Efficient Approximation of Certain and Possible Answers for Ranking and Window Queries over Uncertain Data

Su Feng
Illinois Institute of Technology
seng14@hawk.iit.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Oliver Kennedy
SUNY Buffalo
okennedy@buffalo.edu

ABSTRACT

Uncertainty arises naturally in many application domains due to, e.g., data entry errors and ambiguity in data cleaning. Prior work in incomplete and probabilistic databases has investigated the semantics and efficient evaluation of ranking and top-k queries over uncertain data. However, most approaches deal with top-k and ranking in isolation and do represent uncertain input data and query results using separate, incompatible data models. We present an efficient approach for under- and over-approximating results of ranking, top-k, and window queries over uncertain data. Our approach integrates well with existing techniques for querying uncertain data, is efficient, and is to the best of our knowledge the first to support windowed aggregation. We design algorithms for physical operators for uncertain sorting and windowed aggregation, and implement them in PostgreSQL. We evaluated our approach on synthetic and real-world datasets, demonstrating that it outperforms all competitors, and often produces more accurate results.

PVLDB Reference Format:
doi:10.14778/3583140.3583151

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fengsu91/uncert-ranking-availability.

1 INTRODUCTION

Many application domains need to deal with uncertainty arising from data entry/extraction errors [36, 51], data lost because of node failures [39], ambiguous data integration [7, 31, 46], heuristic data wrangling [13, 21, 58], and bias in machine learning training data [26, 50]. Incomplete and probabilistic databases [18, 55] model uncertainty as a set of so-called possible worlds. Each world is a deterministic database representing one possible state of the real world. The commonly used possible world semantics [55] returns for each world the (deterministic) query answer in this world. Instead of this set of possible answer relations, most systems produce either certain answers [33] (result tuples that are returned in every world), or possible answers [33] (result tuples that are returned in at least one world). Unfortunately, incomplete databases lack the expressiveness of deterministic databases and have high computational complexity.

Notably, uncertain versions of order-based operators like SORT / LIMIT (i.e., Top-K) have been studied extensively in the past [19, 41, 48, 54]. However, the resulting semantics often lacks closure. That is, composing such operators with other operators typically requires a complete rethinking of the entire system [52], because the model that the operator expects its inputs to be encoded with differs from the model encoding the operator’s outputs.

In [23, 24], we started addressing the linked challenges of computational complexity, closure, and expressiveness in incomplete database systems, by proposing AU-DBs, an approach to uncertainty management that can be competitive with deterministic query processing. Rather than trying to encode a set of possible worlds losslessly, each AU-DB tuple is defined by one range of possible values for each of its attributes and a range of (bag) multiplicities. Each tuple of an AU-DB is a hypercube that bounds a region of the attribute space, and together, the tuples bound the set of possible worlds between an under-approximation of certain answers and an over-approximation of possible answers. This model is closed under relational algebra [23] with aggregation [24] (RAagg). That is, if an AU-DB D bounds a set of possible worlds, the result of any RAagg query over D bounds the set of possible query results. We refer to this correctness criteria as bound preservation. In this paper, we add support for bounds-preserving order-based operators to the AU-DB model, along with a set of (nontrivial) operator implementations that make this extension efficient. The closure of the AU-DB model under RAagg, its efficiency, its property of bounding certain and possible answers, and its capability to compactly represent large sets of possible tuples using attribute-level uncertainty are the main factors for our choice to extend this model.

When sorting uncertain attribute values, the possible order-by attribute values of two tuples t1 and t2 may overlap, which leads to multiple possible sort orders. Thus, supporting order-based operators over AU-DBs requires encoding multiple sort orders. Unfortunately, a dataset can only have one physical ordering. We address this limitation by introducing a position attribute, decoupling the physical order in which the tuples are stored from the set of possible logical orderings. With a tuple’s position in a sort order encoded as a numerical attribute, operations that act on this order (i.e., LIMIT) can be redefined in terms of standard relational operators, which, already have well-defined semantics in the AU-DB model. In short, by virtualizing sort order into a position attribute, the existing AU-DB model is sufficient to express the output of SQL’s order-dependent operations in the presence of uncertainty.

We start by (i) formalizing uncertain orders within the AU-DB model and present a semantics of sorting and windowed aggregation operations that can be implemented as query rewrites.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097. doi:10.14778/3583140.3583151
combined with existing AU-DB rewrites [23, 24], any \textit{RA_{app}} query with order-based operations can be executed using a deterministic DBMS. Unfortunately, these rewrites introduce SQL constructs that necessitate computationally expensive operations, driving a central contribution of this paper: (ii) new algorithms for sort, top-k, and windowed aggregation operators for AU-DBs.

To understand the intuition behind these operators, consider the logical sort operator, which extends each input row with a new attribute storing the row’s position wrt. to ordering the input relation on a list \( O \) of order-by attributes. If the order-by attributes’ values are uncertain, we have to reason about each tuple \( t \)’s lowest possible position (the number of tuples that certainly precede it in all possible worlds), and highest possible position (the number of tuples that possibly precede it in at least one possible world). We can naively compute a lower (resp., upper) bound by joining every tuple \( t \) with every other tuple, counting pairs where \( t \) is certainly (resp., possibly) preceded by the tuple it is paired with. We refer to this approach as the rewrite method, as it can be implemented in SQL. However, the rewrite approach has quadratic runtime. Inspired by techniques for aggregation over interval-temporal databases such as [47], we propose a one-pass algorithm to compute the bounds on a tuple’s position that also supports top-k queries.

**Example 1 (Uncertain Sorting and Top-k).** Fig. 1a shows a sales DB, extracted from 3 press releases. Uncertainty arises for the 4th term in \( D_1 \). The task of finding the two terms with the most sales is semantically ambiguous for uncertain data. Consider the following semantics for uncertain ranking: (i) U-top [54] (Fig. 1c) returns the most likely ranked order; (ii) U-rank [54] (Fig. 1e) returns the most likely tuple at each position (term 4 is more likely than any other value for both the 1st and 2nd position); and (iii) Probabilistic threshold queries (PT-k) [32, 59] return tuples that appear in the top-k with a probability exceeding a threshold (PT), generalizing both possible (PT > 0; Fig. 1d) and certain (PT ≥ 1; Fig. 1e) answers.

With the exception of U-Top, none of these semantics return both information about certain and possible results, making it difficult for users to gauge the (i) trustworthiness or (ii) completeness of an answer. Risk assessment on the results produced by these semantics is difficult, preventing their use for critical applications in, e.g., the medical or financial domains. Furthermore, the outputs of uncertain ranking operators like U-Top are not valid as inputs to further uncertainty-aware queries, because they lose information about uncertainty in the source data. These disadvantages motivate our choice of the AU-DB data model. First, AU-DBs naturally encode query result reliability. By providing each attribute value (and tuple multiplicity) as a range, users can quickly assess the precision of an answer. Second, the model is complete: the full set of possible answers is represented. Finally, the model admits a closed, efficiently computable, and bounds-preserving semantics for \textit{RA_{app}}.

**Example 2 (AU-DB top-2 query).** Fig. 1f (right) shows the result of computing the top-2 answers sorted on sales. AU-DB admits additional worlds with 5 sales in term 4. For example, the tuples \([4,7,5]\) or \([4,4,7]\) are certainly not in the result. The AU-DB compactly represents an under-approximation of certain answers and an over-approximation of all the possible answers, e.g., for our example, the AU-DB admits additional worlds with 5 sales in term 4.

Implementing windowed aggregation requires determining the (uncertain) membership of tuples in windows, which may be affected both by uncertainty in sort position and in group-by attributes. Furthermore, we have to reason about which of the tuples possibly belonging to a window minimize / maximize the aggregation function result. It is possible to implement this reasoning in SQL, albeit at the cost of range self-joins (this is the rewrite method we will discuss in detail in [22] and evaluate in Sec. 8). We propose a one-pass algorithm for windowed aggregation over AU-DBs, which we will refer to as the native method.

The intuition behind our algorithm is to share state between multiple windows. For example, consider the window \texttt{ROWS BETWEEN 3 PRECEDING AND CURRENT ROW}. In the deterministic case, with each

---

1. The process of obtaining a selected-guess world is domain-specific, but [23, 24] suggest the most likely world, if it can be feasibly obtained.
We build on prior research in incomplete and probabilistic databases, with ranges to support aggregation. MCDB [34] and Pip [37] sample possible outcomes, but can only obtain probabilistic bounds on their from the set of possible worlds to generate expectations of possible outcomes. Selected-guess and lower bounds-based approach of [23], adding an 29, 43, 49 can be computed in complexity [6, 33]. However, under-approximations [ 17, 23, 28, 45, 52, 57]. These result encodings are not closed (i.e., not useful for subsequent queries), and are also expensive to compute (often NP-hard). Symbolic models [12, 25, 40] that are closed under aggregation permit PTIME data complexity, but extracting certain / possible answers is still intractable. We proposed AU-DBs [24] which are closed under and achieve efficiency through approximation.

Uncertain Top-k. A key challenge in uncertain top-k ranking is defining a meaningful semantics. The set of tuples certainly (resp., possibly) in the top-k may have fewer (more) than k tuples. U-Top [54] picks the top-k set with the highest probability. U-Rank [54] assigns to each rank the tuple which is most-likely to have this rank. Global-Topk [59] first ranks tuples by their probability of being in the top-k and returns the k most likely tuples. Probabilistic threshold top-k (PT-k) [32] returns all tuples that have a probability of being in the top-k that exceeds a pre-defined threshold. Expected rank [19] calculates the expected rank for each tuple across all possible worlds and picks the k tuples with the highest expected rank. Ré et al. [48] proposed a multi-simulation algorithm that stops when a guaranteed top-k probability can be guaranteed. Soliman et al. [53] proposed a framework that integrates tuple retrieval, grouping, aggregation, uncertainty management, and ranking in a pipelined fashion. Each of these generalizations necessarily breaks some intuitions about top-k, producing more (or fewer) than k tuples, or producing results that are not the top-k in any world.

Uncertain Order. Amarilli et. al. extends the relational model with a partial order to encodes uncertainty in the sort order of a relation [10, 11]. For more general use cases where posets can not represent all possible worlds, Amarilli et. al. also develop a symbolic model of provenance [9] whose expressions encode possible orders. Both approaches are limited to set semantics.

3 NOTATION AND BACKGROUND

A database schema $\text{Sch}(D) = \{\text{Sch}(R_1), \ldots, \text{Sch}(R_n)\}$ is a set of relation schemas $\text{Sch}(R_i) = (A_{1i}, \ldots, A_{ni})$. Use $\text{arity}(\text{Sch}(R))$ to denote the number of attributes in $\text{Sch}(R)$. An instance $D$ for schema $\text{Sch}(D)$ is a set of relation instances with one relation per schema in $\text{Sch}(D): D = \{R_1, \ldots, R_n\}$. Assuming a universal value domain $D$, a tuple with schema $\text{Sch}(R)$ is an element from $\mathbb{D}^{\text{arity}(\text{Sch}(R))}$.

A $K$-relation [27] annotates each tuple with an element of a (commutative) semiring. In this paper, we focus on $\mathbb{N}$-relations. An $\mathbb{N}$-relation of arity $n$ is a function that maps each tuple $(\mathbb{D}^n)$ in the relation to an annotation in $\mathbb{N}$ representing the tuple’s multiplicity. Tuples not in the relation are mapped to multiplicity 0. $\mathbb{N}$-relations are required to have finite support (tuples not mapped to 0). Since $K$-relations are functions from tuples to annotations, it is customary to denote the annotation of a tuple $t$ in relation $R$ as $\text{R}(t)$. A $K$-database is a set of $K$-relations. Green et al. [27] did use the semiring operations to express positive relational algebra ($\mathbb{R}^+\mathbb{A}^+$) operations.
AU-DBs encode bounds on the multiplicities of tuples by using
bounds on the multiplicities of the tuple, the multiplicity of the tuple in the SGW, and
codes the selected-guess value of all of the possible world’s tuples, and the total multiplicity of tuples
for range-annotated each hypercube tuple is also a range of possible annotations (e.g.,
zero or more tuples contained inside it). Second, the annotation of
a range-annotated tuple with schema
(a_1, \ldots, a_n) and t be a tuple with the same schema as t. t bounds t (denoted t \sqsubseteq t) iff \forall i \in \{1, \ldots, n\} : t.a_i \leq t.a_i^1
Note that a single AU-DB tuple may bound multiple deterministic tuples, and conversely that a single deterministic tuple may be bound by multiple AU-DB tuples. Informally, an AU-DB relation bounds a possible world if we can distribute the multiplicity of each tuple in the possible world over the AU-DB relation’s tuples. This idea is formalized through tuple matchings. A tuple matching \mathcal{T}M from an n-ary AU-DB relation to an n-ary relation is a function \( (\mathcal{D}t)^n \rightarrow \mathcal{N} \) that fully allocates the multiplicity of every tuple of \( R \):
\[
\forall t \in (\mathcal{D}t)^n : \forall \not\in t : \mathcal{T}M(t, t) = 0 \quad \forall t \in \mathcal{D}t^n : \sum_{t \in \mathcal{D}t^n} \mathcal{T}M(t, t) = R(t)
\]
R bounds R (denoted \( R \sqsubseteq R \)) iff there exists a tuple matching \( \mathcal{T}M \) where the total multiplicity allocated to each \( t \in \mathcal{R} \) falls within the bounds annotating t:
\[
\forall t \in \mathcal{D}t^n : \sum_{t \in \mathcal{D}t^n} \mathcal{T}M(t, t) \geq R(t)^\downarrow \quad \text{and} \quad \sum_{t \in \mathcal{D}t^n} \mathcal{T}M(t, t) \leq R(t)^\uparrow
\]
An AU-DB relation R bounds an incomplete \( N \)-relation \( R \) (denoted \( R \sqsubseteq R \)) iff it bounds every possible world (i.e., \( \forall R \in \mathcal{R} : R \sqsubseteq R \)), and if projecting down to the selected guess attribute of R results in a possible world of \( \mathcal{R} \). As shown in [23, 24], (i) AU-DB query semantics is closed under \( \mathcal{R}A^* \), set difference and aggregations, and (ii) queries preserve bounds. That is, if every relation \( R_i \in \mathcal{D} \) bounds the corresponding relation of an incomplete database \( R_i \in \mathcal{D} \) (i.e., \( \forall i : R_i \sqsubseteq R_i \)), then for any query Q, the results over \( \mathcal{D} \) bound the results over \( \mathcal{D} \) (i.e., \( Q(\mathcal{D}) \sqsubseteq Q(\mathcal{D}) \)).

Expression Evaluation. In [24], we defined a semantics \([e]_t\) for evaluating primitive-valued expressions e over the attributes of a range tuple t. These semantics preserves bounds; given any expression e and any deterministic tuple t bounded by t (i.e., t \sqsubseteq t), the result of deterministically evaluating the expression ([e]_t) is guaranteed to be bounded by the ranged evaluation \([e]_t\).

\[
\forall t \sqsubseteq t : c = [e]_t, (c, c^\downarrow, c^\uparrow) = [e]_t \quad \rightarrow \quad c^\downarrow \leq c \leq c^\uparrow
\]
[24] proved this property for any e composed of attributes, constants, arithmetic and boolean operators, and comparisons. For example, \([a^2/a^2/a^2]^1 + [b^2/b^2/b^2]^1 = [a^2 + b^2/a^2a^2 + b^2a^2 + b^2] \]

4 DETERMINISTIC SEMANTICS
Before introducing the AU-DB semantics for ranking and windowed aggregation, we first formalize the corresponding deterministic algebra operators that materialize sort positions of rows as data.

Sort order. Assume a total order \( < \) for the domains of all attributes. For simplicity, we only consider sorting in ascending order.
The extension for supporting both ascending and descending order is straightforward. For any two tuples \( t \) and \( t' \) with schema \( (A_1, \ldots, A_n) \) and sort attributes \( O = (A_{i_1}, \ldots, A_{i_m}) \) we define:
\[
t <_O t' \iff \exists j \in \{1, \ldots, m\} : \forall k \in \{1, \ldots, j - 1\} : t.A_{i_k} = t'.A_{i_k} \land A_{i_j} < t.A_{i_j}
\]
The less-than or equals comparison operator \( \leq_O \) generalizes this definition in the usual way. Note that SQL sorting (ORDER BY) and some window bounds (ROW BETWEEN . . .) may be non-deterministic. For instance, consider a relation \( R \) with schema \( (A, B) \) with two rows \( t_1 = (1, 1) \) and \( t_2 = (1, 2) \) each with multiplicity 1; Sorting this relation on attribute \( A \) (the tuples are indistinguishable on this attribute), can return the tuples in either order. Without loss of generality, we ensure a fully deterministic semantics (up to tuple equality) by extending the ordering on attributes \( O \), using the remaining attributes of the relation as a tiebreaker: The total order \( t <_O t' \) for tuples from a relation \( R \) is defined as \( t <_O \text{Sch}(R) = O t' \) (assuming some arbitrary order of the attributes in \( \text{Sch}(R) \)). We first introduce operators for windowed aggregation, because sorting can be defined as a special case of windowed aggregation.

4.1 Windowed Aggregation

A windowed aggregate is defined by an aggregate function, a sort order (ORDER BY), and a window bound specification. A window boundary is relative to the defining tuple, by the order-by attribute value (RANGE BETWEEN . . .) or by position (ROWS BETWEEN). In the interest of space, we will limit our discussion to row-based windows, as range-based windows are strictly simpler. A window includes every tuple within a specified interval of the defining tuple. Windowed aggregation extends each input tuple with the aggregate value computed over the tuple’s window. If a PARTITION BY clause is present, then window boundaries are evaluated within a tuple’s partition. In SQL, a single query may define a separate window for each aggregate function (SQL’s OVER clause). This can be modeled by applying multiple window operators in sequence.

Example 4 (Row-Based Windows). Consider the bag relation below and consider the windowed aggregation \( \sum(B) \) on \( A \) with bounds \([-2, 0]\) (including the two preceding tuples and the tuple itself). The window for the first duplicate of \( t_1 = (a, 5, 3) \) contains tuple \( t_1 \) with multiplicity 1, the window for the second duplicate of \( t_1 \) contains \( t_1 \) with multiplicity 2 and so on. Because each duplicate of \( t_1 \) ends up in a different window, there are three result tuples produced for \( t_1 \), each with a different \( \sum(B) \) value. Furthermore, tuples \( t_2 = (b, 3, 1) \) and \( t_3 = (b, 3, 4) \) have the same position in the sort order, demonstrating the need to use \( <_O^\text{total} \) to avoid non-determinism in what their windows are. We have \( t_2 <_O^\text{total} t_3 \) and, thus, the window for \( t_2 \) contains \( t_2 \) with multiplicity 1 and \( t_1 \) with multiplicity 2 while the window for \( t_3 \) contains \( t_1, t_2 \) and \( t_3 \) each with multiplicity 1.

\[
\begin{array}{|c|c|c|c|} \hline
A & B & C & N \\
\hline
a & 5 & 3 & 3 \\
& 3 & 1 & 1 \\
b & 3 & 4 & 1 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|c|c|} \hline
A & B & C & \sum(B) & N \\
\hline
a & 5 & 3 & 3 & 1 \\
& 5 & 3 & 10 & 1 \\
& 5 & 3 & 15 & 1 \\
b & 3 & 1 & 13 & 1 \\
b & 3 & 4 & 11 & 1 \\
\hline
\end{array}
\]

The semantics of the row-based window aggregate operator \( \omega \) is shown in Fig. 3. The parameters of \( \omega \) are partition-by attributes \( G \), order-by attributes \( O \), an aggregate function \( f(A) \) with
\[
\omega_{f(A)}^{[J_u]} \rightarrow X; G; O \end{equation}(R)(t) = \pi_{\text{Sch}(R),X}(\text{ROW}(R))
\]
\[
\text{ROW}(R)(t) = \begin{cases} 1 & \text{if } t = t' = f(\pi_A(W_{R,i}, t)) \circ i \\
\land i \in [0, R(t') - 1] & \text{otherwise} \\
0 & \text{otherwise} \\
\end{cases}
\]
\[
\text{cover}(R, t) = [\text{pos}(R, t), 0, \text{pos}(R, t, R(t) - 1)]
\]
\[
\text{bounds}(R, t, i) = [\text{pos}(R, t, i) + 1, \text{pos}(R, t, i) + u]
\]

Figure 3: Windowed Aggregation

A \( \subseteq \text{Sch}(R) \), and an interval \([l, u] \). For simplicity, we hide some arguments \((G, O, l, u) \) in the definitions and assume they passed to intermediate definitions where needed. The operator outputs a relation with schema \( \text{Sch}(R) \times X \).

The heavy lifting occurs in the definition of relation \( \text{ROW}(R) \), which “explodes” relation \( R \), adding an attribute \( i \) to replace each tuple of multiplicity \( n \) with \( n \) distinct tuples. \( \text{ROW}(R) \) computes the windowed aggregate over the window defined for the pair \((t, i)\), denoted as \( W_{R,t,i}(t') \). To construct this window, we define the multiplicity of tuple \( t' \) in the partition for tuple \( t \) (denoted as \( \text{P} \)). The range of sort positions the tuple \( t \) covers (\( \text{cover}(R, t) \)), and the range of positions in its window (\( \text{bounds}(R, t, i) \)). The multiplicity of tuple \( t' \) in the window for the \( i \)th duplicate of \( t \) is the size of the overlap between the bounds \( \text{bounds}(R, t, i) \), and the cover of \( t' \).

4.2 Sort Operator

We now define a sort operator \( \text{SORT}_{A \rightarrow R} \) which extends each row of \( R \) with an attribute \( r \) that stores the position of this row in \( R \) according to \( <_O^\text{total} \). This operator is just “syntactic sugar” as it can be expressed using windowed aggregation.

Definition 1 (Sort Operator). Consider a relation \( R \) with schema \( (A_1, \ldots, A_n) \), list of attributes \( O = (B_1, \ldots, B_m) \) where each \( B_j \) is in \( \text{Sch}(R) \). The sort operator \( \text{SORT}_{A \rightarrow R}(R) \) returns a relation with schema \( (A_1, \ldots, A_n, r) \) as defined below.

\[
\text{SORT}_{A \rightarrow R} = \pi_{\text{Sch}(R),r \leftarrow 1 \rightarrow r}(\pi_{\text{count}(1) \leftarrow r, 0}(R))
\]

Top-k queries can be expressed using the sort operator followed by a selection. For instance, the SQL query shown below can be written as \( \pi_{A,B}(\sigma_{r \leq 3}(\text{SORT}_{A \rightarrow R}(R))) \).

SELECT A, B FROM R ORDER BY A LIMIT 3;

5 AU-DB Sorting and Top-k Semantics

We now develop a bound-preserving semantics for sorting and top-k queries over AU-DBs. Recall that each tuple in an AU-DB is annotated with a triple of multiplicities and that each (range-annotated) value is likewise a triple. Elements of a range-annotated value \( c = [c_1, c_2, c_3] \) or multiplicity triple \((p_1, p_2, p_3)\) are accessed as: \( c^1 = c_1, c^{pq} = c_2, \) and \( c^3 = c_3 \). We use bold face to denote range-annotated tuples, relations, values, and databases. Both the
uncertainty of a tuple's multiplicity and the uncertainty of the values of order-by attributes create uncertainty in a tuple's position in the sort order. The former, because it determines how many duplicates of a tuple appear in the sort order which affects the position of tuples which may be larger wrt. the sort order and the latter because it affects which tuples are smaller than a tuple wrt. the sort order. As mentioned before, a top-k query is a selection over the result of a sort operator which checks that the sort position of a tuple is less than or equal to k. A bound-preserving semantics for selection was already presented in [24]. Thus, we focus on sorting and use the existing selection semantics for top-k queries.

**Comparison of Uncertain Values.** Before introducing sorting over AU-DBs, we first discuss the evaluation of \( <_O \) over tuples with uncertain values (recall that \(<_O^{total} \) is defined in terms of \( <_O \)). Per [24], a Boolean expression over range-annotated values evaluates to a bounding tuple (using the order \( \perp < \top \) where \( \perp \) denotes false and \( \top \) denotes true). The result of an evaluation of an expression \( e \) is denoted as \( [e] \). For instance, \([1/1/3] < [2/2/2] \) is \([\perp/\top/\top] \), because the expression may evaluate to false (e.g., if the first value is 3 and the second values is 2), evaluates to true in the selected-guess world, and may evaluate to true (if the \( 1^{st} \) value is 1 and the \( 2^{nd} \) value is 2). The extension of \( < \) to comparison of tuples on attributes \( O \) using \( <_O \) is shown below. For example, consider tuples \( t_1 = (1/1/3, [a/a/a]) \) and \( t_2 = ([2/2/2], [b/b/b]) \) over schema \( R(A,B) \). We have \( t_1 <_{AB} t_2 = [\perp/\top/\top] \), because \( t_2 \) could be ordered before \( t_2 \) (if \( t_1.A = 1 \), is ordered before \( t_2 \) in the selected-guess world \( (1 < 2) \), and may be ordered after \( t_2 \) (if \( t_1.A = 3 \)).

\[
[t <_O t']^1 = \exists i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, i-1\} : [t.A_j < t'.A_j]^1 \\
[t <_O t']^2 = \exists i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, i-1\} : [t.A_j < t'.A_j]^2 \\
[t <_O t']^3 = \exists i \in \{1, \ldots, n\} : [t.A_i < t'.A_i]^3 \\
\]

To simplify notation, we will use \( t <_O t' \) instead of \( [t <_O t'] \).

**Tuple Rank and Position.** To define windowed aggregation and sorting over AU-DBs, we generalize pos using the uncertain version of \( <_O \). The lowest possible position of the first duplicate of a tuple \( t \) in an AU-DB relation \( R \) is the total multiplicity of tuples \( t' \) that certainly exist \( (R(t')^1 > 0) \) and are certainly smaller than \( t \) (i.e., \([t' <_O t]^1 = \top) \). The selected-guess position of a tuple is the position of the tuple in the selected-guess world, and the greatest possible position of \( t \) is the total multiplicity of tuples that possibly exist \( (R(t')^1 > 0) \) and possibly precede \( t \) (i.e., \([t' <_O t]^1 = \top) \). The sort position of the \( i^{th} \) duplicate (with the first duplicate being 0) is computed by adding \( i \) to the position of the first duplicate.

\[
\text{pos}(R, O, t, i) = i + \sum_{(t' <_O t)} R(t')^1 \\
\text{pos}(R, O, t, i)^2 = i + \sum_{(t' <_O t)^2} R(t')^2 \\
\text{pos}(R, O, t, i)^3 = i + \sum_{(t' <_O t)^3} R(t')^3 \\
\]

**5.1 AU-DB Sorting Semantics**

To define AU-DB sorting, we split the possible duplicates of a tuple and extend the resulting tuples with a range-annotated value denoting the tuple’s (possible) positions in the sort order. The certain multiplicity of the \( i^{th} \) duplicate of a tuple \( t \) in the result is either 1 for duplicates that are guaranteed to exist \( (i < R(t')^1) \) and 0 otherwise. The selected-guess multiplicity is 1 for duplicates that do not certainly exist (in some possible world there may be less than \( i \) duplicates of the tuple), but are in the selected-guess world (the selected-guess world has \( i \) or more duplicates of the tuple). Finally, the possible multiplicity is always 1.

**Definition 2 (AU-DB Sorting Operator).** Let \( R \) be an AU-DB relation and \( O \subseteq \text{Sch}(R) \). The result of applying the sort operator \( \text{sort}_{O \rightarrow R} \) to \( R \) is defined in Fig. 4.

Every tuple in the result of sorting is constructed by extending an input tuple \( t' \) with the range of positions \( \text{pos}(R, O, t', i) \) it may occupy wrt. the sort order. The definition decomposes \( t \) into a base tuple \( t' \), and a position tuple for each duplicate of \( t \) in \( R \). We annotate all certain duplicates as certain \( (1,1,1) \), remaining selected-guess (but uncertain) duplicates as uncertain \( (0,1,1) \) and non-selected guess duplicates as possible \( (0,0,1) \).

**Example 5 (AU-DB Sorting).** Consider the AU-DB relation \( R \) shown on the left below with certain, selected guess and possible multiplicities from \( \mathbb{N}^3 \) assigned to each tuple. For values or multiplicities that are certain, we write only the certain value instead of the triple. The result of the sort relation on attributes \( A, B \) using AU-DB sorting semantics and storing the sort positions in column \( \text{pos} \) \( (\text{sort}_{A,B \rightarrow \text{pos}}(R)) \) is shown below on the right. Observe how the \( 1^{st} \) input tuple \( t_1 = (1, [1/1/3]) \) was split into two result tuples occupying adjacent sort positions. The \( 3^{rd} \) input tuple \( t_3 = ([1/1/2], 2) \) could be the \( 1^{st} \) in sort order (if its \( A \) value is 1 and the \( B \) values of the duplicates of \( t_1 \) are equal to 3) or be at the \( 3^{rd} \) position if two duplicates of \( t_1 \) exist and either \( A = 2 \) or the \( B \) values of \( t_1 \) are all \( \leq 3 \).

**5.2 Bound Preservation**

We now prove that our semantics for the sorting operator on AU-DB relations is bound preserving, i.e., given an AU-DB \( R \) that bounds an incomplete bag database \( \mathcal{R} \), the result of a sort operator \( \text{sort}_{O \rightarrow R} \) applied to \( R \) bounds the result of \( \text{sort}_{O \rightarrow \mathcal{R}} \) evaluated over \( \mathcal{R} \).

**Theorem 1 (Bound Preservation of Sorting).** Given an AU-DB relation \( R \) and incomplete bag relation \( \mathcal{R} \) such that \( \mathcal{R} \subseteq R \), and
We now introduce a bound preserving semantics for windowed aggregation over AU-DBs. We have to account for three types of uncertainty: (i) uncertain partition membership if a tuple may not exist; (ii) uncertain window membership if a tuple’s partition membership, position, or multiplicity are uncertain; and (iii) uncertain aggregation results from either preceding type of uncertainty, or if we are aggregating over uncertain values. We compute the windowed aggregation result for each input tuple in multiple steps: (i) we first use AU-DB sorting to split each input tuple into tuples whose multiplicities are at most one. This is necessary, because the aggregation function result may differ among the duplicates of a tuple (as is already the case for deterministic windowed aggregation); (ii) we then compute for each tuple t an AU-DB relation \( P_t(R) \) storing the tuples that certainly and possibly belong to the partition for that tuple; (iii) we then compute an AU-DB relation \( W_R \) encoding which tuples certainly and possibly belong to the tuple’s window; (iv) since row-based windows contain a fixed number of tuples, we then determine from the tuples that possibly belong to the window, the subset that together with the tuples that certainly belong to the window (these tuples will be in the window in every possible world) minimizes / maximizes the aggregation function result. This then enables us to bound the aggregation result for each input tuple from below and above. For instance, for a row-based window \([-2, 0]\), we know that the window for a tuple t will never contain more than 3 tuples. If we know that two tuples certainly belong to the window, then at most one additional possible tuple can belong to the window.

### 6.1 Windowed Aggregation Semantics

As before, we omit windowed aggregation parameters \((G, O, l, u, f, A)\) from the arguments of intermediate constructs and assume they are passed along where needed.

#### Partitions

We start by defining AU-DB relation \( P_t(R) \) which encodes the multiplicity of tuple t’ in the partition for t based on partition-by attributes G. This is achieved using selection, comparing a tuple’s values in G with the values of t.G on equality. AU-DB selection sets the certainty of the selected-guess, or possible multiplicity) of a tuple to 0 if the tuple possibly (in the selected-guess world, or certainly) does not fulfill the selection condition.

\[
P_t(R) = \left[ \sigma_{G=\tau.G}(R) \right]
\]

#### Certain and Possible Windows

We need to be able to reason about which tuples (and with which multiplicity) belong certainly to the window for a tuple and which tuples (with which multiplicity) could possibly belong to a window. For a tuple t, we model the window’s tuples as an AU-DB relation \( W_R \) where a tuple’s lower bound multiplicity encodes the number of duplicates of the tuple that are certainty in the window, the selected-guess multiplicity encodes the multiplicity of the tuple in the selected-guess world, and the upper bound encodes the largest possible multiplicity with which the tuple may occur in the window minus the certain multiplicity. In the remainder of this paper we omit the definition of the select-guess, because it can be computed using the deterministic semantics for windowed aggregation. For completeness, we include it in the extended version of this paper [22]. We formally define \( W_R \) in Fig. 6. Recall that in the first step we used sort to split the duplicates of each tuple into tuples with multiplicity upper bound of 1. Thus, the windows we are constructing here are for tuples instead of for individual duplicates of a tuple. A tuple t’ is guaranteed to belong to the window for of a tuple t with a multiplicity of \( n = R(t')^+ \) (the number of duplicates of the tuple that certainly exist) if the tuple certainly belongs to the partition for t and all possible positions that these n duplicates of the tuple occupy in the sort order are guaranteed to be contained in the smallest possible interval of sort positions contained in the bounds of the window for t. Tuples t’ possibly belong to the window of t if any of its possible positions falls within the interval of all possible positions of t. As an example consider Fig. 5 which shows the sort positions that certainly (red) and possibly (green) belong to tuple t’s window (window bounds \([-1, 4]\)). For any window \([l, u]\), sort positions certainly covered by the window start from latest possible starting position for t’s window which is \( t.l + u \) \((4 + 4 = 8 \text{ in our example})\) and end at the earliest possible upper bound for the window which is \( t.l + u \) \((4 + 4 = 8 \text{ in our example})\). Furthermore, Fig. 5 shows the membership of three tuples in the window. Tuples t1 does certainly not belong to the window, because none of its possible sort positions are in the window’s set of possible sort positions, t2 does certainly belong to the window, because all of its possible sort positions are in the set of positions certainly in the window. Finally, t3 possibly belongs to the window, because some of its sort positions are in the set of positions possibly covered by the window.

#### Combining and Filtering Certain and Possible Windows

As mentioned above, row-based windows contain a fixed maximal number of tuples based on their bounds. We use \( \text{size}(\{l, u\}) \) to denote the size of a window with bounds \([l, u]\), i.e., \( \text{size}(\{l, u\}) = (u - l) + 1 \). This limit on the number of tuples in a window should be taken into account when computing bounds on the result of an aggregation function. For that, we combine the tuples certainly in the window (say there are \( m \) such tuples) with a selected bag of up to \( \text{size}(\{l, u\}) - m \) rows possibly in the window that minimizes (for the lower aggregation result bound) or maximizes (for the upper
aggregation result bound) the aggregation function result for an input tuple. Let us use \( \text{posn}(R, t) \) to denote size \((l, u) - m:\)

\[
\text{posn}(R, t) = \text{size}(\{l, u\}) - \sum \mathcal{W}_{R_1}(t')
\]

Which bag of up to \( \text{posn}(R, t) \) tuples minimizes / maximizes the aggregation result depends on what aggregation function is applied. For \( \text{sum} \), the up to \( \text{posn}(R, t) \) rows with the smallest negative values are included in the lower bound and the up to \( \text{posn}(R, t) \) rows with the greatest positive values for the upper bound. For \( \text{count} \) no additional row are included for the lower bound and up to \( \text{posn}(R, t) \) rows for the upper bound.

For each tuple \( t \), we define AU-DB relation \( \mathcal{R} \mathcal{W}_{R_1} \) where each tuple’s lower/upper bound multiplicities encode the multiplicity of this tuple contributing to the lower and upper bound aggregation result, respectively. We only show the definition for \( \text{sum} \), the definitions for other aggregation functions are similar. In the definition, we make use \( R^\downarrow \) and \( R^\uparrow \):

\[
R^\downarrow(t) = R(t)^\downarrow \quad \quad R^\uparrow(t) = R(t)^\uparrow
\]

Note that \( R^\downarrow \) and \( R^\uparrow \) are \((l, u)\)-relations over range-annotated tuples. Furthermore, we define \( \text{min}-k(R, t, A) \) (and \( \text{max}-k(R, t, A) \)) that are computed by restricting \( \mathcal{W}_{R_1} \) to the tuples with the smallest negative values (largest positive values) as lower (upper) bounds on attribute \( A \) that could contribute to the aggregation, keeping tuples with a total multiplicity of up to \( \text{posn}(R, t) \). Note that the deterministic conditions / expressions in the definition of \( \text{min}-k(R, t, A) \) (and \( \text{max}-k(R, t, A) \)) are well-defined, because single values are extracted from all range-annotated values. For \( \text{max} \) (resp., \( \text{min} \)) and similar idempotent aggregates, it suffices to know the greatest (resp., least) value possibly in the window.

\[
\mathcal{R} \mathcal{W}_{R_1}(t') = \mathcal{W}_{R_1}(t') + \min-k(R, t, A)(t')
\]

\[
\mathcal{R} \mathcal{W}_{R_1}(t') = \mathcal{W}_{R_1}(t') + \max-k(R, t, A)(t')
\]

\[
\min-k(R, t, A) = \sigma_{\text{posn}(R, t)}(\text{sort}_{A \rightarrow t}(\sigma_{A < 0}(\mathcal{W}_{R_1})))
\]

\[
\max-k(R, t, A) = \sigma_{\text{posn}(R, t)}(\text{sort}_{A \rightarrow t}(\sigma_{A > 0}(\mathcal{W}_{R_1})))
\]

Windowed Aggregation. Using the filtered combined windows we are ready to define row-based windowed aggregation over AU-DBs. To compute aggregation results, we utilize the operation \( \oplus_f \) defined in [24] for aggregation function \( f \) that combines the range-annotated aggregation attribute value of a tuple with the tuple’s multiplicity bounds. For instance, for \( \text{sum} \), \( \oplus_{\text{sum}} \) is multiplication, e.g., if a tuple with \( A \) value \( [10,20] \) has multiplicity \((1,2,3)\) it contributes \( [10,40,90] \) to the sum. Here, \( \oplus \) denotes the application of the aggregation function over a set of elements (e.g., \( \sum \) for \( \text{sum} \)). Note that, as explained above, the purpose of \( \text{expand}(R) \) is to split a tuple with \( n \) possible duplicates into \( n \) tuples with a multiplicity of 1. Furthermore, note that the bounds on the aggregation result may be the same for the \( i^{th} \) and \( j^{th} \) duplicate of a tuple. To deal with that we apply a final projection to merge such duplicate result tuples.

**Definition 3 (Row-based Windowed Aggregation).** Let \( R \) be an AU-DB relation. We define window operator \( \omega_{\{l,u\}}^f(A) \rightarrow \times; G; O \) as:

\[
\omega_{\{l,u\}}^f(A) \rightarrow \times; G; O(R)(t) = \pi_{\text{Sch}(R),X}(\mathcal{R} \mathcal{W}_{R_1}(t \circ \text{aggres}) = \text{expand}(R)(t)
\]

\[
\text{expand}(R) = \sigma_{\text{Sch}(R),\text{rad}}(\text{sort}_{\text{Sch}(R)}\rightarrow \text{rad}(R))
\]

**Example 6 (AU-DB Windowed Aggregation).** Consider the AU-DB relation \( R \) shown below and query \( \sigma_{\{l,u\}}^f(A \rightarrow \times; G; O) \) i.e., windowed aggregation partitioning by \( A \), ordering \( B \), and computing \( \text{sum}(C) \) over windows including \( 1 \) preceding and the current row. For convenience we show an identifier for each tuple on the left. As mentioned above, we first expand each tuple with a possible multiplicity larger then one using sorting. Consider tuple \( t_3 \). Both \( t_1 \) and \( t_2 \) may belong to the same partition as \( t_3 \) as their \( A \) value ranges overlap. There is no tuple that certainly belongs to the same partition as \( t_3 \). Thus, only tuple \( t_3 \) itself will certainly belong to the window. To compute the bounds on the aggregation result we first determine which tuples (in the expansion created through sorting) may belong to the window for \( t_3 \). These are the two tuples corresponding to the duplicates of \( t_3 \), because these tuples may belong to the partition for \( t_3 \) and their possible sort positions \((0,0,1)\) and \((1,1,2)\) overlap with the sort positions possibly covered by the window for \( t_3 \) \((0,1,2)\). Since the size of the window is \( 2 \) tuples, the bounds on the sum are computed using the lower / upper bound on the \( C \) value of \( t_3 \) \((1,2,4,5)\) and no additional tuple from the possible window (because the \( C \) value of \( t_3 \) is positive) for the lower bound and the largest possible \( C \) value of one copy (we can only fit one additional tuple into the window) of \( t_3 \) \((7)\) for the upper bound. Thus, we get the aggregation result \([21/11/12]\) as shown below.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( C )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( A )</td>
</tr>
<tr>
<td>( t_1 )</td>
<td>( 1 )</td>
<td>([1/1/3])</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( [2/3/1] )</td>
<td>( 15 )</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>([1/1/2])</td>
<td>( 2 )</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>([2/3/3])</td>
<td>( 15 )</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( A )</td>
</tr>
<tr>
<td>( r_1 )</td>
<td>( 1 )</td>
<td>([1/1/3])</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>( 1 )</td>
<td>([1/1/3])</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>([1/1/2])</td>
<td>( 4 )</td>
</tr>
<tr>
<td>( r_4 )</td>
<td>([2/3/3])</td>
<td>( 15 )</td>
</tr>
</tbody>
</table>

**6.2 Bound Preservation**

We now prove this semantics for group-based and row-based windowed aggregation over AU-DBs to be bound preserving.

**Theorem 2 (Bound Preservation for Windowed Aggregation).** Consider an AU-DB relation \( R \) and incomplete bag relation...
such that $\mathcal{R} \subseteq \mathcal{R}$, and $\mathcal{O} \subseteq \text{Sch}(\mathcal{R})$. For any row-based windowed aggregation $\omega_{f(A)}^{[\mathcal{U}]}(\mathcal{M}) \rightarrow \mathcal{X}$, we have:

$$\omega_{f(A)}^{[\mathcal{U}]}(\mathcal{M}) \rightarrow \mathcal{X}, \mathcal{G}; \omega_{f(A) \rightarrow X}^{[\mathcal{U}]}(\mathcal{M}) \rightarrow \mathcal{X}, \mathcal{G}$$

Proof Sketch: As in the proof for sorting over AU-DBs, we consider WLOG one of the possible worlds $\mathcal{R} \subseteq \mathcal{R}$ and a tuple matching $\mathcal{M}$ based on which $\mathcal{R}$ is bounding $\mathcal{R}$. We then construct a tuple matching $\mathcal{M}'$ for the output of windowed aggregation. In the proof, we utilize the fact that windowed aggregation produces one output tuple $t$ for each input tuple $t'$ such that $t$ extends the input tuple $t'$ with the aggregation result for $t'$'s window and has the same multiplicity as the input tuple $t'$. Thus, we only need to show that the bounds on the aggregation function result bound the values in the result for the possible world $\mathcal{R}$ and that tuples with multiplicity $n$ are split into $n$ output tuples with multiplicity 1.

7 NATIVE ALGORITHMS

We now introduce optimized algorithms for ranking and windowed aggregation over AU-DBs that are more efficient than their SQL counterparts presented in [22]. Through a connected heap data structure, these algorithms leverage the fact that the lower and upper position bounds are typically close approximations of one another to avoid performing multiple passes over the data. We assume a physical encoding of an AU-DB relation $\mathcal{R}$ as a classical relation [24] where each range-annotated value of an attribute $A$ is stored as three attributes $A^1$, $A^{bg}$, and $A^\dagger$. In this encoding, attributes $t.#^1$, $t.#^{bg}$, and $t.#^\dagger$ store the tuple’s multiplicity bounds.

7.1 Non-deterministic Sort, Top-k

Algorithm 1 sorts an input AU-DB $\mathcal{R}$. The algorithm assigns to each tuple its position $\tau$ given as lower and upper bounds: $t.\tau^1$, $t.\tau^\dagger$, respectively. Given a parameter $k$, the algorithm can also be used to find the top-k elements; otherwise we set $k = |\mathcal{R}|$ (the maximal possible size of the input relation). Algorithm 1 takes as input the relational encoding of an AU-DB relation $\mathcal{R}$ sorted on $O^1$, the lower-bound of the sort order attributes. Recall from Equation (1) that to determine a lower bound on the sort position of a tuple $t$ we have to sum up the smallest multiplicity of tuples $s$ that are certainly sorted before $t$, i.e., where $s.\tau^1 \leq \mathcal{O}^1$ (for fast eviction), sorted on $O^1$. Since $s.\tau^1 \leq \mathcal{O}^1$ holds for any tuple, we know that these tuples are visited by Algorithm 1 before $t$. We store tuples in a min-heap $\text{todo}$ sorted on $O^1$ and maintain a variable $\text{rank}^1$ to store the current lower bound.

For every incoming tuple $t$, we first determine all tuples $s$ from $\text{todo}$ certainly preceding $t$ ($s.\mathcal{O}^1 < t.\mathcal{O}^1$) and update $\text{rank}^1$ with their multiplicity. Since $t$ is the first tuple certainly ranked after any such tuple $s$ and all tuples following $t$ will also certainly ranked after $s$, we can now determine the upper bound on $s$'s position. Based on Equation (3) this is the sum of the maximal multiplicity of all tuples that may precede $s$. These are all tuples $s$ such that $s.\mathcal{O}^1 \geq u.\mathcal{O}^1$, i.e., all tuples we have processed so far. We store the sum of the maximal multiplicity of these tuples in a variable $\text{rank}^1$ which is updated for every incoming tuple. We use a function $\text{emit}$ to compute $s$'s upper bound sort position, adapt $s.#^1$ (for a top-k query, $s$ may not exist in the result if its position may be larger than $k$), add $s$ to the result, and adapt $\text{rank}^1$ (all tuples processed in the following are certainly ranked higher than $s$). Function $\text{split}$ splits a tuple with $t.# > 1$ into multiple tuples as required by Def. 2. If we are only interested in the top-k results, then we can stop processing the input once $\text{rank}^1$ is larger than $k$, because all following tuples will be certainly not in the top-k. Once all inputs have been processed, the heap may still contain tuples whose relative sort position wrt. to each other is uncertain. We flush these tuples at the end. Algorithm 1’s worst-case runtime is $O(n \cdot \log n)$ and worst-case memory requirement is $O(n)$ for $n = |\mathcal{R}|$ (see [22]).

7.2 Connected Heaps

In our algorithm for windowed aggregation that we will present in Sec. 7.3, we need to maintain the tuples possibly in a window ordered increasingly on $\tau^1$ (for fast eviction), sorted on $A^1$ to compute $\min_{\mathcal{R}}(k, \mathcal{R}, A)$, and sorted decreasingly on $A^\dagger$ to compute $\max_{\mathcal{R}}(k, \mathcal{R}, A)$. We could use separate heaps to access the smallest element(s) wrt. to any of these orders efficiently. However, if a tuple needs to be deleted, the tuple will likely not be the root element in all heaps which means we have to remove non-root elements from some heaps which is inefficient (linear in the heap size). Of course it would be possible to utilize other data structures that maintain order such as balanced binary trees. However, such data structures do not achieve the $O(1)$ lookup performance for the smallest element that heaps provide. Instead, we introduce a simple, yet effective, data structure we refer to as a connected heap.

A connected heap is comprised of $H$ heaps which store pointers to a shared set of records. Each heap has its own sort order. A record stored in a connected heap consists of a tuple (the payload) and $H$ backwards pointers that point to the nodes of the individual heaps storing this tuple. These backward pointers enable efficient deletion ($O(H \cdot \log n)$) of a tuple from all heaps when it is popped as the root of one of the component heaps. In [22] we explain how the standard sift-up and sift-down heap operations are used to restore the heap property in $O(\log n)$ when removing a non-root element from a component heap. When a tuple is inserted into a connected heap, it is inserted into each component heap in $O(\log n)$
Without loss of generality, we focus on window specifications with connected heaps. Even for small databases (10k tuples) and a small fraction of uncertain order-by values (1%), connected heaps outperform heaps by a factor of ~ 2. Larger databases / more uncertain data result in larger heaps and, thus, better performance.

**Example 7 (Connected heap).** Consider the connected heap shown below on the left storing tuples $t_1 = (1,3)$, $t_2 = (2,6)$, $t_3 = (3,2)$, and $t_4 = (4,1)$. Heap $h_1$ ($h_2$) is sorted on the first (second) attribute. Calling pop() on $h_1$ removes $t_1$ from $h_1$. Using the backwards pointer from $t_1$ to the corresponding node in $h_2$ (shown in red), we also remove $t_1$ from $h_2$. The node pointing to $t_1$ from $h_2$ is replaced with the right most leaf node of $h_2$ (pointing to $t_2$ in $h_2$). In this case the heap property is not violated and, thus, no sift-down / up is required.

### 7.3 Ranged Windowed Aggregation

Without loss of generality, we focus on window specifications with only a `ROWS PRECEDING` clause; a `FOLLOWING` clause can be simulated by offsetting the window, i.e., a window bound of $[-N, 0]$. Algorithm 2 uses a function `compBounds` to compute the bounds on the aggregation function result based on the certain and possible content of a window. We present the definition of this function for several aggregation functions in [22]. Algorithm 2 follows a sweeping pattern similar to Algorithm 1 to compute the windowed aggregate in a single pass over the data which has been preprocessed by applying `SORTORDER(R)` and then has been sorted on $r^i$. The algorithm uses a minheap `openw` which is sorted on $r^i$ to store tuples for which have not been seen yet all tuples that could belong to their window. Additionally, the algorithm maintains the following data structures: cert is a map from a sort position $i$ to a tree storing tuples $t$ that certainly exist and for which $t.r^i = i$ sorted on $r^i$. This data structure is used to determine which tuples certainly belong to the window of a tuple; `(poss, pagg^i, pagg^j)` is a connected minheap where `poss`, `pagg^i`, and `pagg^j` are sorted on $r^i$, $A^i$, $-A^i$, respectively. This connected heap stores tuples possibly in a window. The different sort orders are needed to compute bounds on the aggregation function result for a window efficiently (we will expand on this later). Finally, we maintain a watermark `c-rank^i` for the lower bound position of the certain part of windows.

Algorithm 2 first inserts each incoming tuple into `openw` (Line 7). If the tuple certainly exists, it is inserted into the tree of certain tuples whose lower bound position is $t.r^i$. Note that each of these trees is sorted on $r^i$ which will be relevant later. Next the algorithm determines for which tuples from `openw`, their windows have been fully observed. These are all tuples $s$ which are certainly ordered before the tuple $t$ we are processing in this iteration ($s.r^i < t.r^i$). To see why this is the case, first observe that (i) we are processing input tuples in increasing order of $r^i$ and (ii) tuples are "finalized" by computing the aggregation bounds in monotonically increasing order of $r^i$. Given that we are using a window bound $[-N, 0]$, all tuples $s$ that could possibly belong to the window of a tuple $t$ have to have $s.r^i \leq t.r^i$. Based on these observations, once we processed a tuple $t$ with $t.r^i > s.r^i$ for a tuple $s$ in `openw`, we know that no tuples that we will process in the future can belong to the window for $s$. In Line 11 we iteratively pop such tuples from `openw`. For each such tuple $s$ we evict tuples from `cert` and update the high watermark `c-rank^i` (Line 12). Recall that for a tuple $u$ to certainly belong to the window for $s$ we have to have $s.r^i - N \geq u.r^i$. Thus, we update `c-rank^i` to $s.r^i - N$ and evict from `cert` all trees storing tuples for sort positions smaller than $s.r^i - N$. Afterwards, we compute the bounds on the aggregation result for $s$ using `cert` and `poss` (we will describe this step in more detail in the following). Finally, we evict tuples from `poss` (and, thus, also `pagg^i` and `pagg^j`) which cannot belong to any windows we will close in the future. These are tuples which are certainly ordered before the lowest possible position in the window of $s$, i.e., tuples $u$ with $u.r^i < s.r^i - N$ (see Fig. 5). Evicting tuples from `poss` based on the tuple for which we are currently computing the aggregation result bounds is safe because we are emitting tuples in increasing order of $r^i$, i.e., for all tuples $u$ emitted after $s$ we have $u.r^i > s.r^i$. Fig. 7 shows an example state for the algorithm when tuple $s$ is about to be emitted. Tuples fully included in the red region ($t_3$ and $t_5$) are currently in `cert[i]` for sort positions certainly in the window for $s$. Tuples with sort position ranges overlapping with the green region are in the possible window (these tuples are stored in `poss`). Tuples like $t_4$ with upper-bound position higher than $s$ will be popped and processed after $s$. Once all input tuples have been processed, we have to close the windows for all tuples remaining in `openw`.

```plaintext
Input: $f, X, O, N, A$, `SORTORDER(R)` sorted on $r^i$
1. $openw \leftarrow \text{minheap}(r^i)$ // tuples with open windows
2. $cert \leftarrow \text{Map}(\text{int}, \text{Tree}(r^i))$ // certain window members by pos.
3. $(\text{poss}, \text{pagg}^i, \text{pagg}^j) \leftarrow \text{connected-minheap}(r^i, A^i, -A^i)$
4. $c-rank^i \leftarrow 0$ // watermark for certain window
5. $res \leftarrow \emptyset$
6. for $t \in R$ do
7.  $openw\text{.insert}(t)$
8.  if $t.r^i > 0$ then // insert into potential certain window
9.    $cert[t.r^i].\text{insert}(t)$
10. while $openw.ppeak().r^i < t.r^i$ do // close windows
11.    $s \leftarrow openw.pop()$
12.    while $cert[c-rank^i] < s.r^i - N$ do // evict certain win.
13.      $cert[c-rank^i] = \text{null}$
14.      $c-rank^i ++$
15.    $s.X \leftarrow \text{compBounds}(f, s, cert, poss)$ // compute agg.
16.    while $\text{poss}.ppeak().r^i < s.r^i - N$ do // evict poss. win.
17.      $\text{poss}.pop()$
18.    $res \leftarrow res \cup \{s\}$
19.    $\text{poss}.\text{insert}(t)$ // insert into poss. win.
20. end for
21. Algorithm 2: Aggregate $f(A) \rightarrow X$, sort on $O$, $N$ preceding
```
process is the same as emitting tuples before we have processed all inputs and, thus, is omitted from Algorithm 2.

Algorithm 2 uses function compBounds to compute the bounds on the aggregation function result for a tuple \( t \) using cer \( t \), pagg\(^1\) and pagg\(^2\) following the definition from Sec. 6.1. First, we fetch all tuples that are certainly in the window from cer \( t \) based on the sort positions that certainly belong to the window for \( t \) (\( \{t.r^1 \cdot N, t.r^1\} \)) and aggregate their \( A \) bounds. Afterwards, we use pagg\(^1\) and pagg\(^2\) to efficiently fetch up possn\( (R, t) \) tuples possibly in the window for \( t \) to calculate the final bounds based on max-k and min-k. The worst-case runtime of the algorithm is \( O(N \cdot n \cdot \log n) \). As mentioned before, the detailed algorithm and further explanations are presented in [22].

8 EXPERIMENTS
We evaluate the efficiency of our rewrite-based approach and the native implementation of the algorithms from Sec. 7 in Postgres and also evaluate the accuracy of the approximations they produce.

Compared Algorithms. We compare against several baselines: Det evaluates queries deterministically ignoring uncertainty in the data. We present these results to show the overhead of the different incomplete query evaluation semantics wrt. deterministic query evaluation; MCDB [34] evaluates queries over a given number of possible worlds sampled from the input incomplete database using deterministic query evaluation. MCDB10 and MCDB20 are MCDB with 10 and 20 sampled worlds, respectively. For MCDB, we treat window for windowed aggregation. Attribute values are uniform randomly distributed. Except where noted, we default to 50k rows and 5% uncertainty with a maximum 1k attribute range on uncertain values.

8.1 Microbenchmarks on Synthetic Data
To evaluate how specific characteristics of the data affect our system’s performance and accuracy, we generated synthetic data consisting of a single table with 2 attributes for sorting and 3 attributes for windowed aggregation. Attribute values are uniform randomly

### Accuracy
Fig. 9 shows the error of the bounds generated by Imp (Rewr produces identical outputs) and MCDB. Recall that Imp is guaranteed to over-approximate the correct bounds, while MCDB is guaranteed to under-approximate the bounds, because it does not compute all possible results. We measure the size of the bounds relative to the size of the correct bound (as computed by Symb and PT-k), and then take the average over all normalized bound sizes. In all cases our approach produces bounds that are closer to the exact bounds than MCDB (~30% over-approximation versus ~70% under-approximation in the worst case). We further note that an over-approximation of possible answers is often preferable to an under-approximation because no possible results will be missed.

8.1.2 Windowed Aggregation. Scaling Data Size. Fig. 12 shows the runtime of windowed aggregation when varying dataset size. We compare two variants of our rewrite-based approach which uses a range overlap join to determine which tuples could possibly belong to a window. Rewr(Index) uses a range index supported by Postgres. We show index creation time and query time separately. We exclude Symb, because for more than 1k tuples, Z3 exceeds the maximal allowable call stack depth and crashes. The performance
We evaluate our approach on real datasets (Iceberg [3], Chicago crime data [4], and Medicare provider data [1]) using realistic sorting and windowed aggregation queries [2]. To prepare the datasets, we perform data cleaning methods (entity resolution and missing value imputation) that output a AU-DB encoding of the space of possible repairs. Fig. 14 shows the performance of real queries on these datasets reporting basic statistics (uncertainty and #rows).

For sorting and top-k queries that contain aggregation which is common in real use-cases, we only measure the performance of Imp is roughly on par with MCDB10. Rewr(index) is almost as fast as MCDB20, but is 5 \times slower than Imp.

Varying window spec, Ranges, and Rate. Fig. 13 shows the runtime of windowed aggregation varying the value ranges of uncertain attribute (on all columns), percentage of uncertain tuples, and window size. For Imp (Fig. 13a) we use a query without partition-by. We also compare runtime of our rewriting based approach (Fig. 13b) using both partition-by and order-by on 8k rows. Imp exhibits similar runtime to MCDB10 and outperforms MCDB20. Rewr is slower than MCDB by several magnitudes due to the range-overlap join.

8.2 Real World Datasets

We evaluate our approach on real datasets (Iceberg [3], Chicago crime data [4], and Medicare provider data [1]) using realistic sorting and windowed aggregation queries [2]. To prepare the datasets, we perform data cleaning methods (entity resolution and missing value imputation) that output a AU-DB encoding of the space of possible repairs. Fig. 14 shows the performance of real queries on these datasets reporting basic statistics (uncertainty and #rows).

For sorting and top-k queries that contain aggregation which is common in real use-cases, we only measure the performance of Imp is roughly on par with MCDB10. Rewr(index) is almost as fast as MCDB20, but is 5 \times slower than Imp.

Varying window spec, Ranges, and Rate. Fig. 13 shows the runtime of windowed aggregation varying the value ranges of uncertain attribute (on all columns), percentage of uncertain tuples, and window size. For Imp (Fig. 13a) we use a query without partition-by. We also compare runtime of our rewriting based approach (Fig. 13b) using both partition-by and order-by on 8k rows. Imp exhibits similar runtime to MCDB10 and outperforms MCDB20. Rewr is slower than MCDB by several magnitudes due to the range-overlap join.

9 CONCLUSIONS AND FUTURE WORK

In this work, we present an efficient approach for under-approximating certain answers and over-approximating possible answers for top-k, sorting, and windowed aggregation queries over incomplete databases. Our approach based on AU-DBs [24] is unique in that it supports windowed aggregation, is also closed under full relational algebra with aggregation, and is implemented as efficient one-pass algorithms in Postgres. We significantly outperform existing algorithms for ranking uncertain data and our approach is applicable to more expressive queries and bounds all certain and possible answers. In future work, we plan to extend our approach to deal more expressive classes of queries, e.g., recursive and fix-point computations as used in ML model training, and will investigate index structures for AU-DBs to further improve performance.