International Journal of Computer Science and Applications,
2008, Vol. 5, No. 3b, pp 165 - 185

© Technomathematics Research Foundation

CONSISTENCY CHECK ALGORITHMS FOR MULTI-DIMENSIONAL PREFERENCE TRADE-OFFS

CHRISTOPH LOFI

Institute for Information Systems
University of Braunschweig
Mühlenpfordtstraße 23
38106 Braunschweig - Germany
lofi@ifis.cs.tu-bs.de
http://www.ifis.cs.tu-bs.de

WOLF-TILO BALKE

Institute for Information Systems
University of Braunschweig
Mühlenpfordtstraße 23
38106 Braunschweig - Germany
balke@ifis.cs.tu-bs.de
http://www.ifis.cs.tu-bs.de

ULRICH GÜNTZER

Institute of Computer Science
University of Tübingen
Sand 13, 72076 Tübingen Germany
ulrich.güntzer@ informatik.uni-tübingen.de

Abstract: Skyline Queries have recently received a lot of attention due to their intuitive query capabilities. Following the concept of Pareto optimality all 'best' database objects are returned to the user. However, this often results in unmanageable large result set sizes hampering the success of this innovative paradigm. As an effective remedy for this problem, trade-offs provide a natural concept for dealing with incomparable choices. But such trade-offs are not reflected by the Pareto paradigm. Thus, incorporating them into the users' preference orders and adjusting skyline results accordingly needs special algorithms beyond traditional skylining.

For the actual integration of trade-offs into skylines, the problem of ensuring the consistency of arbitrary trade-off sets poses a demanding challenge. Consistency is a crucial aspect when dealing with multi-dimensional trade-offs spanning over several attributes. If the consistency should be violated, cyclic preferences may occur in the result set. But such cyclic preferences cannot be handled by information systems in a sensible way. Often, this problem is circumvented by restricting the trade-offs' expressiveness, e.g. by altogether ignoring some classes of possibly inconsistent trade-offs. In this paper, we will present a new algorithm capable of efficiently verifying the consistency of any arbitrary set of trade-offs. After motivating its basic concepts and introducing the algorithm itself, we will also show that it exhibits superior average-case performance. The benefits of our approach promise to pave the way towards personalized and cooperative information systems.

Keywords: Database Query Processing, Personalization, Skyline Queries, Preferences, Trade-Offs, Pareto Optimality, Multi-Objective Optimization.

1. Introduction

Preference-based queries are important to personalize the massive amount of information residing in today's databases and information systems. In contrast to simple SQL-style queries users are enabled to state their wishes and a suitable degree of relaxation and filtering can be applied when queries are evaluated against a specific database instance. Since users cannot be expected to compare database items in a pairwise fashion, preferences are usually specified separately with respect to each query attribute. To answer a complex query, the attribute preferences are then combined following the Pareto semantics, i.e. all items showing equal or worse values with respect to all attributes than some other items are removed from the result set. This class of queries is referred to as *skyline queries* [Börzsonyi, (2001)].
However, the good personalization behavior comes at the cost of rather big result set sizes, because especially for larger numbers of query attributes, many items become incomparable. This is even more problematic for real world user preferences which often have to face anti-correlation between at least some attributes in the underlying database instance. For instance, in e-commerce applications users tend to prefer less expensive offers, but also like a high degree of quality. Thus, in practical applications suitable compromises have to be found. But such compromises cannot be expressed in the framework of Pareto optimality: incomparable results (e.g. better in quality, but worse in price vs. less expensive and worse in quality) are always part of the skyline.
In [Balke *et al*., (2007a)] the concept of trade-offs was first introduced to extend the Pareto semantics and allow for compromises based on user interaction. A user simply specifies a domination or equivalence relationship between two combinations of values with respect to several attributes. The Pareto product order then can be extended by this new user-specific information. This concept is valuable for extending the expressiveness of skyline queries. But stating such trade-off information 'amalgamates' the attributes concerned. This complicates the query evaluation, since the separability characteristic of the Pareto semantics (i.e. the ability to decompose a relation on database objects back to relations on attribute values) is lost [Bradley et al., (2005)].
Moreover, to be consistent Pareto product orders just need the underlying base preferences to be acyclic, which is easy to ensure. In contrast, trade-offs might introduce inconsistencies 'through the back door'. That means, if several trade-offs span over a set of non-disjoint attribute sets, the induced preference order might contain cyclic preference. Unfortunately, these cycles are hard to detect. A first criterion about how preference cycles can be detected even in complex product orders has been given in [Balke *et al*., (2007a)]. But when a larger number of attributes is concerned, materializing the complete product order is a costly (and mostly impractical) operation. In this paper we propose a way to check a set of trade-offs for consistency without having to materialize the product order.
Extending our original contribution [Lofi *et al.*, (2008)], to be self-contained, we start by presenting a simple algorithm that uses a tree-shaped data structure which incrementally integrates one trade-off after the other. This algorithm uses templates of possible trade-off instantiations to derive all possible inconsistencies in an iterative fashion. For each trade-off, it checks the consistency with all trade-offs that have been integrated before. If the new trade-off may lead to inconsistencies, it will immediately be rejected. Otherwise the entire set of trade-offs can be safely evaluated against the database instance.
As new contributions in this work, our initial *pattern-algorithm* will serve as a baseline to further extensions and optimizations. We will introduce two new versions of the algo-

rithm, namely the *PrePost-algorithm* and the *pruning-algorithm*, and will highlight their superior performance compared to the original approach.
Our extensive experiments investigate typical features like the runtime, memory usage, and scalability and show the practical applicability of the approach. In brief, we present novel algorithms that efficiently allow to guarantee the consistency of a set of user trade-offs without having to access any database items and thus prevents costly failures at query evaluation time.
The paper is organized as follows: First, we provide a brief overview on related approaches for skyline evaluation and result refinement in chapter 2. Then, we introduce the observations and definitions necessary to develop an algorithm for checking trade-off consistency in chapter 3. The resulting algorithms are the topic of chapter 4. In chapter 5, we evaluate the algorithm and will outline its superior performance. We close with a short summary and outlook.

2. Related Work

The skyline paradigm gained a lot of momentum in recent years. The easy querying and the generally intuitive result sets comprising all optimal items are appealing characteristics. Since deriving the actual Pareto-optimal sets is computationally expensive, much research work has focused on the efficient evaluation of skyline queries using different styles of algorithms and further optimizations like indexes, see e.g.[Börzsonyi *et al.,* (2001)], [Kossmann *et al.*, (2002)]. However, not only the consumed computation time for query evaluation poses restrictions for the practical applicability of the skyline paradigm. For larger numbers of query attributes, also result set sizes often become unmanageable, especially when facing a high degree of anti-correlation between different attributes in the data set. In fact, already for only about 5 to 10 attributes, skylines tend to become unmanageably large and can easily contain 30% or more of the entire database instance (see experiments in e.g., [Balke *et al.*, (2005)], [Börzsonyi *et al.,* (2001)], [Godfrey, (2004)], [Godfrey *et al.*, (2005)]).
In order to keep result sets manageable, two major approaches have been devised: On one hand, there are techniques using user independent structural properties and heuristics to reduce the skyline set without any further interaction (often called user-oblivious approaches). Here, techniques range from extending the Pareto semantics [Balke *et al.*, (2007d)], over providing representative samples of the skyline [Balke *et al.*, (2005)], or giving users an overview by (approximately dominating) representatives [Koltun and Papadimitriou, (2005)], [Lin *et al.*, (2007)], to a structural result set ranking based on subspace-skyline occurrences of database items [Pei *et al.*, (2006)].
On the other hand, a more user-centered reduction and focusing the skyline set can be archived by incorporating additional information directly provided by the user. Most prominent among these approaches are techniques that allow users to interactively modify and extend their preferences in a cooperative fashion, see e.g. [Chomicki, (2006)], [Lee *et al.*, 2007]. In particular, a user might consider some attributes in the query to be more important than others. This fact is used in [Lee *et al.*, (2007)] where the user is enabled to provide a total order on attributes. Then, the evaluation first focuses on those subspace-skylines containing only the more interesting attributes.
Trade-Offs [Balke *et al.*, (2007c)] represent another approach for integrating user-provided information. Trade-offs, as part of our everyday's decision making, are indeed a very natural concept. They are able to introduce information about compromises a user is willing to accept. Such information cannot be expressed by traditional Pareto skylines. A

summary of basic concepts for integrating trade-offs and interfaces for the necessary user interactions can be found in [Balke *et al.*, (2007c)] and [Balke *et al.*, (2007b)].
The importance of avoiding cyclic preferences for deterministic query processing is easy to see. First criteria and consistency checks were already established by [Balke *et al.*, (2007a)] which, however, still relied on the materialization of the entire product order and were therefore computationally prohibitive. An obviously sufficient criterion to ensure the consistency of trade-offs is to restrict trade-offs to disjoint attribute sets only. In [Balke *et al.*, (2005)], a slightly less restrictive and easy to ensure criterion was introduced. However, this so-called characteristic of being 'Cartesian' was only proven to be sufficient, but not necessary. Hence, none of the former approaches were able to provide a complete and manageable algorithm for checking a set of arbitrary trade-offs for consistency.

3. Concepts for Integrating Preference Trade-Offs

In this section we introduce the basic concepts needed to specify base preferences, aggregate them to a product order, and enhance the resulting order consistently with user-specific trade-off information. We will always assume that a query containing a set of attributes is given and for each attribute, a base preference is defined.

Definition 1: (base preferences and Pareto aggregation)
A base preference P on an attribute A with domain D is a strict partial order on values in D. A base equivalence Q is an equivalence relation on D compatible with P (i.e. $P \circ Q \subseteq P$ and $Q \circ P \subseteq P$). Then several base preferences $P_1, \ldots, P_n$ together with base equivalences $Q_1, \ldots, Q_n$ can be aggregated to a product order on $(D_1 \times \ldots \times D_n)^2$ under the Pareto semantics such that $\forall\, x, y \in (D_1 \times \ldots \times D_n)$:
$(y \rhd x) \Leftrightarrow \forall\, i{:}\, y_i \gtrsim_i x_i \wedge \exists\, i{:}\, y_i >_i x_i \,, (1 \leq i \leq n)$
and also
$(y \approx x) \Leftrightarrow \forall\, i{:}\, y_i \approx_i x_i \,, (1 \leq i \leq n)$

$y_i \approx_i x_i$ stands for $(y_i,\, x_i) \in Q_i$, and $y_i >_i x_i$ stands for $(y_i,\, x_i) \in P_i$ while $y_i \gtrsim_i x_i$ represents $(y_i,\, x_i) \in P_i \cup Q_i$. The resulting product order can be enhanced by amalgamating subsets of attributes and specifying trade-offs between specific (previously incomparable) values.

Definition 2: (trade-offs)
A trade-off $t := (y_\mu \rhd x_\mu)$ on a (sub-)set $\mu \subseteq \{1, \ldots, n\}$ of attribute indices with $x_\mu, y_\mu \in \times_{i \in \mu} D_i$ defines a preference relationship between two sets of (previously incomparable) domain values. It can be defined on all attributes or just on a subset of attributes. In the latter case it automatically enhances the preference order by all possible object extensions following the ceteris paribus semantics. That means preference relationships are added between all objects featuring domain values y_μ, resp. x_μ regarding attributes with index in μ and equivalent values regarding attributes with index not in μ.
The value y_μ is referred to as the trade-offs *precondition*, whereas the value x_μ is called the *postcondition*.

In the following we will always refer to the aggregation of all base preferences following the Pareto semantics as the '*product order*', whereas we will refer to a product order into which one or several trade-offs have been integrated as '*enhanced product order*'. For ease of use, in the following we will notate trade-offs in the form of generalized object pairs in square brackets with the values for attributes not in μ replaced by wildcards '⋆'. Also, we will focus only on strict preference relations and trade-offs for sake of brevity. Dealing with equivalences or non-strict preferences is analogous to the observations in this work.

Example: Let us assume we have three attributes over binary domains {0, 1} in a query and the base preferences are given by the natural order 1 > 0 and the base equivalences are simply the natural equality relations. A user might specify a trade-off spanning over the first two attributes to make a specific combination comparable, e.g. [1, 0, ⋆] ⊳ [0, 1, ⋆]. Following the ceteris paribus semantics this trade-off would introduce two new preference relationships into the enhanced product order, namely (1, 0, 0) ⊳ (0, 1, 0) and (1, 0, 1) ⊳ (0, 1, 1).

Such trade-offs extend the expressiveness of the preference order beyond the Pareto semantics. However, their consistent integration is not easy. In contrast to the simple product order, preference cycles in the enhanced product order can occur as more trade-offs are added, even if each individual trade-off is consistent.

Example: Consider a set of three trade-offs:
t_1: [1, 0, ⋆] ⊳ [0, 1, ⋆]
t_2: [⋆, 1, 0] ⊳ [⋆, 0, 1]
t_3: [0, ⋆, 1] ⊳ [1, ⋆, 0]
Now, let's consider some database object (1, 0, 0). Applying trade-off t_1 we can deduce that (1, 0, 0) ⊳ (0, 1, 0). Now trade-off t_2 applies, resulting in (0, 1, 0) ⊳ (0, 0, 1) and using trade-off t_3 finally results in (0, 0, 1) ⊳ (1, 0, 0). We thus get the complete trade-off sequence (1, 0, 0) ⊳ (0, 1, 0) ⊳ (0, 0, 1) ⊳ (1, 0, 0) which is obviously a cyclic preference, i.e. trade-offs t_1, t_2, t_3 and the base preferences are not consistent.

Thus, trade-offs on overlapping attribute sets can easily become inconsistent. But the inconsistency of several trade-offs is often very hard to see for the user. This is especially true, if the trade-offs have been elicited implicitly, e.g. by comparing sample database items like in example critiquing frameworks [Viappiani *et al.*, (2006)]. To protect the information system from running into cyclic preferences, the user input therefore has to be checked for inconsistent information. In case that an inconsistent trade-off is detected, the system has to reject at least the last specified trade-off.

We will now investigate the nature of the problem and show a way to detect inconsistencies between trade-offs. Before the first trade-off is added, the Pareto product order is obviously acyclic, if and only if all base preferences are acyclic. This is a direct result of the separability characteristics of the Pareto semantics. However, when adding trade-offs, the separability is always lost and inconsistencies may occur by concatenations of base preferences with trade-offs.

In any case the question of consistency of a set of trade-offs with a set of base preferences should always be independent of the particular database instance. Hence, we have

to make sure that no combination of preferences and trade-offs for an arbitrary set of objects (i.e. arbitrary attribute value combinations) can cause a cyclic preference. The following observation states that such preference cycles in the enhanced product order are caused by sequences of trade-offs, where the post condition of the previous trade-off always matches or dominates the pre-condition of the following trade-off.

Observation 1: A cyclic preference in the enhanced product order causes that some value tuple $(a_1, \dots, a_n)$ is preferred over itself. This of course violates the irreflexivity of strict preference orders, in this case the enhanced product order. The cycle may stretch over several value combinations using some (finite) sequence of (base) preferences interleaved with trade-offs. This sequence can be applied on $(a_1, \dots, a_n)$ and also ends with $(a_1, \dots, a_n)$. Moreover, for any adjacent value combinations during the sequence $(b_1, \dots, b_n)$ and $(c_1, \dots, c_n)$ holds either $\forall 1 \leq i \leq n: b_i \geq c_i$, or there exists some trade-off $t_i := (y_\mu \rhd x_\mu)$ on a subset of attributes μ such that restricted to μ holds: $(b_1, \dots, b_n)|_\mu \geq y_\mu$ and $x_\mu \geq (c_1, \dots, c_n)|_\mu$ and $\forall i \notin \mu: b_i \geq c_i$.

We now know how trade-offs are integrated into the enhanced product order and how they can connect to each other. The sequences of connections are also the key to verify the consistency of a set of trade-offs and base preferences.

Observation 2: Since base preferences are required to be acyclic, using only preferences from the product order can never result in cyclic preferences in the enhanced product order. Thus, the consistency check has only to focus on the trade-offs. In case that a preference cycle exists, any sequence of trade-offs $(t_{i_1}, \dots, t_{i_l})$ instantiating the cycle can be repeated in the same way and therefore forms a typical pattern of two adjacent duplicate applications of trade-offs $(t_{i_1}, \dots, t_{i_l}, t_{i_1}, \dots, t_{i_l})$. Hence, as cyclic preferences always create a typical pattern (of any length), we can deduce that if and only if such adjacent duplicate patterns exist with respect to a set of trade-offs, a cyclic preference exits in the enhanced product order.

This observation directly leads to the following lemma that will serve as a basis for the fault detection of our later algorithm:

Lemma 1: If a trade-off sequence $(t_{i_1}, \dots, t_{i_l}, t_{i_1}, \dots, t_{i_l})$ can be applied to any generic tuple of domain values, the enhanced product order contains a cyclic preference (and is thus inconsistent).

Proof: Let $\mu := \mu_{i_1} \cup \dots \cup \mu_{i_l}$ be a set of indices and υ be some index $\upsilon \in \mu$. Let c_υ be the value of the υ-th attribute of an value tuple that is a result of applying the trade-off sequence $(t_{i_1}, \dots, t_{i_l})$.
Then every c_υ can be characterized as follows:
Let $j_1, \dots, j_r$ be the set of sub-indices with $1 \leq j_1 < \dots < j_r \leq l$ such that $\upsilon \in \mu_{i_{j1}} \cap \mu_{i_{j2}} \cap \dots \cap \mu_{i_{jr}}$. Then $t_{i_{jr}}$ is the last trade-off modifying the υ-th attribute. This enforces that c_υ has the value of the υ-th attribute of the post-condition of $t_{i_{jr}}$. This is still true when $(t_{i_1}, \dots, t_{i_l})$ is immediately applied for a second time.

Moreover, since this holds for every υ, we can conclude that after applying $(t_{i_1}, \dots, t_{i_l})$ twice directly adjacent, the resulting value tuple after l and $2l$ trade-offs actually have the same values c_υ for all attributes with $\upsilon \in \mu$. All other attributes were not affected by $(t_{i_1}, \dots, t_{i_l})$ at all (and have thus still their initial value), i.e. the value tuples after the application of l and $2l$ trade-offs are identical and therefore there exists a cyclic preference induced by the trade-offs. ■

Checking all possible trade-off sequences seems like a lot of effort. But even when applying arbitrary trade-off sequences on a generic object, the values in the postconditions of the trade-offs have to be adhered to, as stated in observation 1. For many value tuples therefore it is simply not possible to append another trade-off and the respective sequence ends. Therefore, the possible length of trade-off sequences in the case of a consistent set of trade-offs is always limited, which is important for the termination of our later algorithm.

Observation 3: When enhancing a product order by trade-offs, the set of trade-offs provided by the user is always finite. Moreover, the intermediate value tuples in each trade-off sequence do only consist of wildcards or specific values provided by the postconditions of those trade-offs already applied.
Since there is only a finite number of trade-offs, also the possible combinations of values in these intermediate tuples occurring during a sequence is limited. Thus, if there is no cyclic preference, there can only be sequences of limited length without creating two identical intermediate value tuples. If on the other hand, two identical intermediate value tuples would have been created, it is obviously possible to apply the same sub-sequence that created the second tuple again and we would be able to detect a directly adjacent duplicate pattern in the trade-off sequence.

4. Algorithms

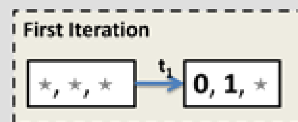
With the above observations and the fault detection condition we are now ready to design a simple algorithm to find all possible inconsistencies in a given set of base preferences enhanced by trade-offs. This algorithm will serve as baseline for further optimizations.

4.1. *The Pattern-Algorithm: Basic Algorithm for Consistency Checks*

We start by constructing a tree structure containing all trade-off sequences that can possibly be built with the current set of trade-offs. The nodes of this tree always represent more or less generic objects with restrictions on certain attributes (imposed by the postconditions of the trade-offs applied).
The tree is initialized as a root node which poses no restrictions on any attribute values. When considering an additional trade-off, all nodes of the existing tree are expanded, whenever the trade-off can be applied to them. Such an expansion will append additional nodes representing the consequences of the applied trade-off. The edges from the parent nodes to the new nodes are then labeled with the responsible trade-offs. Finally, we try to apply all already stated trade-offs to the new nodes.
Every time a node is added, we check the sequence of trade-offs from the root to the current node. If that sequence contains an adjacent duplicate pattern according to observation 2 in the previous section, the set of current trade-offs is not consistent and the algorithm terminates with a failure.

Let us reconsider the previous example and the successive integration of the three trade-offs t_1, t_2, and t_3.

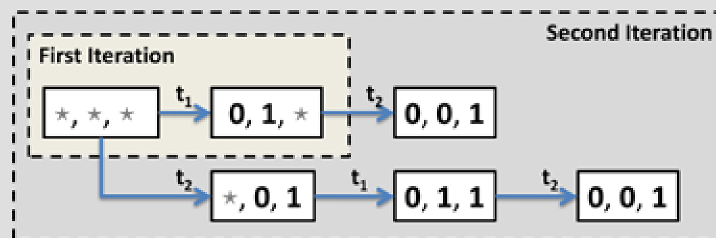
Example: We aim at detecting potential trade-off inconsistencies while abstracting from specific database instances. Therefore, we start with a generic root object (⋆, ⋆, ⋆). Each trade-off applies to this generic object as ⋆ can be replaced with any value.
First, we integrate the trade-off t_1: [1, 0, ⋆] ⊳ [0, 1, ⋆] – this is possible since the root node (⋆, ⋆, ⋆) dominates the trade-off's precondition (1, 0, ⋆). The trade-off will end at some object of the form (0, 1, ⋆) which is added as a new node to the tree.
After applying only t_1, we obtain the following tree:



Now, we try to apply t_1 to the new node. Since (0, 1, ⋆) does not dominate the trade-off's precondition [1, 0, ⋆], it cannot be applied again. In case we would have been able to apply t_1 again, we would immediately have been able to detect the adjacent duplicate pattern (t_1, t_1) in the trade-off sequence and thus an inconsistency.

As the next step, the second trade-off t_2 has to be added. The tree therefore has to be expanded accordingly for both nodes.

Example: Trade-off t_2: $[\star, 1, 0] \rhd [\star, 0, 1]$ can be applied for both nodes of the current tree, since (⋆,⋆,⋆) fulfills t_2's precondition trivially and also the new node of the previous step $(0, 1, \star)$ dominates the precondition $[\star, 1, 0]$. This in turn results in two new nodes: $(0, 0, 1)$ and $(\star, 0, 1)$. For each of these nodes, we again have to apply all previous trade-offs, i.e. t_1 and t_2, recursively.
This creates following tree:



Example: The third trade-off t_3: $[0, \star, 1] \rhd [1, \star, 0]$ is applied in a similar fashion. Again, t_3 extends all nodes that match its precondition $[0, \star, 1]$. Figure 1 illustrates the effects of the application of t_3: when extending node (0,0,1) recursivly, we soon encounter a sequence of trade-offs showing an adjacent duplicate pattern $(t_1, t_2, t_3, t_1, t_2, t_3)$ and thus proving that the set of trade-offs t_1, t_2, t_3 is indeed inconsistent and the algorithm consequentially terminates with a failure.

Please note that in Figure 1 all nodes are omitted which would have been induced by t_3 after the inconsistency was detected (this especially includes those nodes of t_3 extending from the root).

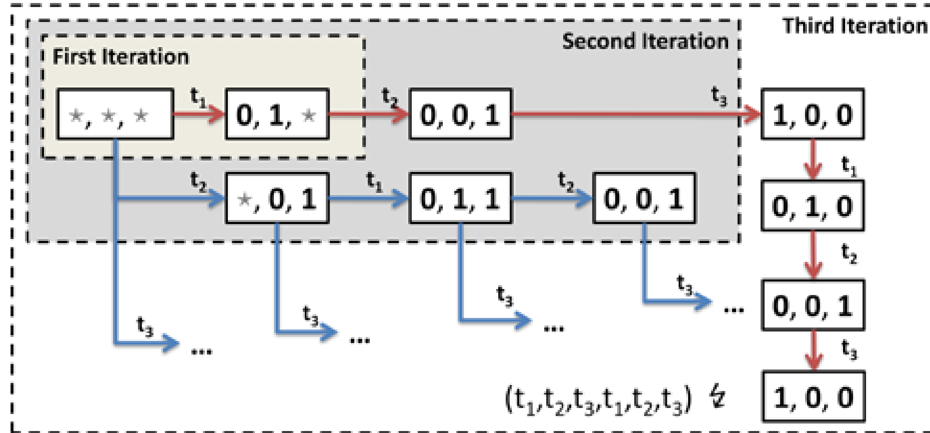


Figure 1: Search tree for three trade-offs until termination due to inconsistency

In the following, we finally present the complete algorithm. It uses following data structures:

- *Template*:
  - Contains an array of values and wildcards (denoted as '⋆'), e.g., [⋆,12,18,⋆]
- *Trade-Off*:
  - Contains two templates *pre* and *post* of equal length and featuring wildcards at the same positions
  - *pre* and *post* do not dominate each other, e.g., *pre:* [⋆,12,18,⋆]; *post:* [⋆,7,22,⋆]
- *Node*:
  - Each node has a set of *labeled* child nodes. The label denotes which trade-off was used to create the child.
  - Each node has a template as content.

For computing the values of new nodes, we will need an additional function merging two templates into a new one. This will be needed, for example, when a trade-off is applied to a node, combing the node's content template with the *post*-template of the trade-off. This function is called *templateMerge* and is denoted by $(template_1 \Lsh template_2)$, meaning that $template_2$ is merged into $template_1$. During the merge operation, all wildcards in $template_1$ are replaced by the respective values in $template_2$.
Example: $[0,\star,\star,1] \Lsh [1,1,\star,\star] = [0,1,\star,1]$

Algorithm: Check Trade-Off Consistency by Detecting Conflicting Patterns

1. Initialize Algorithm

1.1. Initialize *root*, a node with a template consisting only of wildcards and an empty list of children

1.2. Initialize *allTradeOffs*, an empty list of trade-offs
2. *Check Trade-Off Consistency*
2.1. Request a new trade-off *t* from the user
2.2. Add *t* to the list *allTradeOffs*
2.3. Call *expandNode(root, {t});*
2.4. **If** the user wants to specify another trade-off
2.4.1. **Goto** 2

Procedure expandNode(Node *node*, List of trade-offs *newTradeOffs*)
1. **For each** *child* **in** *node.children*
1.1. Call *expandNode(child, newTradeOffs)*
2. **For each** *t* **in** *newTradeOffs* **with** $node.template \trianglerighteq t.pre$ *(i.e. t is applicable on the current node)*
2.1. **If** the trade-off labels of the path from the root-node to the current node followed by *t* contains an adjacent duplicate sequence
2.1.1. **Exit** Algorithm with failure: Set of trade-offs inconsistent
2.2. Create template *newTemplate* := $t.post \looparrowleft node.template$
2.3. Create a new node *newNode* having *newTemplate* as template
2.4. Call *expandNode(newNode, allTradeOffs)*
2.5. Add *newNode* to *node.children* and label the respective edge with *t*

4.2. *The PrePost-Algorithm: Optimizing Cycle Detection*

The *pattern-algorithm* introduced in the previous chapter ensured consistency of multiple trade-offs by checking for trade-off chains of unlimited length as described in observation 3. The basic idea is that whenever there is such an unlimited consecutive application of trade-offs possible, the original set of trade-offs is inconsistent. On the other hand, if it is not possible the original trade-off set is consistent. The algorithm detected such unlimited chains by using the observation that any unlimited chain of trade-offs contains at least one pair of similar, directly adjacent subpatterns. However, detecting these chains directly as performed by the *pattern-algorithm* is not the most efficient solution. Weak points that can be identified are

- Performing the actual check for adjacent identical patterns in the original algorithm is computationally expensive as, for each check, the whole path of the actual node within the tree has to be traced back to the root.
- Also, the algorithm materializes the two duplicate subpatterns to their full extent. However, it is possible to detect them just after the first subpattern is materialized as shown below.

Following observation will lead to a more efficient solution:

Observation 4: Each chain of trade-offs with duplicate and directly adjacent subpatterns is at some point of their construction of the following form:

For example, consider the chain $(t_1, t_2, t_3, t_2, t_3)$. Then $subpattern_1$, the prefix pattern, is (t_1) while $subpattern_2$, which is repeated, is (t_2, t_3).
Furthermore, whenever a chain of the above form can be constructed, also another chain can be constructed where $subpattern_1$ only contains the tree root which consists of only

wildcards:

root *	subpattern$_2$	subpattern$_2$

This can be explained by the fact that when $subpattern_2$ may follow after $subpattern_1$, it may also follow after the less restrictive root node. Checking for directly adjacent subpatterns without prefix can be done in an efficient manner in such a way that only one of the two subpatterns need to be materialized in the tree and without backtracking each node to the root.
This can be archived by checking if the pattern's last template dominates the least-restrictive *combined* pre-condition of all trade-offs involved in the pattern. If this is the case, then $subpattern_2$ can be reapplied directly after itself and thus leads to an infinite trade-off chain indicating an inconsistency.

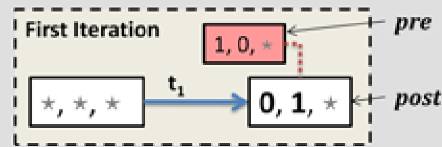
In the above observation, the notion of least-restrictive combined pre-condition is used. To clarify this concept, consider a set of n trade-offs $(t_1 : pre_1 \rhd \mathrm{post}_1)$, …, $(t_n : pre_n \rhd post_n)$ which can be combined to a chain $(t_1, t_2, \dots, t_n)$, i.e. $post_1 \trianglerighteq pre_2 \wedge \dots \wedge post_{n-1} \trianglerighteq pre_n$. This chain of trade-offs could be seen as a single, more complex trade-off $(t_c : pre_c \rhd post_c)$, i.e. applying t_c would yield the exact same effects as applying $t_1, \dots, t_n$ consecutively. The least restrictive combined pre-condition of the trade-offs $c_1, \dots, c_n$ is equivalent to pre_c of t_c and can be expressed by $pre_c \coloneqq pre_1 \leftarrowtail pre_2 \leftarrowtail \cdots \leftarrowtail pre_n$. The same consideration also holds for the combined post-condition in reverse order, i.e. $post_c \coloneqq post_n \leftarrowtail post_{n-1} \leftarrowtail \cdots \leftarrowtail post_1$. Please note that the combined post-condition is represented by the templates currently used in the nodes of the pattern-algorithm tree.
To adapt observation 4 into the pattern-algorithm, each node in the tree is additionally annotated with a template representing the least-restrictive combined pre-condition. Thus, each node consists of two templates which are abbreviated with $post$ for the original content (combined post-condition) and pre (combined pre-condition), hence the name PrePost-algorithm for the new implementation.
Modifying the pattern-algorithm requires an additional change as originally, a depth-first approach for expanding and checking the tree was used. However, if only the condition comparing combined pre- and post-conditions as described in observation 4 is used, the algorithm will not be able to detect infinite trade-off chains having a non-empty prefix-pattern and thus does not necessarily terminate. Utilizing a breadth-first approach instead will avoid any termination problems as for each unlimited chain with a prefix, a shorter chain without the prefix will be constructed which will be detected as inconsistent and thus terminating the algorithm.

Example: Re-consider the example already utilized for demonstrating the pattern-algorithm. Again, we aim at detecting potential trade-off inconsistencies between the three trade-offs t_1, t_2, and t_3, which are provided incrementally:

$t_1 : [1, 0, \star] \rhd [0, 1, \star]$
$t_2 : [\star, 1, 0] \rhd [\star, 0, 1]$
$t_3 : [0, \star, 1] \rhd [1, \star, 0]$

Again, we start with the generic root $[\star, \star, \star]$ and trade-off t_1. t_1 can be applied to the root as it dominates it's pre-condition and a new node is created with the combined pre-condition pre (illustrated as smaller and darker rectangle) and the combined post-

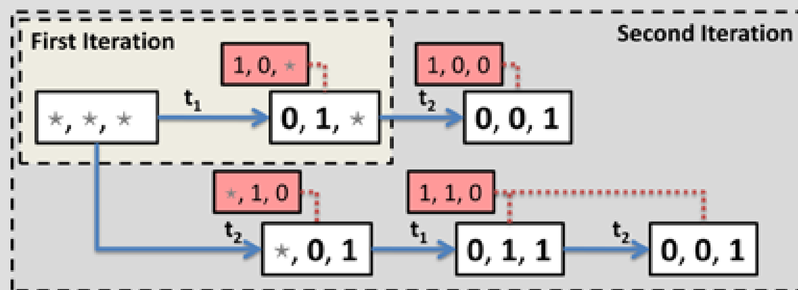
condition $post$ (illustrated by a larger white rectangle).



Every time a new node n is created, its pre and $post$ templates are checked for dominance. As soon as $n.post \trianglerighteq n.pre$ holds for any node n, the algorithm terminates as there is an inconsistency within the set of trade-offs.

In the next step, the second trade-off t_2 is added. The tree therefore has to be expanded accordingly for both nodes in breadth-first manner.

Example: Again, trade-off t_2: $[\star, 1, 0] \rhd [\star, 0, 1]$ can be applied for both nodes of the current tree. Please note that the new pre-templates are computed from the pre-template of the new node's predecessor by merging (i.e. replacing wildcards with actual values). Also note that multiple nodes may share a pre-template if it did not change during application of a new trade-off.

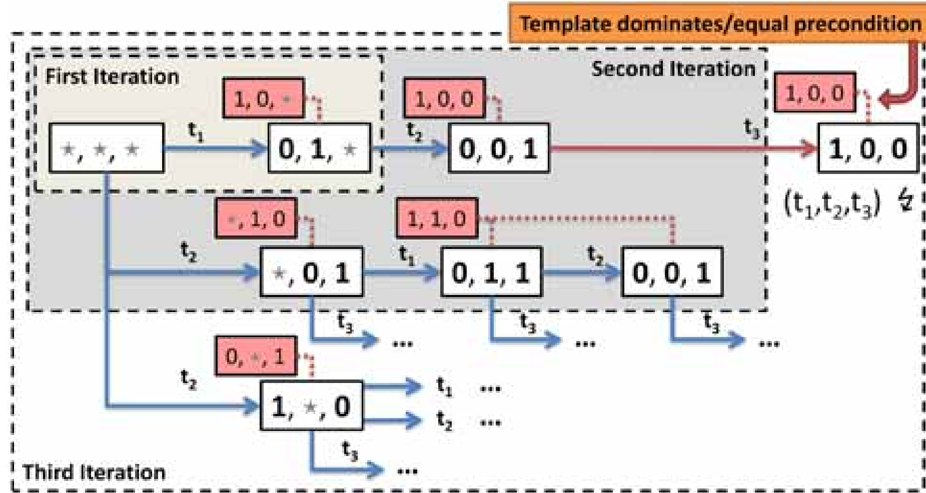


Again, for no node n holds: $n.post \trianglerighteq n.pre$.

Finally, t_3 is added to the tree and an inconsistency is detected.

Example: The third trade-off t_3: $[0, \star, 1] \rhd [1, \star, 0]$ is applied in a similar fashion. The tree is expanded in a breadth-first manner and for each new node n, the condition $n.post \trianglerighteq n.pre$ is checked. Parts of the resulting tree are illustrated in Figure 2. As soon as the chain $\{t_1, t_2\}$ is expanded with t_3, an inconsistency is detected as $n.post \trianglerighteq n.pre$ holds for the newly appended node. This means that directly after $\{t_1, t_2, t_3\}$ an adjacent duplicate could be attached indicating an inconsistency.

Figure 2: Search tree for three trade-offs until termination due to inconsistency



For implementing this new condition, the procedure *expandNode()* and the definition of node needs to be changed:

- *Node*:
  - Has a set of child nodes
  - Has a template *post* representing the combined post-condition
  - Has a template *pre* representing the combined pre-condition
  - Has a flag *new* indicating if the node is new or not. Defaults to *true*.

Procedure expandNode(Node *node*, List of trade-offs *newTradeOffs*)

1. *expandQueue* := {*root*}
2. **While** *expandQueue* not empty
   - 2.1. *currentNode* := First element of *expandQueue*; remove element from queue
   - 2.2. **For each** *child* **in** *currentNode.children*
     - *2.2.1. expandQueue.add(child)*
   - 2.3. **If** *currentNode.new* = true
     - *2.3.1.* **For each** *t* **in** *allTradeOffs*
       - *2.3.1.1.* applyTradeOff(*currentNode, t*)
   - 2.4. **For each** *t* **in** *newTradeOffs*
     - *2.4.1.* applyTradeOff(*currentNode, t*)

Procedure applyTradeOff (Node *node*, Trade-offs *tradeOffs*)

1. **If** *node.template* ⊵ *tradeOff.pre* // *is trade-off is applicable*
   - 1.1. *node.new* := false
   - 1.2. Create a new node *newNode*
   - 1.3. Create template *newNode.post* := *tradeOff.post* ⇤ *node.post*
   - 1.4. Create template *newNode.pre* := *node.pre* ⇤ *tradeOff.pre*
   - 1.5. **If** *newNode.post* ⊵ *newNode.pre* // *check if inconsistent*

1.5.1. **Exit** Algorithm with failure: Set of trade-offs inconsistent
1.6. Add *newNode* to *expandQueue*
1.7. Add *newNode* to *node.children*

4.3. *The pruning-algorithm: Optimizing the tree structure*

Both previous algorithms still show a major weakness: In both cases, the tree contains many redundant branches which are, for each trade-off, unnecessarily expanded and checked. This deficit can be compensated by *pruning* the tree. Correct pruning will remove all branches of the tree which are contained completely within another branch while also being more restrictive within its preconditions and provide thus no additional information.

Observation 5: Given two nodes *n* and *m* of a tree as used in the PrePost-algorithm, i.e. nodes representing a chain of trade-offs as a single combined trade-off containing the respective combined pre-condition *pre* and combined post-condition *post*. The node *n* can be pruned (i.e. removed) if it does not provide any additional information over node *m*. This is the case if

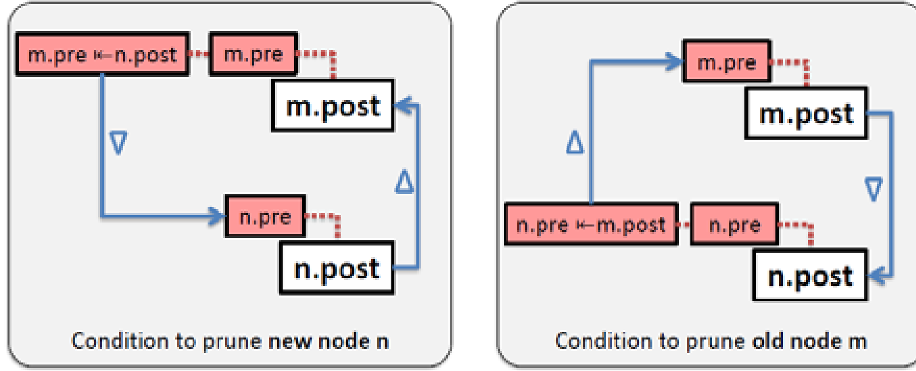
1) The precondition of *n* is more restrictive than the precondition of *m* in the same *context*, i.e. only a proper subset of all database objects which would be affected by *m* would also be affected by *n*. This can be expressed by $(m.pre \Lsh n.post) \rhd n.pre$
2) The effect of *n* is more general than the effect of *m*, i.e. any trade-off which can be applied on *m* can also be applied on *n*. This is expressed by $n.post \rhd m.post$

Thus, a node *n* can be pruned if there is a node *m* with $(m.pre \Lsh n.post) \rhd n.pre \wedge n.post \rhd m.post$.

We can now modify our PrePost-algorithm in the following way: Each time a new node *n* is created, the whole tree is scanned and for each already existing node *m*, the algorithm checks if either *n* or *m* can be pruned (refer to figure 3). Every time a node is pruned, all its children are recursively removed and also the node itself is deleted.
To extend the previous PrePost-algorithm, a new procedure, *pruneNode()* for pruning a node is needed which also has to be integrated into the *applyTradeOff()* procedure.

Procedure applyTradeOff (Node *node*, Trade-offs *tradeOffs*)
1. **If** $node.template \trianglerighteq tradeOff.pre$ // *is trade-off is applicable*
   1.1. Create a new node *newNode*
   1.2. Create template *newNode.post* := $tradeOff.post \Lsh node.post$
   1.3. Create template *newNode.pre* := $node.pre \Lsh tradeOff.pre$
   1.4. *node.new* := false
   1.5. **If** $newNode.post \trianglerighteq newNode.pre$
      1.5.1. **Exit** Algorithm with failure: Set of trade-offs inconsistent
   1.6. **If not** *pruneNode(newNode)* // *check if inconsistent*
      1.6.1. Add *newNode* to *expandQueue*
      1.6.2. Add *newNode* to *node.children*

Figure 3: Pruning conditions



$(m.pre \Lsh n.post) \rhd n.pre \wedge n.post \rhd m.post$ $(n.pre \Lsh m.post) \rhd m.pre \wedge m.post \rhd n.post$

Procedure pruneNode (Node *n*) : {true/false}
1. **For each** node *m* of the current tree in depth-first order
   1.1. **If** $(m.pre \Lsh n.post) \rhd n.pre \wedge n.post \rhd m.post$
      1.1.1. **Return** true // *prune n, expandNode() will not add n to tree*
   1.2. **If** $(n.pre \Lsh m.post) \rhd m.pre \wedge m.post \rhd n.post$
      1.2.1. Delete m and also recursivly all it's children
      1.2.2. **Return** false // *expandNode() will add n to tree*

4.4. *Optimization of Matching Pre-/Postconditions*

After establishing the basic algorithm, we can focus on an additional optimization. It aims at reducing the number of template comparisons required in *expandNode* step 2 ($node.post \trianglerighteq t.pre$). While a single template comparison is a relatively cheap operation, it has to be performed for each node of the tree for every new trade-off. With growing tree sizes, template comparisons easily contribute a significant amount of the overall effort required.

However, in each set of trade-offs usually some sequences are impossible because the pre- and postconditions of trade-offs are incomparable, i.e. they will never match. Such trade-offs obviously cannot be applied adjacently in a sequence. Therefore in many cases the result of pre-/post-condition comparisons can be predicted by simply considering the current trade-off and the trade-off labeling the edge which leads to the current node.

Example: Consider two trade-offs t_1: $[1, 0, \star] \rhd [0, 1, \star]$ and t_4: $[1, \star, 0] \rhd [0, \star, 1]$. t_4 can never be applied directly after t_1 as the postcondition of t_1 does not dominate the precondition of t_4 regardless of the values of the wildcards. In the same way t_4's post-condition will never dominate t_1's precondition. In fact, t_1 and t_4 can never occur adjacently in any sequence of trade-offs.

Hence, a good optimization of the algorithm is to precompute a matrix containing the information whether it possible at all (in the most general case) that a given trade-off can be applied directly after some other. The algorithm is now modified in such a way that it performs an (extremely cheap) lookup in the matrix before starting to compare the two respective templates. If the value in the matrix indicates that the current trade-off can never be applied on the actual node (by considering the trade-off which created the current node), the actual comparison can be skipped. Comparisons only need to be performed, if it is generally possible to apply the trade-off. Moreover, it can be restricted to comparing only those attributes containing wildcards in any of the two trade-offs.

5. Experimental Evaluation

In this section, we evaluate our algorithm using various synthetic performance tests. The usage of synthetic tests was necessary as there is no practical test set of typical user trade-offs readily available. In each evaluation, we check different sets of trade-offs using the three introduced algorithms for consistency and measure the resulting resource and time consumption.

5.1. *Experimental Setup*

We performed our experiments on a notebook computer featuring an Intel Pentium M 2.0 GHz CPU with 1.5 GB RAM and Sun Java 6.

For the various evaluations, randomly generated trade-off sets were used. Trade-off sets could be adjusted in a) their size (default 20 trade-offs per set if not mentioned otherwise), b) the number of total attributes (default 6), c) the number of attributes involved in a trade-off (default randomly chosen between 2 and 4 for each trade-off), and d) the size of the attribute domains (default 20).

Please note that the evaluation settings and algorithms slightly differ from the evaluation performed in our original work in [Lofi *et al.*, (2008)], thus measured values are not identical. However, they show identical tendencies.

5.2. *Sequences of Fixed Length*

In this section, random trade-off sets of the size 20 are examined. The procedure for this evaluation is as follows: 15,000 sets of 20 trade-offs each are randomly generated. For each set, the trade-offs are incrementally checked for consistency as described in chapter 4, one after the other. During these checks, the search tree of all possible trade-off chains (i.e. all possibilities to apply the current trade-offs to generic database objects and objects generated by other trade-offs) is constructed. The tree is then tested for an inconsistency of the trade-off set using the three introduced algorithm variants: the pattern-algorithm, the PrePost-algorithm and the pruning-algorithm.

During that course, four key measurements are taken: The time needed to check a full set of 20 trade-offs for consistency, the size of the search tree constructed to perform that check, and the depth of that tree. Finally, it is measured after how many trade-offs the sequence became inconsistent (if at all). The summarized values of the 15,000 trade-off sets each are compiled in Table 1, Table 2 and Figure 4.

A sequence of 20 different trade-offs already contains a large amount of user provided information. In most real-world applications, we anticipate a considerably smaller number of trade-offs per user-interaction. During our evaluation, 41% of all randomly generated trade-off sequences had been inconsistent; most of them broke after stating more than 14 trade-offs – shorter trade-off chains were usually consistent.

The size of the search tree and the time needed to check for inconsistencies is highly correlated (Pearson coefficient >0.99). Both are influenced by the specific nature of trade-offs within a set. Some trade-offs are highly active and can often be applied after others. Many such active trade-offs in a set will result in the construction of many and long trade-off sequences and thus result in the creation of a larger tree. While it is hard to grasp which properties of trade-offs allow for the construction of a larger number of trade-off sequences, it fortunately seems to be quite a rare event.
When comparing the three algorithm implementations, the pattern-algorithm and the PrePost-algorithm perform similar (with the PrePost-algorithm being slightly better over-all and clearly better when considering median values). However, the pruning-algorithm shows a significantly superior performance. When considering quantile computation times (i.e., 25%, 50%, 75%, and 98% of all trade-off chains), the pruning-algorithm is always at least 4.5-times more efficient than the PrePost-algorithm.
In more detail, the median computation time for the pattern-algorithm is 17ms, for the PrePost-algorithm 3ms and for the pruning algorithm less than a millisecond. The respective 98%-quantiles (i.e. the computation time which is not exceeded by 98% of all trade-off chains) are 7,068 ms, 3,977ms and 862ms. Unfortunately, 2% of all trade-off chains perform rather bad for all algorithms and can reach computation times up to 70-80 seconds. These cases need to be detected during computation and treated individually. Please refer to Table 1, Table 2, and Figure 4 for a more detailed compilation of performance evaluation results.

We can conclude the following: When using the pruning-algorithm, the check for trade-off consistency can be performed fast, efficient and in real-time for nearly all occuring trade-off chains. However, this cannot be guaranteed – under rare, unfortunate circumstances, the computation may take considerably more time.

5.3. *Varying the Number of Trade-Offs*

The tree size (and thus the runtime) increases with every additional trade-off that needs to be checked. From a complexity theoretical point of view, all our algorithms perform exponentially within the number of trade-offs. However, we will show in the following that this usually does not pose a problem for practical application.
In this evaluation, we examine the impact of the size of trade-off set on the number of nodes of the search tree. We reuse the trade-offs generated during the last evaluation. Now, incrementally up to 20 times, a single trade-off is added to the present set of trade-offs to be checked for consistency. After each trade-off added, the tree-size required by the algorithm is measured.
The median and maximum tree sizes of 15,000 generated trade-off sets are plotted in Figure 5 for the pattern-algorithm (note the usage of a logarithmic scale on the y-axis). The exponential nature of the algorithm is clearly visible. However, note that the median size most of the times stays 2-4 orders of magnitude below the worst case measured. Also, it should be mentioned that we expect that users in a trade-off based system rarely state more than 5-10 trade-offs – our assumption of 20 trade-offs clearly beyond the expected complexity of the problem.

Table 1. Measured Statistics for 15,000 sequences of 20 trade-offs

		Pattern-Algorithm		
Measure	**Inconsistent at**	**Time**	**Tree Size**	**Tree Depth**
Median	14	17 ms	9556	11
Quartil 1	10	4 ms	2427	9
Quartil 3	17	81 ms	40789	13
98%-Quant	19	7,068 ms	2,582,913	22
Min	2	<1 ms	5	3
Max	20	77 sec	8,548,700	37
Mean	13.23	542 ms	195,540	11.5

Table 2. Measured Statistics for 15,000 sequences of 20 trade-offs

	PrePost-Algorithm			Pruning-Algorithm		
Measure	**Time**	**Tree Size**	**Tree Depth**	**Time**	**Tree Size**	**Tree Depth**
Median	14 ms	9,337	11	3 ms	1,212	9
Quartil 1	3 ms	2,390	8	<1 ms	359	7
Quartil 3	73 ms	38,278	13	11 ms	4,632	11
98%-Q	3,977 ms	1,110,879	21	862 ms	193,202	19
Min	<1 ms	4	2	<1 ms	4	2
Max	84 sec	5,744,852	32	79 sec	5,622,241	31
Mean	346 ms	102,423	11.0	153 ms	23,215	9.5

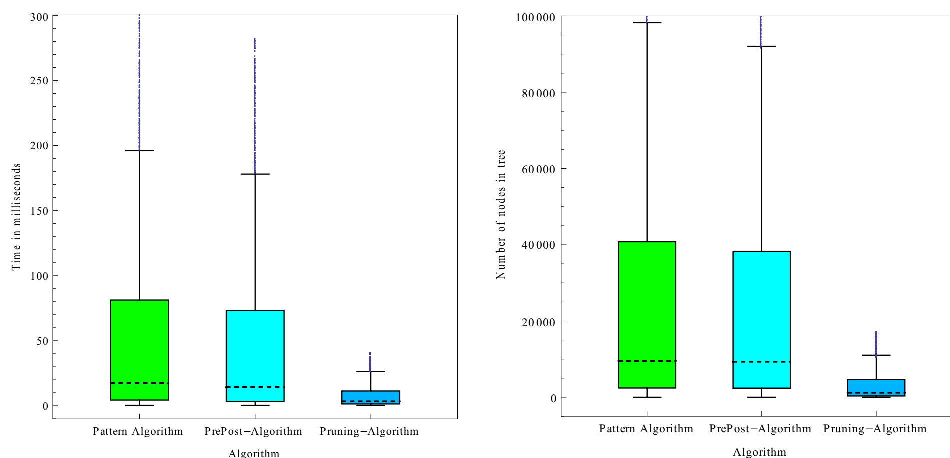


Figure 4: Time needed (left) and tree size (right) plotted for the three algorithms. Plots show quartial 1, median, quartile 3, outlier borders and mild outliers.

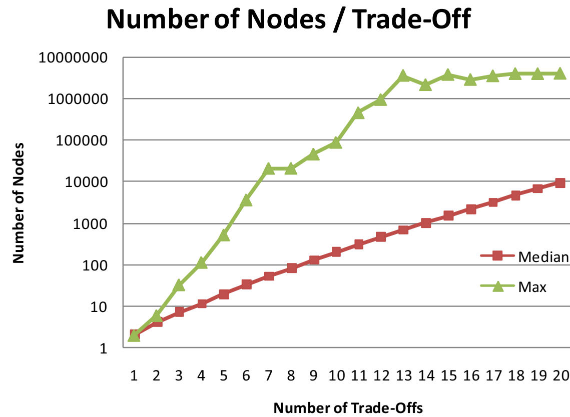


Figure 5: Tree size plotted by number of trade-offs for the pattern-algorithm

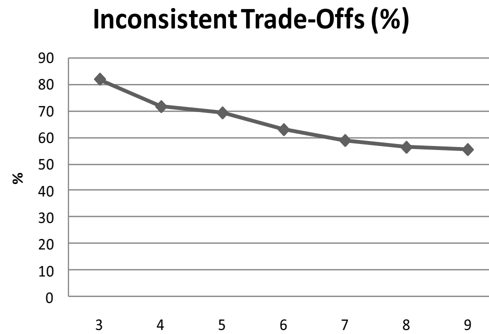


Figure 6: Percentage of inconsistent trade-offs for increasing number of attributes

5.4. *Varying the Number of Attributes*

For this evaluation, the number of considered attributes per database object has been varied. The number of attributes was adjusted starting from three up to nine attributes. For each number of attributes, we again generated 15,000 sets of 20 trade-offs each. The generated trade-offs randomly span at least two up to 75% of all available attributes (i.e., for 8 total attributes, we had trade-offs spanning 2 to 6 attributes).

The first observation is the decrease of inconsistent trade-off sets with increasing number of attributes: When regarding three attributes per possible database object, 82% off all generated trade-off sets have shown to be inconsistent. This number decreased to 51% when considering nine attributes. Figure 6 illustrates this relationship. The reason is that the average amount of overlap of each two trade-offs in the set decreases and there are fewer options to create preference cycles.

Other significant relationships could not really be established. For any number of attributes, computation time still stayed below 1 second for around 95% of all trade-off sets. For three and four attributes, the median tree size was measured at 653 and 1358 nodes, respectively, which are lower than those for more attributes. However, this trend could not be further observed for increasing numbers of attributes. This suggests that the actual runtime of the algorithm may vary significantly and is hard to estimate.

5.5. *Effectiveness of Precomputation of Matching Tables*

In chapter 4.4, we introduced an optimization which can save template comparisons. This was facilitated by pre-computing whether two given trade-offs can be applied directly after each other in the most general case. By evaluating 15,000 sets of 20-trade-offs each, it turned out that this heuristic can save 59% of all template comparisons in average (min: 36%, max: 84%, standard deviation: 6%).

6. Summary and Outlook

Trade-Offs have shown to be a natural and powerful addition to skyline queries. By extending the Pareto semantics, they can successfully remedy the fast growing result set sizes, one of the paradigm's major weaknesses. However, before trade-offs can be reliably integrated into the skyline computation, their consistency has to be ensured. In this paper, we presented a novel algorithm that is able to solve the essential problem of verifying arbitrary sets of multi-dimensional trade-offs. This ability helps to meet the long-term research challenge for cooperative and personalized information systems.
In contrast to former approaches, our algorithms are able to reliably detect any inconsistency in a set of trade-offs without posing restrictions on their expressiveness in a time-efficient manner. This was archived by abstracting from materializing prohibitively complex product order and without the need to access actual database instances. Instead, we focus on discovering specific patterns within possible trade-off sequences which indicate a cyclic preference in the product order. We have proven that certain patterns containing adjacent duplicate subsequences of trade-offs will always lead to such cyclic preferences and thus the trade-off set is inconsistent. This method was further refined into more efficient algorithms which can reduce the tree-size and computation times significantly.
In our evaluations, the latter algorithm has also shown a superior average-case performance. In fact, using the pruning-algorithm the consistency check for 75% of all evaluated trade-offs could be performed below 11 ms and a further 23% stayed below 800 milliseconds.
In our future work, we intend to focus on the problem of actually incorporating trade-offs into a user's skyline after the consistency check has been performed. Also, intuitive methods for eliciting suitable trade-offs from the user and guiding them over the entire process of applying compromises to the selection problem is a promising research area. Additionally, heuristics for further improving the performance for those rare cases of longer runtimes will also help to better understand the nature of trade-offs in human decision-making.

References

Balke, W.-T.; Güntzer. U. (2004): Multi-objective Query Processing for Database Systems. *Int. Conf. on Very Large Data Bases (VLDB),* Toronto, Canada.

Balke, W.-T.; Zheng, J.; Güntzer, U. (2005): Approaching the Efficient Frontier: Cooperative Database Retrieval Using High-Dimensional Skylines. In *Proc. of the Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, Beijing, China.

Balke, W.-T.; Güntzer, U.; Lofi, C. (2007a): Eliciting Matters – Controlling Skyline Sizes by Incremental Integration of User Preferences. *Int. Conf. on Database Systems for Advanced Applications (DASFAA),* Bangkok, Thailand.

Balke, W.-T.; Güntzer, U.; Lofi, C. (2007b): User Interaction Support for Incremental Refinement of Preference-Based Queries. *1st International Conference on Research Challenges in Information Science (RCIS),* Ouarzazate, Morocco.

Balke, W.-T.; Güntzer, U.; Lofi, C. (2007c): Incremental Trade-Off Management for Preference Based Queries. In *International Journal of Computer Science & Applications (IJCSA)*, Vol. 4(2).

Balke, W.-T.; Güntzer, U.; Siberski, W. (2007d): Restricting Skyline Sizes using Weak Pareto Dominance. In *Informatik - Forschung und Entwicklung (IFE)*, Vol. 21(3), Springer.

Balke, W.-T.; Güntzer, U. (2007). Consistently Adding Amalgamations to Preference Orders. *3rd Multidisciplinary Workshop on Advances in Preference Handling (M-Pref 2007)*, Vienna, Austria.

Börzsonyi, S.; Stocker, K., Kossmann, D. (2001): The Skyline Operator. *In Proc. of the Int. IEEE Conf. on Data Engineering (ICDE)*, Heidelberg, Germany.

Bradley, W. J.; Hodge, J.K.; Kilgour, D. M. (2005): Separable discrete preferences. *Mathematical Social Sciences*, 49(3):335-353.

Chomicki, J. (2006): Iterative Modification and Incremental Evaluation of Preference Queries. In *Proc. of the Int. Symp. on Found. of Inf. and Knowledge Systems (FoIKS)*, Budapest, Hungary.

Godfrey, P. (2004): Skyline cardinality for relational processing. How many vectors are maximal? In *Proc. of the Int. Symp. on Foundations of Information and Knowledge Systems (FoIKS 2004)*. Vienna, Austria.

Godfrey, P.; Shipley, R.; Gryz. J. (2005): Maximal Vector Computation in Large Data. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Trondheim, Norway.

Koltun, V.; Papadimitriou, C. (2005): Approximately Dominating Representatives. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Edinburgh, UK.

Kossmann, D.; Ramsak, F.; Rost, S. (2002): Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Hong Kong, China.

Lee, J.; You, G.; Hwang, S. (2007): Telescope: Zooming to Interesting Skylines. In *Proc. of the Int. Conf. on Database Systems for Advanced Applications (DASFAA)*, Bangkok, Thailand.

Lofi, C.; Balke, W.-T.; Güntzer, U. (2008): Efficiently Performing Consistency Checks for Multi-Dimensional Preference Trade-Offs. In *Proc. of the Int. Conf. on Research Challenges in Computer Science (RCIS)*, Marrakech, Marokko.

Lin, X.; Yuan, Y.; Zhang, Q.; Zhang, Y. (2007): Selecting Stars: The k Most Representative Skyline Operator. In *Proc. of the Int. IEEE Conf. on Data Engineering (ICDE),* Istanbul, Turkey.

Pei, J.; Yuan, Y; Lin, X.; Jin, W.; Ester, M.; Liu, Q.; Wang, W.; Tao, Y-; Yu, J. X.; Zhang, Q. (2006): Towards multidimensional subspace skyline analysis. In *ACM Trans. on Database Systems (TODS)*. Vol. 31(4).

Viappiani, P.; Faltings, B.; Pu, P. (2006): Preference-based Search using Example-Critiquing with Suggestions. In *Journal of Artificial Intelligence Research (JAIR)*, Vol. 27