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.

PVLDB Reference Format:
Su Feng, Boris Glavic, and Oliver Kennedy. Efficient Approximation of Certain and Possible Answers for Ranking and Window Queries over Uncertain Data. PVLDB, 16(6): 1346 - 1358, 2023.
doi:10.14778/3583140.3583151

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

1 INTRODUCTION

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

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

In [23, 24], we started addressing the linked challenges of computational complexity, closure, and expressiveness in incomplete database systems, by proposing **AU-DBs**, an approach to uncertainty management that can be competitive with deterministic query processing. Rather than trying to encode a set of possible worlds losslessly, each AU-DB tuple is defined by one range of possible values for each of its attributes and a range of (bag) multiplicities. Each tuple of an AU-DB is a hypercube that bounds a region of the attribute space, and together, the tuples bound the set of possible worlds between an *under-approximation of certain answers* and an *over-approximation of possible answers*. This model is closed under relational algebra [23] with aggregation [24] ($\mathcal{RA}^{agg}$). 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 factors for our choice to extend this model.

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

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

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583151

combined with existing AU-DB rewrites [23, 24], any $\mathcal{RA}^{agg}$ 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 in all possible worlds), and highest possible position (the number of tuples that possibly precede it in at least one possible world). We can naively compute a lower (resp., upper) bound by joining every tuple t with every other tuple, counting pairs where t is certainly (resp., possibly) preceded by the tuple it is paired with. We refer to this approach as the *rewrite method*, as it can be implemented in SQL. However, the rewrite approach has quadratic runtime. Inspired by techniques for aggregation over interval-temporal databases such as [47], we propose a one-pass algorithm to compute the bounds on a tuple's position that also supports top-k queries.

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

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

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.*

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 terms high-lighted and the result of the rolling sum of sales for the current and next term.

Term
4
3

(b) U-Top

Term
4
4

(c) U-Rank

Term
3
4
5

(d) PT(0)

Term
4

(e) PT(1)

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

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

Figure 1: Ranking, top-k, and windowed aggregation queries over an incomplete (probabilistic) database and a bounding AU-DBs.

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 heuristic data cleaning systems that return one repair (possible world) from the space of all repairs [14, 38]. 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 admits additional worlds with 5 sales in term 4.*

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

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

[1] The process of obtaining a selected-guess world is domain-specific, but [23, 24] suggest the most likely world, if it can be feasibly obtained.

new window one row enters the window and one row leaves. Sum-based aggregates (sum, count, avg) can leverage commutativity and associativity of addition, i.e., updating the window requires only constant time. Similar techniques [8] can maintain 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 to iterate over tuples sorted on both the upper- and lower-bounds of their possible sort positions. 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 AU-DB aggregation semantics from [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 in decreasing (increasing) order of their upper (lower) bound to find the k-subset with the minimal/maximal aggregation result. Furthermore, we have to separately maintain tuples that certainly belong to a window (which contribute to both bounds). To maintain sets of tuples such that they can be accessed in several sort orders efficiently, we develop a 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 on 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.*

2 RELATED WORK

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

Probabilistic/Incomplete databases. Certain answer semantics [6, 28, 29, 33, 43, 44] 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, 43, 49] 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 only obtain probabilistic bounds on their estimates. Queries over symbolic models for incomplete data like C-tables [33] and m-tables [56] 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, 42, 45, 52, 57]. These result encodings are not closed (i.e., not useful for subsequent queries), and are also expensive to compute (often NP-hard). Symbolic models [12, 25, 40] that are closed under aggregation permit PTIME data complexity, but extracting certain / possible answers is still intractable. We proposed AU-DBs [24] which are closed under $\mathcal{RA}^{agg}$ 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 [54] picks the top-k set with the highest probability. U-Rank [54] assigns to each rank the tuple which is most-likely to have this rank. Global-Topk [59] first ranks tuples by their probability of being in the top-k and returns the k most likely tuples. Probabilistic threshold top-k (PT-k) [32] returns all tuples that have a probability of being in the top-k that exceeds a pre-defined threshold. Expected rank [19] calculates the expected rank for each tuple across all possible worlds and picks the k tuples with the highest expected rank. Ré et al. [48] proposed a multi-simulation algorithm that stops when a guaranteed top-k probability can be guaranteed. Soliman et al. [53] proposed a framework that integrates tuple retrieval, grouping, aggregation, uncertainty management, and ranking in a pipelined fashion. Each of these generalizations necessarily breaks some intuitions about top-k, producing more (or fewer) than k tuples, or producing results that are not the top-k in any world.

Uncertain Order. Amarilli et. al. extends the relational model with a partial order to 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.

3 NOTATION AND BACKGROUND

A database schema $\textsc{Sch}(D) = \{\text{Sch}(R_1), \ldots, \text{Sch}(R_n)\}$ is a set of relation schemas $\text{Sch}(R_i) = (A_1, \ldots, A_n)$. Use $arity(\textsc{Sch}(R))$ to denote the number of attributes in $\textsc{Sch}(R)$. An instance D for schema $\textsc{Sch}(D)$ is a set of relation instances with one relation per schema in $\textsc{Sch}(D)$: $D = \{R_1, \ldots, R_n\}$. Assuming a universal value domain $\mathbb{D}$, a tuple with schema $\textsc{Sch}(R)$ is an element from $\mathbb{D}^{arity(\textsc{Sch}(R))}$.

A $\mathcal{K}$-relation [27] annotates each tuple with an element of a (commutative) semiring. In this paper, we focus on $\mathbb{N}$-relations. An $\mathbb{N}$-relation of arity n is a function that maps each tuple ($\mathbb{D}^n$) in the relation to an annotation in $\mathbb{N}$ representing the tuple's multiplicity. Tuples not in the relation are mapped to multiplicity 0. $\mathbb{N}$-relations are required to have finite support (tuples not mapped to 0). Since $\mathcal{K}$-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

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

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

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

over $\mathcal{K}$-relations as shown in Fig. 2. Notably for us, positive bag-relational algebra is equivalent to $\mathcal{K}$-relational semantics for the natural numbers semiring $\mathbb{N} = (\mathbb{N}, +, \times, 0, 1)$.

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:

$$\textsc{cert}_{\mathbb{N}}(\mathcal{R}, t) := \min(\{R(t) \mid R \in \mathcal{R}\})$$
$$\textsc{poss}_{\mathbb{N}}(\mathcal{R}, t) := \max(\{R(t) \mid R \in \mathcal{R}\})$$

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, a tuple is not defined as a point in $\mathbb{D}^n$, but rather as a bounding hypercube specified as upper and lower bounds (and a selected-guess) for each attribute value. Every such hypercube represents 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 possible world fall into the range annotating the hypercubes. 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 distinguished 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. 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 $(\mathbb{D}_I)^n \times \mathbb{D}^n \rightarrow \mathbb{N}$ 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 equality) 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 $\mathrm{Sch}(R)$). We first introduce operators for windowed aggregation, because sorting can be defined as a special case of windowed aggregation.

4.1 Windowed Aggregation

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

Example 4 (Row-Based Windows). *Consider the bag relation below and consider the windowed aggregation $sum(B)$ 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	ℕ
a	5	3	3
b	3	1	1
b	3	4	1

A	B	C	sum(B)	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

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

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

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

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

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

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

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

Figure 3: Windowed Aggregation

$A \subseteq \mathrm{Sch}(R)$, 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 multiplicity of tuple t' in the partition for tuple t (denoted as $\mathcal{P}_{R,t}(t')$), the range of 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 window for the i^{th} duplicate of t is the size of the overlap between the bounds $bounds(R, t, i)$, and the cover of t'.

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 $<_O^{total}$. This operator is just "syntactic sugar" as it can be expressed using windowed aggregation.

Definition 1 (Sort Operator). *Consider a relation R with schema $(A_1, \ldots, A_n)$, list of attributes $O = (B_1, \ldots, B_m)$ where each B_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-DB 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 create uncertainty in a tuple's position in the sort order. The former, because it determines how many duplicates of a tuple appear in the sort order which affects the position of tuples which may be larger wrt. the sort order and the latter because it affects which tuples are smaller than a tuple wrt. the sort order. As mentioned before, a top-k query is a selection over the result of a sort operator which checks that the sort position of a tuple is less than or equal to k. A bound-preserving semantics for selection was already presented in [24]. Thus, we focus on sorting and use the existing selection semantics for top-k queries.

Comparison of Uncertain Values. Before introducing sorting over AU-DBs, we first discuss the evaluation of $<_O$ over tuples with uncertain values (recall that $<_O^{total}$ is defined in terms of $<_O$). Per [24], a Boolean expression over range-annotated values evaluates to a bounding 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 after $\mathbf{t_2}$ (if $\mathbf{t_1}.A$ is 3).

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

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

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

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.

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

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

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

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

Figure 4: Range-annotated sort operator semantics.

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 duplicates that do not certainly exist (in some possible world there may be less than i duplicates of the tuple), but are in the selected-guess world (the selected-guess world has i or more duplicates of the tuple). Finally, the possible multiplicity is always 1.

Definition 2 (AU-DB Sorting Operator). *Let* $\mathbf{R}$ *be an AU-DB relation and* $O \subseteq \mathrm{Sch}(\mathbf{R})$. *The result of applying the sort operator* $\textsc{sort}_{O\to\tau}$ *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 $\mathrm{pos}(\mathbf{R}, O, \mathbf{t}', i)$ 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 5 (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 *(*$\textsc{sort}_{A,B\to pos}(\mathbf{R})$*) 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 its* 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 $\textsc{sort}_{O\to\tau}$ applied to $\mathbf{R}$ bounds the result of $\textsc{sort}_{O\to\tau}$ 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*

$O \subseteq \mathrm{Sch}(\mathcal{R})$. *We have:*

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

Proof Sketch: We prove the theorem by taking a tuple matching $\mathcal{TM}$ for each possible world R in the input (that is guaranteed to exist, because $\mathcal{R} \sqsubset \mathbf{R}$) and construct a tuple matching $\mathcal{TM}'$ for the output of sorting based on which $\textsc{sort}_{O\to\tau}(\mathcal{R}) \sqsubset \textsc{sort}_{O\to\tau}(\mathbf{R})$ holds. In the proof we make use of the fact that the sort operator distributes the multiplicity of an input tuple t to multiple output tuples which each are extensions of t with a sort position, keeping all other attributes the same as in the input. ■

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 additional possible tuple can belong to the window.

6.1 Windowed Aggregation Semantics

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

Partitions We start by defining AU-DB relation $\mathcal{P}_{\mathbf{t}}(\mathbf{R})$ which encodes the multiplicity of tuple $\mathbf{t}'$ in 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, or possible multiplicity) of a tuple to 0 if the tuple possibly (in the selected-guess world, or certainly) does not fulfill the selection condition.

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

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 bound 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 $\mathbf{t}.\tau^{\downarrow} + u$ ($4 + 4 = 8$ 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 $\mathrm{size}([l, u])$ to denote the size of a window with bounds $[l, u]$, i.e., $\mathrm{size}([l, u]) = (u - l) + 1$. 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 $\mathrm{size}([l, u]) - m$ rows possibly in the window that minimizes (for the lower aggregation result bound) or maximizes (for the upper

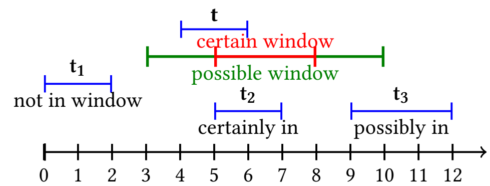


Figure 5: Possible and certain window membership of tuples in the window [-1,4] for t based on their possible sort positions.

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

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

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

aggregation result bound) the aggregation function result for an input tuple. Let us use $\text{possn}(\mathbf{R}, \mathbf{t})$ to denote $\text{size}([l, u]) - m$:

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

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

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

Note that $\mathbf{R}^{\downarrow}$ and $\mathbf{R}^{\uparrow}$ are bags ($\mathbb{N}$-relations) over range-annotated tuples. Furthermore, we define $\text{min-k}(\mathbf{R}, \mathbf{t}, A)$ (and $\text{max-k}(\mathbf{R}, \mathbf{t}, A)$) that are computed by restricting $\mathcal{W}_{\mathbf{R},\mathbf{t}}$ to the tuples with the smallest negative values (largest positive values) as lower (upper) bounds on attribute A that could contribute to the aggregation, keeping tuples with a total multiplicity of up to $\text{possn}(\mathbf{R}, \mathbf{t})$. Note that the deterministic conditions / expressions in the definition of $\text{min-k}(\mathbf{R}, \mathbf{t}, A)$ (and $\text{max-k}(\mathbf{R}, \mathbf{t}, A)$) 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.

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

Windowed Aggregation. Using the filtered combined windows we are ready to define row-based windowed aggregation over AU-DBs. To compute aggregation results, we utilize the operation $\circledast_f$ defined in [24] for aggregation function f that combines the range-annotated aggregation attribute value of a tuple with the tuple's multiplicity bounds. For instance, for **sum**, $\circledast_{\mathbf{sum}}$ 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 $\text{expand}(\mathbf{R})$ 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 AU-DB relation. We define window operator* $\omega^{[l,u]}_{f(A) \to X;\, G;\, O}$ *as:*

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

Example 6 (AU-DB Windowed Aggregation). *Consider the AU-DB relation* $\mathbf{R}$ *shown below and query* $\omega^{[-1,0]}_{sum(C) \to SumA;\, A;\, B}(\mathbf{R})$*, i.e., 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 the 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* $[2/11/12]$ *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**	**SumC**	$\mathbb{N}^3$
$\mathbf{r}_1$	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	[2/4/5]	[2/11/12]	1
$\mathbf{r}_4$	[2/3/3]	15	4	[4/4/9]	(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). *Consider an AU-DB relation* $\mathbf{R}$ *and incomplete bag relation*

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

7 NATIVE ALGORITHMS

We now introduce optimized algorithms for ranking and windowed aggregation over AU-DBs that are more efficient than their SQL counterparts presented in [22]. Through a *connected heap* data structure, these algorithms leverage the fact that the lower and upper position bounds are typically close approximations of one another to avoid performing multiple passes over the data. We assume a physical encoding of an AU-DB relation $\mathbf{R}$ 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.

7.1 Non-deterministic Sort, Top-k

Algorithm 1 sorts an input AU-DB $\mathbf{R}$. The algorithm assigns to each tuple its position τ given as lower and upper bounds: $\mathtt{t}.\tau^{\downarrow}$, $\mathtt{t}.\tau^{\uparrow}$, respectively. Given a parameter k, the algorithm can also be used to find the top-k elements; otherwise we set $\mathsf{k} = |\mathbf{R}|$ (the maximal possible size of the input relation). Algorithm 1 takes as input the relational encoding of an AU-DB relation $\mathbf{R}$ 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 $\mathbf{t}$ we have to sum up the smallest multiplicity of tuples $\mathbf{s}$ that are certainly sorted before $\mathbf{t}$, i.e., where $\mathbf{s}.O^{\uparrow} <^{total}_{O} \mathbf{t}.O^{\downarrow}$. Since $\mathbf{s}.O^{\downarrow} <^{total}_{O} \mathbf{s}.O^{\uparrow}$ holds for any tuple, we know that these tuples are visited by Algorithm 1 before $\mathbf{t}$. 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 $\mathbf{t}$, we first determine all tuples $\mathbf{s}$ from `todo` certainly preceding $\mathbf{t}$ ($\mathbf{s}.O^{\uparrow} < \mathbf{t}.O^{\downarrow}$) and update $\mathtt{rank}^{\downarrow}$ with their multiplicity. Since $\mathbf{t}$ is the first tuple certainly ranked after any such tuple $\mathbf{s}$ and all tuples following $\mathbf{t}$ will also certainly ranked after $\mathbf{s}$, we can now determine the upper bound on $\mathbf{s}$'s position. Based on Equation (3) this is the sum of the maximal multiplicity of all tuples that may precede $\mathbf{s}$. These are all tuples $\mathbf{u}$ 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 $\mathbf{s}$'s upper bound sort position, adapt $\mathbf{s}.\#^{\downarrow}$ (for a top-k query, $\mathbf{s}$ may not exist in the result if its position may be larger than k), add $\mathbf{s}$ to the result, and adapt $\mathtt{rank}^{\downarrow}$ (all tuples processed in the following are certainly

Input: $\mathbf{R}$ (sorted on $O^{\downarrow}$), $\mathsf{k} \in \mathbb{N}$ (or $\mathsf{k} = |\mathbf{R}^{\uparrow}|$)

todo ← minheap($O^{\uparrow}$) ; rank$^{\downarrow}$ ← 0 ; rank$^{\uparrow}$ ← 0 ; res ← $\emptyset$
for $\mathbf{t} \in \mathbf{R}$ **do**
  while todo.peek().$O^{\uparrow}$ < $\mathbf{t}.O^{\downarrow}$ **do** // emit tuples
    emit(todo.pop())
    if rank$^{\downarrow}$ > k **then** // tuples certainly out of top-k?
      return res
  $\mathbf{t}.\tau^{\downarrow}$ ← rank$^{\downarrow}$ // set position lower bound
  todo.insert($\mathbf{t}$) // insert into todo heap
  rank$^{\uparrow}$ += $\mathbf{t}.\#^{\uparrow}$ // update position upper bound
while not todo.isEmpty() **do** // flush remaining tuples
  emit(todo.pop())
return res

def emit($\mathbf{s}$)
  $\mathbf{s}.\tau^{\uparrow}$ ← min(k, rank$^{\uparrow}$) // position upper bound capped at k
  if rank$^{\uparrow}$ > k **then** // s may not be in result if $\mathbf{s}.\tau^{\uparrow} > k$
    $\mathbf{s}.\#^{\downarrow}$ ← 0
  res ← res ∪ split({$\mathbf{s}$})
  rank$^{\downarrow}$ += $\mathbf{s}.\#^{\downarrow}$ // update position lower bound

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

ranked higher than $\mathbf{s}$). Function `split` splits a tuple with $\mathbf{t}.\# > 1$ into multiple tuples as required by Def. 2. If we are only interested in the top-k results, then we can stop processing the input once $\mathtt{rank}^{\downarrow}$ 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 1's worst-case runtime is $O(n \cdot \log n)$ and worst-case memory requirement is $O(n)$ for $n = |\mathbf{R}|$ (see [22]).

7.2 Connected Heaps

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

A *connected heap* is comprised of H heaps which store pointers to a shared set of records. Each heap has its own sort order. A record stored in a connected heap consists of a tuple (the payload) and H backwards pointers that point to the nodes of the individual heaps storing this tuple. These backward pointers enable efficient deletion ($O(H \cdot \log n)$) of a tuple from all heaps when it is popped as the root of one of the component heaps. In [22] we explain how the standard sift-up and sift-down heap operations are used to restore the heap property in $O(\log n)$ when removing a non-root element from a component heap. When a tuple is inserted into a connected heap, it is inserted into each component heap in $O(\log n)$

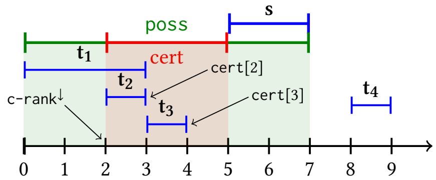
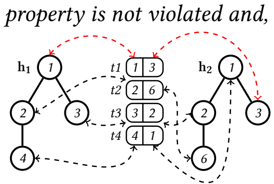
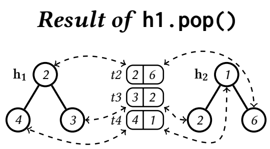


Figure 7: Example state for Algorithm 2, N=5, `c-rank`$^{\downarrow}$=2.

in the usual way with the exception that the backwards pointers are populated. In [22], we experimentally compare the performance of heaps with connected heaps. Even for small databases (10k tuples) and a small fraction of uncertain order-by values (1%), connected heaps outperform heaps by a factor of ~ 2. Larger databases / more uncertain data result in larger heaps and, thus, better performance.

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





7.3 Ranged Windowed Aggregation

Without loss of generality, we focus on window specifications with only a **ROWS PRECEDING** clause; a **FOLLOWING** clause can be simulated by offsetting the window, i.e., a window bound of $[-N, 0]$. Algorithm 2 uses a function `compBounds` to compute the bounds on the aggregation function result based on the certain and possible content of a window. We present the definition of this function for several aggregation functions in [22]. Algorithm 2 follows a sweeping pattern similar to Algorithm 1 to compute the windowed aggregate in a single pass over the data which has been preprocessed by applying $\text{SORT}_{O \to \tau}(\mathbf{R})$ and then has been sorted on $\tau^{\downarrow}$. The 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`, `pagg`$^{\downarrow}$, `pagg`$^{\uparrow}$) is a connected minheap where `poss`, `pagg`$^{\downarrow}$, and `pagg`$^{\uparrow}$ are sorted on $\tau^{\uparrow}$, $A^{\downarrow}$, $-A^{\uparrow}$, 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 `c-rank`$^{\downarrow}$ for the lower bound position of the certain part of windows.

Algorithm 2 first inserts each incoming tuple into `openw` (Line 7). If the tuple certainly exists, it is inserted into the tree of certain tuples whose lower bound position is $\mathbf{t}.\tau^{\downarrow}$. Note that each of these trees is sorted on $\tau^{\uparrow}$ which will be relevant later. Next the algorithm

```
   Input: f, X, O, N, A, SORT_{O→τ}(R) sorted on τ↓
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 2: Aggregate $f(A) \to X$, sort on O, N preceding

determines for which tuples from `openw`, their windows have been fully observed. These are all tuples $\mathbf{s}$ which are certainly ordered before the tuple $\mathbf{t}$ 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 $\mathbf{s}$ that could possibly belong to the window of a tuple $\mathbf{t}$ have to have $\mathbf{s}.\tau^{\downarrow} \leq \mathbf{t}.\tau^{\uparrow}$. Based on these observations, once we processed a tuple $\mathbf{t}$ with $\mathbf{t}.\tau^{\downarrow} > \mathbf{s}.\tau^{\uparrow}$ for a tuple $\mathbf{s}$ in `openw`, we know that no tuples that we will process in the future can belong to the window for $\mathbf{s}$. In Line 11 we iteratively pop such tuples from `openw`. For each such tuple $\mathbf{s}$ we evict tuples from `cert` and update the high watermark `c-rank`$^{\downarrow}$ (Line 12). Recall that for a tuple $\mathbf{u}$ to certainly belong to the window for $\mathbf{s}$ 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 $\mathbf{s}$ using `cert` and `poss` (we will describe this step in more detail in the following). Finally, we evict tuples from `poss` (and, thus, also `pagg`$^{\downarrow}$ 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 $\mathbf{s}$, i.e., tuples $\mathbf{u}$ 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 $\mathbf{u}$ emitted after $\mathbf{s}$ we have $\mathbf{u}.\tau^{\uparrow} > \mathbf{s}.\tau^{\uparrow}$. Fig. 7 shows an example state for the algorithm when tuple $\mathbf{s}$ 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 $\mathbf{s}$. Tuples with sort position ranges overlapping with the green region are in the possible window (these tuples are stored in `poss`). Tuples like $\mathbf{t}_4$ with upper-bound position higher than $\mathbf{s}$ will be popped and processed after $\mathbf{s}$. 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 from Algorithm 2.

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

8 EXPERIMENTS

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

Compared Algorithms. We compare against several baselines: *Det* evaluates queries deterministically ignoring uncertainty in the data. We present these results to show the overhead of the different incomplete query evaluation semantics wrt. deterministic query evaluation; *MCDB* [34] evaluates queries over a given number of possible worlds sampled from the input incomplete database using deterministic query evaluation. *MCDB10* and *MCDB20* are MCDB with 10 and 20 sampled worlds, respectively. For MCDB, we treat the highest and lowest possible value across all samples as the upper and lower bounds and compare against the tight bounds produced by the other algorithms (since computing optimal bounds is often intractable). Given a exact 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. For *PT-k* [32], we set its threshold to 1 (0) to compute all certain (possible) answers. *Symb* represents ranking and aggregation results as symbolic expressions similar to [9, 12]. We use an SMT solver (Z3 [20]) to compute tight bounds on the possible sort positions / aggregation results for tuples. *Rewr* is our rewrite-based approach [22] that has to process the input relation twice for sorting and uses range self-joins to determine the content of windows. *Imp* is implemented as a native extension for Postgres 13.3. 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 Windows binary. We run *PT-k* on a separate Windows machine with an 8-core 3800MHz 32MB cache AMD Ryzen 5800x CPU, 64G RAM, and 2TB HD. *PT-k* is single-threaded and in-memory. Since we deactivated intra-query parallelism in Postgres, but still have to go to disk, the comparison is in favor of *PT-k*. 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 a maximum 1k attribute range on uncertain values.

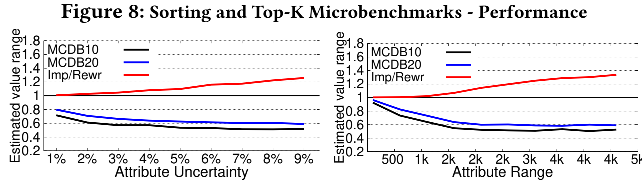
8.1.1 Sorting and Top-k Queries. **Scaling Data Size.** Fig. 11 shows the runtime of sorting, varying the dataset size. Since *Symb* and *PT-k* perform significantly worse, we only include these methods for smaller datasets (Fig. 11a). MCDB and our techniques significantly outperform *Symb* and *PT-k* (~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. 8 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 8: Sorting and Top-K Microbenchmarks - Performance

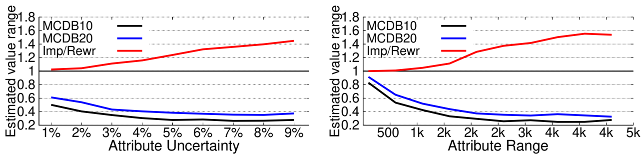


(a) Varying uncertainty **(b) Varying range**

Figure 9: Sorting microbenchmarks - approximation quality

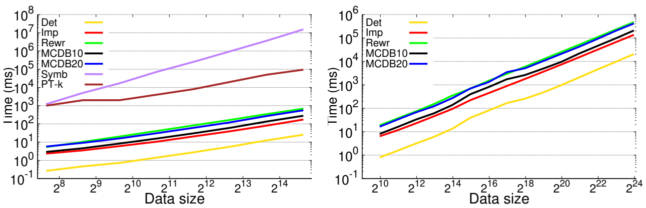
Accuracy. Fig. 9 shows the error of the bounds generated by *Imp* (*Rewr* produces identical outputs) and *MCDB*. Recall that *Imp* is guaranteed to over-approximate the correct bounds, while *MCDB* is guaranteed to under-approximate the bounds, because it does not compute all possible results. We measure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

8.1.2 Windowed Aggregation. **Scaling Data Size.** Fig. 12 shows the runtime of windowed aggregation when varying dataset size. We compare two variants of our rewrite-based approach which uses a range overlap join to determine which tuples could possibly belong to a window. *Rewr(Index)* uses a range index supported by Postgres. We show index creation time and query time separately. We exclude *Symb*, because for more than 1k tuples, Z3 exceeds the maximal allowable call stack depth and crashes. The performance



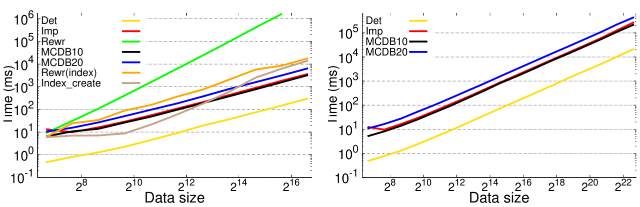
(a) Varying uncertainty **(b) Varying range**

Figure 10: Window microbenchmarks - approximation quality



(a) Smaller datasets **(b) Larger datasets**

Figure 11: Sorting performance varying dataset size



(a) Smaller datasets **(b) Larger datasets**

Figure 12: 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 + Partiton-by + Window size	w=3,r=1k,u=5%	105.1ms	73.5s	1209.4ms	2127.1ms
	w=3,r=10k,u=5%	101.7ms	75.2s	1231.3ms	2142.9ms
	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 13: Windowed aggregation microbenchmarks - 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. 13 shows the runtime of windowed aggregation varying the value ranges of uncertain attribute (on all columns), percentage of uncertain tuples, and window size. For *Imp* (Fig. 13a) we use a query without partition-by. We also compare runtime of our rewriting based approach (Fig. 13b) using both partition-by and order-by on 8k rows. *Imp* exhibits similar runtime to *MCDB10* and outperforms *MCDB20*. *Rewr* is slower than *MCDB* by several magnitudes due to the range-overlap join.

8.2 Real World Datasets

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

For sorting and top-k queries that contain aggregation which is common in real use-cases, we only measure the performance

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

Figure 14: Real world data - performance

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

Figure 15: 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 16: Real world data - windowed aggregation accuracy and recall

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. 15 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. 16 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. We significantly outperform existing algorithms for ranking uncertain data and our approach is applicable to more expressive queries and bounds all certain and possible answers. In future work, we plan to extend our approach to deal more expressive classes of queries, e.g., recursive and fix-point computations as used in ML model training, and will investigate index structures for AU-DBs to further improve performance.

REFERENCES

[1] https://data.medicare.gov/data/hospital-compare. Medicare Hospital Dataset. (https://data.medicare.gov/data/hospital-compare).

[2] https://github.com/fengsu91/uncert-ranking-availability. Paper Artifacts. (https://github.com/fengsu91/uncert-ranking-availability).

[3] https://nsidc.org/data/g00807. Iceberg Dataset. (https://nsidc.org/data/g00807).

[4] https://www.kaggle.com/currie32/crimes-in-chicago. Chicago Crimes Dataset. (https://www.kaggle.com/currie32/crimes-in-chicago).

[5] Serge Abiteboul, T.-H. Hubert Chan, Evgeny Kharlamov, Werner Nutt, and Pierre Senellart. 2010. Aggregate queries for discrete and continuous probabilistic XML. In *ICDT*. 50–61.

[6] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. 1991. On the Representation and Querying of Sets of Possible Worlds. *Theor. Comput. Sci.* 78, 1 (1991), 158–187.

[7] Parag Agrawal, Anish Das Sarma, Jeffrey Ullman, and Jennifer Widom. 2010. Foundations of uncertain-data integration. *PVLDB* 3, 1-2 (2010), 1080–1090.

[8] Robert Albright, Alan J. Demers, Johannes Gehrke, Nitin Gupta, Hooyeon Lee, Rick Keilty, Gregory Sadowski, Ben Sowell, and Walker M. White. 2008. SGL: a scalable language for data-driven games. In *SIGMOD*. 1217–1222.

[9] Antoine Amarilli, M Lamine Ba, Daniel Deutch, and Pierre Senellart. 2014. Provenance for Non-deterministic Order-Aware Queries. *Prepr int: http://a3nm. net/publications/amarilli2014provenance. pdf* (2014).

[10] Antoine Amarilli, M. Lamine Ba, Daniel Deutch, and Pierre Senellart. 2017. Possible and Certain Answers for Queries over Order-Incomplete Data. In *Proc. TIME*. 4:1–4:19.

[11] Antoine Amarilli, Mouhamadou Lamine Ba, Daniel Deutch, and Pierre Senellart. 2019. Computing possible and certain answers over order-incomplete data. *Theor. Comput. Sci.* 797 (2019), 42–76.

[12] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In *PODS*. 153–164.

[13] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. 2014. Sampling from Repairs of Conditional Functional Dependency Violations. *VLDBJ* 23, 1 (2014), 103–128.

[14] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Müller, Sonia Castel, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *CIDR*.

[15] Douglas Burdick, Prasad M. Deshpande, T. S. Jayram, Raghu Ramakrishnan, and Shivakumar Vaithyanathan. 2007. OLAP over uncertain and imprecise data. *VLDBJ* 16, 1 (2007), 123–144.

[16] Arbee L. P. Chen, Jui-Shang Chiu, and Frank Shou-Cheng Tseng. 1996. Evaluating Aggregate Operations Over Imprecise Data. *IEEE Trans. Knowl. Data Eng.* 8, 2 (1996), 273–284.

[17] Marco Console, Paolo Guagliardo, and Leonid Libkin. 2019. Fragments of Bag Relational Algebra: Expressiveness and Certain Answers. In *ICDT*. 8:1–8:16.

[18] Marco Console, Paolo Guagliardo, Leonid Libkin, and Etienne Toussaint. 2020. Coping with Incomplete Data: Recent Advances. In *PODS*. ACM, 33–47.

[19] Graham Cormode, Feifei Li, and Ke Yi. 2009. Semantics of Ranking Queries for Probabilistic Data and Expected Ranks. In *ICDE*. 305–316.

[20] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *ETAPS*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. 337–340.

[21] Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *PODS*. 159–170.

[22] Su Feng, Boris Glavic, and Oliver Kennedy. 2022. Efficient Approximation of Certain and Possible Answers for Ranking and Window Queries over Uncertain Data (extended version). (2022). arXiv:2302.08676 [cs.DB]

[23] Su Feng, Aaron Huber, Boris Glavic, and Oliver Kennedy. 2019. Uncertainty Annotated Databases - A Lightweight Approach for Approximating Certain Answers. In *SIGMOD*.

[24] Su Feng, Aaron Huber, Boris Glavic, and Oliver Kennedy. 2021. Efficient Uncertainty Tracking for Complex Queries with Attribute-level Bounds. In *SIGMOD*. 528–540.

[25] Robert Fink, Larisa Han, and Dan Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. *PVLDB* 5, 5 (2012), 490–501.

[26] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2022. Towards data-centric what-if analysis for native machine learning pipelines. In *DEEM@SIGMOD*. 3:1–3:5.

[27] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*.

[28] Paolo Guagliardo and Leonid Libkin. 2016. Making SQL Queries Correct on Incomplete Databases: A Feasibility Study. In *PODS*.

[29] Paolo Guagliardo and Leonid Libkin. 2017. Correctness of SQL Queries on Databases with Nulls. *SIGMOD Record* 46, 3 (2017), 5–16.

[30] Paolo Guagliardo and Leonid Libkin. 2019. On the Codd semantics of SQL nulls. *Inf. Syst.* 86 (2019), 46–60.

[31] Alon Halevy, Anand Rajaraman, and Joann Ordille. 2006. Data integration: the teenage years. In *VLDB*. 9–16.

[32] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. 2008. Ranking Queries on Uncertain Data: A Probabilistic Threshold Approach. In *SIGMOD*. 673–686.

[33] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984), 761–791.

[34] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. 2008. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD*.

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

[36] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. 2006. Declarative Support for Sensor Data Cleaning. In *PERVASIVE*. 83–100.

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

[38] Poonam Kumari, Said Achmiz, and Oliver Kennedy. 2016. Communicating Data Quality in On-Demand Curation. In *QDB*.

[39] Willis Lang, Rimma V. Nehme, Eric Robinson, and Jeffrey F. Naughton. 2014. Partial results in database systems. In *SIGMOD*. 1275–1286.

[40] Jens Lechtenbörger, Hua Shu, and Gottfried Vossen. 2002. Aggregate Queries Over Conditional Tables. *J. Intell. Inf. Syst.* 19, 3 (2002), 343–362.

[41] Jian Li, Barna Saha, and Amol Deshpande. 2009. A Unified Approach to Ranking in Probabilistic Databases. *PVLDB* 2, 1 (2009), 502–513.

[42] Xi Liang, Zechao Shang, Sanjay Krishnan, Aaron J. Elmore, and Michael J. Franklin. 2020. Fast and Reliable Missing Data Contingency Analysis with Predicate-Constraints. In *SIGMOD*. 285–295.

[43] Leonid Libkin. 2016. SQL's Three-Valued Logic and Certain Answers. *TODS* 41, 1 (2016), 1:1–1:28.

[44] Witold Lipski. 1979. On Semantic Issues Connected with Incomplete Information Databases. *TODS* 4, 3 (1979), 262–296.

[45] Raghotham Murthy, Robert Ikeda, and Jennifer Widom. 2011. Making Aggregation Work in Uncertain and Probabilistic Databases. *IEEE Trans. Knowl. Data Eng.* 23, 8 (2011), 1261–1273.

[46] Dan Olteanu, Lampros Papageorgiou, and Sebastiaan J van Schaik. 2013. Pigora: An Integration System for Probabilistic Data. In *ICDE*. 1324–1327.

[47] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *Advances in Spatial and Temporal Databases*, Michael Gertz, Matthias Renz, Xiaofang Zhou, Erik Hoel, Wei-Shinn Ku, Agnes Voisard, Chengyang Zhang, Haiquan Chen, Liang Tang, Yan Huang, Chang-Tien Lu, and Siva Ravada (Eds.). Springer International Publishing, Cham, 125–144.

[48] Christopher Re, Nilesh Dalvi, and Dan Suciu. 2007. Efficient Top-k Query Evaluation on Probabilistic Data. In *ICDE*. 886–895.

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

[50] Babak Salimi, Romila Pradhan, Jiongli Zhu, and Boris Glavic. 2022. Interpretable Data-Based Explanations for Fairness Debugging. In *SIGMOD*. 247–261.

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

[52] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. 2008. Probabilistic top-k and ranking-aggregate queries. *TODS* 33, 3 (2008), 13:1–13:54.

[53] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. 2008. Probabilistic Top-k and Ranking-Aggregate Queries. *TODS* 33, 3, Article 13 (2008), 54 pages.

[54] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. 2007. Top-k Query Processing in Uncertain Databases. In *ICDE*. 896–905.

[55] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. Probabilistic databases. *Synthesis Lectures on Data Management* 3, 2 (2011), 1–180.

[56] Bruhathi Sundarmurthy, Paraschos Koutris, Willis Lang, Jeffrey F. Naughton, and Val Tannen. 2017. m-tables: Representing Missing Data. In *ICDT*.

[57] Mohan Yang, Haixun Wang, Haiquan Chen, and Wei-Shinn Ku. 2011. Querying uncertain data with aggregate constraints. In *SIGMOD*. 817–828.

[58] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. 2015. Lenses: An On-demand Approach to ETL. *PVLDB* 8, 12 (2015), 1578–1589.

[59] Xi Zhang and Jan Chomicki. 2008. On the semantics and evaluation of top-k queries in probabilistic databases. In *ICDE*. 556–563.