Stable Skyline*

Parke Godfrey[1]
[1]York University
Toronto, ON M3J 1P3
CANADA

Wei Ning[1,2]
[2]Microsoft
Redmond, WA 98052-8300
U.S.A.

Abstract

Many applications today—from e-commerce to analyzing scientific data-sets—require the notion of *best-match*. That is, one needs data that *best* satisfies one's criteria. Relational query languages like SQL, however, are ill-suited for this. An SQL query returns tuples that match precisely its conditions.

The *skyline clause* was proposed, in part, with this goal in mind. Skyline combines multiple preference criteria in parallel. (The skyline clause filters out any tuple if there is another one better than, or equal to, it on all criteria, and strictly better than it on at least one.) How to compute skyline queries efficiently has been well studied. However, SQL with the skyline clause does not capture many types of preferences.

Skyline conflates the notion of *filtering* for the *best* answers, and the *preference semantics* that defines best. A preference semantics defines an ordering over the answers, the *preference graph*. For skyline, the resulting preference graph is a partial order, both acyclic and transitive. We believe, however, that more general, natural preference semantics may not preserve transitivity, but result more generally in directed-acyclic preference graphs instead.

We introduce a generalization of skyline that can result in directed, acyclic preference graphs instead of partial orders. We develop the *stable skyline semantics* to accommodate this. We show how stable skylines can be computed.

1. Introduction

"Find me a house in the Annex in good condition for less than $500,000 with at least three bedrooms and a backyard."[1] 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* (Kaplan 1981; 1982).

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 person 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.[2] Furthermore, the house hunter's criteria are not *conditions*, per se. Rather, they are *preferences*: *in reasonable condition*, *inexpensive*, *many bedrooms*, and *has a backyard*. The house hunter may not even really expect to find a house that actually satisfies all these preferences, or that satisfies them in equal measure. Rather, the house hunter is looking for the *best* options, as measured against these preferences.

The skyline clause was proposed in (Börzsönyi, Kossmann, and Stocker 2001) as an extension to SQL, with syntax as in Fig. 1. Skyline offers an elegant approach to combining multiple preference criteria in parallel.

```
select ... from ... where ...
    group by ... having ...
    skyline of A_1 [min | max | diff], ...,
               A_n [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.[3] The skyline criteria taken together (conjunctively) then impose a *partial order* over the input tuples.[4] The skyline set is the *crown* of this partial order.

*This work was investigated while Wei Vicky Ning was a graduate student at York University. She is now with Microsoft.

[1]The Annex is a neighborhood in midtown Toronto.

[2]A house for $500,000 in the Annex? Really!

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

[4]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.

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

Figure 2: Skyline query for the house hunter.

The query in Fig. 2 is a skyline query expressing the house hunter's query. 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. Backyard 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).

One may criticize that the idea skyline is not new. Indeed, this very same idea has been studied before as the *maximal vector problem*, and is the same concept as *Pareto optimal*. What is new is that skyline introduced the concept to queries as a means to handle preferences in a natural way within existing relational query languages.

One might criticize that skyline is too weak in itself to provide for a useful preference query language. Skyline by itself provides us only with a limited way to express and combine preferences. We agree.

Lastly, from the point of view of work in preferences, one might criticize that the skyline operator conflates the selection of the best answers and the preference semantics that orders the answers. Indeed, we will want to separate these concerns carefully, to consider how we can generalize the notion of skyline to cover richer types of preferences.

```
skyline of Price min, #bdrm max,
     case style
        when 'modern'   then Yr_built max
        when 'Victorian' then Yr_built min
     end
```

Figure 3: Query with conditional preferences.

Consider the query fragment in Fig. 3. For modern-style houses, we are interested in ones that are newer. However, for Victorian-style houses, we prefer the opposite, those that are older. We want different preferences to apply to different data. Whether we want a newer house or an older house is conditional on whether the house's style is modern or Victorian, respectively. Case is not a preference construct provided in skyline, but would be a seemingly useful addition. However, it is far from simple to add such a construction. It does not work with the skyline semantics as defined.

We propose a way to generalize skyline to provide a basis for supporting richer preferences such as case. In 2., we review related work. In 3., we motivate and define stable skyline. First, we formalize skyline queries, to show how "implement" conditional preferences, and to lay the groundwork for implementing other potentially useful preference constructions in the future. In 4., we define the *stable skyline semantics* which accommodates the loss of transitivity. This preserves the desirable properties that the crown semantics for SQL with the skyline clause has. We present an algorithm to compute the stable skyline set and discuss further means to accomplish this efficiently. In 5., we discuss further issues, future work, and conclude.

2. Related Work

People have long recognized a need for preferences in database queries. In (Chang 1976), a deductive query language called DEDUCE was proposed for relational databases which includes preferences. Lacroix and Pirotte (Lacroix and Pirotte 1977) introduced the *domain relational calculus* (DRC) and the *intermediate level language*, ILL, as an English-like language for structured expressions with the goal of more natural, more expressive query languages. In (Lacroix and Lavency 1987), Lacroix and Lavency extended the DRC to provide a preference mechanism. Preferences in a query are satisfied if possible, but "ignored" when not.

Chomicki 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 (Chomicki 2002; 2003). His model and winnow are quite expressive. 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* (graph), the order the preferences induce over the potential answer tuples. Thus, winnow is a quite rich model. However, 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 in a relational system. In (Chomicki 2004), 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 (Kießling 2002), 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 (Kießling and Köstler 2002), 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.

How to compose preferences in Preference SQL meaningfully can be challenging. Because Preference SQL introduces many new constructs, how to realize it efficiently is a challenge. It 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 (Kießling

2004), 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 (Börzsönyi, Kossmann, and Stocker 2001) (as discussed in §1.). Much work since has gone into developing efficient, external, relationally well-behaved algorithms for evaluating skyline queries (Börzsönyi, Kossmann, and Stocker 2001; Chomicki et al. 2003; Eng, Ooi, and Tan 2003; Godfrey 2004; Godfrey, Shipley, and Gryz 2007; Kossmann, Ramsak, and Rost 2002). 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. The ideas leading to stable skyline are discussed in depth in (Godfrey and Ning 2004; Ning 2005).

3. Motivation & Background

Let the skyline operator, '∇', correspond to the skyline clause. It can be treated as a new relational algebra operator. In the skyline clause in Fig. 1, the A_i's are columns. (Of course, these may include derived columns, as well.) Each A_i can be considered 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. Call such a function-directive pair a skyline *comparator*, and the set of function-directive pairs of a skyline clause the skyline *filter*.

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.

The single operator '$>$' (max) would logically suffice; both '$<$' and '$\neq$' are algebraically redundant, given '$>$'. A skyline filter $\mathcal{F} \cup \{\neq\mathsf{A}\}$ is equivalent to $\mathcal{F} \cup \{<\mathsf{A}, >\mathsf{A}\}$.[5] A skyline filter $\mathcal{F} \cup \{<\mathsf{A}\}$ is equivalent to $\mathcal{F} \cup \{>(-1 \cdot \mathsf{A})\}$. In proofs then, we restrict our attention to '$>$', dismissing '$<$' and '$\neq$'. In discussion and in 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 '$>$'.

The skyline filter—a set of comparators—then defines how tuples are to be compared, to determine which will be returned in the skyline set. 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 graph* over $\mathbb{T}$ with respect to $\mathcal{F}$. Let $\mathcal{F}$ be composed of just '$>$' comparators, without loss of generalization. Denote the preference graph (relation) by '$\succ_{\mathcal{F}}$'; $r \succ_{\mathcal{F}} s$ *iff* $(\forall(>\mathsf{A}) \in \mathcal{F}.\ \mathsf{A}(r) \geq \mathsf{A}(s)) \wedge (\exists(>\mathsf{A}) \in \mathcal{F}.\ \mathsf{A}(r) > \mathsf{A}(s))$.

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 $r \sim_{\mathcal{F}} s$.

For any skyline filter $\mathcal{F}$ built over '$>$' comparators, the preference graph '$\succ_{\mathcal{F}}$' 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.[6]

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*

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

or equivalently

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

By soundness, we mean that each skyline tuple represents a *best* tuple; that is, there is no tuple that is better than it—that trumps 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.

We can now consider how we might extend this formalism in useful ways, in particular so that we can implement conditional preferences (as used in the query in Fig. 3).

Note that two tuples r and s are incomparable either if, for every $(>\mathsf{A}) \in \mathcal{F}$, $\mathsf{A}(r) = \mathsf{A}(s)$, *or* there are two comparators $(>\mathsf{A}), (>\mathsf{B}) \in \mathcal{F}$ such that $\mathsf{A}(r) < \mathsf{A}(s)$ and $\mathsf{B}(r) > \mathsf{B}(s)$. A single directive diff ('$\neq$') by itself can deem two tuples as incomparable if their values on the corresponding function differ, regardless of the other comparators.

We add a new directive, equal ('$=$'), as the dual of diff. With equal, two tuples are deemed incomparable if their values on the corresponding function are the *same*, regardless of the other comparators. If they differ in value on the function, the other comparators then determine how they relate. Interestingly, this addition increases skyline's expressiveness. Skyline with '$=$' can express preference graphs that skyline without '$=$' cannot. Thus, equal is useful in composing more complex skyline queries.

Definition 3 *The meaning of* equal *is defined via its effect in a skyline filter.*

[5] This may not seem natural for nominal values such as *modern* and *Victorian*. However, we consider that the column function has translated these to reals, or integers, so they *are* ordered. Likewise, string data can simply be lexicographically ordered. While it would be nonsensical to use a max comparator for nominal data by itself, we can sensibly exploit this mechanism to define '$\neq$' (diff) in terms of '$>$' (max).

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

- *If* $(=A) \in \mathcal{F}$ *and* $A(r) = A(s)$, $r \sim_{\mathcal{F}} s$, *regardless of any other comparators in* $\mathcal{F}$.
- *If* $(=A) \in \mathcal{F}$ *and* $A(r) \neq A(s)$, $r \succ_{\mathcal{F}} s$ iff $r \succ_{\mathcal{F}-\{=A\}} s$;
- *If* $A(r) \neq A(s)$ *but* $r \not\succ_{\mathcal{F}-\{=A\}} s$ *and* $s \not\succ_{\mathcal{F}-\{=A\}} r$, $r \sim_{\mathcal{F}} s$ *(as the tuples are considered equivalent with respect to the conditions).*

Example 4 *Consider a slightly simplified version of the example from Fig. 2. The house hunter is interested only in* Victorian *and* modern*-style houses. In both cases,* Price *should be minimized and #bdrm should be maximized. For Victorian-style houses, the house hunter prefers older houses, and so* yr_built *should be minimized. For modern-style houses, however, the house hunter prefers newer houses, and so* yr_built *should be maximized.*

Let $P =$ Price, $B =$ #bdrm, $Y =$ Yr_built, *and* $S =$ Style. *Let*

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

The skyline query can be composed as

$$\mathbb{Q}\text{: } \nabla_{\{<_P, >_B, =_S\}}(\nabla_{\{<_P, >_B, <_Y\}}(\sigma_{style='Victorian'}(\mathbb{H})) \cup \nabla_{\{<_P, >_B, >_Y\}}(\sigma_{style='modern'}(\mathbb{H})))$$

The use of equal in Example 4 is necessary for us to achieve what we intend. The outer '∇' combines the results of the skyline over the Victorian houses (with older as a criterion) and the skyline over the modern-style houses (with newer as a criterion). We want that the Victorian 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 '$=$' directive is essential to ensure that the Victorian houses are not compared against one another *again*, but this time with respect to the fewer criteria (Price and #bdrm); and likewise, that the modern-style houses are not either.[7]

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 graph is still guaranteed to be irreflexive and antisymmetric; hence, it is a directed acyclic graph (DAG).

Diff comparators ('$\neq$') 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 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 graph 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.

[7] Ideally, this query would be written with a single '∇' operator inducing a single preference relation over $\mathbb{H}$. With a generalization of equal, this can be done (Godfrey and Ning 2004). We have not done it here out of the need for brevity.

HouseListing

#	Price	Style	Yr_built	#bdrm
r	$340k	Victorian	1920	3
s	$350k	modern	1934	3
t	$370k	Victorian	1903	3

Figure 4: Table for example breaking transitivity.

Example 5 *Consider the table* $\mathbb{T}$ *in Fig. 4, and the skyline filter* $\mathcal{F} = \{<$Price$, =$Style$\}$. *Thus,* $r \succ_{\mathcal{F}} s$ *and* $s \succ_{\mathcal{F}} t$. *However,* $r \sim_{\mathcal{F}} t$.

Consider Ex. 4 again, the *implicit* preference relation, '$\succ$', induced by the query over $\mathbb{H}$, and the example tuples from Fig. 4. Because house r is less expensive than house s (and they tie on number of bedrooms), $r \succ s$. Note that since they are of different styles, we do not compare Yr_built between them. Likewise, because house s is less expensive than house t, $s \succ t$. However, $r \sim t$, because they are of the same style and t is older than r, preferred for Victorian houses.

4. Stable Skyline Semantics

Considerations

We would like to maintain the same type of semantics for our generalized skyline as for the original skyline: an answer tuple is one that is not trumped by any other with respect to the preference graph. So the answer set with respect to a query should seemingly just be the set of all tuples that are not trumped.

Of course, the original semantics for skyline—the crown skyline semantics, Def. 1—is with transitivity in mind. Since we now permit DAG preference graphs, we must re-examine this definition. There are three possible ways to proceed:

1. recover a partial order from the directed acyclic graph as the preference graph;
2. continue using the crown skyline semantics anyway; or
3. develop a new skyline semantics that accommodates directed-acyclic-graph preference graphs 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, for example, 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, Def. 1, for skyline: it is those tuples not trumped by any others, with respect to $\mathcal{F}$. Interestingly, Def. 1 does not depend on '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ being a partial order. However, this is not an ideal solution either. Once '$\succ_{\mathcal{F}}$' over $\mathbb{T}$ is not transitive, we can no longer have both *soundness* and *completeness*, as defined in Def. 2. Both these properties are really intended as part of skyline's semantics. They are consequences of the skyline set in Def. 1, *when* the preference graph is a PO. The crown skyline set—as defined in Def. 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 Def. 1, that is—because other non-skyline tuples trump them.

Therefore, idea #3 is the direction in which we 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.

Preference semantics that allow for cyclic preference graphs are, of course, possible. Let us, however, assume that any query induces an *acyclic* preference graph. Skyline extended with equal, as in §3., will only result in directed, acyclic preference graphs.

Definition

Loss of transitivity results in that the crown skyline set is no longer "stable". Consider table $\mathbb{T}$ and filter $\mathcal{F}$ from Fig. 4 and Example 5 (p. 4) 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 6 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 $\mathcal{F}$, $\nabla_{\mathcal{F}}(\mathbb{T})$ from Def. 1 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 7 *Define $\mathbf{S}_{\mathcal{F},\mathbb{T}}$, given table $\mathbb{S}$, 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 Def. 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 5: 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 Fig. 5 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 8 *For a tuple $t \in \mathbb{T}$ of depth i (as assigned by* mark_depths *in Fig. 5), either $\forall j \geq 2(i+1).\ t \in \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$ or $\forall j \geq 2(i+1).\ t \notin \mathbf{S}^j_{\mathcal{F},\mathbb{T}}$.*

Theorem 9 *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 $\mathbf{S}^k_{\mathcal{F},\mathbb{T}} = \mathbf{S}^{k+1}_{\mathcal{F},\mathbb{T}}$.*

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 10 *Given a finite table $\mathbb{T}$ and skyline filter $\mathcal{F}$, there exists only one stable skyline set (Def. 6). That is, the stable skyline set is unique.*

Definition 11 *Define the* stable skyline operator, '$\boldsymbol{\nabla}$'*, with respect to filter $\mathcal{F}$ and table $\mathbb{T}$ as*

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

Theorem 12 *$\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$ is equivalent to the unique stable skyline set.*

Definition 13 *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 Def. 13 is a direct consequence of the definition of $\mathbf{S}_{\mathcal{F},\mathbb{T}}$ (Def. 7).

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

1. It preserves *completeness* of the set (Def. 2).
2. It has a stability property (Def. 6), which is epistemically appealing.
3. It is easier to compute than is the crown skyline set. (This is discussed next.)
4. It enables skyline operations to be composed in sound ways.

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

Computing Stable Skyline

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 14 *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.[8]

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

The algorithm sort-filter-skyline (SFS) in Fig. 6 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 (Chomicki et al. 2002; 2003). 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. So any topological sort of that order suffices. In (Chomicki et al. 2002; 2003), we show it is straightforward to find a suitable topological sort.

Of course, $\mathcal{F}$ may be a DAG filter. However, it can be extended in a similar way. Consider the filter that results from $\mathcal{F}$ when all equal comparators are removed. It induces a preference relation that is a partial order. Any topological sort of that order is also a topological sort of the DAG induced by $\mathcal{F}$.

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 '$\succ_{\mathcal{F}}$'.

Theorem 15 *Algorithm SFS in Fig. 6 computes the stable skyline set,* $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$.

None of proposed algorithms for skyline will work to compute the (crown) skyline set, $\nabla_{\mathcal{F}}(\mathbb{T})$, once the preference graph, '$\succ_{\mathcal{F}}$' over $\mathbb{T}$, is no longer transitive. It seems none could be easily fixed. The algorithms rely inherently on transitivity of the preference graph. None were designed, of course, to compute the stable skyline set, $\boldsymbol{\nabla}_{\mathcal{F}}(\mathbb{T})$. Since they rely on transitivity, it is doubtful they could be adapted.

Once we admit non-transitivity, computing the crown semantics becomes inherently expensive. This is because tuples may be trumped by non-skyline tuples while by no crown-skyline tuples. Therefore, comparisons that can be avoided when transitivity is guaranteed now cannot be. With stable skyline, however, every non-skyline tuple is trumped by some stable-skyline tuple. Thus, many comparisons can be avoided, as is possible with crown skyline over partial orders. Stable skyline is much more efficient to compute than crown skyline for DAG preference relations.

In (Godfrey, Shipley, and Gryz 2007), we establish the average-case run-time complexities of the generic[9] skyline algorithms. We present a new skyline algorithm, LESS, that extends SFS, and establish that it is $\mathcal{O}(N)$ average-case running time. LESS improves on SFS's performance by eliminating tuples more quickly, but assumes transitivity to do this. It is interesting future work to see if LESS or other algorithms can be adapted for stable skyline.

5. Conclusions

Skyline queries offer an elegant way to combine preference criteria, but they are limited in the preferences that can be expressed and how they can be combined. We believe that a generalization of skyline that could handle conditionals (case) and related preference constructs would go a long ways towards making skyline a viable basis for a relational preference query language.

To handle case requires that skyline be defined over preference graphs that are directed acyclic graphs, without transitivity. We introduced *stable skyline* that extends skyline

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

[9] Algorithms that do not require pre-processing nor pre-existing data-structures such as indexes.

with a new directive, equal. This provides a natural way to implement case for skyline—and, potentially, many other preference constructs—but with the loss of transitivity. We developed the *stable skyline semantics* which recovers the properties of *soundness* and *completeness* of the skyline set in an elegant and natural way over directed, acyclic preference graphs. We showed how stable skyline could be computed efficiently.

We would like to be able to cover very rich preferences and preference constructs in skyline SQL queries. Thus, there is much work still to be done to extend semantics for skyline further to be able to define these meaningfully.

```
select Address, Agent, Lockbox#, Cond,
       Price, #bdrm, Backyard, Style
    from HouseListing
    skyline of #bdrm max, Backyard max,                    1
        Cond max by 1,                                     2
        case style                                         3
            when 'modern' then Yr_built max
            when 'Victorian' then Yr_built min
        end,
        case delta of Cond                                 4
            when 0 then Price min by 5000
            when 1 then Price min by 10000
            else Price min by 30000
        end,
        if Cond ≤ 2 then HasGarage max,                    5
        if neighbourhood = 'Downtown'
            then dist_to_subway min by 1 // km
```

Figure 7: Future "skyline" query.

Consider the hypothetical "skyline" query in Fig. 7, with a "natural" interpretation of what the preference constructs may intend. Type 1 is provided by original skyline. Type 3 is the case-conditional we define. The preference types 2, 4, and 5 were not accommodated here.

Type 4 demonstrates a different type of conditional we may wish to support: additional preferences are evaluated depending on the difference (the delta) between the functional values (in the condition) of the two tuples. In this case, the greater the difference between the houses's condition, we expect a larger price difference between them in order not to rule out the house in worse condition. For example, if the first house has condition 5 and the second, 3, the first house will not trump the second only when the second is less expensive by more than $30,000. Type 5 represents a general if-conditional, allowing any general condition to posit additional preference criteria. For example, for houses of condition 2 or worse, one that has a garage cannot be trumped by one that does not.

We plan to study how to define such queries under new semantics such as the stable skyline, and how to evaluate efficiently these queries. We want also to map semantic equivalences, for the types of queries above. These equivalences may enlighten us on how preferences can be composed, and how they can be used to optimize generalized skyline queries.

References

[1] Börzsönyi, S.; Kossmann, D.; and Stocker, K. 2001. The skyline operator. In *Proc. of ICDE*, 421–430.

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

[3] Chomicki, J.; Godfrey, P.; Gryz, J.; and Liang, D. 2002. Skyline with presorting. Technical Report 2002-04, CS, York University, Toronto, ON, Canada. Long version of (Chomicki et al. 2003).

[4] Chomicki, J.; Godfrey, P.; Gryz, J.; and Liang, D. 2003. Skyline with presorting. In *Proc. of ICDE*, 717–719.

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

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

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

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

[9] Godfrey, P., and Ning, W. 2004. Relational preference queries via stable skyline. CS-2004-03, technical report, Computer Science, York University.

[10] Godfrey, P.; Shipley, R.; and Gryz, J. 2007. Algorithms and analyses for maximal vector computation. *VLDB J.* 16(1):5–28.

[11] Godfrey, P. 2004. Skyline cardinality for relational processing. In *Proc. of FoIKS*, 78–97. Springer.

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

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

[14] Kießling, W., and Köstler, G. 2002. Preference SQL: Design, implementation, experiences. In *Proc. of VLDB*, 990–1001.

[15] Kießling, W. 2002. Foundations of preferences in database systems. In *Proc. of VLDB*, 311–322.

[16] Kießling, W. 2004. Preference constructors for deeply personalized database queries. Technical Report 2004-7, University of Augsburg, Augsburg, Germany.

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

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

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

[20] Ning, W. 2005. SPQL: The design of a relational preference query language. Masters Thesis, York University (P. Godfrey, supervisor).