into tuple weights. For each tuple $r \in R$, we compute its weight as $w(r) = \sum_{x \in \vec{X}} w_x(r(x))$. Thus, the weights $w(r)$ are the weights of the query answers. This takes $O(n)$ for the $O(n)$ tuples of R. Then, applying linear-time selection [6] on R gives us the kth smallest query result.

A.9 Proof of Lemma 6.10

First, for $\alpha_{\text{free}}(Q) = 1$, we have by Lemma 5.4 that an atom contains all free variables, thus $\mathrm{mh}(Q) = 1$ For the case of $\alpha_{\text{free}}(Q) = 2$, let x and u be the two independent variables. Because they do not appear together in the same atom, there exist two different atoms e_R, e_T such that e_R contains x but not u and e_T contains u but not x. Without loss of generality, we can further assume that the variable sets in these atoms are not strictly contained in others (if they are, we can choose those instead). We can also assume that our choice of independent variables x, y and atoms e_R, e_T is such that these two atoms do not have any variables in common, otherwise Q would be cyclic. We also have at least one more maximal atom-hyperedge e_S, that is not absorbed by e_R or e_T because $\mathrm{mh}(Q) > 2$. For the variables of e_S, we claim that $\mathtt{var}(e_S) \subseteq (\mathtt{var}(e_R) \cup \mathtt{var}(e_T))$. Suppose that e_S contains a variable s s.t. $s' \notin (\mathtt{var}(e_R) \cup \mathtt{var}(e_T))$. Then because t cannot be independent, there must exist an atom e_U that contains x and t (or u and t). However, in that case, e_R, e_S, e_U (or e_T, e_S, e_U) create a cycle violating the acylicity of Q. Let $\vec{Y}$ be the variables in $\mathtt{var}(e_R) \cap \mathtt{var}(e_S)$ and $\vec{Z}$ those in $\mathtt{var}(e_S) \cap \mathtt{var}(e_T)$. We have $\vec{Y} \neq \emptyset$ and $\vec{Z} \neq \emptyset$, otherwise e_S would be absorbed by e_R or e_T respectively. Conversely, $\mathtt{var}(e_R) \not\subset \mathtt{var}(e_S)$ because e_R would be absorbed by e_S, and the same is true for e_T. At this point, the other atoms of the query can only be absorbed by the existing ones, otherwise we introduce an independent variable or a cycle.

B INVERTED ACCESS BY LEXICOGRAPHIC ORDER

A straightforward adaptation of Algorithm 1 can be used to achieve *inverted access*: given a query result as the input, we return its index according to the lexicographic order. Algorithm 2 is almost the same algorithm as Algorithm 1 except that the choices in each iteration are made according to the given answer and the corresponding index is constructed (instead of the opposite). The algorithm runs in constant time per answer since every operation can be done within that time (unlike Algorithm 1, there is no need for binary search here).