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

Su Feng
Illinois Institute of Technology
sfeng14@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.

1 INTRODUCTION

Many application domains need to deal with uncertainty arising from data entry/extraction errors [36, 53], data lost because of node failures [40], ambiguous data integration [7, 31, 48], heuristic data wrangling [13, 21, 62], and bias in machine learning training data [26, 52]. Incomplete and probabilistic databases [18, 57] 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* [57] 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, 42, 50, 56]. However, the resulting semantics often lacks *closure*. That is, composing such operators with other operators typically requires a complete rethinking of the entire system [54], 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 aggregates [24] ($\mathcal{RA}^{agg}$). That is, if an AU-DB D bounds a set of possible worlds, the result of any $\mathcal{RA}^{agg}$ 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 $\mathcal{RA}^{agg}$, 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 factor for our choice to extend this model in this work.

When sorting uncertain attribute values, the possible order-by attribute values of two tuples t_1 and t_2 may overlap, which leads to multiple possible sort orders. Supporting order-based operators over AU-DBs requires encoding multiple possible 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, crucially 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 this paper 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. When combined with existing AU-DB rewrites [23, 24], any $\mathcal{RA}^{agg}$ 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: (iii) 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 over 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 its pairing. 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 [49], 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 a variety of reasons, including extraction errors (e.g., D_3 includes term 5) or missing information (e.g., only preliminary data is available for the 4th term in D_1). The task of finding the two terms with the most sales is semantically ambiguous for uncertain data. Several attempts to define semantics include (i) U-top [56] (Fig. 1c), which returns the most likely ranked order; (ii) U-rank [56] (Fig. 1c), which returns the most likely tuple at each position (term 4 is more likely than any other value for both the 1st and 2nd position); or (iii) Probabilistic threshold queries (PT-k) [32, 64], which 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 resulting data is difficult, preventing its use for critical applications, e.g., in the medical, engineering, 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 factors motivate our choice of the AU-DB data model. First, the data model naturally encodes query result reliability. By providing each attribute value (and tuple multiplicity) as a range, users can quickly assess the precision of each answer. Second, the data model is complete: the full set of possible answers is represented. Finally, the model admits a closed, efficiently computable, and bounds-preserving semantics for $\mathcal{RA}^{agg}$.

Example 2 (AU-DB top-2 query). *Fig. 1f (left) shows an AU-DB, which uses triples, consisting of a lower bound, a selected-guess value (defined shortly), and an upper bound to bound the value range of an attribute (**Term**, **Sales**) and the multiplicity of a tuple ($\mathbb{N}^3$). The AU-DB* bounds *all of the possible worlds of our running example. Intuitively, each world's tuples fit into the ranges defined by the AU-DB. The selected-guess values encode one distinguished world (here, D_1) — supplementing the bounds with an educated guess about which possible world correctly reflects the real world [1], providing backwards compatibility with existing systems, and a convenient reference point for users [14, 39]. Fig. 1f (right) shows the result of computing the top-2 answers sorted on* term. *The rows marked in grey encode all tuples that could exist in the top-2 result in some possible world. For example, the tuples* $(3, 4)$ *(D_1),* $(3, 7)$ *(D_2), and* $(5, 7)$ *(D_3) are all encoded by the AU-DB tuple* $([3/3/5], [4/7/7]) \rightarrow (1, 1, 1)$. *Results with a row multiplicity range of (0,0,0) 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 admit additional worlds with 5 sales in term 4.*

[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.

Implementing windowed aggregation requires determining the (uncertain) membership of each window, 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 implemented this reasoning in SQL, albeit at the cost of range self-joins on the relation (this *rewrite method* is discussed in detail in [22] and evaluated 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 `ROWS BETWEEN 3 PRECEDING AND CURRENT ROW`. In the deterministic case, with each new window one row enters the window and one row leaves. Sum-based aggregates (`sum`, `count`, `average`) can leverage commutativity and associativity of addition, i.e., updating the window requires only constant time. Similar techniques [8] can maintain of `min`/`max` aggregates in time logarithmic in the window size.

Non-determinism in the row position makes such resource sharing problematic. First, tuples with non-deterministic positions do not necessarily leave the window in FIFO order; We need iteration over tuples sorted on both the upper- and lower-bounds of their position. Second, the number of tuples that could *possibly* belong to the window may be significantly larger than the window size. Considering all possible rows for a k-row window (using the naive AU-DB aggregation operator [24]) results in a looser bound than if only subsets of size k are considered. For that, we need access to rows possibly in a window sorted on the bounds of the aggregation attribute values (e.g., to find the k-subset with the minimal/maximal sum) in both decreasing order of their upper bound and increasing order of their lower bound. Furthermore, we have to separate maintain tuples that certainly belong to a window (which must contribute to both bounds). To efficiently maintain sets of tuples such that they can be accessed in several sort orders efficiently, we develop a new data structure which we refer to as a *connected heap*. A connected heap is a set of heaps where an element popped from one heap can be efficiently ($O(\log n)$) removed from the other heaps even if their sort orders differ from the heap we popped the element from. This data structure allows us to efficiently maintain sufficient state for computing AU-DB results for windowed aggregation. In preliminary experiments, we demonstrated that, connected heaps significantly outperform a solution based classical heaps.

Example 3 (Windowed Aggregation). *Consider the following windowed aggregation query:*

```
SELECT *, sum(Sales) OVER (ORDER BY term ASC
BETWEEN CURRENT ROW AND 1 FOLLOWING) as sum FROM R;
```

Fig. 1g shows the result of this query over our running example AU-DB. The column Sum *bounds all possible windowed aggregation results for each AU-DB tuple and the entire AU-DB relation bounds the windowed aggregation result for all possible worlds. Notice that AU-DBs ignore correlations which causes an over-approximation of ranges in the result. For example, term 1 has a maximum aggregation result value of 6 according to the AU-DB representation but the maximum possible aggregation value across all possible world is 5.*

D_1	Term	Sales
	1	2
	2	3
	3	7
	4	4

Term	Sales	Sum
1	2	5
2	3	10
3	7	11
4	4	4

D_2	Term	Sales
	1	3
	2	2
	3	4
	4	6

Term	Sales	Sum
1	3	5
2	2	6
3	4	10
4	6	6

D_3	Term	Sales
	1	2
	2	2
	5	4
	4	7

Term	Sales	Sum
1	2	4
2	2	9
5	4	4
4	7	11

(a) An uncertain sales database with three possible worlds (with probability .4, .3 and .3 respectively) with top-2 highest selling term in each world high-lighted and the result of the rolling sum of sales for the current and next term.

Term
4
3

(b) U-Top Top-k result

Term
4
4

(c) U-Rank query result

Term
3
4
5

(d) PT(0) query result (possible answers)

Term
4

(e) PT(1) Top-k result (certain answers)

Term	Sales	$\mathbb{N}^3$	Term	Sales	Position	$\mathbb{N}^3$
1	[2/2/3]	(1,1,1)	1	[2/2/3]	[2/3/3]	(0,0,0)
2	[2/3/3]	(1,1,1)	2	[2/3/3]	[2/2/3]	(0,0,0)
[3/3/5]	[4/7/7]	(1,1,1)	[3/3/5]	[4/7/7]	[0/0/1]	(1,1,1)
4	[4/4/7]	(1,1,1)	4	[4/4/7]	[0/1/1]	(1,1,1)

(f) AU-DB bounding the worlds and top-2 result produced by our approach

Term	Sales	Sum	$\mathbb{N}^3$
1	[2/2/3]	[4/5/6]	(1,1,1)
2	[2/3/3]	[6/10/10]	(1,1,1)
[3/3/5]	[4/7/7]	[4/11/14]	(1,1,1)
4	[4/4/7]	[4/4/14]	(1,1,1)

(g) AU-DB windowed aggregation result produced by our approach

Figure 1: Ranking, Top-k, and Window Queries over an Incomplete (Probabilistic) Database. We get different results for the various semantics proposed in related work. Our approach stands out in that it bounds both certain and possible answers and is closed not just under these specific query types, but also $\mathcal{RA}^{agg}$.

2 RELATED WORK

We build on prior research in incomplete and probabilistic databases, uncertain aggregation, uncertain top-k, uncertain sorting, and temporal databases.

Probabilistic/Incomplete databases. Certain answer semantics [6, 28, 29, 33, 44, 45] only returns answers that are guaranteed to be correct. Computing certain answers is coNP-complete in data-complexity [6, 33]. However, under-approximations [17, 23, 28, 29, 44, 51] can be computed in PTIME. AU-DBs [24] build on the selected-guess and lower bounds-based approach of [23], adding an upper bound on possible answers and attribute-level uncertainty with ranges to support aggregation. MCDB [34] and Pip [37] sample from the set of possible worlds to generate expectations of possible outcomes, but can not generally obtain bounds on their estimates. Queries over symbolic models for incomplete data like C-tables [33] and m-tables [58] often have PTIME data complexity, but obtaining certain answers from query results is intractable.

Aggregation in Incomplete/Probabilistic Databases. General solutions for non-windowed aggregation over uncertain data remain an open problem [18]. Due to the complexity of uncertain aggregation, most approaches focus on identifying tractable cases and producing lossy representations [5, 15, 16, 35, 37, 43, 47, 54, 61]. 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, 41] 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 $\mathcal{RA}^{agg}$ 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 [56] picks the top-k set with the highest probability. U-Rank [56] assigns to each rank the tuple which is most-likely to have this rank. Global-Topk [64] 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. [50] proposed a multi-simulation algorithm that stops when a guaranteed top-k probability can be guaranteed. Soliman et al. [55] proposed a framework that integrates tuple retrieval, grouping, aggregation, uncertainty management, and ranking in a pipelined fashion. Li et al. [42] proposed a unified ranking approach for top-k based on generating functions which use and/xor trees to reason about complex correlations. 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 encode 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. **Temporal Aggregation.** Temporal databases must reason about tuples associated with partially overlapping intervals. For example, a window aggregate may be recast as an interval self-join, where one table defines the set of windows and each of its tuples is joined with the tuples in the window. We take inspiration from temporal databases for our own operator implementations. The first temporal aggregation algorithm was given in [59]. Moon et al. [46] proposed a balanced tree algorithm for **count**, **sum** and **avg** aggregates, and a divide-and-conquer algorithm for **min** and **max**. Kline and Snodgrass proposed the *aggregation tree* [38], an in-memory data structure that supports incremental computation of temporal aggregates. Yang et al. [60] proposed a materialized version called the *SB-tree* that can be used as an index for incremental temporal aggregation computations. The *MVSB-tree* [63] is and extension of the SB-tree that supports

$$[\![\pi_A(R)]\!](t) = \sum_{t' :\, t = \pi_A t'} R(t') \qquad [\![\sigma_\theta(R)]\!](t) = \begin{cases} R(t) & \textbf{if } \theta(t) \\ 0 & \textbf{otherwise} \end{cases}$$

$$[\![R \cup S]\!](t) = R(t) + S(t) \qquad [\![R \times S]\!](t) = R(t) \times S(t)$$

Figure 2: Evaluation semantics $[\![\cdot]\!]$ that lift the operations of a semiring $\mathcal{K}$ to $\mathcal{RA}^+$ operations over $\mathcal{K}$-relations.

predicates in the aggregation query. Piatov and Helmer [49] proposed a sweep-line based approach the reduces the space needed to compute **min** and **max** aggregates over temporal data.

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 $\text{Sch}(R_i) = (A_1, \ldots, A_n)$. Use $arity(\text{Sch}(R))$ 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 $\mathbb{D}$, a tuple with schema $\text{Sch}(R)$ is an element from $\mathbb{D}^{arity(\text{Sch}(R))}$.

A $\mathcal{K}$-relations [27] annotates each tuple with an element of a (commutative) semiring. In this paper, we focus on which $\mathbb{N}$-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 have finite support (tuple not mapped to 0). Since $\mathcal{K}$-relations are functions from tuples to annotations, it is customary to denote the annotation of a tuple t in relation R as $R(t)$. A $\mathcal{K}$-database is a set of $\mathcal{K}$-relations. Green et al. [27] did use the semiring operations to express positive relational algebra ($\mathcal{RA}^+$) operations over $\mathcal{K}$-relations as shown in Fig. 2. Notably for us, for the natural numbers semiring $\mathbb{N} = (\mathbb{N}, +, \times, 0, 1)$, this semantics are equivalent to those of positive bag-relational algebra.

3.1 Incomplete N-Relations

An incomplete $\mathbb{N}$-database $\mathcal{D} = \{D_1, \ldots, D_n\}$ (resp., incomplete $\mathbb{N}$-relation $\mathcal{R} = \{R_1, \ldots, R_n\}$) is a set of $\mathbb{N}$-databases D_i (resp., $\mathbb{N}$-relations R_i) called possible worlds. Queries over incomplete $\mathbb{N}$-databases use possible world semantics: The result of a query Q over an incomplete $\mathbb{N}$-database $\mathcal{D}$ is the set of relations $\mathcal{R}$ (possible worlds) derived by evaluating Q over every world in $\mathcal{D}$ using the semantics of Fig. 2. In addition to enumerating all possible query results, past work has introduced the concept of *certain* and *possible answers* for set semantics, which are respectively the set of tuples present in all worlds or in at least one world. Certain and possible answers have been generalized [23, 30] to bag semantics as the extrema of the tuple's annotations across all possible worlds. Formally, the certain and possible annotations of a tuple t in $\mathcal{R}$ are:

$$\text{CERT}_{\mathbb{N}}(\mathcal{R}, t) := \min(\{R(t) \mid R \in \mathcal{R}\})$$
$$\text{POSS}_{\mathbb{N}}(\mathcal{R}, t) := \max(\{R(t) \mid R \in \mathcal{R}\})$$

3.2 AU-Databases (AU-DBs)

Using $\mathcal{K}$-relations , we introduced *AU-DBs* [23] (*attribute-annotated uncertain databases*), a special type of $\mathcal{K}$-relation that summarizes an incomplete $\mathcal{K}$-relation by bounding its set of possible worlds. An AU-DB differs from the classical relational model in two key ways: First, tuples are not defined as individual points $\mathbb{D}^n$, but rather as a bounding hypercube specified as upper and lower bounds (and selected-guess) for each attribute value. Every such hypercube can represent zero or more tuples contained inside it. Second, the annotation of each hypercube tuple is also a range of possible annotations (e.g., multiplicities for range-annotated $\mathbb{N}$-relations). Intuitively, an AU-DB *bounds* a possible world if the hypercubes of its tuples contain all of the possible world's tuples, and the total multiplicity of tuples in the hypercube fall into the range annotating the hypercube. An AU-DB *bounds* an incomplete $\mathcal{K}$-database $\mathcal{D}$ if it bounds all of $\mathcal{D}$'s possible worlds. To be able to model, e.g., the choice of repair made by a heuristic data repair algorithm, the value and annotation domains of an AU-DB also contain a third component: a *selected-guess* (SGW) that encodes one world.

Formally, in an AU-DB, attribute values are *range-annotated values* $c = [c^{\downarrow}/c^{sg}/c^{\uparrow}]$ from a *range-annotated domain* $\mathbb{D}_I$ that encodes the selected-guess value $c^{sg} \in \mathbb{D}$ and two values ($c^{\downarrow}, c^{\uparrow} \in \mathbb{D}$) that bound c^{sg} from below and above. For any $c \in \mathbb{D}_I$ we have $c^{\downarrow} \leq c^{sg} \leq c^{\uparrow}$. We call a value $c \in \mathbb{D}_I$ certain if $c^{\downarrow} = c^{sg} = c^{\uparrow}$. AU-DBs encode bounds on the multiplicities of tuples by using $\mathbb{N}^3 = (\mathbb{N}^3, +_{\mathbb{N}^3}, \cdot_{\mathbb{N}^3}, \mathbb{0}_{\mathbb{N}^3}, \mathbb{1}_{\mathbb{N}^3})$ annotations on tuples in ${\mathbb{D}_I}^n$. The annotation $(k^{\downarrow}, k^{sg}, k^{\uparrow})$ encodes a lower bound on the certain multiplicity of the tuple, the multiplicity of the tuple in the SGW, and an over-approximation of the tuple's possible multiplicity. We note that an AU-DB can be encoded in a relational database by encoding each annotated value as three columns encoding the lower bound, selected-guess and upper bound value. Consider the AU-DB relation $\mathbf{R}(A, B)$ with a tuple $([1/3/5], [a/a/a])$ annotated with $(1, 1, 2)$. This tuple represents the fact that each world consists of either 1 and 2 tuples with $B = a$ and A between 1 and 5. The SGW contains a tuple $(3, a)$ with multiplicity 1.

Bounding Databases. As noted above, an AU-DB summarizes an incomplete $\mathbb{N}$-relation by defining bounds over the possible worlds that comprise it. To formalize bounds over $\mathbb{N}$-relations, we first define what it means for a range-annotated tuple to bound a set of deterministic tuples. Let $\mathbf{t}$ be a range-annotated tuple with schema $(a_1, \ldots, a_n)$ and t be a tuple with the same schema as $\mathbf{t}$. $\mathbf{t}$ bounds t (denoted $t \sqsubseteq \mathbf{t}$) iff $\forall i \in \{1, \ldots, n\} : \mathbf{t}.a_i^{\downarrow} \leq t.a_i \leq \mathbf{t}.a_i^{\uparrow}$

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-relation bounds a possible world if we can distribute the multiplicity of each tuple in the possible world over the AU-relation's tuples. This idea is formalized through *tuple matchings*. A tuple matching $\mathcal{TM}$ from an n-ary AU-relation $\mathbf{R}$ to an n-ary relation R is a function $({\mathbb{D}_I})^n \times \mathbb{D}^n \to \mathbb{N}$ that fully allocates the multiplicity of every tuple of R:

$$\forall \mathbf{t} \in {\mathbb{D}_I}^n : \forall t \not\sqsubseteq \mathbf{t} : \mathcal{TM}(\mathbf{t}, t) = 0 \quad \forall t \in \mathbb{D}^n : \sum_{\mathbf{t} \in {\mathbb{D}_I}^n} \mathcal{TM}(\mathbf{t}, t) = R(t)$$

$\mathbf{R}$ bounds R (denoted $R \sqsubset \mathbf{R}$) iff there exists a tuple matching $\mathcal{TM}$ where the total multiplicity allocated to each $\mathbf{t} \in \mathbf{R}$ falls within the bounds annotating $\mathbf{t}$:

$$\forall \mathbf{t} \in {\mathbb{D}_I}^n : \sum_{t \in \mathbb{D}^n} \mathcal{TM}(\mathbf{t}, t) \geq \mathbf{R}(\mathbf{t})^{\downarrow} \textbf{ and } \sum_{t \in \mathbb{D}^n} \mathcal{TM}(\mathbf{t}, t) \leq \mathbf{R}(\mathbf{t})^{\uparrow}$$

An AU-DB relation $\mathbf{R}$ bounds an incomplete $\mathbb{N}$-relation $\mathcal{R}$ (denoted $\mathcal{R} \sqsubset \mathbf{R}$) iff it bounds every possible world (i.e., $\forall R \in \mathcal{R} : R \sqsubset \mathbf{R}$), and if projecting down to the selected guess attribute of $\mathbf{R}$ results

in a possible world of $\mathcal{R}$. As shown in [23, 24], (i) AU-DB query semantics is closed under $\mathcal{RA}^+$, set difference and aggregations, and (ii) queries preserve bounds. That is, if every relation $\mathbf{R}_i \in \mathbf{D}$ bounds the corresponding relation of an incomplete database $\mathcal{R}_i \in \mathcal{D}$ (i.e., $\forall i : \mathcal{R}_i \sqsubset \mathbf{R}_i$), then for any query Q, the results over $\mathbf{D}$ bound the results over $\mathcal{D}$ (i.e., $Q(\mathcal{D}) \sqsubset Q(\mathbf{D})$).

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

$$\forall t \sqsubseteq \mathbf{t} : c = [\![e]\!]_t, (c^{\downarrow}, c^{sg}, c^{\uparrow}) = [\![e]\!]_{\mathbf{t}} \quad \rightarrow \quad c^{\downarrow} \leq c \leq c^{\uparrow}$$

[24] proved this property for any e composed of attributes, constants, arithmetic and boolean operators, and comparisons. For example, $[a^{\downarrow}/a^{sg}/a^{\uparrow}] + [b^{\downarrow}/b^{sg}/b^{\uparrow}] = [a^{\downarrow} + b^{\downarrow}/a^{sg} + b^{sg}/a^{\uparrow} + b^{\uparrow}]$

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:

$$t <_O t' \Leftrightarrow \exists j \in \{1, \ldots, m\} : \forall k \in \{1, \ldots, j-1\} : t.A_{i_k} = t'.A_{i_k} \wedge t'.A_{i_j} < t.A_{i_j}$$

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 equivalence) by extending the ordering on attributes O, using the remaining attributes of the relation as a tiebreaker: The total order $t <_O^{total} t'$ for tuples from a relation R is defined as $t <_{O,\mathrm{Sch}(R)-O} t'$ (assuming some arbitrary order of the attributes in $\mathrm{Sch}(R)$).

Example 4 (Sorting). *Consider the relation R shown on the left below. The multiplicity from $\mathbb{N}$ assigned to each tuple is shown on the right. The result of sorting the relation on attribute A using our deterministic semantics and storing the sort positions in column* pos *is shown below on the right. Note the order of tuples $t_1 = (3, 15)$ and $t_2 = (1, 1)$ is made deterministic by $t_2 <_A^{total} t_1$, because $t_2.B < t_1.B$. Note also that the two copies of t_2 are each assigned a different position.*

A	B	$\mathbb{N}$
3	15	1
1	1	2

A	B	pos	$\mathbb{N}$
1	1	0	1
1	1	1	1
3	15	4	1

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 5 (Row-Based Windows). *Consider the bag relation below and consider the windowed aggregation sum(B) sorting 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 $<_O^{total}$ to avoid non-determinism in what their windows are. We have $t_2 <_O^{total} t_3$ 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.*

A	B	C	$\mathbb{N}$
a	5	3	3
b	3	1	1
b	3	4	1

A	B	C	sum(B)	$\mathbb{N}$
a	5	3	5	1
a	5	3	10	1
a	5	3	15	1
b	3	1	13	1
b	3	4	11	1

The semantics of the row-based window aggregate operator ω is shown in Fig. 3. The parameters of ω are partition-by attributes G, order-by attributes O, an aggregate function $f(A)$ with $A \subseteq \mathrm{Sch}(R)$, 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 $\mathrm{Sch}(R) \circ X$.

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

Ω computes the dense-rank windowed aggregate. $\mathrm{rank}(R, O, t)$ computes the dense rank of tuple $t \in R$ using $<_O$: the number of tuple groups (tuples with matching values of O) preceding t. We define the window for tuple t (denoted $\mathcal{W}^{groups}_{G,O,l,u,R,t}$) point-wise for each tuple by considering the tuple's dense rank within its partition (denoted $\mathcal{P}_{R,G,t}$).

$$\Omega^{[l,u]}_{f(A)\to X;\, G;\, O}(R)(t) = \begin{cases} R(t') & \textbf{if } t = t' \circ f(\pi_A(\mathcal{W}^{groups}_{G,O,l,u,R,t'})) \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathcal{W}^{groups}_{G,O,l,u,R,t}(t') = \begin{cases} R(t') & \textbf{if } (\mathsf{rank}(\mathcal{P}_{R,G,t}, O, t) \\ & \quad -\mathsf{rank}(\mathcal{P}_{R,G,t}, O, t')) \in [l,u] \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathsf{rank}(R, O, t) = |\{\ t'.O \mid R(t') > 0 \wedge t' <_O t\ \}|$$

$$\omega^{[l,u]}_{f(A)\to X;\, G;\, O}(R)(t) = \pi_{\mathrm{Sch}(R),X}(\mathcal{ROW}(R))$$

$$\mathcal{ROW}(R)(t) = \begin{cases} 1 & \textbf{if } t = t' \circ f(\pi_A(\mathcal{W}_{R,t',i})) \circ i \\ & \quad \wedge i \in [0, R(t') - 1] \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathcal{P}_{R,t}(t') = \begin{cases} R(t') & \textbf{if } t'.G = t.G \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathcal{W}_{R,t,i}(t') = |\mathit{cover}(\mathcal{P}_{R,t}, t') \cap \mathit{bounds}(\mathcal{P}_{R,t}, t, i)|$$

$$\mathsf{pos}(R, t, i) = i + \sum_{t' <^{total}_O t} R(t')$$

$$\mathit{cover}(R, t) = [\mathsf{pos}(R, t, 0), \mathsf{pos}(R, t, R(t) - 1)]$$

$$\mathit{bounds}(R, t, i) = [\mathsf{pos}(R, t, i) + l, \mathsf{pos}(R, t, i) + u]$$

Figure 3: Windowed Aggregation

Because dense-rank windows are computed over tuple groups, the multiplicity of teach tuple in the window is taken directly from the relation. For sparse rank windows, we need to ensure that the window contains exactly the desired number of rows, e.g., a window with bounds $[-2, 0]$ should contain exactly 3 rows, e.g., 1 row with multiplicity 2 and one row with multiplicity 1. Since we need to ensure that windows contain a fixed number of tuples, it may be the case that a tuple will be included with a multiplicity in the window that is less than the tuple's multiplicity in the input relation. Furthermore, the window for one duplicate of a tuple may differ from the window of another duplicate of the same tuple.

4.2 Sort Operator

We now define a sort operator $\textsc{sort}_{O\to\tau}(R)$ which extends each row of R with an attribute τ that stores the position of this row in R according to $<^{total}_O$. 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_i is in* $\mathrm{Sch}(R)$. *The* sort operator $\textsc{sort}_{O\to\tau}(R)$ *returns a relation with schema $(A_1, \ldots, A_n, \tau)$ as defined below.*

$$\textsc{sort}_{O\to\tau}(R) = \pi_{\mathrm{Sch}(R),\tau-1\to\tau}(\omega^{[-\infty,0]}_{count(1)\to\tau;\, \emptyset;\, O}(R))$$

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 $\pi_{A,B}(\sigma_{r\leq 3}(\textsc{sort}_{A\to r}(R)))$.

```
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-DBs is annotated with a triple of multiplicities and that each (range-annotated) value is likewise a triple. Elements of a range-annotated value $\mathbf{c} = [c_1/c_2/c_3]$ or multiplicity triple (n_1, n_2, n_3) are accessed as: $\mathbf{c}^{\downarrow} = c_1$, $\mathbf{c}^{sg} = c_2$, and $\mathbf{c}^{\uparrow} = c_3$. 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 creates uncertainty in a tuple's position in the sort order. The earlier, 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. to 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 $<^{total}_O$ is defined in terms of $<_O$). Per [24], a Boolean expression over range-annotated values evaluates to a bounding triple (using the order $\bot < \top$ where $\bot$ denotes false and $\top$ denotes true). The result of an evaluation of an expression e is denoted as $[\![e]\!]$. For instance, $[\![[1/1/3] < [2/2/2]]\!]_= = [\bot/\top/\top]$, 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 $\mathbf{t_1} = ([1/1/3], [a/a/a])$ and $\mathbf{t_2} = ([2/2/2], [b/b/b])$ over schema $R(A, B)$. We have $\mathbf{t_1} <_{A,B} \mathbf{t_2} = [\bot/\top/\top]$, because $\mathbf{t_1}$ could be ordered before $\mathbf{t_2}$ (if $\mathbf{t_1}.A$ is 1), is ordered before $\mathbf{t_2}$ in the selected-guess world ($1 < 2$), and may be ordered before $\mathbf{t_2}$ (if A is 3).

$$[\![\mathbf{t} <_O \mathbf{t}']\!]^{\downarrow} = \exists i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, i-1\} : [\![\mathbf{t}.A_j = \mathbf{t}'.A_j]\!]^{\downarrow} \wedge [\![\mathbf{t}.A_i < \mathbf{t}'.A_i]\!]^{\downarrow}$$

$$[\![\mathbf{t} <_O \mathbf{t}']\!]^{sg} = \exists i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, i-1\} : [\![\mathbf{t}.A_j = \mathbf{t}'.A_j]\!]^{sg} \wedge [\![\mathbf{t}.A_i < \mathbf{t}'.A_i]\!]^{sg}$$

$$[\![\mathbf{t} <_O \mathbf{t}']\!]^{\uparrow} = \exists i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, i-1\} : [\![\mathbf{t}.A_j = \mathbf{t}'.A_j]\!]^{\uparrow} \wedge [\![\mathbf{t}.A_i < \mathbf{t}'.A_i]\!]^{\uparrow}$$

To simplify notation, we will use $\mathbf{t} <_O \mathbf{t}'$ instead of $[\![\mathbf{t} <_O \mathbf{t}']\!]$.

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 $\mathbf{t}$ in an AU-DB relation $\mathbf{R}$ is the total multiplicity of tuples $\mathbf{t}'$ that certainly exist ($\mathbf{R}(\mathbf{t}')^{\downarrow} > 0$) and are certainly smaller than $\mathbf{t}$ (i.e., $[\![\mathbf{t}' <_O \mathbf{t}]\!]^{\downarrow} = \top$). The selected-guess position of a tuple is the position of the tuple in the selected-guess world, and the greatest possible position of $\mathbf{t}$ is the total multiplicity of tuples that possibly exist ($\mathbf{R}(\mathbf{t}')^{\uparrow} > 0$) and possibly precede $\mathbf{t}$ (i.e., $[\![\mathbf{t}' <_O \mathbf{t}]\!]^{\uparrow} = \top$). The sort position of the i^{th} duplicate (with the first duplicate being 0) is computed by adding i to the position bounds of the first duplicate.

$$\text{sort}_{O\to\tau}(\mathbf{R})(\mathbf{t}) = \begin{cases} (1,1,1) & \textbf{if } \mathbf{t} = \mathbf{t}' \circ \text{pos}(\mathbf{R}, O, \mathbf{t}', i) \wedge i \in \left[0, \mathbf{R}(\mathbf{t}')^{\downarrow}\right) \\ (0,1,1) & \textbf{if } \mathbf{t} = \mathbf{t}' \circ \text{pos}(\mathbf{R}, O, \mathbf{t}', i) \wedge i \in \left[\mathbf{R}(\mathbf{t}')^{\downarrow}, \mathbf{R}(\mathbf{t}')^{sg}\right) \\ (0,0,1) & \textbf{if } \mathbf{t} = \mathbf{t}' \circ \text{pos}(\mathbf{R}, O, \mathbf{t}', i) \wedge i \in \left[\mathbf{R}(\mathbf{t}')^{sg}, \mathbf{R}(\mathbf{t}')^{\uparrow}\right) \\ (0,0,0) & \textbf{otherwise} \end{cases}$$

Figure 4: Range-annotated sort operator semantics.

$$\text{pos}(\mathbf{R}, O, \mathbf{t}, i)^{\downarrow} = i + \textstyle\sum_{(\mathbf{t}' <_O \mathbf{t})^{\downarrow}} \mathbf{R}(\mathbf{t}')^{\downarrow} \quad (1)$$

$$\text{pos}(\mathbf{R}, O, \mathbf{t}, i)^{sg} = i + \textstyle\sum_{(\mathbf{t}' <_O \mathbf{t})^{sg}} \mathbf{R}(\mathbf{t}')^{sg} \quad (2)$$

$$\text{pos}(\mathbf{R}, O, \mathbf{t}, i)^{\uparrow} = i + \textstyle\sum_{(\mathbf{t}' <_O \mathbf{t})^{\uparrow}} \mathbf{R}(\mathbf{t}')^{\uparrow} \quad (3)$$

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 $\mathbf{t}$ in the result is either 1 for duplicates that are guaranteed to exist ($i < \mathbf{R}(\mathbf{t})^{\downarrow}$) and 0 otherwise. The selected-guess multiplicity is 1 for duplicate 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 $\mathbf{R}$ be an AU-DB relation and $O \subseteq \text{Sch}(\mathbf{R})$. The result of applying the sort operator $\text{sort}_{O\to\tau}$ to $\mathbf{R}$ is defined in Fig. 4*

Every tuple in the result of sorting is constructed by extending an input tuple $\mathbf{t}'$ with the range of positions $\text{pos}(\mathbf{R}, O, \mathbf{t}', i)$ it may occupy wrt. the sort order. The definition decomposes $\mathbf{t}$ into a base tuple $\mathbf{t}'$, and a position triple for each duplicate of $\mathbf{t}$ in $\mathbf{R}$. 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 6 (AU-DB Sorting). *Consider the AU-DB relation $\mathbf{R}$ 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 sorting the relation on attributes A, B using AU-DB sorting semantics and storing the sort positions in column* pos *($\text{sort}_{A,B\to pos}(\mathbf{R})$) is shown below on the right. Observe how the 1^{th} input tuple $\mathbf{t_1} = (1, [1/1/3])$ was split into two result tuples occupying adjacent sort positions. The 3^{rd} input tuple $\mathbf{t_3} = ([1/1/2], 2)$ could be the 1^{th} in sort order (if it's A value is 1 and the B values of the duplicates of $\mathbf{t_1}$ are equal to 3) or be at the 3^{rd} position if two duplicates of $\mathbf{t_1}$ exist and either A is 2 or the B values of $\mathbf{t_1}$ are all < 3.*

A	*B*	$\mathbb{N}^3$
1	[1/1/3]	*(1,1,2)*
[2/3/3]	*15*	*(0,1,1)*
[1/1/2]	*2*	*(1,1,1)*

A	*B*	*pos*	$\mathbb{N}^3$
1	[1/1/3]	[0/0/1]	*(1,1,1)*
1	[1/1/3]	[1/1/2]	*(0,0,1)*
[1/1/2]	*2*	[0/1/2]	*(1,1,1)*
[2/3/3]	*15*	[2/2/3]	*(0,1,1)*

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 $\mathbf{R}$ that bounds an incomplete bag database $\mathcal{R}$, the result of a sort operator $\text{sort}_{O\to\tau}$ applied to $\mathbf{R}$ bounds the result of $\text{sort}_{O\to\tau}$ evaluated over $\mathcal{R}$.

Theorem 1 (Bound Preservation of Sorting). *Given an AU-DB relation $\mathbf{R}$ and incomplete bag relation $\mathcal{R}$ such that $\mathcal{R} \sqsubset \mathbf{R}$, and $O \subseteq \text{Sch}(\mathcal{R})$. We have:*

$$\text{sort}_{O\to\tau}(\mathcal{R}) \sqsubset \text{sort}_{O\to\tau}(\mathbf{R})$$

Proof: Since $\mathcal{R} \sqsubset \mathbf{R}$, for every possible world $R \in \mathcal{R}$, there has to exist a tuple matching $\mathcal{TM}$ (see Sec. 3.2) based on which this property holds. We will show that based on $\mathcal{TM}$ we can generate a tuple matching for $\text{sort}_{O\to\tau}(\mathbf{R})$ and $\text{sort}_{O\to\tau}(\mathcal{R})$. The existence of such a tuple matching for every $\text{sort}_{O\to\tau}(R) \in \text{sort}_{O\to\tau}(\mathcal{R})$ implies that $\text{sort}_{O\to\tau}(\mathcal{R}) \sqsubset \text{sort}_{O\to\tau}(\mathbf{R})$. Define $S(t)$ as the set of tuples smaller than t in sort order. WLOG consider one tuple $\mathbf{t}$ with $\mathbf{R}(\mathbf{t}) > 0$ and let $t_1, \ldots, t_n$ be the tuples in R for which $\mathcal{TM}(\mathbf{t}, t_i) > 0$. Recall that the sort operator splits each input tuple t_i with $R(t) = n$ into n output tuples t_{ij} with multiplicity 1 each. Thus, the sum of the multiplicities of tuples t_{ij} for $j \in [0, R(t_i)-1]$ is equal to $R(t_i)$. Both the deterministic and AU-DB version of $\text{sort}_{\to}$ does preserve all attribute values of input tuples and adds another attribute for sorting and ranged sorting results. We first prove that $\{G(t_i)\} \sqsubset G(\mathbf{t})$. For all instances $G(t_i)$ where $t_i \in t_1, \ldots, t_n$, we have $G(t_i) \sqsubset G(\mathbf{t})$ given $t_i <_O t_x \sqsubset \mathbf{t} <_O \mathbf{t}_x$ if $t_i \sqsubset \mathbf{t}$ and $t_x \sqsubset \mathbf{t}_x$. Given ranged result value $\mathbf{p}$ for $\mathbf{t}$ and result value p_i for t_i. By the ranged sorting definition, we have $p_i \sqsubset \mathbf{p}$. So $t_i \circ p_i \sqsubseteq \mathbf{t} \circ \mathbf{p}$ holds for all i. So output tuple matching $\mathcal{TM}(\mathbf{t} \circ \mathbf{p}, t_i \circ p_i) = \mathcal{TM}(\mathbf{t}, t_i)$ exists.

6 AU-DB WINDOWED AGGREGATION

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 ($\mathbf{R}(\mathbf{t})^{\downarrow} = 0$) or has uncertain partition attributes; (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 $\mathbf{t}$ an AU-DB relation $\mathcal{P}_{\mathbf{t}}(\mathbf{R})$ storing the tuples that certainly and possibly belong to the partition for that tuple; (iii) we then compute an AU-DB relation $\mathcal{W}_{\mathbf{R},\mathbf{t}}$ 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 $\mathbf{t}$ will never contain more than 3 tuples. If we know that two tuples certainly belong to the window, then at most one of the possible tuples can be part of 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 which $\mathcal{P}_{\mathbf{t}}(\mathbf{R})$ encodes the multiplicity with which a tuple $\mathbf{t}'$ belongs to the partition for $\mathbf{t}$ based on partition-by attributes G. This is achieved using selection, comparing a tuple's values in G with the values of $\mathbf{t}.G$ on equality. AU-DB selection sets the certain, selected-guess, possible multiplicity of a tuple to 0 if the tuple possibly, in the selected-guess world, or certainly does not fulfill the selection condition.

$$\mathcal{P}_{\mathbf{t}}(\mathbf{R}) = [\![\sigma_{G=\mathbf{t}.G}(\mathbf{R})]\!]$$

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 $\mathbf{t}$, we model the window's tuples as an AU-DB relation $\mathcal{W}_{\mathbf{R},\mathbf{t}}$ 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 $\mathcal{W}_{\mathbf{R},\mathbf{t}}$ in Fig. 6. Recall that in the first step we used sort to split the duplicates of each tuple into tuples with multiplicity upper bounds of 1. Thus, the windows we are constructing here are for tuples instead of for individual duplicates of a tuple. A tuple $\mathbf{t}'$ is guaranteed to belong to the window for of a tuple $\mathbf{t}$ with a multiplicity of $n = \mathbf{R}(\mathbf{t}')^{\downarrow}$ (the number of duplicates of the tuple that certainly exist) if the tuple certainly belongs to the partition for $\mathbf{t}$ 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 $\mathbf{t}$. Tuple $\mathbf{t}'$ possibly belongs to the window of $\mathbf{t}$ if any of its possible positions falls within the interval of all possible positions of $\mathbf{t}$. As an example consider Fig. 5 which shows the sort positions that certainly (red) and possibly (green) belong to tuple $\mathbf{t}$'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 $\mathbf{t}$'s window which is $\mathbf{t}.\tau^{\uparrow} + l$ ($6 + (-1) = 5$ in our example) and end at the earliest possible upper bound for the window which is $\mathbf{t}.\tau^{\uparrow} + u$ ($4 + 4 = 8$ in our example). Furthermore, Fig. 5 shows the membership of three tuples in the window. Tuple $\mathbf{t_1}$ does certainly not belong to the window, because none of its possible sort positions are in the window's set of possible sort positions, $\mathbf{t_2}$ does certainly belong to the window, because all of its possible sort positions are in the set of positions certainly in the window. Finally, $\mathbf{t_3}$ 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 size($[l, u]$) to denote the size of a window with bounds $[l, u]$, i.e., 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 size($[l, u]$) $- m$ rows possibly in the window that minimizes (for the lower aggregation result bound) or maximizes the (for the upper aggregation result bound) the aggregation function result for an input tuple. Let us use possn($\mathbf{R}, \mathbf{t}$) to denote size($[l, u]$) $- m$:

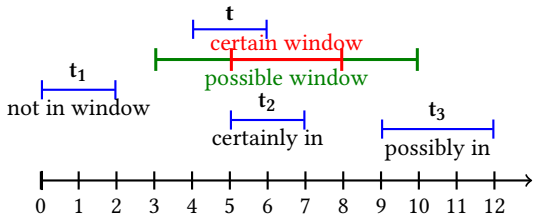


Figure 5: Possible and certain window membership of tuples in window of t based on their possible sort positions for window spec [-1,4].

$$\text{possn}(\mathbf{R}, \mathbf{t}) = \text{size}([l, u]) - \sum_{\mathbf{t}'} \mathbf{R}(\mathbf{t}')^{\downarrow}$$

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

For each tuple $\mathbf{t}$, we define AU-DB relation $\mathcal{RW}_{\mathbf{R},\mathbf{t}}$ 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 **sum**, the definitions for other aggregation functions are similar. In the definition, we make use $\mathbf{R}^{\downarrow}$ and $\mathbf{R}^{\uparrow}$:

$$\mathbf{R}^{\downarrow}(\mathbf{t}) = \mathbf{R}(\mathbf{t})^{\downarrow} \qquad \mathbf{R}^{\uparrow}(\mathbf{t}) = \mathbf{R}(\mathbf{t})^{\uparrow}$$

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

$$\mathcal{RW}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{\downarrow} = \mathcal{W}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{\downarrow} + \text{min-k}(\mathcal{W}_{\mathbf{R},\mathbf{t}}, A)(\mathbf{t}')$$
$$\mathcal{RW}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{\uparrow} = \mathcal{W}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{\downarrow} + \text{max-k}(\mathcal{W}_{\mathbf{R},\mathbf{t}}, A)(\mathbf{t}')$$
$$\mathcal{RW}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{sg} = \mathcal{W}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{sg}$$
$$\text{min-k}(\mathbf{R}, A) = \sigma_{\tau<\text{possn}(\mathbf{R},\mathbf{t})}(\text{SORT}_{A^{\downarrow}\to\tau}(\sigma_{A^{\downarrow}<0}(\mathbf{R}^{\downarrow})))$$
$$\text{max-k}(\mathbf{R}, A) = \sigma_{\tau<\text{possn}(\mathbf{R},\mathbf{t})}(\text{SORT}_{-A^{\uparrow}\to\tau}(\sigma_{A^{\uparrow}>0}(\mathbf{R}^{\uparrow})))$$

$$\mathcal{W}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{\downarrow} = \begin{cases} \mathcal{P}_{\mathbf{t}}(\mathbf{R})(\mathbf{t}')^{\downarrow} & \textbf{if } [\text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}', 0)^{\downarrow}, \text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}', 0)^{\uparrow}] \subseteq [\text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}, 0)^{\uparrow} + l, \text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}, 0)^{\downarrow} + u] \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathcal{W}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{\uparrow} = \begin{cases} \mathcal{P}_{\mathbf{t}}(\mathbf{R})(\mathbf{t}')^{\uparrow} - \mathcal{W}_{\mathbf{R},\mathbf{t},i}(\mathbf{t}')^{\downarrow} & \textbf{if } ([\text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}', 0)^{\downarrow}, \text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}', 0)^{\uparrow}] \cap [\text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}, 0)^{\downarrow} + l, \text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}, 0)^{\uparrow} + u]) \neq \emptyset \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathcal{W}_{\mathbf{R},\mathbf{t}}(\mathbf{t}')^{sg} = \begin{cases} \mathcal{P}_{\mathbf{t}}(\mathbf{R})(\mathbf{t}')^{sg} & \textbf{if } \text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}', 0)^{sg} \subseteq [\text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}, 0)^{sg} + l, \text{pos}(\mathcal{P}_{\mathbf{t}}(\mathbf{R}), O, \mathbf{t}, 0)^{sg} + u] \\ 0 & \textbf{otherwise} \end{cases}$$

Figure 6: Certain and possible windows for row-based windowed aggregation over AU-DBs

Windowed Aggregation. Using the filtered combined windows we are ready to define row-based windowed aggregation over UA-DBs. To compute aggregation results, we utilize the operation $\circledast_f$ defined in [24] for aggregation function f that combines the range-annotated values of tuples in the aggregation attribute with the tuple's multiplicity bounds. For instance, for **sum**, $\circledast_{\textbf{sum}}$ is multiplication, e.g., if a tuple with A value $[10/20/30]$ has multiplicity $(1, 2, 3)$ it contributes $[10/40/90]$ to the sum. Here, $\bigoplus$ denotes the application of the aggregation function over a set of elements (e.g., $\sum$ for **sum**). Note that, as explained above, the purpose of expand($\mathbf{R}$) 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* $\mathbf{R}$ *be an UA-DB relation. We define window operator* $\omega^{[l,u]}_{f(A)\to X;\, G;\, O}$ *as:*

$$\omega^{[l,u]}_{f(A)\to X;\, G;\, O}(\mathbf{R})(\boldsymbol{t}) = \pi_{\text{Sch}(\mathbf{R}),X}(\mathcal{ROW}(\mathbf{R}))$$
$$\mathcal{ROW}(\mathbf{R})(\boldsymbol{t} \circ \mathit{aggres}(\boldsymbol{t})) = \mathit{expand}(\mathbf{R})(\boldsymbol{t})$$
$$\mathit{aggres}(\boldsymbol{t}) = \bigoplus_{\mathbf{t}'} \mathbf{t}'.A \circledast_f \mathcal{RW}_{\mathit{expand}(\mathbf{R}),\boldsymbol{t}}(\mathbf{t}')$$
$$\mathit{expand}(\mathbf{R}) = \pi_{\text{Sch}(\mathbf{R}),\tau_{id}}(\text{SORT}_{\text{Sch}(\mathbf{R})\to\tau_{id}}(\mathbf{R}))$$

Example 7 (AU-DB Windowed Aggregation). *Consider the AU-DB relation* $\mathbf{R}$ *shown below and query* $\omega^{[-1,0]}_{sum(C)\to SumA;\, A;\, B}(\mathbf{R})$*, ire., windowed aggregation partitioning by A, ordering on B, and computing 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* $\mathbf{t_3}$*. Both* $\mathbf{t_1}$ *and* $\mathbf{t_2}$ *may belong to the same partition as* $\mathbf{t_3}$ *as their A value ranges overlap. There is no tuple that certainly belongs to the same partition as* $\mathbf{t_3}$*. Thus, only tuple* $\mathbf{t_3}$ *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* $\mathbf{t_3}$*. These are the two tuples corresponding to the duplicates of* $\mathbf{t_1}$*, because these tuples may belong to the partition for* $\mathbf{t_3}$ *and their possible sort positions (*$[0/0/1]$ *and* $[1/1/2]$*) overlap with the sort positions possibly covered by the window for* $\mathbf{t_3}$ *(*$[0/1/2]$*). 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* $\mathbf{t_3}$ *(*$[2/4/5]$*) and no additional tuple from the possible window (because the C value of* $\mathbf{t_1}$ *is positive) for lower bound and the largest possible C value of one copy (we can only fit one additional tuple into the window) of* $\mathbf{t_1}$ *(7) for the upper bound. Thus, we get the aggregation result* $[4/11/11]$ *as shown below.*

	A	B	C	$\mathbb{N}^3$
$\mathbf{t_1}$	1	[1/1/3]	7	(1,1,2)
$\mathbf{t_2}$	[2/3/3]	15	4	(0,1,1)
$\mathbf{t_3}$	[1/1/2]	2	[2/4/5]	1

	A	B	C	SumA	$\mathbb{N}^3$
$\mathbf{r_1}$	1	[1/1/3]	7	[7/7/14]	1
$\mathbf{r_2}$	1	[1/1/3]	7	[7/7/14]	(0,0,1)
$\mathbf{r_3}$	[1/1/2]	2	4	[4/11/11]	1
$\mathbf{r_4}$	[2/3/3]	15	[2/4/5]	[2/4/9]	(0,1,1)

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). *Given an AU-DB relation* $\mathbf{R}$ *and incomplete bag relation* $\mathcal{R}$ *such that* $\mathcal{R} \sqsubset \mathbf{R}$*, and* $O \subseteq \text{Sch}(\mathcal{R})$*. For any row-based windowed aggregation* $\omega^{[l,u]}_{f(A)\to X;\, G;\, O}$*, we have:*

$$\omega^{[l,u]}_{f(A)\to X;\, G;\, O}(\mathcal{R}) \sqsubset \omega^{[l,u]}_{f(A)\to X;\, G;\, O}(\mathbf{R})$$

Proof: For ranged tuple $\mathbf{t} \in \mathbf{R}$, define group of tuples in possible window of $\mathbf{t}$ as $W_{\mathbf{t}} = \{\mathbf{t}' \mid \mathcal{W}_{\mathbf{R},\mathbf{t},i}{}^{\uparrow}(\mathbf{t}') + \mathcal{W}_{\mathbf{R},\mathbf{t},i}{}^{\downarrow}(\mathbf{t}') > 0\}$. We first prove that $\bigcup_{\mathcal{TM}(\mathbf{t},t_i)>0} W_{t_i} \sqsubset W_{\mathbf{t}}$. By definition of pos we know that $\text{pos}(t_i) \subseteq \text{pos}(\mathbf{t})$. So for $\mathcal{TM}(\mathbf{t}_1, t_1 i) > 0$ and $\mathcal{TM}(\mathbf{t}_2, t_2 i) > 0$ we have $W_{\mathbf{t}_1}(\mathbf{t}_2) \geq W_{t_1 i}(t_2 i)$. Thus $\bigcup_{\mathcal{TM}(\mathbf{t},t_i)>0} W_{t_i} \sqsubset W_{\mathbf{t}}$. Then $f(A)$ of top u for $W_{\mathbf{t}}$ is greater or equal to top $f(A)$ of top u for any possible worlds. In case of sum calculating lower-bound, assume $\mathcal{TM}(\mathbf{t}, t) > 0$, if $F < \mathbb{F}^{\downarrow}$, then there must exist $W_t > 0$ s.t. $\forall_{W_{\mathbf{t}}>0}$: $t_x.A < \mathbf{t}.A^{\downarrow}$, which implies $\bigcup_{\mathcal{TM}(\mathbf{t},t)=0}$ and contradict with our assumption. So $F \geq \mathbb{F}^{\downarrow}$. Symmetrically we can prove that $F \leq \mathbb{F}^{\uparrow}$. So $\mathbb{F}$ bounds F. So tuple matching $\mathcal{TM}(\mathbf{t} \circ \mathbb{F}, t_i \circ F) = \mathcal{TM}(\mathbf{t}, t)$ exists. Other aggregation functions can be proved in a similar fashion.

7 NATIVE ALGORITHMS

We now introduce optimized algorithms for ranking and windowed aggregation over UA-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 $\mathbf{R}$ as a classical relation [24] where each range-annotated value of an attribute A is stored as three attributes $A^{\downarrow}$, A^{sg}, and $A^{\uparrow}$. In this encoding, attributes $\mathbf{t}.\#^{\downarrow}$, $\mathbf{t}.\#^{sg}$, and $\mathbf{t}.\#^{\uparrow}$ store the tuple's multiplicity bounds.

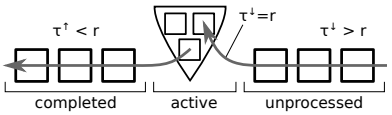


Figure 7: The lifecycle of tuples in Algorithm 1

7.1 Non-deterministic Sort, Top-k

Algorithm 1 sorts an input AU-DB **R**. The algorithm assigns to each tuple its position τ given as lower and upper bounds: $\mathtt{t}.\tau^{\downarrow}$, $\mathtt{t}.\tau^{\uparrow}$, respectively[2]. Given a parameter k, the algorithm can also be used to find the top-k elements; otherwise we set $\mathtt{k} = |R|$ (the size of the input relation). Algorithm 1 (Fig. 7) takes as input the relational encoding of an AU-DB relation **R** sorted on $O^{\downarrow}$, 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 $\mathbf{s}.O^{\uparrow} <_O^{total} \mathbf{t}.O^{\downarrow}$. Since $\mathbf{s}.O^{\downarrow} <_O^{total} \mathbf{s}.O^{\uparrow}$ holds for any tuple, we know that these tuples are visited by Algorithm 1 before **t**. We store tuples in a min-heap `todo` sorted on $O^{\uparrow}$ and maintain a variable $\mathtt{rank}^{\downarrow}$ to store the current lower bound. For every incoming tuple **t**, we first determine all tuples **s** from `todo` certainly preceding **t** ($\mathbf{s}.O^{\uparrow} < \mathbf{t}.O^{\downarrow}$) and update $\mathtt{rank}^{\downarrow}$ 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 **u** such that $\mathbf{s}.O^{\uparrow} \geq \mathbf{u}.O^{\downarrow}$, i.e., all tuples we have processed so far. We store the sum of the maximal multiplicity of these tuples in a variable $\mathtt{rank}^{\uparrow}$ which is updated for every incoming tuple. We use a function `emit` to compute **s**'s upper bound sort position, adapt $\mathbf{s}.\#^{\downarrow}$ (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 $\mathtt{rank}^{\downarrow}$ (all tuples processed in the following are certainly ranked higher than **s**). Function `split` splits a tuple with $\mathbf{t}.\# > 1$ into multiple tuple as required by Def. 2. If we are only interested in the top-k results, then we can stop processing the input once $\mathtt{rank}^{\downarrow}$ 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 2 defines function $split(\mathbf{t})$ split multiplicities of range tuple **t** into multiplicity of ones using the semantics in **??**.

Complexity Analysis. Let $n = |\mathbf{R}|$. The algorithm requires $O(n \cdot \log n)$ to sort the input. It then processes the data in one-pass. For each tuple, we compare $\mathbf{t}.O^{\uparrow}$ in $O(1)$ with the root of the heap and insert the tuple into the heap in $O(\log |heap|)$. Tuples are removed from the heap just once in $O(\log |heap|)$. In the worst-case, if the sort positions of all tuples may be less than k, then the heap will contain all n tuples at the end before flushing. Thus, $|heap|$ is bound n and we get $O(n \cdot \log n)$ as the worst-case runtime complexity for our algorithm requiring $O(n)$ memory. However, in practice, heap sizes are typically much smaller.

[2]The selected guess position τ^{sg} is trivially obtained using an additional linked heap, and omitted here for clarity.

```
Input: R (sorted by O↓), k ∈ ℕ (or k = |R|)
1  todo ← minheap(O↑) ; rank↓ ← 0 ; rank↑ ← 0 ; res ← ∅
2  for t ∈ R do
3  |  while todo.peek().O↑ < t.O↓ do                    // emit tuples
4  |  |  emit(todo.pop())
5  |  |  if rank↓ > k then          // tuples certainly out of top-k?
6  |  |  |  return res
7  |  t.τ↓ ← rank↓                          // set position lower bound
8  |  todo.insert(t)                          // insert into todo heap
9  |  rank↑ += t.#↑                       // update position upper bound
10 while not todo.isEmpty() do                // flush remaining tuples
11 |  emit(todo.pop())
12 return res

13 def emit(s)
14 |  s.τ↑ ← min(k, rank↑)   // position upper bound capped at k
15 |  if rank↑ > k then       // s may not be in result if s.τ↑ > k
16 |  |  s.#↓ ← 0
17 |  res ← res ∪ split({s})
18 |  rank↓ += s.#↓                       // update position lower bound
```

Algorithm 1: Non-deterministic sort on O (top-k)

```
1 Function split(t):
2 |  for i ∈ [1, t.t.#↑] do
3 |  |  t_i = copy(t) ;
4 |  |  t_i.t.#^sg = t.t.#^sg < i : 1?0 ;
5 |  |  t_i.t.#↑ = 1 ;
6 |  |  t_i.t.#↓ = t.t.#↓ < i : 1?0 ;
7 |  |  t_i.τ↑ += i ;
8 |  |  t_i.τ↓ += i ;
9 |  |  return t_i ;
```

Algorithm 2: Split bag tuple

7.2 Connected Heaps

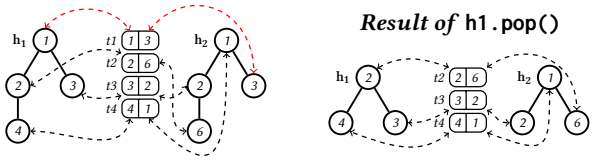
In our algorithm for windowed aggregation that we will present in sec. 7.3, we need to maintain the tuples possibly in a window ordered increasingly on $\tau^{\uparrow}$ (for fast eviction), sorted on $A^{\downarrow}$ to compute min-k($\mathbf{R}, A$), and sorted decreasingly on $A^{\uparrow}$ to compute max-k($\mathbf{R}, A$). 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. Furthermore, trees are typically are not as efficient in practice as heaps which can be implemented as arrays. 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. When a tuple is inserted into a connected heap, it is inserted into each component heap in $O(\log n)$ in the usual way with the exception that the backwards pointers are populated.

Preliminary experiments. To measure the impact of the back-pointers in connected heaps on performance, we did a preliminary experimental comparison with using a set of independent heaps. Without the backlinks, removing an non-root element from a heap is linear in the size of the heap in the worst-case, because it may require a search over the whole heap to find the position of such an element. Afterwards, the element can be deleted and the heap property can be restored in $O(\log n)$. Using the backlinks, finding the positions of an element in other heaps is $O(1)$ and so popping the root element of one heap and removing it from all other heaps is in $O((\log n) \cdot m)$ where n is the size of the largest heap and m is the number of heaps. The table below shows the execution times in milliseconds using connected heaps (back pointers) versus classical unconnected heaps (linear search). This experiment was run on a database with 50k tuples and 1%-5% uncertainty (amount of tuples that are uncertain) varying the size of the ranges for the attribute we are aggregating over. The main factor distinguish linear search performance from back pointers is the heap size which for our windowed aggregation algorithm is affected by attribute range size, percentage of tuples which have uncertain order-by and data size. Even though in this experiment the amount of uncertain data and database size are quite low, we already see 25% up to a factor of $\sim$ 10 improvement. For larger databases or larger percentage of uncertain data, the sizes of heaps will increase and, thus, we will see even more significant performance improvements.

Uncert	Range	Connected heaps (Back pointers)(ms)	Unconnected heaps (Linear search)(ms)
1%	2000	1979.272	3479.042
1%	15000	2045.162	6676.732
1%	30000	2103.974	9646.330
5%	2000	1976.651	4078.487
5%	15000	2149.990	15186.657
5%	30000	2191.823	22866.713

Deletion from a connected heap. When a node is popped from one of the component heaps the nodes of the other heaps storing the tuple are identified in $O(H)$ using the backwards pointers. Like in standard deletion of nodes from a heap, a deleted node is replaced with the right-most node at the leaf level. Standard sift-down and sift-up are then used to restore the heap property in $O(\log n)$. Recall that the heap property for a min-heap requires that for each node in the heap its value is larger than the value of its parent. Insertion of a new node v into a heap places the new element at the next free position at the leaf level. This may violate the heap property. The heap property can be restored in $O(\log n)$ using sift-up (repeatedly replacing a node with its parent). Similarly, to delete the root of a heap, we replace the root with the right-most child. This again may violate the heap property if the new root is larger than one of its children. The heap property can be restored in $O(\log n)$ steps using sift-down, i.e., replacing a node that is larger than a child with the smaller of its children. For a connected heap, deletion may cause a node to be deleted that is currently not the root of the heap. Like in standard heaps, we replace the node v to be deleted with the right-most node v_l from the leaf level. This may violate the heap property (every child is larger than its parent) in two possible ways: either v_l is smaller than the parent of v or v_l is larger than one of the children of v. Note that it is not possible for both cases to occur at the same time, because the heap was valid before and, thus, if v_l is larger than a child of v, then it has to be larger than the parent of v. If v_l is smaller than the parent of v, then it has to be smaller than all other nodes in the subtree rooted at v. We can restore the heap property by sifting up v_l. Now consider the case where v_l is larger than one of the children of v and let v_c denote that child (or the smaller child if v_l is larger than both children). Note that the subtree rooted at v was a valid heap. Thus, replacing v with v_l is replacing the root element of this subheap and the heap property for the subheap can be restored using sift-down. Since v_l is larger than the parent of v this restores the heap property for the whole heap.

Example 8 (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 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 mimicked by offsetting the window, i.e., a window bound of $[-N, 0]$. Algorithm 3 uses a function `compBounds` to compute the bounds on the aggregation function result based on the certain and possible content of a window. We discuss the code for these functions below for aggregation functions **min**, **max**, and **sum** (**count** uses the same algorithm as **sum** using $[1/1/1]$ instead of the values of an attribute A). Algorithm 3 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 $\text{SORT}_{O \to \tau}(\mathbf{R})$ and then has been sorted on $\tau^{\downarrow}$. This algorithm uses a minheap `openw` which is sorted on $\tau^{\uparrow}$ to store tuples for which have not 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 $\mathbf{t}$ that certainly exist and for which $\mathbf{t}.\tau^{\downarrow} = i$ sorted on $\tau^{\uparrow}$. This data structure is used to determine which tuples certainly belong to the window of a tuple; (`poss`, $\text{pagg}^{\downarrow}$, $\text{pagg}^{\uparrow}$) is a connected minheap with `poss`, $\text{pagg}^{\downarrow}$, and $\text{pagg}^{\uparrow}$ are sorted on $\tau^{\uparrow}$, $A^{\uparrow}$, $A^{\uparrow}$, 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 $\texttt{c-rank}^{\downarrow}$ for the lower bound of the certain part of windows.

Algorithm 3 first inserts each incoming tuple into `openw` (Line 7). If the tuple certainly exists, it is inserted into the tree of certain

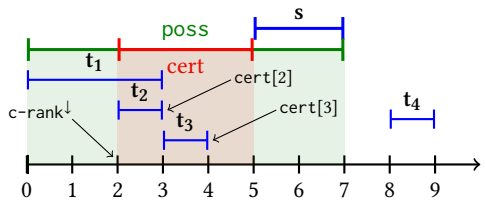


Figure 8: State example for Algorithm 3, N=5, c-rank$^{\downarrow}$=2.

Input: f, X,O, N, A, $\text{SORT}_{O\to\tau}(\text{R})$ **sorted on** $\tau^{\downarrow}$

```
1  openw ← minheap(τ↑)                          // tuples with open windows
2  cert ← Map(int, Tree(τ↑))                    // certain window members by pos.
3  (poss, pagg↓, pagg↑) ← connected-minheap(τ↑, A↓, A↑)
4  c-rank↓ ← 0                                  // watermark for certain window
5  res ← ∅
6  for t ∈ R do
7      openw.insert(t)
8      if t.#↓ > 0 then                         // insert into potential certain window
9          cert[t.τ↓].insert(t)
10     while openw.peek().τ↑ < t.τ↓ do          // close windows
11         s ← openw.pop()
12         while c-rank↓ < s.τ↑ − N do          // evict certain win.
13             cert[c-rank↓] = null
14             c-rank↓++
15         s.X ← compBounds(f, s, cert, poss)   // compute agg.
16         while poss.peek.τ↑ < s.τ↓ − N do     // evict poss. win.
17             poss.pop()
18         res ← res ∪ {s}
19     poss.insert(t)                           // insert into poss. win.
```

Algorithm 3: Aggregate $f(A) \to X$, sort on O, N preceding

```
1  def compBounds(f, t, cert, poss)   // compute bounds on sum(A)
2      if f = sum then
3          return computeSumBounds(t,cert, poss)
4      if f = min then
5          return computeMinBounds(t,cert, poss)
6      if f = max then
7          return computeMaxBounds(t,cert, poss)
8      if f = count then
9          return computeCountBounds(t,cert, poss)
10     if f = avg then
11         return computeMinBounds(t,cert, poss)
```

Algorithm 4: Computing bounds for $f(A) \to X$ for tuple **t**

```
1  def ComputeSumBounds(t, cert, poss)      // compute bounds on sum(A)
2      n ← N − 1; X↓ ← t.A↑; X↑ ← t.A↑      // pos. and bounds
3      for x ∈ [t.τ↑ − N, t.τ↓] do
4          for s ∈ cert[x] do
5              if s.τ↑ ≤ t.τ↓ then          // belongs to cert. window of s
6                  X↑ += s.A↑; X↓ += s.A↓
7                  n − −
8              else break;
9      lb_poss ← copy(pagg↓); ub_poss ← copy(pagg↑)
10     n_lb ← n; n_ub ← n                   // max. num. of tuples possibly in win.
11     while n_lb > 0 ∧ ¬lb_poss.isEmpty() do   // compute X↓
12         s ← lb_poss.pop()
13         if s.A↓ < 0 then                 // only values < 0 contribute to X↓
14             X↓ += s.A↓
15             n_lb − −
16         else break;
17     while n_ub > 0 ∧ ¬ub_poss.isEmpty() do   // compute X↑
18         s ← ub_poss.pop()
19         if s.A↑ > 0 then                 // only values > 0 contribute to X↑
20             X↑ += s.A↑
21             n_ub − −
22         else break;
23     return [X↓, X↑]
```

Algorithm 5: Computing bounds for $\textbf{sum}(A) \to X$ for tuple **t**

```
1  def computeMinBounds(t, cert, poss)      // compute bounds on min(A)
2      n ← N − 1; X↓ ← t.A↑; X↑ ← t.A↑      // pos. and bounds
3      for x ∈ [t.τ↑ − N, t.τ↓] do
4          if cert[x] then                  // min of certain lower-bound
5              X↑ ← x
6              break
7      X↓ = pagg↓.peek().A↓                 // min of possible lower-bound
8      return [X↓, X↑]
```

Algorithm 6: Computing bounds for $\textbf{min}(A) \to X$ for tuple **t**

tuples whose lower bound position is $\mathbf{t}.\tau^{\downarrow}$. Note that each of these trees is sorted on $\tau^{\uparrow}$ 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 ($\mathbf{s}.\tau^{\uparrow} < \mathbf{t}.\tau^{\downarrow}$). To see why this is the case first observe that (i) we are processing input tuples in increasing order of $\tau^{\downarrow}$ and (ii) tuples are "finalized" by computing the aggregation bounds in monotonically increasing order of $\tau^{\uparrow}$. 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 $\mathbf{s}.\tau^{\downarrow} \leq \mathbf{t}.\tau^{\uparrow}$. Based on these observations, once we processed a tuple **t** with $\mathbf{t}.\tau^{\downarrow} > \mathbf{s}.\tau^{\uparrow}$ for a tuple **s** in openw, we know that no tuples that we will processed 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$^{\downarrow}$ (Line 12). Recall that for a tuple **u** to certainly belong to the window for **s** we have to have $\mathbf{s}.\tau^{\uparrow} - N \geq \mathbf{t}.\tau^{\downarrow}$. Thus, we update c-rank$^{\downarrow}$ to $\mathbf{s}.\tau^{\uparrow} - N$ and evict from cert all trees storing tuples for sort positions smaller than $\mathbf{s}.\tau^{\uparrow} - N$. 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, evict tuples from poss (and, thus, also pagg$^{\downarrow}$ and pagg$^{\uparrow}$) 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 $\mathbf{u}.\tau^{\uparrow} < \mathbf{s}.s^{\downarrow} - N$ (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 $\tau^{\uparrow}$, i.e., for all tuples **u** emitted after **s** we have $\mathbf{u}.\tau^{\uparrow} > \mathbf{s}.\tau^{\uparrow}$. Fig. 8 shows an example state for the algorithm when tuple **s** is about to be emitted. Tuples fully included in the red region ($\mathbf{t}_2$ and $\mathbf{t}_3$) are currently in cert$[i]$ for sort positions certainly in the window for **s**. Tuples with sort position ranges overlapping with green region are in the possible window (these tuples are stored in poss). Tuples like $\mathbf{t}_4$ 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. This process is the same as emitting tuples before we have processed all inputs and, thus, is omitted form Algorithm 3.

Algorithm 3 uses function compBounds to compute the bounds on the aggregation function result for a tuple $\mathbf{t}$ using cert, $\text{pagg}^{\downarrow}$ and $\text{pagg}^{\uparrow}$ following the definition from sec. 6.1.

Complexity Analysis. Algorithm 3 first sorts the input in $O(n \log n)$ time using Algorithm 1 followed by a deterministic sort on $\tau^{\downarrow}$. Each tuple is inserted into openw, poss, and cert at most once and poped from openw exactly once. The size of the heaps the algorithm maintains is certainly less than n at all times. To compute the aggregation function bounds, we have to look at the certain tuples in $\text{cert}[i]$ at most $size([N, 0]) = N + 1$ sort positions i and at most $N + 1$ tuples from poss that can be accessed using the connected heaps in $O(N \cdot \log n)$. Thus, the overall runtime of the algorithm is $O(N \cdot n \cdot \log n)$.

8 EXPERIMENTS

We evaluate the efficiency of our rewrite-based approach and the native implementation of the algorithms presented in Sec. 7 in Postgres and the accuracy the of 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 tests, we treat the highest and lowest possible value for all samples as the upper and lower bounds and compare against the tight bounds produced by the compared algorithms (since computing optimal bounds is often intractable). Given a tightest bound $[c, d]$, we define the recall of a bound $[a, b]$ as $\frac{min(b,d)-max(a,c)}{d-c}$ and the accuracy of $[a, b]$ as $\frac{max(b,d)-min(a,c)}{min(b,d)-max(a,c)}$. The recall/accuracy for a relation is then the average recall/accuracy of all tuples. *PT-k* [32] only supports sorting and returns all answers with a probability larger than a user-provided threshold of being among the top-k answers. By setting the threshold to 1 (0) we can use this approach to compute all certain (possible) answers. *Symb* represents aggregation results, rank of tuples, and window membership as symbolic expressions which compactly encode the incomplete database produced by possible world semantics using the model from [12] for representing aggregation results and a representation similar to [9] to encode uncertainty in the rank of tuples. We use an SMT solver (Z3 [20]) to compute tight bounds on the possible ranks / aggregation results for tuples. *Rewr* is a rewrite-based approach we implemented uses self-unions for sorting queries and self-joins for windowed aggregation queries. *Imp* is the native implementation of our algorithms in Postgres. All experiments are run on a 2×6 core 3300MHz 8MB cache AMD Opteron 4238 CPUs, 128GB RAM, 4×1TB 7.2K HDs (RAID 5) with the exception of *PT-k* which was provided by the authors as a binary for Windows only. We run *PT-k* on a separate Windows machine with a 8 core 3800MHz 32MB cache AMD Ryzen 5800x CPU, 64G RAM, 2TB HD. Because the *PT-k* implementation is single-threaded and in-memory, we consider our comparisons are in favor of *PT-k*. We implement our algorithms as an extension for Postgres 13.3 and evaluate all algorithms on Postgres. We report the average of 10 runs.

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 distributed. Except where noted, we default to 50k rows and 5% uncertainty with maximum 1k attribute range on uncertain values.

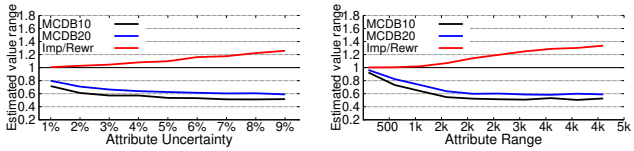
8.1.1 Sorting and Top-k Queries. **Scaling Data Size.** Fig. 12 shows the runtime of sorting, varying the dataset size. Since *Symb* and *PT-k* perform significantly worse, we only include these methods for smaller datasets (Fig. 12a). MCDB and our techniques significantly outperform *Symb* and *PT-k* (~2+ OOM). *Rewr* is roughly on par with *MCDB20* while *Imp* outperforms *MCDB10*. Given their poor performance and their lack of support for windowed aggregation, we exclude *Symb* and *PT-k* from the remaining microbenchmarks.

Varying k, Ranges, and Rate. Fig. 9 shows runtime of top-k (k is specified) and sorting queries (k is not specified) when varying (i) the number of tuples returned k, (ii) the size of the ranges of uncertain order-by attributes (*range*), and (iii) the fraction of tuples with uncertain order-by attributes. *Imp* is the fastest method, with an overhead of deterministic query processing between 3.5 (top-k) and 10 (full sorting). *Rewr* has higher overhead over *Det* than *MCDB*. Notably, the performance of *MCDB* and *Rewr* is independent of all three varied parameters. Uncertainty and *range* have small impact on the performance of *Imp* while computing top-k results is significantly faster than full sorting when k is small.

Configurations	*Det*	*Imp*	*Rewr*	*MCDB10*	*MCDB20*
r=1k,u=5%	31.5ms	233.1ms	786.7ms	310.1ms	639.3ms
r=10k,u=5%	30.9ms	286.1ms	792.6ms	314.3ms	621.2ms
r=1k,u=20%	31.8ms	266.3ms	794.9ms	325.8ms	651.2ms
r=1k,u=5%,k=2	13.4ms	48.3ms	750.4ms	149.1ms	295.2ms
r=1k,u=5%,k=10	13.4ms	48.2ms	751.1ms	150.4ms	296.1ms

Range(r),Uncertainty(u),k or full sorting

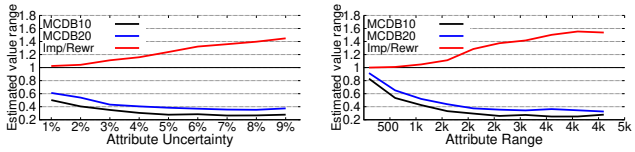
Figure 9: Sorting and Top-K Microbenchmarks - Performance



(a) Varying uncertainty **(b) Varying range**

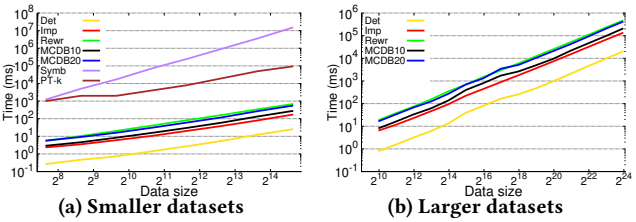
Figure 10: Sorting microbenchmarks - approximation quality

Accuracy. Fig. 10 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 related 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



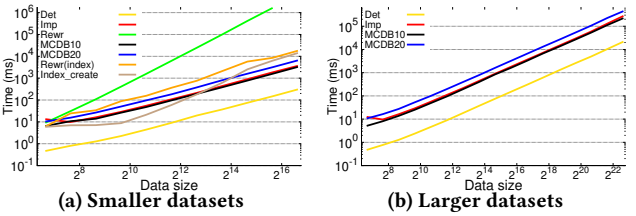
(a) Varying uncertainty (b) Varying range

Figure 11: Window microbenchmarks - approximation quality



(a) Smaller datasets (b) Larger datasets

Figure 12: Sorting performance varying dataset size



(a) Smaller datasets (b) Larger datasets

Figure 13: Windowed aggregation performance varying dataset size

Configurations		*Det*	*Imp*	*MCDB10*	*MCDB20*
Order-by + Window size	w=3,r=1k,u=5%	85.3ms	895.3ms	948.6ms	1850.4ms
	w=3,r=10k,u=5%	87.1ms	899.7ms	931.3ms	1877.5ms
	w=3,r=1k,u=20%	88.7ms	903.2ms	944.7ms	1869.7ms
	w=6,r=1k,u=5%	86.2ms	1008.3ms	953.1ms	1885.1ms

(a) Order-by, Window size (w), Range (r), Uncertainty (u)

Configurations		*Det*	*Rewr*	*MCDB10*	*MCDB20*
Order-by	w=3,r=1k,u=5%	105.1ms	73.5s	1209.4ms	2127.1ms
+ Partiton-by	w=3,r=10k,u=5%	101.7ms	75.2s	1231.3ms	2142.9ms
+ Window size	w=3,r=1k,u=20%	104.2ms	81.1s	1201.1ms	2102.3ms

(b) Order-by + partition-by, Window size (w), Range (r), Uncertainty (u)

Figure 14: Windowed aggregation microbenchmarks - Performance

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. 13 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 of *Imp* is roughly on par with *MCDB10*. *Rewr(Index)* is almost as fast as *MCDB20*, but is 5 × slower than *Imp*.

Varying window spec, Ranges, and Rate. Fig. 14 shows the runtime of windowed aggregation varying attribute uncertain value ranges (on all columns), percentage of uncertain tuples, and window size. For *Imp* (Fig. 14a) we use a query without partition-by. We also compare runtime of our rewriting based approach (Fig. 14b) using both partition-by and order-by on 8k rows. *Imp* exhibits similar runtime to *MCDB10* and outperforms *MCDB20*. Doubling the window size have only a slight impact (about 10%) on our

Datasets & Queries		*Imp* (time)	*Det* (time)	*MCDB20* (time)	*Rewr* (time)	*Symb* (time)	*PT-k* (time)
Iceberg [3]	Rank	0.816msms	0.123ms	2.337ms	1.269ms	278ms	1s
(1.1%, 167K)	Window	2.964ms	0.363ms	7.582ms	1.046ms	589ms	N.A.
Crimes [4]	Rank	1043.505ms	94.306ms	2001.12ms	14787.723ms	>10min	>10min
(0.1%, 1.45M)	Window	3.050ms	0.416ms	8.337ms	2.226ms	>10min	N.A.
Healthcare [1]	Rank	287.515ms	72.289ms	1451.232ms	4226.260ms	15s	8s
(1.0%, 171K)	Window	130.496ms	15.212ms	323.911ms	13713.218ms	>10min	N.A.

Figure 15: Real world data - performance

Datasets & Measures		*Imp/Rewr*	*MCDB20*	*PT-k/Symb*
Iceberg	bound accuracy	0.891	1	1
[3]	bound recall	1	0.765	1
Crimes	bound accuracy	0.996	1	1
[4]	bound recall	1	0.919	1
Healthcare	bound accuracy	0.990	1	1
[1]	bound recall	1	0.767	1

Figure 16: Real world data - sort position accuracy and recall

Datasets & Methods		Grouping/Order accuracy	Grouping/Order recall	Aggregation accuracy	Aggregation recall
Iceberg [3]	*Imp/Rewr*	0.977	1	0.925	1
	MCDB20	1	0.745	1	0.604
	Symb	1	1	1	1
Crimes [4]	*Imp/Rewr*	0.995	1	0.989	1
	MCDB20	1	0.916	1	0.825
	Symb	1	1	1	1
Healthcare [1]	*Imp/Rewr*	0.998	1	0.998	1
	MCDB20	1	0.967	1	0.967
	Symb	1	1	1	1

Figure 17: Real world data - windowed aggregation accuracy and recall

implementation performance. *Rewr* is slower than *MCDB* by several magnitudes due to the range-overlap join. Our techniques are not significantly affected by the range and uncertainty rate.

8.2 Real World Datasets

We evaluate our approach on real datasets (Iceberg [3], Chicago crime data [4], and Medicare provide 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. 15 shows the performance of real queries on these datasets reporting basic statistics (uncertainty and #rows).

We use the following queries.

iceberg.

Find top 3 sizes of ice-bergs mostly observed.

```
SELECT size,count(*) AS ct FROM iceberg
GROUP BY size
ORDER BY ct DESC LIMIT 3;
```

Window: For each day, find rolling sum of number of icebergs observed on that day and following 3 days.

```
SELECT date, sum(number) OVER (ORDER BY date
  BETWEEN CURRENT ROW AND 3 FOLLOWING) AS r_sum
FROM iceberg;
```

Crimes.

Rank: Find top three days with most incidents of crimes.

```
SELECT date, count(*) AS ct
  FROM crimes GROUP BY date
  ORDER BY ct DESC LIMIT 3;
```

Window: For each crime in 2016, find the earliest year among the crime itself and nearest crime at north and south of it.

```
SELECT rid, min(year) OVER
  (ORDER BY latitude BETWEEN
    1 PRECEDING AND 1 FOLLOWING) AS min_year
```

```
FROM crimes WHERE year='2016';
```

healthcare.
Rank: Find top 5 facility with highest score on MRSA Bacteremia.

```
SELECT facility_id,facility_name,score FROM healthcare
WHERE measure_name = 'MRSA Bacteremia'
ORDER BY score LIMIT 5;
```

Window: get in-line rank of facility on MRSA Bacteremia scores.

```
SELECT facility_id,facility_name,count(*) OVER
(ORDER BY score DESC) AS rank
FROM healthcare
WHERE measure_name = 'MRSA Bacteremia';
```

For sorting and top-k queries that contain aggregation which commonly seen in real use-cases, we only measure the performance of the sorting/top-k part over pre-aggregated data (see [24] for an evaluation of the performance of aggregation over AU-DBs) In general, our approach (*Imp*) is faster than *MCDB20*. *Symb* and *PT-k* are significantly more expensive. Fig. 16 shows the approximation quality for our approach and *MCDB*. Our approach has precision close to 100% except for sorting on the Iceberg dataset which has a larger fraction of uncertain tuples and wider ranges of uncertain attribute values due to the pre-aggregation. *MCDB* has lower recall on Iceberg and Healthcare sorting queries since these two datasets have more uncertain tuples (10 times more than the Crimes dataset). Fig. 17 shows the approximation quality of our approach and *MCDB* for windowed aggregation queries. We measured both the approximation quality of grouping of tuples to windows and for the aggregation result values. For Crimes and Iceberg, the aggregation accuracy is affected by the partition-by/order-by attribute accuracy and the uncertainty of the aggregation attribute itself. The healthcare query computes a count, i.e., there is no uncertainty in the aggregation attribute and approximation quality is similar to the one for sorting. Overall, we provide good approximation quality at a significantly lower cost than the two exact competitors.

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 under full relational algebra with aggregation, and is implemented as efficient one-pass algorithms in Postgres. Our approach significantly outperforms existing algorithms for ranking uncertain data while being applicable to more expressive queries and bounding all certain and possible query answers. Thus, our approach enables the efficient evaluational of complex queries involving sorting over incomplete databases. We present a SQL-based implementation as well as the aforementioned one-pass algorithms. Using an implementation of these algorithms in Postgres, we demonstrate that our approach significantly outperforms the SQL-based implementation and for windowed aggregations we have performance close to sampling based approach with 10 samples while 10 sample produces low recall comparing with the accuracy our approach have. Furthermore, our approach significantly outperforms an existing algorithm for ranking uncertain data while being applicable to more expressive queries and bounding all certain and possible query answers. In future work, we plan to extend our approach to deal more expressive classes of queries, e.g., recursive queries, and will investigate index structures for AU-DBs to further improve performance.

REFERENCES

[1] Medicare hospital dataset. https://data.medicare.gov/data/hospital-compare.
[2] Paper artifacts. https://github.com/fengsu91/uncert-ranking-availability.
[3] Iceberg dataset. https://nsidc.org/data/g00807.
[4] Chicago crimes dataset. https://www.kaggle.com/currie32/crimes-in-chicago.
[5] S. Abiteboul, T.-H. H. Chan, E. Kharlamov, W. Nutt, and P. Senellart. Aggregate queries for discrete and continuous probabilistic xml. In *ICDT*, pages 50–61, 2010.
[6] S. Abiteboul, P. C. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):158–187, 1991.
[7] P. Agrawal, A. D. Sarma, J. Ullman, and J. Widom. Foundations of uncertain-data integration. *PVLDB*, 3(1-2):1080–1090, 2010.
[8] R. Albright, A. J. Demers, J. Gehrke, N. Gupta, H. Lee, R. Keilty, G. Sadowski, B. Sowell, and W. M. White. SGL: a scalable language for data-driven games. In *SIGMOD Conference*, pages 1217–1222. ACM, 2008.
[9] A. Amarilli, M. L. Ba, D. Deutch, and P. Senellart. Provenance for nondeterministic order-aware queries. *Prepr int: http://a3nm. net/publications/a-marilli2014provenance. pdf*, 2014.
[10] A. Amarilli, M. L. Ba, D. Deutch, and P. Senellart. Possible and certain answers for queries over order-incomplete data. In *Proc. TIME*, pages 4:1–4:19, Mons, Belgium, oct 2017.
[11] A. Amarilli, M. L. Ba, D. Deutch, and P. Senellart. Computing possible and certain answers over order-incomplete data. *Theor. Comput. Sci.*, 797:42–76, 2019.
[12] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.
[13] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. Sampling from repairs of conditional functional dependency violations. *VLDBJ*, 23(1):103–128, 2014.
[14] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Müller, S. Castel, C. Bautista, and J. Freire. Your notebook is not crumby enough, replace it. In *CIDR*, 2020.
[15] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Olap over uncertain and imprecise data. *VLDBJ*, 16(1):123–144, 2007.
[16] A. L. P. Chen, J.-S. Chiu, and F. S.-C. Tseng. Evaluating aggregate operations over imprecise data. *IEEE Trans. Knowl. Data Eng.*, 8(2):273–284, 1996.
[17] M. Console, P. Guagliardo, and L. Libkin. Fragments of bag relational algebra: Expressiveness and certain answers. In *ICDT*, pages 8:1–8:16, 2019.
[18] M. Console, P. Guagliardo, L. Libkin, and E. Toussaint. Coping with incomplete data: Recent advances. In *PODS*, pages 33–47. ACM, 2020.
[19] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 305–316, 2009.
[20] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
[21] W. Fan. Dependencies revisited for improving data quality. In *PODS*, pages 159–170, 2008.
[22] S. Feng, B. Glavic, and O. Kennedy. Efficient approximation of certain and possible answers for ranking and window queries over uncertain data (extended version). 2022.
[23] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - a lightweight approach for approximating certain answers. In *SIGMOD*, 2019.
[24] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Efficient uncertainty tracking for complex queries with attribute-level bounds. In *Proceedings of the 46th International Conference on Management of Data*, page 528 – 540, 2021.
[25] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5):490–501, 2012.
[26] S. Grafberger, P. Groth, and S. Schelter. Towards data-centric what-if analysis for native machine learning pipelines. In *DEEM@SIGMOD*, pages 3:1–3:5. ACM, 2022.
[27] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.
[28] P. Guagliardo and L. Libkin. Making sql queries correct on incomplete databases: A feasibility study. In *PODS*, 2016.
[29] P. Guagliardo and L. Libkin. Correctness of sql queries on databases with nulls. *SIGMOD Record*, 46(3):5–16, 2017.
[30] P. Guagliardo and L. Libkin. On the codd semantics of sql nulls. *Inf. Syst.*, 86:46–60, 2019.

[31] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: the teenage years. In *VLDB*, pages 9–16, 2006.

[32] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 673–686, New York, NY, USA, 2008. Association for Computing Machinery.

[33] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[34] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[35] T. S. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, pages 346–355, 2007.

[36] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *PERVASIVE*, pages 83–100, 2006.

[37] O. Kennedy and C. Koch. Pip: A database system for great and small expectations. In *ICDE*, pages 157–168, 2010.

[38] N. Kline and R. Snodgrass. Computing temporal aggregates. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 222–231, 1995.

[39] P. Kumari, S. Achmiz, and O. Kennedy. Communicating data quality in on-demand curation. In *QDB*, 2016.

[40] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *SIGMOD*, pages 1275–1286, 2014.

[41] J. Lechtenbörger, H. Shu, and G. Vossen. Aggregate queries over conditional tables. *J. Intell. Inf. Syst.*, 19(3):343–362, 2002.

[42] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *Proc. VLDB Endow.*, 2(1):502–513, aug 2009.

[43] X. Liang, Z. Shang, S. Krishnan, A. J. Elmore, and M. J. Franklin. Fast and reliable missing data contingency analysis with predicate-constraints. In *SIGMOD*, pages 285–295, 2020.

[44] L. Libkin. Sql's three-valued logic and certain answers. *TODS*, 41(1):1:1–1:28, 2016.

[45] W. Lipski. On semantic issues connected with incomplete information databases. *TODS*, 4(3):262–296, 1979.

[46] B. Moon, I. Lopez, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 145–154, 2000.

[47] R. Murthy, R. Ikeda, and J. Widom. Making aggregation work in uncertain and probabilistic databases. *IEEE Trans. Knowl. Data Eng.*, 23(8):1261–1273, 2011.

[48] D. Olteanu, L. Papageorgiou, and S. J. van Schaik. Pigora: An integration system for probabilistic data. In *ICDE*, pages 1324–1327, 2013.

[49] D. Piatov and S. Helmer. Sweeping-based temporal aggregation. In M. Gertz, M. Renz, X. Zhou, E. Hoel, W.-S. Ku, A. Voisard, C. Zhang, H. Chen, L. Tang, Y. Huang, C.-T. Lu, and S. Ravada, editors, *Advances in Spatial and Temporal Databases*, pages 125–144, Cham, 2017. Springer International Publishing.

[50] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 886–895, 2007.

[51] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *J. ACM*, 33(2):349–370, 1986.

[52] B. Salimi, R. Pradhan, J. Zhu, and B. Glavic. Interpretable data-based explanations for fairness debugging. In *SIGMOD*, pages 247–261, 2022.

[53] S. Sarawagi et al. Information extraction. *Foundations and Trends® in Databases*, 1(3):261–377, 2008.

[54] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Probabilistic top-k and ranking-aggregate queries. *TODS*, 33(3):13:1–13:54, 2008.

[55] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Probabilistic top-k and ranking-aggregate queries. *ACM Trans. Database Syst.*, 33(3), Sept. 2008.

[56] M. A. Soliman, I. F. Ilyas, and K. Chen-Chuan Chang. Top-k query processing in uncertain databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 896–905, 2007.

[57] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[58] B. Sundarmurthy, P. Koutris, W. Lang, J. F. Naughton, and V. Tannen. m-tables: Representing missing data. In *ICDT*, 2017.

[59] P. Tuma. *Implementing Historical Aggregates in Tempis*. Wayne State University, 1993.

[60] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings 17th International Conference on Data Engineering*, pages 51–60, 2001.

[61] M. Yang, H. Wang, H. Chen, and W.-S. Ku. Querying uncertain data with aggregate constraints. In *SIGMOD*, pages 817–828, 2011.

[62] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: An on-demand approach to etl. *PVLDB*, 8(12):1578–1589, 2015.

[63] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, page 237–245, New York, NY, USA, 2001. Association for Computing Machinery.

[64] X. Zhang and J. Chomicki. On the semantics and evaluation of top-k queries in probabilistic databases. In *2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 556–563, 2008.