Relational Preference Queries via Stable Skyline

Parke Godfrey

Wei Ning

Technical Report CS-2004-03

August 2004

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Relational Preference Queries via Stable Skyline

Parke Godfrey Wei Ning

York University
Toronto, ON M3J 1P3
Canada
{godfrey, ningwei}@cs.yorku.ca

Abstract

We advocate the extension of relational database systems to support preference queries. Many database applications today—from e-commerce to queries over scientific data-sets—are essentially *best-match* searches. Relational queries are ill-suited for these. Supporting preference criteria in the query language can extend its expresssiveness to cover best-match queries in a natural way.

We study *skyline queries* as a foundation for preference queries. Skyline offers a natural way to combine multiple preference criteria in parallel. Skyline as it was introduced, however, is limited in its expressiveness, and does not capture many types of preferences and compositions people would like to support. We present a formal model of skyline and motivate two extensions to skyline that greatly increase its expressiveness. These extensions destroy though the partial-order semantics of skyline as originally defined. We develop the *stable skyline semantics* that accommodates the extensions and the loss of transitivity in the preference relation in a natural manner. This also opens the door to other, potentially useful extensions. We present a high-level algorithm that computes the stable skyline set. Lastly, we show how skyline criteria can be *grounded* in a natural way in cases when the preference relation may otherwise have cycles.

1 Introduction

1.1 What is Wrong?

answer the question
i find there are no answers
then make something up

"Find me a house in Monterey in good condition for less than $300,000 with at least three bedrooms and an ocean view." With an appropriate database at hand, one could compose a query (say, in SQL) to express this. But what if the query comes up empty? Our house hunter must try again with a new query, perhaps by modifying (weakening) the criteria from the original query. This process can be long and arduous, and is often unsuccessful because of it. Kaplan named this seeming behavior of the database system to withhold information *stonewalling* [22, 23].

Relational database systems—and, for that matter, other common information system technologies—do not offer a solution. They stonewall. A relational query *selects* the tuples that satisfy the query's conditions. The querier must know something—or quite a bit usually—about the *data* in the database in order to specify these conditions suitably.

There are many applications and tasks today much like the house hunter's. The person might not know much about the data. (A house for $300,000 in Monterey? Really!) Furthermore, the house hunter's criteria are not *conditions*, per se. Rather, they are *preferences*: *in reasonable condition*, *inexpensive*, *many bedrooms*, and *a good view*. The house hunter may not even be expecting to find a house that actually satisfies all these preferences, or that satisfies them in equal measure. The house hunter is looking for the best options, with *bestness* measured against the preferences.

E-commerce tasks are similar to the house hunter's. Customers search for products and services with certain criteria—requirements *and* preferences—in mind. The relational systems that back-end e-commerce systems require coercing the preferences into conditions, leading to stonewalling and frustration.

In scientific domains, a researcher often queries a database in an investigatory manner. An astronomer queries an astronomical body database to find objects most similar to the description of a hypothetical object. A geneticist queries a human genome database to find genes that *best-match* given criteria. Relational queries are ill-suited to these tasks.

What solutions are there to stonewalling? What would an information system or query language need to address these issues? Is the notion of best-match and preferences at odds with the relational approach? There is much interest in these issues, and the need for

good approaches is ever more pressing. There is quite a bit of research within the area, especially as of late. However, there remains little practical progress so far.

1.2 Why Preferences?

choice everywhere
how to rank from best to worst
the way is not clear

Numerous solutions to stonewalling have been pursued. SQL allows for the ordering of results. Thus, if one could *rank* answer tuples by *bestness*, they could be returned in order from best to worst. This requires providing each tuple a score. For this, two things are needed: the user's preference criteria; and a *utility function* that combines the "preference" scores of a tuple into a single score.

If one is able to specify the preference criteria well, though, one could write a query in a preference query language. Devising an appropriate utility function is quite hard. How does one combine the value of a house's *condition* with its *number of bedrooms*? With the Boolean *ocean view*?

The utility function is opaque, so it is often difficult for the user to interpret *why* results are ranked as they are. Ideally, a preference query language removes the need for a utility function.

Ranking approaches return *all* the "items", although ranked. This can be expensive to evaluate. There has been substantial work to address this. Top-k queries limit the results to the top k scoring tuples. This permits optimization. Determining a natural k for a given query in advance, though, is hard.

Cooperative query answering (CQA) has sought to address the stonewalling problem in various ways. Some techniques relay information back to the user after a query that would help the user in designing follow-up queries. Other CQA techniques generate follow-up queries automatically when it is determined the initial query was insufficient. These approaches, however, do not provide for the most part means by which the user can express which are the preferential components of the query.

These different approaches, in truth, are orthogonal. There is no reason a system employing a preference query language could not also rank the results by *bestness*. There is no reason why information retrieval (IR) techniques could not be integrated in a full-fledged system. In many ways, CQA and preference queries are opposite sides of the same coin: preference queries allow the user to state the preference criteria up front; and CQA helps remedy the case when the the query is unsuccessful. Furthermore, we do not believe relational queries and preferences to be at odds.

A preference query language brings tools to bear that the other approaches do not. This warrants the study of this approach.

1.3 Why Skyline?

i see the skyline
transcending geometry
a surface no more

The skyline clause was proposed in [5] as an extension to SQL, with syntax as in Figure 1. Skyline offers an elegant approach to combining multiple preference criteria in parallel.

```
select ... from ... where ...
    group by ... having ...
    skyline of A1 [min | max | diff], ...,
               An [min | max | diff]
```

Figure 1: The proposed skyline clause for SQL.

The skyline operator *filters* the set of tuples derived by (the rest of) the query. Any tuple r is removed if there is another tuple s that is better than, or equal to, tuple r on each skyline criterion (A_i), *and* is strictly better than tuple r on at least one criterion. In this case, we say that s *trumps* r. Tuple s is better than tuple r on criterion A_i max if s's A_i value is greater than r's. Tuple s is better than tuple r on criterion A_i min if s's A_i value is less than r's. If there is a criterion A_i diff, then r cannot be trumped by s if s's A_i value is different from r's. The answer set is the set of tuples never trumped, called the *skyline set*.

Each skyline criterion, A_i max or A_i min, imposes a *weak order* over the input tuples.[1] The skyline criteria taken together (conjunctively) then impose a *partial order* over the input tuples.[2] The skyline set is the *crown* of this partial order.

```
select Address, Agent, Lockbox#, Cond,
       Price, #bdrm, OceanView, Style
  from HouseListing
  where area = 'Monterey'
  skyline of Cond max, Price min,
             #bdrm max, OceanView max,
             Style diff;
```

Figure 2: Skyline query for the house hunter.

The query in Figure 2 is a skyline query expressing the house hunter's query from Section 1.1. It is assumed here that Cond (the *condition* of the house) is a numeric score, say, from 1..5, with 5 as the best. Price and #bdrm (number of bedrooms) are as one would expect. OceanView is a Boolean: 1 for *true*; and 0 for *false*. We have added the criterion Style diff; this means the query will find the *best* houses per house-style (e.g., bungalow, modern, and Victorian).

[1] It is not a *total order* since two tuples may have the same A_i value, thus tying. A weak order is a total order except for allowing ties.

[2] We can ignore diff here. Note that any criterion A_i diff can be replaced by the criteria A_i max *and* A_i min.

Note that Cond and OceanView might not be available (and in the expected format) in a base House table. HouseListing though can be a view. Or it can be replaced by a sub-query in the query in Figure 2. The requisite information for the skyline criteria can be assembled via any relational query. The skyline operator is composable with the relational algebra.

We believe that skyline offers a natural foundation for a preference query language. It provides a way to combine multiple preference criteria. And via composition—with itself, as in nested skyline clauses, and with other relational operations—it provides a fairly rich language to compose and prioritize preferences. Furthermore, skyline is readily understandable.

While the skyline clause is a readily understandable extension to SQL, a relational engine can be extended with a *skyline operator* to optimize and execute skyline queries efficiently. There has been a fair amount of work as of late on algorithms to compute skyline efficiently. Thus, a fairly rich, skyline-based preference query language could be supported with the addition of a single operator to the relational engine. Other studied preference query languages would seemingly require extensive changes to the relational system.

However, SQL with skyline is not expressive enough. There are many natural preferences and preference compositions that have been proposed that cannot be expressed by skyline. Prioritizing preferences can be accomplished to some degree by nested skyline clauses, but often not naturally. To realize a skyline-based preference query language, skyline needs to be extended somehow.

1.4 In This Paper

what is accomplished
in the span of a season
the weather changes

We demonstrate specific ways that skyline can be extended that greatly increase its expressiveness. We focus on two extensions particularly: a new criterion directive equal as a dual to diff (and, as a logical consequence, another directive maxeq); and a criterion modifier by. For example, the criterion Price min by 5000 means that house (tuple) *s* does not trump house *r* with respect to the criterion *unless* *s* is at least $5,000 less expensive than *r*.

These extensions affect skyline's partial-order semantics. Equal can mean loss of transitivity. We are no longer guaranteed a partial order, but are guaranteed a directed acyclic graph. We introduce a change to the skyline semantics that we call *stable skyline* which accommodates the loss of transitivity.

The addition of by can mean the introduction of cycles into the *preference relation* (that is, the algebraic relation induced by the skyline query over the input set of tuples). We do not believe that there is any reasonable "skyline" semantics which would accommodate naturally cycles. However, we show how skyline queries can be made *safe* in a natural way, which guarantees a cycle-free preference relation. This is possible with the addition of equal, and thus in stable skyline, but not in original skyline.

We consider related work in Section 2. We formalize skyline in Section 3, and we extend it, introducing the *stable skyline semantics*, in Section 4. We conclude in Section 5.

2 Related Work

questions are sacred
answers are often profane
ask me what you will

People have recognized a need for preferences in queries nearly since the the introduction of expressive, declarative query languages for databases. In [6], a deductive query language called DEDUCE was proposed for relational databases which includes preferences. Lacroix and Pirotte [30] introduced the *domain relational calculus* (DRC) and the *intermediate level language*, ILL, as an English-like language allowing structured expressions with the goal of more natural, more expressive query languages. In [29], Lacroix and Lavency showed how to extend the DRC to provide a preference mechanism. Preferences within a query are satisfied if possible, but "ignored" when not.

Approaches to preferences in queries can be classified as *qualitative* and *quantitative*. Qualitative approaches allow for preferences to be represented in queries, offer means to compose preference criteria—with themselves and with the other query constructs and conditions—and extend the query semantics to encompass preferences, as did the extension to DRC in [29]. In quantitative approaches, answers are scored with respect to "preferences". The preferences are not usually composed in the query, but are reflected in the scoring function. Answers then with the best scores are deemed to match the user's preference best, and the answers are returned in best-to-worst order.

Our main interest is in qualitative approaches. In designing a qualitative preference query facility, we have the following criteria (that which we hold *sacred*).

- *Simple:* Queries are understandable and natural.
- *Adjustable:* It is easy to rewrite queries, to compose followup queries.
- *Composable:* It is easy to compose preference elements with themselves and with other (say, relational) operators.
- *Realizable:* Preference queries can be optimized and executed efficiently.
- *Expressive:* Many types of preference queries can be expressed.
- *Declarative:* There is a declarative semantics.

Lacroix and Lavency's preference extension to the DRC [29] is *simple* and understandable which enables preferred conditions to be stated. They addressed

composability in showing how multiple preference conditions could be combined and prioritized. As their extended query language inherits the same first-order logic definition as the DRC, their language has a *declarative* semantics. The approach is limited in *expressiveness*, though. Each preference condition is evaluated in a Boolean manner: either there are answers that satisfy the query *and* it; or there are not (and so the query is evaluated as without that preference). Also, there was no consideration of *realizability*. In fact, the addition of multiple preferences can lead to exponentially larger representations in the DRC.

In current work, Chomicki has introduced a general logical framework for preferences as preference formulas, and has proposed a relational operator winnow for composing preference relations in the relational algebra [8, 9]. His model and winnow are quite *expressive*, and he has investigated the types of preferences that can be expressed, and has shown how they can be *composed*. Winnow offers a *declarative* semantics. Chomicki has investigated the effects of types of preference formulas on the *preference relation*, the order the preferences induce over the potential answer tuples. Much work in preferences restricts consideration to *partial orders*. Chomicki has found useful preference formulas that violate partial orders, but shows how this can be accommodated. (Likewise, we consider the ramifications of giving up partial orders with skyline in this paper.) Thus, winnow is a quite rich model. However, it is not *simple*; it can be complex to understand how to write preferences and how to compose them. It is also not clear so far how to *realize* winnow, say, in a relational system. In [10], Chomicki derives some special cases of winnow-based queries that can be evaluated efficiently. More work is needed though to identify significant, useful sub-classes of winnow that can be handled well.

In [24], Kiessling has taken an algebraic approach to constructing a rich preference query language as an extension to SQL that he calls Preference SQL. A number of preference operators are introduced, and how they compose is defined. Preference SQL allows users to write *best-match* queries by composing their preference criteria via the preference operators. Preference SQL has been on the market since 1999, and is used in several commercial ventures. The system compiles preference queries into SQL for evaluation. In [26], Kiessling and Koestler investigate further how to extend SQL and XPATH for the Preference SQL operators, and present rich examples of the types of queries that can be composed.

Preference SQL is not *simple*. How to compose preferences in the language meaningfully can be challenging. Because Preference SQL introduces many new constructs, how to *realize* it efficiently is a challenge. The current system translates queries into SQL. It would be hard to integrate the preference mechanisms within a relational engine because of the extensive additions. Preference SQL has an operational semantics, but not a defined *declarative* semantics. In particular, composition of the preference operators can raise difficulties. The intended semantics is that the preference relation be a partial order, but certain compositions can violate this. In [25], Kiessling proposes the concept of *substitutable values* (SV's) and *SV relations* to address sound composition of Preference SQL's Pareto and prioritized preferences.

The skyline operator was introduced in [5], as discussed in Section 1.3. Skyline queries have uses beyond preference queries. In fact, the problem has been studied earlier as the *maximal vector problem* [28]. Skyline imports the concept to relational databases, and supporting preferences is a key motivation behind the research. Skyline offers a *simple*, understandable construct for combining multiple preference criteria in parallel, and is *composable* with the relational algebra. Skyline as introduced has a clear partial-order semantics. Much recent work has gone into developing efficient, external, relationally well-behaved algorithms for evaluating skyline queries [5, 12, 14, 20, 27]. Thus skyline is a promising approach for supporting preference queries. By itself, however, skyline is not as expressive as winnow or Preference SQL. To serve as a foundation for preference queries, the *expressiveness* of skyline needs to be improved.

Quantitative approaches to preferences provide means to score potential answer tuples based on preference criteria. At its simplest, this can be done in SQL via the order by clause. This requires a single *metric*, or score, to be computed for each tuple. The metric is a *utility function* that combines the preference "values" somehow into a score. A disadvantage is that devising this utility function is not *simple*, as discussed in Section 1.2. On the other hand, ranking is a natural way to respond to preference. There is work on how to address better *composability* in ranking approaches. In [2], Agrawal et al. explore how numeric as well as categorical attributes can be combined, and Agrawal and Wimmers [1] present a framework for combining preferences. In [36], preferences are expressed as expressions stored as data, and evaluated during query execution. There has also been much research on how to rank efficiently. Often, one may be willing to accept just the top k answers from the ranking. Top-k queries can be optimized [7, 21].

There are many hybrid approaches that add constructs to the query language for preferences and that partially automate devising the utility function. In [15], Fagin et al. study utility functions that are *best* with respect to minimizing the discrepancy between the partial order and the weak orders of the preferences. In [6], membership functions are defined to measure a tuples adherence to stated preference conditions. In [32], Motro uses functions that measure

distance between tuples to measure adherence to *goals* expressed in the query. In [3], Bosc and Pivert integrate fuzzy sets with relational queries. The fuzzy conditions indicate degree of adherence to preferences.

Cooperative query answering, as discussed in Section 1.2, attempts to address many of the same issues preference queries do, and is the genesis of much preference query work [22, 23]. In [13], a cooperative information system, CoBase, is discussed, which allows for approximate answers. Thus "preferences" can be added to the query as conditions; if the query cannot be answered absolutely, conditions are "relaxed" to the point it has answers, thus approximate with respect to the original query. In [19], we explored the theoretical complexity of automatically eliminating as few conditions as necessary from the query if it fails so that the modified query has answers. This is a dual approach to the (Boolean) preference satisfaction that the system in [29] performs. See [31] for a broader survey of cooperative answering work.

Others have studied how query answers may be *annotated* to indicate the degree of adherence to, say, "preferences" [16, 33, 34]. These annotations are not metric scores, per se, as in the ranking approaches, but are deduced during query evaluation based upon rules that are part of the database. Thus, the annotation process is part of the query semantics, and is not specified as a function in the query.

Under the topic of *ceteris paribus* (*with other things equal*), researchers are studying the nature of preference, and when one situation—or solution or tuple—may be considered better than another [4]. Such work is clearly relevant to our endeavors here. There is more work in artificial intelligence and intelligent databases that seeks to embed *preference* and *priority into* the semantics. Preferences can disambiguate indefinite information, and prioritize certain sources over others. Thus preferences affect the canonical *models* of the database, and thus the semantics of the database itself. Reiter's default logic [35] is an example. There are parallels in this work in methods and goals with the pursuit of preference query languages, but they are distinct endeavors. Preferences for us identify *best* tuples, but they cannot affect answers to queries with no preferences involved.

3 Skyline Formalized

four values, not two
express the truth in our world
what would you prefer

We carefully formalize skyline so that we might propose extensions in a principled way and address the extensions's ramifications. In the skyline clause in Figure 1, the A_i's are columns; these may be derived columns (e.g., (age + salary)/#bdrm + 3). Therefore, we can consider each A_i as a *function* with its domain as all potential tuples over that schema, and its range as the real numbers. Paired with each function, A_i, is a directive that indicates how the tuples are to be compared with respect to A_i. Let us call a function-directive pair a skyline *comparator*. Let us call the set of function-directive pairs of a skyline-of clause the skyline *filter*.

We introduce a corresponding skyline operator, '∇', for conciseness's sake, and to facilitate formal discussion. Figure 3 enumerates the types of comparators, and how we shall designate them in our notation. So, for example, skyline of A max, B min, C diff is denoted by $\nabla_{\{>\mathsf{A},<\mathsf{B},\neq\mathsf{C}\}}$. The directives max, min, and diff essentially are the equality *operators* '$>$', '$<$', and '$\neq$', respectively, from the perspective of comparing tuples to compute the skyline.

skyline of	∇
F max	$>$F
F min	$<$F
F diff	$\neq$F

Figure 3: Original skyline comparators.

Note that the single operator '$>$' (max) would suffice; both '$<$' and '$\neq$' are algebraically redundant, given '$>$'. A skyline filter $\{\neq\mathsf{A}\} \cup \mathcal{F}$ is equivalent to $\{<\mathsf{A}, >\mathsf{A}\} \cup \mathcal{F}$. A skyline filter $\{<\mathsf{A}\} \cup \mathcal{F}$ is equivalent to $\{>(-1 \cdot A)\} \cup \mathcal{F}$. In proofs then, we restrict our attention to '$>$', dismissing '$<$' and '$\neq$'. In discussion and definitions, we consider both '$>$' and $\neq$', as '$\neq$' will help to motivate the extensions we propose. In examples, we shall still employ '$<$' for naturalness, with it understood that it can be rewritten via '$>$'.

```
compare (tuple r, tuple s, comparator OP F ± ε) {
    if ((F(r) OP F(s) + ε) ∧ (F(s) OP F(r) + ε))
        return ⊤;
    else if (F(r) OP F(s) + ε)
        return t;
    else if (F(s) OP F(r) + ε)
        return f;
    else
        return ⊥;
}
```

Figure 4: The compare evaluation with a comparator.

The skyline filter (a set of comparators) defines how tuples are to be compared, to determine which will be returned in the skyline set. The compare procedure in Figure 4 defines how two tuples are compared with respect to a comparator. F is a tuple function. OP is either '$>$' or '$\neq$' (for now). We consider an extension, '$\pm\epsilon$', later on. (So for now, let $\epsilon = 0$.) The procedure returns one of four *truth values*:

- $\top$, denoting *over-defined* (nicknamed *top*);
- *t*, denoting *true*;
- *f*, denoting *false*;
- $\bot$, denoting *under-defined* (nicknamed *bottom*);

For '$>$', *t*, *f*, and $\bot$ are the possible return values.

	$\top$	t	f	$\bot$
$>$	never	$\mathsf{F}(r) > \mathsf{F}(s)$	$\mathsf{F}(r) < \mathsf{F}(s)$	$\mathsf{F}(r) = \mathsf{F}(s)$
$\neq$	$\mathsf{F}(r) \neq \mathsf{F}(s)$	never	never	$\mathsf{F}(r) = \mathsf{F}(s)$
$\geq$	$\mathsf{F}(r) = \mathsf{F}(s)$	$\mathsf{F}(r) > \mathsf{F}(s)$	$\mathsf{F}(r) < \mathsf{F}(s)$	never
$=$	$\mathsf{F}(r) = \mathsf{F}(s)$	never	never	$\mathsf{F}(r) \neq \mathsf{F}(s)$

Figure 5: Return values for compare(r, s, OP F).

If $\mathsf{F}(r) > \mathsf{F}(s)$, t is returned. If $\mathsf{F}(s) > \mathsf{F}(r)$ instead, f is returned. However, if $\mathsf{F}(s) = \mathsf{F}(r)$, then $\bot$ is returned. For '$\neq$', only $\top$ and $\bot$ are possible. If $\mathsf{F}(s) \neq \mathsf{F}(r)$, then $\top$ is returned; else ($\mathsf{F}(s) = \mathsf{F}(r)$), $\bot$ is returned. Figure 5 summarizes the return values. ('$\geq$' / maxeq and '=' / equal will be introduced and discussed next in Section 4.)

$\circ$	$\top$	t	f	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
t	$\top$	t	$\top$	t
f	$\top$	$\top$	f	f
$\bot$	$\top$	t	f	$\bot$

Figure 6: How compare's truth values compose.

For any pair of tuples, r and s, we want to combine the results of the comparators (applied to r and s) in the filter, $\mathcal{F}$, to determine how the tuples relate. If for all (OP F) $\in \mathcal{F}$, compare(r, s, OP F) returns t or $\bot$, *and* for some (OP F) $\in \mathcal{F}$, it returns t, we say that r *trumps* s.

However, if for one comparator, (OP_1 F) $\in \mathcal{F}$, compare(r, s, OP_1 F) $= t$, but for another, (OP_2 G) $\in \mathcal{F}$, compare(r, s, OP_2 G) $= f$, then r and s are *incomparable*. There are two additional ways that r and s can be *incomparable*. If for all (OP F) $\in \mathcal{F}$, compare(r, s, OP F) $= \bot$, they are incomparable. Finally, if for any (OP F) $\in \mathcal{F}$, compare(r, s, OP F) $= \top$, they are incomparable. The last case can occur when OP is '$\neq$', which implements the intended semantics for diff correctly.

```
compare (tuple r, tuple s, set F) {
    result := ⊥;
    foreach F in F {
        result := result ∘ compare(r, s, F);
        if (result == ⊤) return ⊥;
    }
    return result;
}
```

Figure 7: The compare evaluation with a filter.

Thus, the truth-value results of the comparators from the filter compose as in Figure 6. The compare procedure in Figure 7 defines how two tuples are compared with respect to a filter.

Denote a set of tuples as $\mathbb{T}$, which we shall call the input *table*. For any $\mathbb{T}$, any skyline filter $\mathcal{F}$ induces an algebraic relation over $\mathbb{T}$. Call this the *preference relation* over $\mathbb{T}$ with respect to $\mathcal{F}$. Denote the preference relation by '$\succ_{\mathcal{F}}$', and define it as $r \succ_{\mathcal{F}} s$ *iff* compare(r, s, $\mathcal{F}$) $= t$. Call tuples r and s *incomparable* with respect to $\mathcal{F}$ *iff* $r \not\succ_{\mathcal{F}} s$ and $s \not\succ_{\mathcal{F}} r$. Denote this by $r \sim_{\mathcal{F}} s$.

Compare for filters (Figure 7) then returns t (*true*) if $r \succ_{\mathcal{F}} s$; f (*false*) if $s \succ_{\mathcal{F}} r$ instead; and $\bot$ (meaning incomparable in this context) if $s \sim_{\mathcal{F}} r$. Note that compare with respect to a filter does not return $\top$. We only use $\top$ when combining the comparison results of the comparators.

For any skyline filter $\mathcal{F}$ built over '$>$' (and '$\neq$') comparators, the preference relation '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ is guaranteed to be a partial order, and thus is *irreflexive*, *antisymmetric*, and *transitive*. The skyline of $\mathbb{T}$ is then defined as the *crown* of the partial order '$\succ_{\mathcal{F}}$' over $\mathbb{T}$. That is, it consists of those tuples in $\mathbb{T}$ that are not trumped with respect to $\mathcal{F}$ by any other tuples in $\mathbb{T}$.

Definition 1 *The skyline set is defined as*

$$\nabla_{\mathcal{F}}(\mathbb{T}) \equiv \{s \in \mathbb{T} \mid \neg\exists r \in \mathbb{T}.\ r \succ_{\mathcal{F}} s\}$$

Call this the crown skyline semantics.[3]

We can characterize the skyline set via *soundness* and *completeness* properties.

Definition 2 *The* soundness *property of skyline sets states that*

$$\forall s \in \nabla_{\mathcal{F}}(\mathbb{T}).\ \neg\exists r \in \mathbb{T}.\ r \succ_{\mathcal{F}} s$$

The completeness *property of skyline sets states that*

$$\neg\exists r \in (\mathbb{T} - \nabla_{\mathcal{F}}(\mathbb{T})).\ \forall s \in \nabla_{\mathcal{F}}(\mathbb{T}).\ s \not\succ_{\mathcal{F}} r$$

By soundness, we mean that each skyline tuple represents a *best* tuple; that is, there is no tuple that is better than it with respect to the skyline criteria. By completeness, we mean that the skyline set represents *all* the best tuples, with respect to the skyline criteria. Hence, every non-skyline tuple is trumped by some skyline tuple. In this way, the skyline set characterizes the table; for any tuple in the table, it is either skyline itself, or there is a skyline tuple that trumps it, and hence, "represents" it in the skyline set.

4 Stable Skyline

4.1 Extending Skyline

the sky is open

[3]The crown skyline semantics is the semantics for skyline as originally defined. We shall consider an alternate semantics, the *stable skyline semantics*, in Section 4.

it is the ground that stops us
up is all there is

We are interested in what extensions can be made to the skyline formalism that might make for a richer, more expressive class of skyline queries. In particular, we seek extensions that would facilitate composing skyline queries in meaningful ways.

In this paper, we present two extensions. First, we add a directive equal ('=') to the skyline comparators. This also consequently provides '≥', which we shall call maxeq. Second, we add a modifier by ('±') for the max ('>') and maxeq ('≥') comparators. These are summarized in Figure 8. The behavior of each of the new comparator constructs is determined as by the compare procedure in Figure 7. For now, let ϵ remain 0. We shall discuss by ('±') in Section 4.4.

skyline of	∇
F max by ϵ	$>\mathsf{F} \pm \epsilon$
F diff	$\neq\mathsf{F}$
F maxeq by ϵ	$\geq\mathsf{F} \pm \epsilon$
F equal	$=\mathsf{F}$

Figure 8: New skyline comparators (Stable Skyline).

4.2 EQUAL and Stable Skyline

transitivity
cannot always be preserved
but not all is lost

Adding '≥' (maxeq) to the skyline repertoire (without adding '=' / equal also) in one respect does not increase skyline's expressiveness.

Theorem 1 *For any skyline filter $\mathcal{F}$ composed of '>' and '≥' comparators and input table $\mathbb{T}$, there exists a skyline filter $\mathcal{F}'$ composed of just '>' comparators such that the preference relation '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ and '$\succ_{\mathcal{F}'}$' over $\mathbb{T}$ are equivalent.*

Proof. *It is possible to represent any partial order as the conjunction of total orders or as the conjunction of weak orders.*[4] □

While there is guaranteed to exist a skyline filter $\mathcal{F}'$ without use of '≥' that is equivalent to a skyline filter $\mathcal{F}$ with '≥', finding it is not straightforward. Such a $\mathcal{F}'$ can be *constructed* from $\mathcal{F}$ by ensuring each function involved in $\mathcal{F}'$ is a total order over $\mathbb{T}$. If F over $\mathbb{T}$ is a total order, then the comparators $\geq$F and $>$F act equivalently over $\mathbb{T}$. Without analyzing $\mathbb{T}$, one could not devise such an $\mathcal{F}'$ guaranteed to provide the same partial order as $\mathcal{F}$. So the addition of '≥' does *effectively* increase skyline's expressiveness. Skyline with '≥' can express the equivalent of the *intersection* operator, '⧫', as well as the *Pareto* operator, '⊗', of the Preference SQL of Kiessling et al. [24].

Adding '=' (equal) to the skyline repertoire increases skyline's expressiveness both effectively *and* logically. Skyline with '=' can express preference relations that skyline without '=' cannot. (This will become evident in the following discussion.) Equal is useful in composing more complex skyline queries. Note that '≥' is algebraically redundant, given '=' and '>'. Filter $\{\geq\mathsf{F}\} \cup \mathcal{F}$ can be written equivalently as $\{>\mathsf{F}, =\mathsf{F}\} \cup \mathcal{F}$. It will be convenient for discussion, however, to keep '≥' in our language.

Example 1 *Say the house hunter is interested only in* bungalow *and* modern*-style houses. In both cases, price should be minimized and #bdrm should be maximized. For bungalows, the house hunter prefers older houses, so yr_built should be minimized. For modern-style houses, however, the house hunter prefers newer houses, and yr_built should be maximized.*

Let P = price, B = #bdrm, Y = yr_built, and S = style. Let

$$\mathbb{H} = \sigma_{style='bungalow' \vee style='modern'}(HouseListing)$$

The skyline query can be composed as

$$\mathbb{Q}\text{: } \nabla_{\{<P,>B,=S\}}\Big(\nabla_{\{<P,>B,<Y\}}(\sigma_{style='bungalow'}(\mathbb{H})) \cup \nabla_{\{<P,>B,>Y\}}(\sigma_{style='modern'}(\mathbb{H}))\Big)$$

The use of equal in Example 1 is necessary for us to achieve what we intend. The outer '∇' combines the results of the skyline over the bungalow houses (with older as a criterion) and the skyline over the modern-style houses (with newer as a criterion). In the outer '∇' operation, we want that the bungalow houses returned be compared against the modern-style houses returned, and vice-versa, with respect to the criteria in common: lower price and more bedrooms. The '=' comparator is essential to ensure that the bungalow houses are not compared against one another *again*, but this time with respect to the fewer criteria; and likewise, that the modern-style houses are not either.

A consequence of adding equal, however, is that a skyline filter is no longer guaranteed to induce a partial order (PO) over the set of tuples. Transitivity may be lost. The preference relation is still guaranteed to be irreflexive and antisymmetric; hence, it is a directed acyclic graph (DAG).

Diff comparators ('≠') do not affect transitivity. This is obvious since a diff comparator can be replaced by max comparators ('>'), and a skyline filter consisting of just max comparators clearly induces a partial

[4] How many total orders are needed, via their conjunction, to represent an arbitrary partial order is known as the *partial order dimension* problem [17]. To determine the *partial order dimension*, given an arbitrary PO, has been proven to be NP-hard [37]. It has been proven that any partial order that can be represented via the conjunction of two or more total orders can also be represented via a conjunction of the same number of weak orders [18].

Of course, skyline is the opposite way around: we provide a conjunction of weak orders, which in turn, define a partial order.

order. Diff simply partitions the tuples, and only tuples in the same partition can relate (i.e., $r \succ_{\mathcal{F}} s$).

How can equal ('=') affect transitivity then? An equal comparator prohibits tuples from the same equality class (partition) to relate. In essence, it punches holes in the partial order of the preference relation that would be induced by the filter without its equal comparators, by making certain pairs of tuples incomparable which would have been comparable otherwise. These "holes" can violate transitivity.

Definition 3 *Given a skyline filter $\mathcal{F}$, define the* partial-order simplification *of $\mathcal{F}$, denoted by $\mathcal{F}^{>}$, as*

$$\{>F \mid (>F \pm \epsilon) \in \mathcal{F} \ \vee \ (\geq F \pm \epsilon) \in \mathcal{F}\}$$

For all r and s, $r \succ_{\mathcal{F}} s$ *if* $r \succ_{\mathcal{F}^{>}} s$; however, there may exist r and s such that $r \sim_{\mathcal{F}} s$ but $r \succ_{\mathcal{F}^{>}} s$.

HouseListing

#	Price	Style
r	$340k	bungalow
s	$350k	modern
t	$370k	bungalow

Figure 9: Table for example breaking transitivity.

Example 2 *Consider the table $\mathbb{T}$ in Figure 9, and the skyline filter $\mathcal{F} = \{<\mathsf{Price}, =\mathsf{Style}\}$. Thus, $r \succ_{\mathcal{F}} s$ and $s \succ_{\mathcal{F}} t$. However, $r \sim_{\mathcal{F}} t$.*

Note that the truth value $\top$ is needed in compare to accomplish this. Via $\top$, one comparator can declare two tuples to be incomparable, regardless of the results of the other comparators in the filter.

Of course, the original semantics for skyline—the crown skyline semantics, Definition 1—was with transitivity in mind. If we are now to permit DAG preference relations, we should re-examine the definition. There are three possible ways to proceed:

1. recover a partial order from the directed acyclic graph as the preference relation;
2. continue using the crown skyline semantics anyway; or
3. develop a new skyline semantics that accommodates directed-acyclic-graph preference relations naturally.

By idea 1, we want to derive a PO from the DAG. An obvious way to accomplish this would be to take the transitive closure of '$\succ_{\mathcal{F}}$', denoted by '$\succ^{*}_{\mathcal{F}}$'. Then the skyline could be defined with respect to '$\succ^{*}_{\mathcal{F}}$' instead. However, this is not good! Our purpose for adding equal is to defeat certain tuples from trumping certain other tuples. By using '$\succ^{*}_{\mathcal{F}}$', we essentially are undoing the effects of the equal comparators. Thus we rule out idea 1.

Idea 2 is simply to keep the same definition, Definition 1, for skyline: it is those tuples not trumped by any others, with respect to $\mathcal{F}$. Interestingly, Definition 1 does not depend on '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ being a partial order. We shall demonstrate, however, that this is not an ideal solution.

Once '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ is not transitive, we can no longer have both *soundness* and *completeness*, as defined in Definition 2. Both these properties are really intended as part of skyline's semantics. They are consequences of the skyline set in Definition 1 *when* the preference relation is a PO. They are not consequences, however, when the preference relation is not transitive. The crown skyline set—as defined in Definition 1—is no longer necessarily complete. There are non-skyline tuples potentially in table $\mathbb{T}$ that are *not* trumped by any skyline tuple. But these are not crown skyline tuples themselves—by Definition 1, that is—because other non-skyline tuples trump them.

Thus idea 3 is the direction in which we must proceed. We need to redefine *skyline* to recapture *soundness* and *completeness*. We shall be able to regain completeness if we are willing to redefine slightly our notion of correctness.

Loss of transitivity results in that the crown skyline set is no longer "stable". Consider table $\mathbb{T}$ and filter $\mathcal{F}$ from Figure 9 and Example 2 again. Only r is in $\nabla_{\mathcal{F}}(\mathbb{T})$. Consider when s is removed from $\mathbb{T}$. Now, $\nabla_{\mathcal{F}}(\mathbb{T}) = \{r, t\}$! So the addition or deletion of *non-skyline* tuples from the table can affect what the skyline set is.

We want a *stability* property for the skyline set. (This will lead back to *completeness*.) Changes to the table over non-skyline tuples should not change the skyline set. To accomplish this, we re-examine our notion of soundness for skyline.

Definition 4 Stability. *Call a subset $\mathbb{S}$ of table $\mathbb{T}$ a* stable skyline set *with respect to filter $\mathcal{F}$ and $\mathbb{T}$* iff

$$\mathbb{S} = \{r \in \mathbb{T} \mid \neg\exists s \in \mathbb{S}.\ s \succ_{\mathcal{F}} r\}$$

For a PO filter $\mathcal{F}$, $\nabla_{\mathcal{F}}(\mathbb{T})$ is a stable skyline set with respect to $\mathbb{T}$. For a DAG filter $\mathcal{F}$, for which transitivity is lost with respect to $\mathbb{T}$, $\nabla_{\mathcal{F}}(\mathbb{T})$ is not necessarily a stable skyline set. All tuples in $\mathbb{S}$ are pair-wise incomparable, as is the case for $\nabla_{\mathcal{F}}(\mathbb{T})$. And for each tuple from $\mathbb{T} - \mathbb{S}$, there is a tuple in $\mathbb{S}$ that trumps it, just as for $\nabla_{\mathcal{F}}(\mathbb{T})$. However, now a (stable) skyline tuple may be trumped by a non-skyline tuple. (For any such non-skyline tuple, though, there is some *other skyline* tuple that trumps it.) So we modify our notion of *soundness*: no skyline tuple is trumped by any other *skyline* tuple.

Can we find such an $\mathbb{S}$? Is it unique? We can, and it is unique. We define this via a transformation and a fixpoint with respect to the transformation.

Definition 5 *Define $\mathbf{S}_{\mathcal{F},\mathbb{T}}$, given argument $\mathbb{S}$, as follows.*

$$\mathbf{S}_{\mathcal{F},\mathbb{T}}(\mathbb{S}) = \{r \in \mathbb{T} \mid \neg\exists s \in \mathbb{S}.\ s \succ_{\mathcal{F}} r\}$$

Define $\mathbf{S}_{\mathcal{F},\mathbb{T}} \uparrow i$ as follows.

- $\mathbf{S}_{\mathcal{F},\mathbb{T}} \uparrow 0 = \emptyset$
- $\mathbf{S}_{\mathcal{F},\mathbb{T}} \uparrow i = \mathbf{S}_{\mathcal{F},\mathbb{T}}(\mathbf{S}_{\mathcal{F},\mathbb{T}} \uparrow (i-1))$, *for* $i > 0$

Let $\mathbf{S}^i_{\mathcal{F},\mathbb{T}}$ *be shorthand for* $\mathbf{S}_{\mathcal{F},\mathbb{T}} \uparrow i$.

Note that $\mathbf{S}^2_{\mathcal{F},\mathbb{T}} = \nabla_{\mathcal{F}}(\mathbb{T})$ (the crown skyline set by Definition 1). When '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ is a PO, $\mathbf{S}^2_{\mathcal{F},\mathbb{T}} = \mathsf{lfp}(\mathbf{S}_{\mathcal{F},\mathbb{T}})$, the *least fixpoint* of $\mathbf{S}_{\mathcal{F},\mathbb{T}}$. When '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ is a DAG, $\nabla_{\mathcal{F}}(\mathbb{T}) = \mathbf{S}^2_{\mathcal{F},\mathbb{T}}$ still, but it is possible that $\mathbf{S}^2_{\mathcal{F},\mathbb{T}} \neq \mathsf{lfp}(\mathbf{S}_{\mathcal{F},\mathbb{T}})$.

```
mark_depths (table 𝕋) {
    i := 0;
    𝔻 := 𝕋;
    while (𝔻 ≠ ∅) {
        𝕊 := {t ∈ 𝔻 | ¬∃r ∈ 𝔻. r ≻_F t};
        foreach t in 𝕊 {
            t.depth := i;
        }
        𝔻 := 𝔻 − 𝕊;
        i++;
    }
}
```

Figure 10: Procedure to mark tuple depths.

We introduce *tuple depth* for use in proving that $\mathbf{S}^i_{\mathcal{F},\mathbb{T}}$ reaches fixpoint. The procedure in Figure 10 assigns a *depth* to each tuple. Any tuple not trumped by any other tuple is assigned depth 0; inductively, any tuple not trumped by any other tuple not of depth i or less is assigned depth $i+1$.

Lemma 1 *For a tuple* $t \in \mathbb{T}$ *of depth* i *(as assigned by* mark_depths *in Figure 10), either* $\forall j \geq 2(i+1).\ t \in \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$ *or* $\forall j \geq 2(i+1).\ t \notin \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$.

Proof. *Proof by induction.*

base: *The set of tuples of depth* 0 *is equivalent to* $\mathbf{S}^2_{\mathcal{F},\mathbb{T}}$ *(and to* $\nabla_{\mathcal{F}}(\mathbb{T})$*). Thus, for any* $t \in \mathbb{T}$ *of depth* 0, $\forall j \geq 2.\ t \in \mathbf{S}^2_{\mathcal{F},\mathbb{T}}$.

hypothesis: *Assume the proposition is* true *for all tuples of depth* i*, for* $0 \leq i < k$*, for given* k.

induction: *Consider tuple* $t \in \mathbb{T}$ *of depth* k*. Consider a tuple* $r \in \mathbb{T}$ *such that* $r \succ_{\mathcal{F}} t$*. Tuple* r*'s depth then is less than* k*. If* $r \notin \mathbf{S}^{2(k+1)}_{\mathcal{F},\mathbb{T}}$*, then* $\forall j \geq 2(k+1).\ r \notin \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$*, by the hypothesis. Else if* $r \in \mathbf{S}^{2(k+1)}_{\mathcal{F},\mathbb{T}}$*, then* $\forall j \geq 2(k+1).\ r \in \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$*, by the hypothesis.*

Therefore, if $\exists r \in \mathbb{T}.\ (r \succ_{\mathcal{F}} t) \wedge r \in \mathbf{S}^{2(k+1)}_{\mathcal{F},\mathbb{T}}$*, then* $\forall j \geq 2(k+1).\ t \notin \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$*. Otherwise,* $\neg\exists r \in \mathbb{T}.\ (r \succ_{\mathcal{F}} t)$*, and* $\forall j \geq 2(k+1).\ t \in \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$. □

Theorem 2 *For any finite* $\mathbb{T}$*, the* least fixpoint *of* $\mathbf{S}_{\mathcal{F},\mathbb{T}}$*,* $\mathsf{lfp}(\mathbf{S}_{\mathcal{F},\mathbb{T}})$*, is obtained in finite iterations; thus, for any skyline filter* $\mathcal{F}$ *and input table* $\mathbb{T}$*, there exists* $k \in \omega$ *such that* $\mathbf{S}^k_{\mathcal{F},\mathbb{T}} = \mathbf{S}^{k+1}_{\mathcal{F},\mathbb{T}}$.

Proof. *Follows directly from Lemma 1.* □

We have established that there is a least fixpoint of $\mathbf{S}_{\mathcal{F},\mathbb{T}}$, that it reaches fixpoint in finite iterations, and that it is equivalent to a stable skyline set. Next, we prove that the stable skyline set is unique.

Lemma 2 *Given a finite table* $\mathbb{T}$ *and skyline filter* $\mathcal{F}$*, there exists only one stable skyline set (Definition 4). That is, the stable skyline set is unique.*

Proof. *For a given* $\mathbb{T}$ *and* $\mathcal{F}$*, assume there exist two distinct stable skyline sets* $\mathbb{S}$ *and* $\mathbb{R}$*. It is not possible that* $\mathbb{S} \supset \mathbb{R}$ *(or* $\mathbb{S} \subset \mathbb{R}$*), by the definition of stable skyline set. Thus, consider* $s \in (\mathbb{S} - \mathbb{R})$ *and* $r \in (\mathbb{R} - \mathbb{S})$*. By* $\mathbb{S}$*'s definition,* $s \succ_{\mathcal{F}} r$*. By* $\mathbb{R}$*'s definition,* $r \succ_{\mathcal{F}} s$*. Contradiction.* □

Definition 6 *Define the* stable skyline operator, '$\boldsymbol{\nabla}$'*, with respect to filter* $\mathcal{F}$ *and table* $\mathbb{T}$ *as*

$$\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T}) = \mathit{lfp}(\mathbf{S}_{\mathcal{F},\mathbb{T}})$$

Theorem 3 $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$ *is equivalent to the unique stable skyline set.*

Proof. *Follows from Theorem 2, Lemma 2, and Definitions 4 (for stable skyline set) and 5 (for* $\mathbf{S}_{\mathcal{F},\mathbb{T}}$*).* □

Definition 7 *The* soundness *property of stable skyline sets states that*

$$\forall s \in \boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T}).\ \neg\exists r \in \boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T}).\ r \succ_{\mathcal{F}} s$$

That '$\boldsymbol{\nabla}$' satisfies the soundness property of Definition 7 is a direct consequence of the definition of $\mathbf{S}_{\mathcal{F},\mathbb{T}}$ (Definition 5).

Stable skyline semantics has the following advantages over the original (crown) skyline semantics when DAG preference relations are permitted.

- It preserves *completeness* of the skyline set (Definition 2).
- It has a stability property (Definition 4), which is epistemically appealing.
- It is seemingly easier to compute than is the crown skyline set. (This is discussed in Section 4.3).
- It enables skyline operations to be composed in semantically sound ways.

When the preference relation is a partial order, stable skyline semantics and the original (crown) skyline semantics concur.

4.3 Computing Stable Skyline

can one hold the sky
within the palms of one's hands
in its completeness

The stable skyline set, $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$, is more straightforward than its formal definition via $\mathbf{S}^i_{\mathcal{F},\mathbb{T}}$ might suggest. It includes the crown skyline tuples ($\nabla_{\mathcal{F}}(\mathbb{T})$), as these are not trumped by any tuples. However, there may be tuples in $(\mathbb{T} - \nabla_{\mathcal{F}}(\mathbb{T}))$ not trumped by any tuples in $\nabla_{\mathcal{F}}(\mathbb{T})$. So $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$ also includes the (crown) skyline of these. And so forth.

Definition 8 *Define the* untrumped *set at stage* i*,* $\mathbf{N}^i_{\mathcal{F},\mathbb{T}}$*, as follows.*

$$\mathbf{N}^i_{\mathcal{F},\mathbb{T}} = \{r \in (\mathbb{T} - \mathbf{S}^i_{\mathcal{F},\mathbb{T}}) \mid \neg\exists s \in \mathbf{S}^i_{\mathcal{F},\mathbb{T}}.\ s \succ_{\mathcal{F}} r\}$$

Then

$$\begin{aligned} \mathbf{S}^{2i+1}_{\mathcal{F},\mathbb{T}} &= \mathbf{S}^{2i}_{\mathcal{F},\mathbb{T}} \cup \mathbf{N}^{2i}_{\mathcal{F},\mathbb{T}} && \text{for } i \geq 0 \\ \mathbf{S}^{2i+2}_{\mathcal{F},\mathbb{T}} &= \mathbf{S}^{2i}_{\mathcal{F},\mathbb{T}} \cup \nabla_{\mathcal{F}}(\mathbf{N}^{2i}_{\mathcal{F},\mathbb{T}}) && \text{for } i \geq 0 \\ \boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T}) &= \bigcup_{i=0}^{\omega} \nabla_{\mathcal{F}}(\mathbf{N}^{2i}_{\mathcal{F},\mathbb{T}}) \end{aligned}$$

Thus the iterations $\mathbf{S}^{i}_{\mathcal{F},\mathbb{T}}$ alternate, adding all presently untrumped tuples in the odd cycles, and reducing these by only retaining the crown of the newly added tuples in the even cycles.[5]

This seems to indicate that we should be able to devise an algorithm in which each tuple is considered just once, with respect to the accumulated skyline tuples so far, and either is discarded or added to the skyline set. We do this next.

```
sfs (array T) {
    // T: The input tuples, topologically sorted.
    array S; // For collecting the stable skyline set.
             // Initialized empty.
    for (i = 0; i < T.length; i++;) {
        trumped := false;
        j := 0;
        while ((j < S.length) ∧ !trumped) {
            if (S[j] ≻_F T[i]) trumped := true;
            j++;
        }
        if (!trumped) S.add(T[i]);
    }
    return S;
}
```

Figure 11: Sort-Filter-Skyline algorithm to compute the stable skyline.

The algorithm sort-filter-skyline (SFS) in Figure 11 computes the stable skyline set, $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$. This algorithm is a main-memory simplification of the external SFS algorithm we presented in [12]. Before the sfs procedure is called, the input tuples are sorted in an order that represents a linear extension of the "partial order" induced by the filter $\mathcal{F}$; that is, a total order that is compatible with the partial order. Of course, $\mathcal{F}$ may be a DAG filter.

Consider $\mathcal{F}^{>}$ from Definition 3. Then '$\succ_{\mathcal{F}^{>}}$' over $\mathbb{T}$ is a partial order. That partial order is an extension of '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ which is a DAG. So any topological sort with respect to '$\succ_{\mathcal{F}^{>}}$' suffices. (In [12], we show it is straightforward to find a suitable topological sort.)

Once the input table $\mathbb{T}$ has been sorted into array T, then skyline tuples are accumulated into array S. Note that a tuple t in array T cannot be trumped with respect to $\mathcal{F}$ by any tuple after it in the array, since the array is topologically sorted with respect to '$\succ_{\mathcal{F}}$'.

[5] Consequently, $\mathbf{N}^{2i+1}_{\mathcal{F},\mathbb{T}} = 0$, for $i \geq 0$.

Theorem 4 *Algorithm SFS in Figure 11 computes the stable skyline set, $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$.*

Proof. *Let $\mathbb{S}$ be the set of tuples returned by procedure sfs (via array S). Let*

$$\mathbb{R} = \{t \in \mathbb{T} \mid \neg\exists s \in \mathbb{S}.\ s \succ_{\mathcal{F}} t\}$$

Does $\mathbb{R} = \mathbb{S}$? Assume not.

$\exists t \in (\mathbb{S} - \mathbb{R})$. Since $t \notin \mathbb{R}$, $\exists s \in \mathbb{S}.\ s \succ_{\mathcal{F}} t$. Tuple s must appear before t then in the topological sort of array T. So it will be in array S when t is compared against the tuples of array S. However, t would then be eliminated by procedure sfs, and so it would not appear in $\mathbb{S}$ itself. Contradiction.

$\exists t \in (\mathbb{R} - \mathbb{S})$. Since $t \in \mathbb{R}$, $\neg\exists s \in \mathbb{S}.\ s \succ_{\mathcal{F}} t$. So procedure sfs would have added t to array S. Contradiction.

Thus, $\mathbb{R} = \mathbb{S}$, and procedure sfs finds a stable skyline set by Definition 4. By Lemma 2, this stable skyline set is unique. By Theorem 3, it follows that procedure sfs computes $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$. □

None of proposed algorithms for skyline—besides SFS—will work to compute the (crown) skyline set, $\nabla_{\mathcal{F}}(\mathbb{T})$, once the preference relation, '$\succ_{\mathcal{F}}$' over $\mathbb{T}$, is no longer transitive. To the best of our knowledge, none could be modified easily to accomplish this. Certainly none computes the stable skyline set, $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$. The algorithms rely inherently on transitivity of the preference relation.

The stable skyline set appears easier to compute than the crown skyline set. An inadvertent advantage of the stable skyline semantics—if skyline has been extended with maxeq and equal—then is that it is easier to evaluate.

A skyline query under the original skyline definition can be written as a regular, albeit awkward, SQL query. Interestingly, it does not appear that a stable skyline query can be written in SQL. From this perspective, an extension to SQL to support stable skyline would constitute a real increase in query expressiveness.[6]

4.4 BY: Cyclic Preferences

don't know what you want
going around in circles
straighten up sail straight

The introduction of by ('$\pm$') allows one to strengthen a criterion: tuple r is deemed better than s on criterion F only if F(r) exceeds F(s) *by* a given amount (ϵ). For instance, when house hunting, one might not feel a price difference of less than $5,000 between two houses is that important. Thus one house should not trump

[6] Within programming extensions of SQL such as Oracle's PL/SQL, of course it could be. Also under a full implementation of recursion in the SQL-4 standards, it could be expressed as well.

a second house on price *unless* it is at least $5,000 less expensive.

It might seem that a simple "solution" to this "by-so-much" issue (without adding by) would be to bin the house prices into 5000-wide buckets; e.g., P = round(Price/5000). Then we would use criterion <P in the query rather than Price. This does *not* work as expected. In [20], we showed that the number of skyline tuples will be *reduced* by binning. The intention here is to *increase* the number of skyline tuples by making it harder for one house to trump another. Thus, counter-intuitively perhaps, binning offers no solution for "by-so-much". The by modifier does give us the behavior we intend. Thus it provides a true increase in expressiveness for skyline.

The introduction of by—as well as certain other extensions we would like to make—has a worse effect on our semantics than before. Not only can the preference relation lose transitivity, as with the addition of '=' comparators, cycles may be introduced to the preference relation.

```
select Address, Price, #bdrm, Cond
    from HouseListing
    skyline of Price min by 5000, #bdrm max by 1,
            Cond max by 1
```

Figure 12: Query with by.

HouseListing

#	Address	Price	#bdrm	Cond
r	32 Elm	$356k	4	4
s	13 Oak	$353k	2	5
t	27 Pine	$350k	3	3

Figure 13: Table for example with cyclic preferences.

Example 3 *Consider the query in Figure 12 and the table HouseListing from Figure 13. Thus* $\mathcal{F} = \{<P \pm 5000, >B \pm 1, >C \pm 1\}$. *Note that* $r \succ_{\mathcal{F}} s$, $s \succ_{\mathcal{F}} t$, *and* $t \succ_{\mathcal{F}} r$.

While we are able to accommodate DAG preference relations with the stable skyline semantics, we do not believe that there is a suitable semantics for preference relations with cycles. There is a natural way that cyclic filters (those that induce cyclic preference relations) can be "repaired", though, to be DAG filters.

Namely, any skyline filter that contains a '$\geq$' comparator (with $\epsilon = 0$) is guaranteed to produce a cycle-free preference relation.

Definition 9 *Call any comparator* $\geq G$ *without a* '$\pm$' *modifier (or equivalently, with* $\epsilon = 0$*) a* ground *comparator. Call any skyline filter which contains a ground comparator a* ground *filter.*

Theorem 5 *For any input table* $\mathbb{T}$*, a ground skyline filter* $\mathcal{F}$ *is guaranteed to induce a cycle-free preference relation,* '$\succ_{\mathcal{F}}$' *over* $\mathbb{T}$.

Proof. *Assume not. Thus,* $\exists r_1, \ldots, r_k \in \mathbb{T}$. $r_i \succ_{\mathcal{F}} r_{i+1}$, *for* $i = 1, \ldots, k-1$, *and* $r_k \succ_{\mathcal{F}} r_1$. $\exists (\geq G) \in \mathcal{F}$. *For any* r *and* s *such that* $r \succ_{\mathcal{F}} s$, $G(r) > G(s)$. *Therefore,* $G(r_1) > \ldots > G(r_k) > G(r_1)$. *Contradiction.* □

For any skyline query that one might compose for which the induced preference relation is not certain to be cycle-free, one can guarantee it cycle-free by making the skyline filter ground. This can be done by adding a *judiciously chosen* ground comparator to the filter.

If we add a ground to the filter to ensure it is cycle-free, we would like that the ground perturb the original preference relation induced by the filter over the table as little as possible. It should, in essence, only affect the cycles. The ground $\geq$G should reflect the preference relation of $\mathcal{F}$ over $\mathbb{T}$.

Definition 10 *Recall filter* $\mathcal{F}^{>}$*, the partial-order simplification of filter* $\mathcal{F}$*, from Definition 3.* '$\succ_{\mathcal{F}^{>}}$' *over* $\mathbb{T}$ *(for any* $\mathbb{T}$*) is a partial order.*

A ground comparator is proper *with respect to filter* $\mathcal{F}$ iff $\forall r, s.\ r \succ_{\mathcal{F}^{>}} s \rightarrow G(r) > G(s)$.[7]

Thus a proper ground changes the filter in a "minimal" way, needed to make it cycle-free. For any filter $\mathcal{F}$, one proper ground $\geq$G is

$$\mathsf{G}(t) = \sum_{>\mathsf{F} \in \mathcal{F}^{>}} \mathsf{F}(t)$$

This G is not unique, however. *Any* monotone weighted linear equation over the functions in $\mathcal{F}^{>}$ would work. In essence, G is a *utility function.* The preference relation is not defined by G though, just influenced by it. One can think of the addition of G as approximating the user's intended preference relation by a cycle-free (well-behaved) preference relation.

Ground comparators are a means to ensure that the preference criteria taken together are, in essence, consistent (i.e., no cycles). Note that this would not be possible without the addition of '$\geq$' to skyline. A skyline filter without a ground comparator may still be cycle-free. A ground comparator is a *sufficient*, but not a *necessary*, condition for a filter to be cycle-free.

4.5 Other Extensions

where to go from here
we need our priorities
to say where to next

With our skyline formalism (Section 3) in place, the stable skyline semantics (Section 4.2) for accommodating DAG preference relations, and a means for rectifying cycles in preference relations (Section 4.4), there are many further extensions to skyline that we are now equipped to explore.

We have discussed the skyline filter as a set of comparators. However, there is no reason to insist that the only allowed element of a filter be a comparator;

[7] Note that this is with respect to *all possible* r and s, and not just with respect to a given table $\mathbb{T}$.

we might allow *skyline filters* as elements of filters. In other words, *nested skyline filters* can be considered. Note that the compare procedure in Figure 7, which evaluates two tuples with respect to a skyline filter, is written generally enough already to accommodate this.

Procedure compare$(r, s, \mathcal{F})$, in which $\mathcal{F}$ is a filter, may return one of three values: t (*true*) if $r \succ_{\mathcal{F}} s$, f (*false*) if $s \succ_{\mathcal{F}} r$, and $\perp$ (*bottom*) if $r \sim_{\mathcal{F}} s$. It never returns $\top$. *Top*, $\top$, is employed as a return value from a comparator to *override* essentially the return values from the other comparators in the filter.

Nested filters can be used to mitigate the results of comparators. Say that price (P) and condition (C) are highly correlated. (Let B be #bdrm.) The query $\boldsymbol{\nabla}_{\{<\mathsf{P},>\mathsf{C},>\mathsf{B}\}}(\mathbb{H})$ then would return most all of the houses.[8] Consider the query $\boldsymbol{\nabla}_{\{\{<\mathsf{P},>\mathsf{C}\},>\mathsf{B}\}}(\mathbb{H})$ instead. Because the conditions $<$P and $>$C are placed together in a sub-filter, this no longer happens. If one house is better on price, and the other is in better condition, when $\{<\mathsf{P}, >\mathsf{C}\}$ is evaluated, it will return $\perp$. In this case, whether the first house trumps the second will then be decided by $>$B.

Example 4 *Consider again the query $\mathbb{Q}$ from Example 1, but with '∇' replaced by '$\boldsymbol{\nabla}$'. Define the tuple function R as follows.*

$$\begin{array}{lll} \mathsf{R}(t) & = t.\mathsf{yr\_built} & \text{if } t.\mathsf{style} = \text{'modern'} \\ & \quad -1 \cdot t.\mathsf{yr\_built} & \text{if } t.\mathsf{style} = \text{'bungalow'} \\ & \quad 0 & \text{otherwise} \end{array}$$

Again, let

$$\mathbb{H} = \sigma_{\mathsf{style}=\text{'bungalow'} \vee \mathsf{style}=\text{'modern'}}(\mathsf{HouseListing})$$

Define query $\mathbb{R}$ as follows.

$$\mathbb{R}: \ \boldsymbol{\nabla}_{\{<\mathsf{P},>\mathsf{B},\{\neq\mathsf{S},>\mathsf{R}\}\}}(\mathbb{H})$$

Queries $\mathbb{Q}$ and $\mathbb{R}$ are semantically equivalent.

Theorem 6 *For any table HouseListing, queries $\mathbb{Q}$ and $\mathbb{R}$ from Example 4 evaluate to the same answer set under the stable skyline semantics.*

Proof. *Recall query $\mathbb{Q}$ from Example 1, but with '∇' replaced by '$\boldsymbol{\nabla}$':*

$$\begin{array}{l} \mathbb{Q}: \boldsymbol{\nabla}_{\{<P,>B,=S\}}(\\ \qquad \boldsymbol{\nabla}_{\{<P,>B,<Y\}}(\sigma_{style='bungalow'}(\mathbb{H})) \\ \qquad \cup \ \boldsymbol{\nabla}_{\{<P,>B,>Y\}}(\sigma_{style='modern'}(\mathbb{H})) \\ \quad) \end{array}$$

We prove by induction over the size of $\mathbb{H}$. Let $\mathbb{Q}_{\mathbb{H}}$ refer to query $\mathbb{Q}$ applied to table $\mathbb{H}$, and likewise, $\mathbb{R}_{\mathbb{H}}$.
base. *Clearly $\mathbb{Q}_{\mathbb{H}} = \mathbb{R}_{\mathbb{H}}$ when $\mathbb{H} = \emptyset$.*
hypothesis. *Assume that $\mathbb{Q}_{\mathbb{H}} = \mathbb{R}_{\mathbb{H}}$ for any $\mathbb{H}$ such that $|\mathbb{H}| < k$ for some $k > 0$.*
induction. *Consider an arbitrary tuple $t \notin \mathbb{H}$.*
case 1. $\mathbb{Q}_{\mathbb{H}} = \mathbb{Q}_{\mathbb{H}\cup\{t\}}$.
$t \notin \mathbb{Q}_{\mathbb{H}\cup\{t\}}$, So, if $t.\mathsf{style} =$ 'modern', $\exists r \in \mathbb{Q}_{\mathbb{H}}$.
$((r.\mathsf{style} =$ 'bungalow' $\wedge \ r \succ_{\{<P,>B\}} t) \ \vee$
$(r.\mathsf{style} =$ 'modern' $\wedge \ r \succ_{\{<P,>B,>Y\}} t))$. If instead $t.\mathsf{style} =$ 'bungalow', $\exists r \in \mathbb{Q}_{\mathbb{H}}$. $((r.\mathsf{style} =$ 'modern' $\wedge \ r \succ_{\{<P,>B\}} t) \ \vee \ (r.\mathsf{style} =$ 'bungalow' $\wedge$ $r \succ_{\{<P,>B,<Y\}} t))$. This implies that $r \succ_{\{<P,>B,\{\neq S,>R\}\}} t$. Thus $r \in \mathbb{R}_{\mathbb{H}}$, by the hypothesis. Therefore, $t \notin \mathbb{R}_{\mathbb{H}}$, by completeness.
$\mathbb{Q}_{\mathbb{H}\cup\{t\}} = \mathbb{R}_{\mathbb{H}\cup\{t\}}$.
case 2. $\mathbb{Q}_{\mathbb{H}} \cup \{t\} = \mathbb{Q}_{\mathbb{H}\cup\{t\}}$.
$\forall r \in \mathbb{Q}_{\mathbb{H}}$. $r \sim_{\{<P,>B,=S\}} t$ by soundness and completeness. Furthermore, $\forall r \in \mathbb{Q}_{\mathbb{H}}$.
$(r.\mathsf{style} =$ 'modern' $\wedge \ t.\mathsf{style} =$ 'modern' $\rightarrow$ $r \sim_{\{<P,>B,>Y\}} t)$ and $\forall r \in \mathbb{Q}_{\mathbb{H}}$.
$(r.\mathsf{style} =$ 'bungalow' $\wedge \ t.\mathsf{style} =$ 'bungalow' $\rightarrow$ $r \sim_{\{<P,>B,<Y\}} t)$. This implies that $\forall r \in \mathbb{Q}_{\mathbb{H}}$. $r \succ_{\{<P,>B,\{\neq S,>R\}\}} t$. Thus, $\mathbb{R}_{\mathbb{H}\cup\{t\}} = \mathbb{R}_{\mathbb{H}} \cup \{t\}$.
$\mathbb{Q}_{\mathbb{H}\cup\{t\}} = \mathbb{R}_{\mathbb{H}\cup\{t\}}$.
case 3. *$t \in \mathbb{Q}_{\mathbb{H}\cup\{t\}}$, but $\mathbb{Q}_{\mathbb{H}} \cup \{t\} \neq \mathbb{Q}_{\mathbb{H}\cup\{t\}}$.*
$\exists r \in \mathbb{Q}_{\mathbb{H}}$. $t \succ_{\{<P,>B,=S\}} r$. Then $\mathbb{Q}_{\mathbb{H}\cup\{t\}} = \mathbb{Q}_{(\mathbb{H}-r)\cup\{t\}}$ by stability.
$\mathbb{R}_{(\mathbb{H}-r)\cup\{t\}} = \mathbb{Q}_{(\mathbb{H}-r)\cup\{t\}}$ then by the hypothesis.
$t \succ_{\{<P,>B,\{\neq S,>R\}\}} r$, as reasoned in cases 1 and 2, so $\mathbb{R}_{(\mathbb{H}-r)\cup\{t\}} = \mathbb{R}_{\mathbb{H}\cup\{t\}}$. $\mathbb{Q}_{\mathbb{H}\cup\{t\}} = \mathbb{R}_{\mathbb{H}\cup\{t\}}$.
Thus, $\mathbb{Q}_{\mathbb{H}} = \mathbb{R}_{\mathbb{H}}$, for all $\mathbb{H}$. □

Nested skyline filters offer a means to combine different partial orders (and DAG's) resulting from different preference sub-queries.

5 Conclusions

desire is perverse
it is easily confused
is that one better

Skyline is an elegant way to combine preference criteria, but has been limited in the preferences that can be expressed and how they can be combined. We believe that our extended skyline—with '=', '$\geq$', and '$\pm$'—and the stable skyline semantics move us beyond this impasse, and offer an initial proof-of-concept that skyline can be used as the basis of an expressive preference query language.

The advantages of our extended skyline are as follows: it is a *simple*, understandable construction; it is more naturally *composed* with itself and the relational operators; it is *realizable*, as by the SFS algorithm in Figure 11, with optimizations and improvements possible; it is *expressive*, allowing us to write many of the preference queries studied with winnow and in the Preference SQL; and the stable skyline semantics is *declarative*.

We must next study how *adjustable* stable skyline queries are, and how to modify stable skyline queries with predictable results. We want to understand the

[8] If P and C are 100% correlated, *all* of the houses (tuples) will be returned.

expressiveness of stable skyline under further extensions and their compositions. These include the nested skyline filters from Section 4.5 and user-defined partial orders. A next major goal is to find *natural* ways that preference criteria may be *prioritized*; e.g., Price and Cond are more important than OceanView. We also need to understand how to use ground comparators, how to determine when a skyline filter is cycle-free already, and how to define what are the *best* proper grounds. We plan to continue to study how to evaluate efficiently skyline queries, with these extensions, and under new semantics such as the stable skyline, and how to optimize these queries. Thus, we want to develop more efficient algorithms for stable skyline. We want also to map semantic equivalences, like queries $\mathbb{Q}$ and $\mathbb{R}$ in Example 4. These equivalences may enlighten us on how preferences can be composed, and how they can be used to optimize skyline queries.

References

[1] Agrawal, R., and Wimmers, E. L. A framework for expressing and combining preferences. In *Proc. of SIGMOD* (May 2000), pp. 297–306.

[2] Agrawal, S., Chaudhuri, S., Das, G., and Gionis, A. Automated ranking of database query results. In *Proc. of CIDR* (Jan. 2003).

[3] Bosc, P., and Pivert, O. Sqlf: A relational database language for fuzzy querying. *IEEE Trans. on Fuzzy Sys. 3*, 1 (Feb. 1995), 1–17.

[4] Boutilier, C., Brafman, R. I., Domshlak, C., Hoos, H. H., and Poole, D. CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *JAIR 21* (2004), 135–191.

[5] Börzsönyi, S., Kossmann, D., and Stocker, K. The skyline operator. In *Proc. of ICDE* (Apr. 2001), pp. 421–430.

[6] Chang, C. L. Deduce: A deductive query language for relational data bases. In *Pattern Rec. and Art. Int.*, C. H. Chen, Ed. Academic Press, New York, 1976, pp. 108–134.

[7] Chaudhuri, S., and Gravano, L. Evaluating top-k selection queries. In *Proc. of VLDB* (Sept. 1999), pp. 397–410.

[8] Chomicki, J. Querying with intrinsic preferences. In *Proc. of EDBT* (2002), Springer (LNCS 2287), pp. 34–51.

[9] Chomicki, J. Preference formulas in relational queries. *ACM TODS 28*, 4 (Dec. 2003), 427–466.

[10] Chomicki, J. Semantic optimization of preference queries. In *1st Int. Sym. on Appl. of Constraint Databases* (2004), Springer (LNCS 3074).

[11] Chomicki, J., Godfrey, P., Gryz, J., and Liang, D. Skyline with presorting. Tech. Rep. 2002-04, CS, York University, Toronto, ON, Canada, Oct. 2002. Long version of [12].

[12] Chomicki, J., Godfrey, P., Gryz, J., and Liang, D. Skyline with presorting. In *Proc. of ICDE* (Mar. 2003), pp. 717–719.

[13] Chu, W. W., Yang, H., Chiang, K., Minock, M., Chow, G., and Larson, C. CoBase: A scalable and extensible cooperative information system. *JIIS 6*, 2/3 (May 1996), 223–259.

[14] Eng, P.-K., Ooi, B. C., and Tan, K.-L. Indexing for progressive skyline computation. *Data and Knowl. Eng. 46*, 2 (2003), 169–201.

[15] Fagin, R., Kumar, R., Sivakumar, D., Mahdian, M., and Vee, E. Comparing and aggregating rankings with ties. In *Proc. of PODS* (June 2004), pp. 47–58.

[16] Gaasterland, T., and Lobo, J. Qualifying answers according to user needs and preferences. *Fundamenta Informaticæ 32*, 2 (1997), 121–137.

[17] Garey, M. R., and Johnson, D. S. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[18] Garg, V. K., and Skawratananond, C. String realizers of posets with applications to distributed computing. In *Proc. of Princ. of Distr. Comp.* (Aug. 2001), pp. 72–80.

[19] Godfrey, P. Minimization in cooperative response to failing database queries. *Int. Journal of Coop. Inf. Sys. 6*, 2 (June 1997), 95–149.

[20] Godfrey, P. Skyline cardinality for relational processing. In *Proc. of FoIKS* (Feb. 2004), Springer, pp. 78–97.

[21] Hristidis, V., Koudas, N., and Papakonstantinou, Y. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proc. of SIGMOD* (May 2001), pp. 259–270.

[22] Kaplan, S. J. Appropriate responses to inappropriate questions. In *Elements of Discourse Understanding*, A. Joshi, B. Webber, and I. Sag, Eds. Cambridge University Press, 1981, pp. 127–144.

[23] Kaplan, S. J. Cooperative responses from a portable natural language query system. *Artificial Intelligence 19*, 2 (Oct. 1982), 165–187.

[24] Kiessling, W. Foundations of preferences in database systems. In *Proc. of VLDB* (Aug. 2002), pp. 311–322.

[25] Kiessling, W. Preference constructors for deeply personalized database queries. Tech. Rep. 2004-7, University of Augsburg, Augsburg, Germany, July 2004.

[26] Kiessling, W., and Köstler, G. Preference SQL: Design, implementation, experiences. In *Proc. of VLDB* (Aug. 2002), pp. 990–1001.

[27] Kossmann, D., Ramsak, F., and Rost, S. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. of VLDB* (Aug. 2002), pp. 275–286.

[28] Kung, H., Luccio, F., and Preparata, F. P. On finding the maxima of a set of vectors. *JACM 22*, 4 (1975), 469–476.

[29] Lacroix, M., and Lavency, P. Preferences: Putting more knowledge into queries. In *Proc. of VLDB* (Sept. 1987), pp. 217–225.

[30] Lacroix, M., and Pirotte, A. Domain-oriented relational languages. In *Proc. of VLDB* (1977), IEEE Computer Society, pp. 370–378.

[31] Minker, J. An overview of cooperative answering in databases. In *Proc. of FQAS* (1998), T. Andreasen, H. Christiansen, and H. L. Larsen, Eds., pp. 282–285.

[32] Motro, A. Supporting goal queries in relational databases. In *Proc. of the 1st Int. Conf. on Expert Database Sys.* (Apr. 1986), pp. 85–96.

[33] Motro, A. Panorama: A database system that annotates its answers to queries with their properties. *JIIS 7*, 1 (Sept. 1996), 51–74.

[34] Pradhan, S. Argumentation databases. In *Proc. of ICLP* (Dec. 2003), Springer (LNCS 2916), pp. 178–193.

[35] Reiter, R. A logic for default reasoning. *Artificial Intelligence 13*, 1–2 (1980), 81–132.

[36] Yalamanchi, A., Srinivasan, J., and Gawlick, D. Managing expressions as data in relational database systems. In *Proc. of CIDR* (Jan. 2003).

[37] Yannakakis, M. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods 3*, 3 (1982), 351–358.