Indexing and Selecting Hierarchical Business Logic

Alessandra Loro
Palantir Technologies
aloro@palantir.com

Anja Gruenheid*,
Donald Kossmann*†
* Systems Group
Dep. of Computer Science
ETH Zurich
† Microsoft Research
{agruen,donaldk}@inf.ethz.ch

Damien Profeta,
Philippe Beaudequin
S.A.S. Amadeus
{firstname.lastname}@amadeus.com

ABSTRACT

Business rule management is the task of storing and maintaining company-specific decision rules and business logic that is queried frequently by application users. These rules can impede efficient query processing when they require the business rule engine to resolve semantic hierarchies. To address this problem, this work discusses hierarchical indexes that are performance and storage-conscious. In the first part of this work, we develop a tree-based hierarchical structure that represents client-defined semantic hierarchies as well as two variants of this structure that improve performance and main memory allocation. The second part of our work focuses on selecting the top rules out of those retrieved from the index. We formally define a priority score-based decision scheme that allows for a conflict-free rule system and efficient rule ranking. Additionally, we introduce a weight-based lazy merging technique for rule selection. All of these techniques are evaluated with real world and synthetic data sets.

1. INTRODUCTION

Business Rule Management Systems (BRMS) handle the retrieval of business logic for internal and external service applications either as dedicated components within a company or as stand-alone commercial systems [10, 18]. This work is motivated by companies like Amadeus that use BRMS for storing and accessing rules with semantic hierarchies such as production line descriptors (i.e., a Mercedes is a type of cars and cars are a type of a transportation device) or geographical knowledge (i.e., Berlin is in Germany which is in Europe and that is part of the world). Given semantic hierarchies, a query to the BRMS needs to retrieve not only rules matching the query values but it additionally needs to consider the semantic predecessors. To illustrate the challenges of semantic hierarchies, take as an example the domain of airline information systems as encountered within the Amadeus BRMS and visualized in Figure 1. Here, the BRMS stores rules ($r_1 - r_5$) that describe which food should be served when flying from one geographic region to another. The hierarchical dependencies of each respective region are shown in Figure 2.

	criteria C			consequence
r_i	c_1	c_2	c_3	d

Rule	Origin	Destination	Class	Food
r_1	TXL	Switzerland	E	Chocolate
r_2	Europe	ZRH	*	Sandwich
r_3	Berlin$_1$	Germany	*	Sausages
r_4	Berlin$_2$	World	E	Croissant
r_5	SXF	BSL	E	Sandwich

Figure 1: Example rule set.

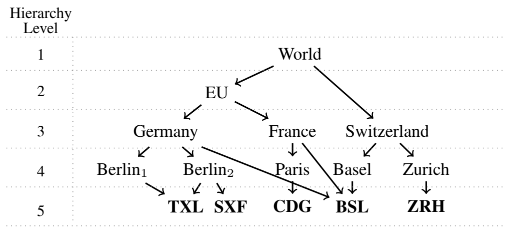

Figure 2: Visualization of the example market hierarchy.

EXAMPLE 1 (HIERARCHICAL RULE RETRIEVAL). *The caterer of an airline needs to know which food is served on each flight. For a flight from Tegel (TXL) to Zurich (ZRH), the request is processed as follows. Value 'TXL' maps to the hierarchical markets {TXL, Berlin$_1$, Berlin$_2$, Germany, Europe, World} while 'ZRH' matches {ZRH, Zurich, Switzerland, World}. To retrieve all candidate rules, the system determines all 24 market combinations ({TXL,ZRH},...,{World,World}) and extracts the matching rules from the rule set. Rules r_1, r_2, and r_4 all contain either 'TXL' or one of its predecessors in criterion 'Origin' and either 'ZRH' or one of its predecessors in criterion 'Destination'. Out of these, a rule is considered a better match if it references an airport instead of a region because it is a more specific (smaller) geographical point, i.e., r_1 is chosen over r_2 and r_4 as top match to the user's query.*

This approach to hierarchical business rule matching is the starting point for our work. It is implemented in the Amadeus BRMS which has the following additional performance specifications: First, the BRMS needs to support read-heavy workloads composed of hundred thousands of point queries such as 'Give me the food served on a flight from TXL to ZRH'. For each of these queries, the system guarantees a processing time of 2ms in a service level agreement (SLA). Furthermore, the BRMS is run on commodity servers where the rule sets of multiple clients are co-located on the same server. This effectively limits the amount of available main memory storage and processing power per rule set and serviced client.

Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

In our work, we enhance the existing Amadeus business rule engine by addressing its two key components **rule retrieval** and **rule selection**. We first introduce a hierarchical index which presents a solution to the problem of rule retrieval, i.e., to finding the candidate rules that match a given query. Remember that this is done in the baseline approach through a) finding all combinations of markets that are a hierarchical match to the query and b) to query a hash index to retrieve the matching rules. A drawback of the existing approach is clearly that the number of index accesses increases drastically with the number of hierarchical criteria: For example, in a rule set explored in the experimental evaluation, one query value maps to up to most 84 markets. With two hierarchical criteria, this can lead to 7056 index accesses in the worst case. We therefore introduce a hierarchical index that minimizes the amount of index accesses to one per hierarchical criterion and furthermore reduces the storage cost by eliminating the need for an explicit value-market mapping such as 'TXL' maps to markets $\{$TXL, Berlin$_1$, Berlin$_2$, Germany, Europe, World$\}$. In our experimental evaluation, we show that a hierarchical index effectively reduces main memory allocation to up to most 23% of the original storage cost for single-criterion indexing. In addition, we show that indexing techniques provide faster response times than any comparable hash-based approach.

The second component addressed in this work is rule selection, i.e., the task of finding the top-k rules that match the user's query. Semantic hierarchies increase the complexity of rule selection when retrieving candidate rules for a single query value: If rules with different hierarchical values match a query, there needs to be a mechanism that decides which rule is the best match from all available candidates. We therefore develop the notion of *specificity*, i.e., a rule is more applicable if its values are more specific than other alternative candidate rules. Furthermore, we show how it can be enforced with strict or fuzzy top-k matching of business rules.

To the best of our knowledge, hierarchical business logic has not been discussed in prior work as it has two key features that make it different from other hierarchical indexing problems: Hierarchies here denote semantic knowledge that is applicable for a value and its hierarchical predecessors. Any query to the system therefore needs to retrieve information about the query value and all its predecessors instead of information about the query value only. It additionally requires novel rule selection mechanisms as the query processor needs to evaluate multiple rule sets that match the query, i.e., one matching set per hierarchical value. We make the following contributions:

- **Rule Retrieval**. We introduce a hierarchical index that is based on the idea of a range index. In this index, rules are not only stored in the leaf but also in internal nodes. To address efficiency, we further present two implementation variations to optimize for main memory storage resp. performance.
- **Rule Selection**. Given a set of candidate rules that match the indexed criteria, the best matching rule(s) that fit the query have to be selected. We present a framework that enables a consistent BRMS and discuss how selection can be efficiently realized after the rule retrieval phase.
- **Experimental Evaluation**. All approaches are evaluated with three real-world datasets obtained from airline services with the same hierarchy structure as our running example. Additionally, we evaluate our techniques with synthetic datasets.

The paper is structured as follows: First, we discuss prior research in the related areas of query and XML processing as well as object-oriented databases in Section 2. We then formally define the problem of hierarchical business rule indexing and its two components, rule retrieval and rule selection in Section 3. Section 4 describes a general framework that shows the interaction between these components. In Section 5 and Section 6, we clarify how they can be addressed in the hierarchical business rule processing context. All of the presented techniques are then evaluated with real-world and synthetic datasets that show the strengths and weaknesses of the proposed mechanisms in Section 7. Section 8 concludes this paper.

2. RELATED WORK

BRMS have been part of commercial information systems for a long time [8, 10, 18]. Hierarchies are supported by these applications as also described in Section 4 but to the best of our knowledge, there has not been any work on using these hierarchical structures to improve query performance. Indexing hierarchical business logic touches on related work in the following areas:

Keyword Search. Work exploring semantic hierarchies has focused on XML processing [14] where queries are keyword requests, exploring (part of) the hierarchy for exact matches to this (set of) keywords [11]. To enhance performance, inverted indexes have been proposed for fast, value-specific accesses within the (XML) hierarchy [7, 21] which corresponds to our baseline solution. Extending this research, keyword search has been recently augmented to not only address keyword-specific requests but to expand it automatically to fit multiple keys [4]. In contrast to our work, this expansion is not hierarchy-driven but aims to match the keyword to the underlying XML structure. Furthermore, the objective of the business rule matching problem presented in this work is to find all rules that match either the defined value or any of its logical predecessors. A challenge that is unique to our problem is client-specific content that may contain overlaps in the parent-child definitions such that a child node may have multiple logical parents (for example city 'Berlin' is associated with two markets because the caterer may not deliver to 'SXF' in the running example). Overlap in hierarchies has been discussed in the literature [16] but not in the context of query processing where these relationship constraints are highly correlated to the performance of any processing technique.

Query Processing for Ordered Structures. Query processing has been extensively studied for XML documents for which their hierarchical structure is used to enhance information retrieval [15]. Similar to research on semantic hierarchies, these approaches focus on retrieving information from a specified node of the hierarchical tree, ignoring the content on the way through the hierarchy to that desired node. Indexes are thus used to optimize path accesses without the notion of semantic consistency along a path [6]. Query languages have been designed in that context to improve (multi-)object accesses of the hierarchical structure [20] but without support for retrieving hierarchy-specific content.

XML Ranking. Finding and sorting the top-k values retrieved from an XML structure is part of this research area and is related to the rule selection problem in our work. The idea of an internal cost function that identifies the top-k elements has been explored in previous work [7] but has not been looked at in the context of parallel queue processing as items are commonly retrieved first and ranked in a second step. Alternative work [2] has focused on achieving a consistent ranking through Bayesian networks and using the queries themselves as indicator for the ranking which is not applicable for our use case. Outside of the XML context, ranking functions such as [5] have looked at processing multiple data sets (here: rule sets) in parallel. In contrast to our work, these assume that the items of these data sets are dependent and influence each other.

Object-Oriented Databases. Hierarchy is a key feature of object-oriented design and has been well-studied for indexing in object-oriented DBMS (OOMS). Here, classes have super and subclass relationships which correspond to the semantic hierarchies studied in this work. To address hierarchical indexing, two types of indexes are commonly applied in OOMS: Key grouping indexes such as the CH-tree [13] and set grouping approaches such as the H-tree and the CG-tree [12, 17]. While the hierarchical criteria in this work have similar child/parent relationships as set grouping approaches, they are more straightforward as their hierarchical relationship is not bound to an encompassing class but to the attribute directly. Key grouping approaches are thus more applicable for our use case but do not take into consideration that one value might be represented differently in each hierarchical level but assumes values to be persistent throughout the hierarchy. Alternatively, hierarchical dependencies can be modelled through adjacency lists in SQL directly, [3]. Note that this line of work, although similarily constructing hierarchies, does not address the problems of optimized rule access and space-efficient rule processing that we solve with our approaches. Furthermore, all of these techniques focus on the retrieval of values only and do not address preference selection in the result set.

Indexing Techniques. Obviously, an important area of related work are indexing techniques which form a large part of our contribution. Indexing for rule bases has been explored explicitly for (non-hierarchical) predicates, [9, 19]. However, indexing hierarchical rules requires better suited indexing techniques as we have shown in our experimental evaluation because especially in use cases with a large number of hierarchical criteria, the search space for point-based queries explodes. As a result, traditional rule indexing techniques are not applicable here.

3. PROBLEM STATEMENT

In this section, we describe briefly how queries are processed in a BRMS and introduce the notation used throughout this paper.

3.1 Overview

In a business rule engine, we define a rule set R, with $|R| = n$, as the set of rules that are accessed for a specific purpose, for example retrieving flight information. Every rule $r_i \in R$ consists of a set of criteria C and exactly one consequence d as shown in Figure 1. We limit the number of consequences at this point to one because their existence is required for retrieving business logic but increasing their number will not influence the results presented in this work. For each rule r_i, a criterion value c_k^i may take one of the following forms for any criterion $c_k \in C$:

- It is a constant value.
- It is a constant value that is part of a hierarchy.
- It is a wildcard (marked through an asterisk).

In this context, we define a hierarchy as a set of constant values that obey a semantic ordering. The relationship of any two hierarchical values h_i and h_j within the same criterion can be classified in either of three categories: A successor, predecessor, or independence relationship. We denote the successor relationship as $h_i \prec h_j$. Alternatively, value h_i can be a predecessor of h_j, marked as $h_i \succ h_j$. In practice, successor and predecessor relationships may span multiple levels of a hierarchy, i.e., if h_k is a predecessor of h_i, then it is also a predecessor of h_j. For example in Figure 2, we see that World is a predecessor of Zurich, making Zurich analogously the successor of World. Last, hierarchical values can be independent of each other denoted as $h_i \parallel h_j$. Note that hierarchical relationships may overlap such that $h_i \parallel h_j$ and $h_k \succ h_j$ and $h_k \succ h_i$ holds. The third type of criterion values, wildcards, can occur in hierarchical and non-hierarchical criteria. It implies indifference to a match of the respective attribute value.

EXAMPLE 2 (CRITERION VALUES). *Figure 1 shows an example rule set R with five rules r_1 to r_5 that define which food should be served during flights. Every rule adheres to a set of criteria C that consists of three values c_1, c_2, and c_3, and has exactly one consequence (Food). Criteria c_1 (Origin) and c_2 (Destination) contain constant values that are part of a hierarchy, for example $h_{TXL} \prec h_{EU}$ applies here. Criterion c_3 (Class) contains constant values and wildcards which are not part of a hierarchy.*

In this example, the goal of querying the BRMS is to determine the food served on a specific flight described by a query q_j. A query consists of a set of attributes A where $|A| = |C|$, i.e., the number of attributes in the query matches the number of criteria in the rule set. For each query, the task of the BRMS is to determine those top-k rules in R that match q_j best. That is, a rule r_i matches a query q_j if for each attribute a_k^j the rule contains either matching criterion values (for criteria that contain constant values) or criterion values that are a hierarchical match to a_k^j.

3.2 Rule Retrieval

If the BRMS stores a rule set R and needs to determine those rules $r_i \in R$ that match a query q_j, it intuitively needs to search R for the rules that match every attribute value a_k^j with their corresponding criterion value c_k^i. As verifying all criteria is expensive, systems commonly apply filtering mechanisms to select a subset R_k of R that contains all rules that eventually match q_j but potentially more rules that have the same attribute value. If c_k is such a criterion, then rule retrieval is the task of finding those rules r_i that match the k-th attribute of q_j.

DEFINITION 1 (RULE RETRIEVAL). *Given a query q_j and criterion c_k, rule retrieval is the process of finding all rules $r_i \in R_k$, where $R_k \subseteq R$ holds, such that the rule criterion values c_k^i fulfill the following condition:*

$$\forall r_i \in R_k : \{a_k^j = c_k^i \ \vee \ c_k^i = `*'\}$$

A rule retrieval definition like this holds if the criteria do not contain hierarchical values, i.e., where criterion values are simple constants or wildcards. If the k-th criterion is hierarchical, a_k^j needs to be translated into its corresponding hierarchical values first. For example if the k-th criterion is 'Origin' in our running example and a_k^j is 'TXL', a function h determines that the matching six criterion values are {TXL,...,World}. Queries against rule sets with hierarchical criteria potentially match not only a single value but possibly multiple criterion values. For each of these, the subset $R_{c_k^i} \subseteq R$ denotes the rules that match value c_k^i in the k-th criterion.

DEFINITION 2 (HIERARCHICAL RULE RETRIEVAL). *Hierarchical rule retrieval requires a function h which determines all hierarchical criterion values for a_k^j such that $h(a_k^j) = \{c_{k_1}^i,\dots,c_{k_{|h|}}^i\}$ with $|h| = |h(a_k^j)|$. Given a query q_j, hierarchical rule retrieval is then the process of selecting all rules r_i that form any subset $R_{c_{k_l}^i} \subseteq R$, i.e., that match any of the hierarchical criteria of a_k^j such that:*

$$\bigcup_{l=1}^{|h|} r_i \in R_{c_{k_l}^i} : \{c_{k_l}^i \in h(a_k^j)\}$$

In this work, we focus exclusively on hierarchical rule retrieval. State-of-the-art mechanisms and their associated drawbacks are shown in Section 4 while we discuss our proposed solutions to rule retrieval in Section 5.

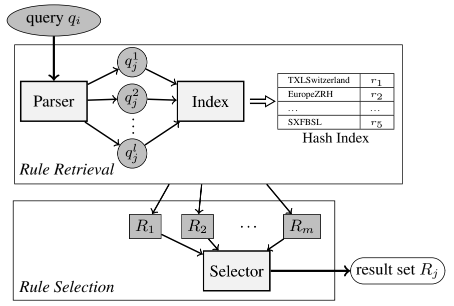

Figure 3: Naive query evaluation framework.

3.3 Rule Selection

Rule selection is the process of finding the top-k rules that match query q_j. For that purpose, it takes as input the rules found in the rule retrieval phase R^* and then determines the top-k rules by examining the weight $\omega(r_i)$ of all rules $r_i \in R^*$ to determine which r_i best matches the issued query. As shown in Definition 2, hierarchical rule retrieval outputs multiple rule subsets per query depending on the indexed hierarchical criterion. For simplicity, we refer to all input rule subsets of rule selection as $R^* = R_1 \dots R_m$. For r_i in any of these subsets to be considered as top-k match, its attributes need to match all rule criteria, even if some of the criteria are not used during rule retrieval.

DEFINITION 3 (RULE SELECTION). *Given multiple rule sets $R^* = R_1 \dots R_m$ with $\sum_{1 \dots m} |R_m| < n$, rule selection determines the top-k rules that are the best matches to query q_j. To be selected as top match, any rule $r_i \in R^*$ needs to fulfill two criteria:*

1. *All attribute values must fulfill the condition in either Definition 1 or Definition 2 for all criteria, independent of whether they are indexed or not.*
2. *The rule weight $\omega(r_i)$ of r_i is at least as high as the weight of $|R'| - k$ rules where $R' \subseteq R^*$ are those rules that conform to condition 1 and k is the number of top rules to be selected.*

Details on rule selection in the Amadeus BRMS are shown in Section 4. These mechanisms are improved with formalized weighting schemes for $\omega(r_i)$ and new techniques for rule ranking in Section 6.

4. AMADEUS BRMS

To put the two components of business rule processing, rule retrieval and rule selection, into context, this section first gives an overview of query processing in the Amadeus BRMS which is a system that amongst others stores airline data of the same type we present in our running example. We will then describe how our work addresses the issues that arise in the existing system and how it modifies the system to handle hierarchical rule content efficiently.

4.1 System Description

The existing Amadeus business rule engine manages queries with hierarchies in a two-step process. First, it resolves the query q_j into a set of subqueries $q_j^1, \dots, q_j^l$ as visualized in Figure 3, reconstructing the hierarchies it contains with a hierarchy map. An excerpt of the hierarchy map for the running example is shown in Table 1. It represents part of the hierarchy shown in Figure 2. Going back to hierarchical rule retrieval as defined in Section 3.2, each of these subqueries corresponds to one hierarchical value $c^i_{k_1} \dots c^i_{k_{|h|}}$ or all combinations thereof if there exist multiple hierarchical criteria. For example on a flight from Berlin-Tegel to Zurich, the hierarchical criteria 'Origin' and 'Destination' would require the query to be split into subqueries {TXL,ZRH} ... {World,World}. After all subqueries have been constructed, the system uses (multi-dimensional) indexes to retrieve all candidate rules that may match q_j. The system employs a (merged) hash index that maps the hash of all hierarchical value combinations (across criteria) to their actual rules. A hash index is an efficient solution to the access problem but it has to be probed for all subqueries which is the trade-off that current BRMS vendors are willing to make. In practice, one hierarchical value can map to a large amount of objects (in the running example an object is a market) based on the amount of semantic knowledge the user wants to convey. As a result, storing a hashmap directly incurs additional storage cost, see Section 7 for details.

Table 1: Example value-market map.

Value	Markets
'TXL'	TXL, Berlin$_1$, Berlin$_2$, Germany, EU, World
'SXF'	SXF, Berlin$_2$, Germany, EU, World
'CDG'	CDG, Paris, France, EU, World
...	...

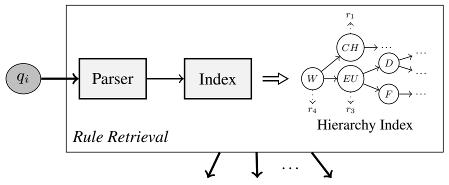

Figure 4: Rule retrieval component with hierarchy index.

As shown in Figure 3, the rule selection component of the Amadeus BRMS takes as input those rule subsets $R^* = R_1, \dots, R_m$ that are obtained in the rule retrieval phase. It then selects the subset of rules R_j^* that are the top-k responses to query q_j, $R_j^* \subseteq R^*$. The challenge for rule selection is to process R^* efficiently as response time is crucial for user satisfaction and computational effort has to be minimized in distributed setups where multiple clients share the same server resources. The Amadeus BRMS resolves this problem by keeping $R_1, \dots, R_m$ sorted throughout the rule retrieval and selection steps according to client-specific weights. The combined set R^* is then computed by merging all subsets. Afterwards, the top matching rules R_j^* can be extracted by scanning R^*, matching each observed rule r_i with q_j, and stopping once the top matches have been found. This solution has several obvious problems: First, it requires the client to assign a weight to each rule manually which means that the client needs domain-specific knowledge for each rule and how it compares to any other rule in the rule set. Second, merging sorted rule sets can be done in $O(n \log n)$ but is inefficient in practice because not all merged rules are actually part of R_j^*. Thus, sorting them causes unnecessary performance overhead.

4.2 Solution Space

In this subsection, we briefly discuss the two design concerns that we address by modifying the rule retrieval and selection components: Performance of the system and its index storage cost.

Performance Requirements. We observe from our experimental evaluation with real-world rule sets that rule retrieval is the bottleneck of Amadeus' current rule processing engine: Given a two-dimensional hash index as would be employed on the hierarchical criteria 'Origin' and 'Destination' in our running example, rule retrieval takes up at least 90% of the rule matching process. To address this problem, we replace the rule retrieval component of the baseline

Table 2: Object references by hierarchy level in the example and two real-world datasets (GCA and MCO).

Hierarchy Level	Running Example	MCO		GCA	
		Mean	Dev.	Mean	Dev.
1	15	19,763.2	11.1	19,965.7	2692.2
2	11	5,012	2968.3	2616.8	1629.7
3	5	489.4	1426	220.7	609.1
4	2	28.4	22.5	5.3	4.6
5	1	48.1	789.7	1.3	4.9

approach with a mechanism that does not resolve the hierarchies up front. Instead, the parser simply forwards this task to the indexing structure as shown in Figure 4. Querying the index multiple times is thus replaced by only one query to the corresponding indexing structure per hierarchical criterion through which all candidate rules are retrieved. We show that with this mechanism, the amount of relative time spent on rule retrieval drastically improves by at least 35% for the same rule sets (Section 7). Implementation details of our hierarchical indexing techniques are discussed in Section 5.

To improve the existing system, our work proposes in-place modifications to the rule selection component of the BRMS. It first introduces a formal weighting scheme which is based on the notion that the properties of the rule criteria such as the hierarchy level or client-specific preferences for one criterion over another can be used to automatically assign weights to rules. Clients can then decide whether the system allows collisions in rule weights, i.e., multiple rules are retrieved through the same query because they have the same weight, or not. To address the inefficiency in filtering the top-k rules from R^* expressed previously, we propose a lazy merge technique that maintains the retrieved rule sets and only merges them on demand. All of these techniques are described in detail in Section 6.2.

Storage Requirements. During the rule retrieval phase, the Amadeus BRMS manages hierarchies with two maps: A hierarchy map that maps a hierarchical value to (a set of) object(s) that represent the hierarchy. In the running example this corresponds to the example value-market map shown in Table 1. The second map it stores is the actual index as an object-rule map. To illustrate the memory storage cost of this approach, we examine two real-world datasets (MCO and GCA). They consist of rules that contain logic for flights where the hierarchies are of geographical nature and have been defined by two different clients. Rule set MCO contains 21,487 different values (airports, cities, etc.) that map to 912 client-specific markets. On average, each of these values references 27.72 markets and each value maps to at least 2 and at most 84 markets. For dataset GCA, we observe 21,520 distinct values that map to 796 different markets. Here, each value is associated with at least 3 different markets and at most with 17 markets. The average number of markets per value is 5. All markets defined by the clients are associated internally with a hierarchy structure similar to the one shown in Figure 2 as geographical knowledge can be easily mapped to different hierarchy levels. These levels differ in their geographic scale, i.e., hierarchy level 5 corresponds to airports while hierarchy level 2 responds to world regions. Table 2 then reports the average number of market enumerations of a certain hierarchy level per possible value as well as their standard deviation. Using the running example, markets associated with the third hierarchy level (countries) are mentioned 6 (Germany), 4 (France), and 5 (Switzerland) times for different value mappings in the hierarchy map. Similar to the running example, we observe as a general trend in the real-world datasets that a smaller

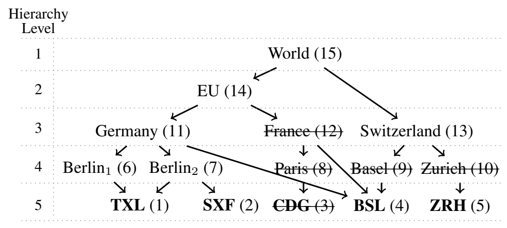

Figure 5: Visualization of compressed example market hierarchy.

hierarchy level often implies a higher number of associations in the hierarchy map. For example, the hierarchy maps in these rule sets store a first-level market reference in up to 92% (92.8%) hierarchical values on average while the higher level markets are stored for a substantially smaller number of hierarchical values (on average in 48.1 (GCA) and 1.3 (MCO) hierarchical mappings).

As all hierarchies need to be completely resolved from the query values to guarantee completeness, all of the possible <value, object> pairs in the hierarchy map have to be stored in the original approach of the Amadeus BRMS. Obviously, storing the same market reference multiple times is unnecessary and incurs high storage cost. This observation motivates memory-consciousness next to efficiency as a requirement for a suitable indexing technique and shows how it is violated by the current baseline approach. The hierarchical indexing techniques presented next explicitly address both dimensions of effective rule processing and shows how the hierarchy map can be transformed into a hierarchy index that incurs lower memory storage but provides good retrieval performance.

5. HIERARCHICAL INDEXING

In this section we establish the notion of a hierarchical index. We first describe how hierarchy graphs can be generated from user-defined hierarchy maps and how they improve the storage cost and efficiency of the rule retrieval component. In this context, we discuss (dis-)advantages of different graph representations for hierarchies. Last, we present two variations of a tree-based hierarchy index which either enhance memory allocation or improve access performance.

5.1 Hierarchy Index

To address the problem of efficient rule retrieval, we propose to replace the hierarchy map as well as the corresponding hash index with a directed acyclic graph $G = (V, E)$. Nodes in this graph correspond to a (combination of) objects (in the running example markets) on different hierarchy levels. Edges then represent the hierarchical relationship between these objects. This graph cannot be disconnected, i.e., it is imperative that there exists exactly one (artificially generated) root node which serves as starting point when accessing the hierarchy index (HI). Rules are annotated on all those nodes that reference the market that they correspond to after the HI has been built. For example rule r_1 of the running example would be associated with TXL (Origin) or Switzerland (Destination), depending on the indexed criterion.

The HI can be constructed from a client-specific hierarchy map as follows: As every value is defined by a set of objects, these object dependencies can be used to establish hierarchical relationships in the graph. More specifically, the sub- and superset relationships of these objects explicitly define the hierarchical dependencies of nodes in the HI.

PROPERTY 1 (SUBSET RELATIONSHIP). *A hierarchical node v_i is the predecessor of v_j ($v_i \succ v_j$) if and only if the objects $M(v_i)$*

that v_i maps to are a strict subset of the objects $M(v_j)$ that v_j is referencing. We denote a subset relationship as $M(v_i) \subset M(v_j)$.

EXAMPLE 3 (SUBSET RELATIONSHIP). *Country Germany maps to markets {Germany, EU, World}. Values Berlin$_1$ and Berlin$_2$ map to the same markets but also one additional market: Their unique Berlin$_1$ resp. Berlin$_2$ markets. Thus, the markets that these values map to are a superset of the markets associated with Germany which is a predecessor market to them.*

Conditioning the structure of the graph on set relationships instead of the underlying client-specific hierarchy has another advantage: Some objects created by clients are generated for a specific purpose but are actually not required because values do not map to them. Take country France in the running example: Though it makes sense to introduce a hierarchical object France on a logical level, it is not used in rules r_1 - r_5 which means that this market definition is superfluous. These markets can be removed from the hierarchy without violating rule retrieval. The unnecessary markets for the running example are marked in Figure 5. Prunable values are often a result of evolving rule (and object) sets as well as content available to the client that he or she does not use.

Unfortunately, representing the map as a DAG albeit intuitive is suboptimal in terms of index access time. Remember the definition of hierarchical indexing to understand why: Per index access, all rules that either match the query value or its predecessors are retrieved. If a value has many predecessors, it will thus incur a higher retrieval cost than a value with few predecessors.

EXAMPLE 4 (DAG RULE RETRIEVAL). *Imagine a query that asks for the flights between airports 'TXL' and 'BSL'. First, values 'TXL' and 'BSL' are resolved to a set of markets. For each of these, we then start at the root, traversing the* HI *using constrained DFS, i.e., exploring only those subtrees that contain a match to any of the markets in the mapping. For example 'TXL' maps to Germany but not France, thus the* HI *is not explored during traversal. The two accesses to the index (one for each Origin/Destination) thus need to process six resp. five nodes to find all possibly matching rules.*

More formally, we can show that the number of processed nodes is in the worst case dependent on the number of vertices in G.

THEOREM 1 (ACCESS COMPLEXITY OF HI). *The number of vertex accesses of* HI *per rule retrieval attempt is bounded by 2^m if m is the number of distinct objects in the hierarchy.*

PROOF. In the worst case, all possible market combinations are generated as nodes in the graph in the following manner: The uppermost (empty) node is the root node which connects to all mapping sets $M(v_i)$ such that $|M(v_i)| = 1$. If each of these nodes v_i is connected to all nodes v_j such that $|M(v_j)| = 2$ and any 2-combinations of object values are represented in the v_j nodes and so on until $|M(v_l)| = m$, then the whole graph is completely connected. If queried for the objects matching v_l, it thus has to process a number of nodes that is equivalent to the number of unordered combinations generated for the m objects, 2^m. ☐

Obviously, this bound is undesirable as it guarantees no feasible access time and though it has minimal index storage cost, it is not optimal in terms total storage cost either: To obtain all rules from the HI, the first step is to use the hierarchy map to find the objects the query value maps to. As a result, the hierarchy map needs to be stored as before which adds again unwanted storage cost. To address these issues, the next section discusses a tree-based hierarchy index that resolves the storage limitations as well as performance guarantee problems of the HI.

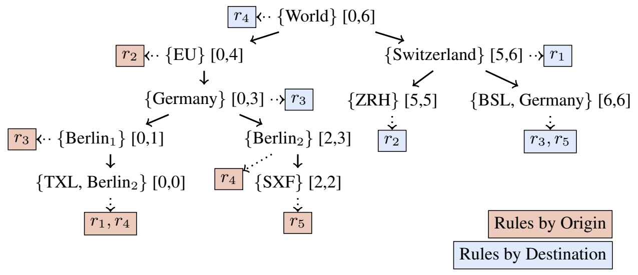

Figure 6: Tree-based hierarchy index for running example.

5.2 Tree-based Hierarchy Index

The tree-based hierarchy index (THI) is a variation of the hierarchy index representing a graph $G_T = (V_T, E_T)$ where a child has at most one parent but a parent may have multiple children. Thus, the parent-child relationship is not a many-to-many but a many-to-one relationship. Given this modification to the HI, we can guarantee predictable access time for the THI as the maximum number of traversed nodes equals the hierarchy depth. Intuitively, the only difference between the HI and THI is a shift in the node descriptions for a child node that has multiple parents, as the remaining nodes adhere to the tree structure already. That is, for every child node with multiple parents the THI construction algorithm removes the link to all parents except for one and adds the object references of the parents directly to the child node when inserting it into the THI.

EXAMPLE 5 (TRANSFORMATION INTO THI). *In our running example, airport market 'TXL' has two markets as parents, 'Berlin$_1$' and 'Berlin$_2$'. The THI algorithm then randomly chooses 'Berlin$_1$' as parent to 'TXL' but extends 'TXL' to node {'TXL','Berlin$_2$'}. A visualization of this modification is shown in Figure 6.*

This structure is obviously not as space-optimized as a HI but has predictable performance because single path traversal is enforced. For a THI that adheres to this property, its tree depth is guaranteed.

PROPOSITION 1 (INDEX DEPTH OF THI). *The depth of a* THI *is at most the depth of its corresponding* HI.

This proposition holds because if there exists no node in HI that has multiple incoming edges, then the HI and THI are the same. Otherwise, vertex insertion will guarantee that a child node is on the same hierarchy level as it was in the HI. Constructing the THI guarantees a lower number of node accesses for THI than HI:

THEOREM 2 (ACCESS COMPLEXITY OF THI). *The number of vertex accesses for* THI *per rule retrieval attempt is bounded by the hierarchy depth δ.*

PROOF. The THI confirms to the tree relationship property, there exists exactly one path from root to a leaf with a maximum depth of δ. In the worst case, $\delta = n$ but commonly $\delta \ll n$ holds. ☐

As disadvantage of this technique, notice that constructing the THI causes an increased number of references to rules because they are duplicated for the parent as well as for the child. Duplication is necessary as otherwise it cannot be guaranteed that the best matching rule is found. For example, if Berlin$_2$ is omitted from the new {'TXL','Berlin$_2$'} node, rule r_4 is not retrieved when querying for value TXL. Similar to the construction of the complete HI, rules are assigned to the vertices in G_T after its construction. Rules are associated directly with objects, i.e., markets in our running example. Since these are present multiple times in the G_T, rule assignment

ALGORITHM 1. Hierarchical index creation (Function INSERTVALUES).

```
Input: Node $v$, Set $M_i$, Tree $G_T$
$Node\ v_{new} \leftarrow new\ Node(M_i)$
// only pursue if candidate objects are novel
if $M(v) \neq M_i \land \forall v_j \in v.children\colon M(v_j) \neq M_i$ then
    // candidate objects are a superset of a child node
    if $\exists v_j \in v.children\colon M(v_j) \subset M_i$ then
        $Node\ v_j \leftarrow pickChildSup(v.children, M_i)$
        INSERTVALUES($v_j$, $M_i$)
    // candidate objects are a subset of a child node
    else if $\exists v_j \in v.children\colon M(v_j) \supset M_i$ then
        $Node\ v_j \leftarrow pickChildSup(v.children, M_i)$
        $v.replace(v_j, v_{new})$
        $v_{new}.addChild(v_j)$
    // candidate objects have overlap with a child node
    else if $\exists v_j \in v.children\colon M(v_j) \cap M_i \neq \emptyset$ then
        $Node\ v_j \leftarrow pickChildOv(v.children, M_i)$
        $M_o \leftarrow M(v_j) \cap$ M(v)
        $Node\ v_o \leftarrow new\ Node(M_o)$
        $v.replace(v_j, v_o)$
        $v_o.addChild(v_j)$
        $v_o.addChild(v_{new})$
    // candidate objects are independent of children
    else $v.addChild(v_{new})$
```

is straightforward. For the running example, we show a complete example THI in Figure 6. Note that this only is one example THI as the choice of parent for both markets 'TXL' and 'BSL' is random.

Furthermore, we observe that criteria with the same underlying hierarchy can share one index: If each criterion uses 50% or more of the combined structure, storing them in the same structured tree is more efficient than keeping a separate index per criterion. Accessing either criterion is realized by adding a flag that determines the current criterion whenever the index is accessed.

Index Construction. Given the hierarchy map of a rule set, a tree-based hierarchy index can be constructed as described in Algorithm 1. Function INSERTVALUES is called for every set of objects M_i that need to be inserted into the THI where the root node is the starting point for insertion and the initial assignment of node v. Before INSERTVALUES is called, the calling function needs to make certain that (i) there exists a root node, (ii) M_i has some overlap with $M(v)$ without which they are independent and M_i forms a new sibling to v under an artificial root node, and (iii) M_i is not a subset of $M(v)$, i.e., the new objects are not higher in the hierarchy than the root node. We then observe that with respect to M_i, an insertion is only necessary if the objects do not match any existing set of objects (Line 3). If M_i is different, one of the following properties determines the point of insertion for M_i:

1. M_i is a superset of any of v's children.
2. M_i is a subset of any of v's children.
3. M_i shares some objects with any of v's children.
4. M_i is independent of v's children.

If M_i is a superset of any v_j that is a child of v, M_i can be pushed down further into the hierarchy. To resolve the insertion point, the algorithm determines a v_j that can be used for further propagating M_i through method $pickChildSup$ (Line 5). If there exist multiple candidates, this function picks the node that has the highest number of matching objects with M_i to minimize storage cost. In case there still exist multiple candidates, it decides upon any v_j at random. The function then calls itself with v_j as new starting node. If M_i is a subset of any of v's children, it needs to be inserted between that child and v. Again, the insertion point is picked at random after the best possible matches are determined through $pickChildSub$ (Line 8). Node v_{new} then simply replaces v_j as v's child after which v_j is attached to v_{new} as child. The generation of an in-between node for overlap with any of the children of v is analogous (Lines 11-17). Last, if none of the above properties is fulfilled, M_i is independent and is therefore inserted as a sibling node.

Query processing with THI's. The tree structure explicitly used by the THI has one big advantage in comparison to the DAG used for the HI in terms of query processing: It does not require a hierarchy map for computation but can be encoded as a range index, [1], which decreases the required storage space for the index. Instead of allocating a variable number of objects depending on the market position in G_T, each node is assigned two integer values, one upper and one lower bound, see Figure 6 for an example range encoding. To maintain update functionality, we propose to store the original hierarchy map nevertheless on disk to be fetched into main memory on demand, for example for index adjustments. The range generation for the THI is straightforward, it can be implemented with a depth first search algorithm to traverse the THI and annotate the children of the node before assigning a weight to the parent. A parent is assigned the lower bound weight of its lowest (in terms of the range value) and the upper bound weight of its highest child. If the lower bound and upper bound have the same value, a parent has exactly one child. To differentiate between parent and child, the range of the parent is expanded by one range value in those cases. An example for the range annotation of a THI is shown in Figure 6.

Updating the THI. Though business logic is not fast-changing, updates have to be supported by any hierarchical index in case part of the logic changes. For this problem, changes can affect either rules or objects (and thus the vertices of G_T). Assuming that rules can only be defined on existing objects, we differentiate between three update types: Objects may be removed, inserted, or modified. Object removal causes G_T to possibly contain empty nodes, i.e., nodes that do not correspond to at least one object. In that case, this node can be collapsed with its predecessor. Object insertion can be realized with only a few changes to the creation mechanism of the THI. Instead of creating tight ranges based on a node's children and single-value ranges for leaf nodes, the ranges can be artifically expanded whenever they are generated. For example if the tight node range of flight market Switzerland is [5,6], a looser range, for example [5,10], would provide space for more children nodes in the hierarchy. In practice, a range increase does not require any additional storage space but allows objects to be inserted with no additional effort as a new node is simply assigned to the overhead range. The third update type, object modification, can be handled by either modifying the node in place or removing and reinserting it into the index. The only costly update operation occurs when a buffer, i.e., a node range, overflows which then requires the index to be restructured. Given slow-changing rule sets such as the flight information datasets that were the inspiration of this work, allocating enough range space is sufficient for most use cases for which the THI is applied.

5.3 THI Variants

Next, we introduce two optimization techniques for the THI: The first one focuses on decreasing the memory allocated for storing the THI while the second one minimizes execution time further at the cost of storage.

ALGORITHM 2. Creation of BTHI from an existing THI

```
Input: Hierarchical tree structure THI
Output: Binary hierarchical tree structure BTHI

FUNCTION: TRANSFORMBINARY($v, b$)
$v.bits \leftarrow b$
$c \leftarrow \lceil log_2(|v.children|) \rceil$
$bit\ array\ b_i \leftarrow new\ bit\ array\ (size\ c)$
forall $v_i \in v.children$ do
    EXPLOREBINARY($v_i, b \oplus b_i$)
    $b_i.increase$
```

Memory-optimized THI. To optimize the THI, we can apply binary instead of integer range encoding. This variant of the THI is inspired by multibit tries that have been used to perform prefix-match look-up for IP address, [22]. The storage requirements are then dependent on the fan-out of the THI instead of the constant cost for two integers to represent the ranges. More specifically, the maximum number of bits needed to represent a key depends on the maximal path from root to leaf. A maximal path is a path where the sum of children per parent is maximal. The transformation of a THI into a binary tree-based hierarchy index BTHI is shown in Algorithm 2. Function TRANSFORMBINARY starts with the root node of the THI and dynamically computes the bit arrays representing each key as follows: First, it checks how many children the current node has. It then stores the upper bound of the required bits to assign a unique key to each one of them in variable c (Line 5). Aftwards, it generates a bit array of that size which is increased per recursive call for each child. Any child node is thus called with its corresponding binary value, a concatenation of its parent and its own unique new bit part. The advantage of a BTHI compared to THI is clearly that instead of reserving a fixed amount of memory, its representation is small if the fan-out per internal node is small. On the other hand, if the dataset is skewed, the number of bits required to represent the nodes is skewed as well.

Performance-optimized THI. The array tree-based hierarchical index ATHI is the complementary approach to the BTHI, optimizing the tree structure not for storage space but performance. The basic idea here is that instead of storing a tree, the hierarchical structure is flattened into an array. Per THI definition, every criterion value references at most δ values which allows for an exact prediction of storage space, i.e., the THI is flattened into a (different) hierarchy map where each value is mapped to exactly δ objects, one for each level of the index. Obviously, this structure requires more memory than either the THI or BTHI as memory is allocated even if no objects are referenced on that hierarchy level. Take as example market World that maps to exactly one market in the running example. As $\delta = 5$, five slots are allocated where only the first one is filled with a reference to the rules matching the world-wide flight market. At the same time it is intuitive that accessing an array is less time-consuming than accessing and traversing a tree.

5.4 Multi-Level Hierarchical Indexing

So far we discussed using an index on one criterion and potentially sharing an index between multiple criteria, annotating the rules per criterion with a different flag to enable correct rule retrieval. An alternative to this one-dimensional index is a multi-level hierarchical index (MHI), i.e., an index is first created on one hierarchical criterion and every node does not contain rules but another index specific based on the criterion value of this node. The trade-offs for this approach are obvious: Its advantage is that it is tailored to the criterion structures which means that there exist no nodes that are not associated with a rule corresponding to the indexed hierarchical criterion. Going back to Figure 6, a multi-level index on criteria 'Origin' and 'Destination' would only have 'Origin' rules in the first level of the index; once the matching nodes are identified, multiple subtrees, one per matching hierarchical node, are explored to find matching nodes for the 'Destination' criterion. Due to the layering of hierarchical indexes, the index structure needs to be accessed only once and all returned rules match all indexed criteria. The drawback of this approach is the overhead caused by storing subtrees for each node in each level of the multi-level tree except for the last one. In Section 7, we specifically examine the trade-off between storage space and performance for different rule sets to evaluate the effectiveness of layering hierarchical indexing structures.

6. RULE SELECTION

Rule retrieval as introduced in the previous section directly influences the performance of the rule selection component as it allows multiple rule subsets as output. Thus, it requires that the rule selection component incorporates a ranking mechanism that determines the top-k rules matching the query. In this section, we discuss how hierarchical values and user preferences may help to define the specificity of a rule formally which defines the rank of a rule. Furthermore, we describe how rule selection can be realized efficiently.

6.1 Rule Specificity

The importance of a rule with respect to others is influenced by two types of specificity:

1. The hierarchy position of value c_k^i (hierarchical specificity)
2. and user preferences concerning c_k (user-defined selectivity).

We will define these types of specificity in the following explicitly and show how they can be interleaved to construct the rule weight.

Hierarchical Selectivity. Hierarchical objects incorporate the idea of specificity implicitly as elements that are further up in the hierarchical structure are considered more general while objects with a deeper hierarchy level are more specific. In the running example, take rules r_1 and r_2. Both answer our initial example query for flight information from Berlin-Tegel to Zurich but as the destination of r_1 (TXL) is more specific than the 'Origin' criterion value of r_2 (Europe) while the markets in criterion 'Destination' have an equal hierarchy level, it is more applicable for this query.

DEFINITION 4 (HIERARCHICAL SELECTIVITY). *A rule criterion value c_k^i is more specific than value c_k^j if a successor relationship holds between the objects they reference in the underlying hierarchy, i.e., the corresponding nodes v_i and v_j in the hierarchy graph maintain a successor relationship such that $v_j \succ v_i$.*

Given the hierarchical index presented previously, HI, the relationship of these two objects can be extracted using set semantics again. Note that v_i and v_j need to be in a hierarchical relationship with each other because otherwise they will not both match the query. Using this relationship, rules associated with values that are deeper in the hierarchy obtain a higher value weight $w(c_k^i)$ for hierarchical value c_k^i of rule r_i. Wildcards for optional rule values as described in Section 3 can be seen as a specific case of a hierarchical value, they are the all-encompassing root nodes of a hierarchy tree.

User-Defined Selectivity. It is intuitive that some criteria are more important to the client than others because of the client's preferences and the intended semantic meaning of those criteria. In the running example, a flight is more distinctly defined through its 'Origin' and 'Destination' criteria than the 'Class' criterion as the geographical regions of a flight determine the length of the flight which usually has higher impact on the served food than the service class. As a result, score computation should allow users to specify the importance of a criterion through an attribute weight α_k for every criterion c_k.

DEFINITION 5 (USER-DEFINED SELECTIVITY). *The weight of criterion c_k in rule r_i is defined as the weight of its original value $w(c_k^i)$ and factor α_k such that $w^*(c_k^i) = w(c_k^i) \times \alpha_k$.*

Alternative to criterion-based rule weights, users may also decide to define a rule weight for the whole rule, thus artificially promoting this rule. Supporting this type of user-defined selectivity is intuitive. To compute the weight of a rule r_i, these types of selectivity can be combined in a simple cost model that assigns each rule a certain weight according to its specificity: The higher the weight, the more specific the rule. The weight ω of a rule r_i can be calculated as the sum of all criteria weights such that

$$\omega(r_i) = \sum_{c_k^i \in C} w^*(c_k^i)$$

.

EXAMPLE 6 (RULE SPECIFICITY). *Given c_1 (Origin), c_2 (Destination), and c_3 (Class), we define a higher weight for c_1 and c_2 than c_3, i.e., $\alpha_1 = \alpha_2 = 2$ and $\alpha_3 = 1$ in the running example. Using the subset relationships from the* HI, *the hierarchical score of value c_1^i (c_2^i) is the hierarchy level of the node v_i it maps to. Furthermore, the optional score of c_3 is either 0 or 1 depending on whether c_3^i is a wildcard or not. If a user requests information for a flight from Berlin-Tegel to Basel, the matching rules are r_1, r_3, and r_4 where $\omega(r_1) = \alpha_1 \times w^*(c_1^1) + \alpha_2 \times w^*(c_2) + \alpha_3 \times w^*(c_3^1) = 2 \times 5 + 2 \times 2 + 1 \times 1 = 15$. The weights for rules r_3 and r_4 can be computed analogously such that $\omega(r_3) = 14$ and $\omega(r_4) = 11$ which means that the engine would return r_1 to the user.*

Complex rule computation functions will not modify the gain achieved through structured rule weight calculations: The important observation here is that the applicability of a rule is reduced to a weight that the system uses to evaluate competing rules. Furthermore, we observe that the computation of rule weights can be done in a static manner which allows the system to be highly efficient during rule retrieval as we will explain in detail in Section 6.2. If every rule r_i is assigned a weight $\omega(r_i)$, the BRMS is immediately able to

1. identify which rule is more specific than alternative rules.
2. identify during an rule set update whether a rule can be added to the system conflict-free.

The second property is important in case strict rule weights are enforced by the system: If a query is issued that retrieves two rules with the same weight, top-k order with a strict number of k rules cannot be guaranteed. On the other hand, if the weight of a rule is directly computed when inserting the rule, the system can raise an alert in case of a weight collision. We therefore differentiate between *strict* and *fuzzy* top-k rule selection which differ only in the conflict consistency for rule weights. In the following, we assume a strict selection strategy for simplicity but all of the presented top-k selection strategies are valid for fuzzy selection as well.

6.2 Top-K Rule Filtering

If every rule is assigned a weight, top-k filtering is the task of selecting the k rules that have the highest weight and match the query. Remember that indexing does not guarantee retrieved rules

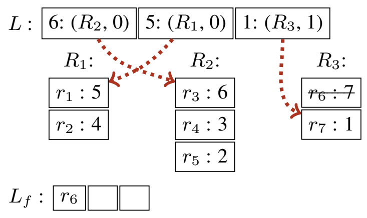

Figure 7: Example for lazy merge rule selection algorithm.

to match the query: It only guarantees that the indexed values are matched. Furthermore, the algorithm has to take multiple rule subsets $R_1, \ldots, R_m$ as input because the query value a_k^j can match multiple c_k^i if an ancestry relationship holds. We next describe the baseline algorithm for rule filtering in the Amadeus BRMS and then introduce a lazy merge technique for top-k filtering.

Sort-Merge Selection (SMS)**.** To unify different rule subsets, the straightforward solution is to employ a sort-merge algorithm. If rules are kept in the indexing structure in a sorted order already, it proceeds by merging all rule sets into one consolidated list. The top-k rules are then determined by parsing the merged list of rules from top to bottom, terminating if k matches have been found.

Lazy Merge Selection (LMS)**.** Lazy merge selection takes as input sorted rule subsets $R_1, \ldots, R_m$ and uses a pointer-based technique to determine the top-k rules. Instead of merging the rule subsets, it integrates top-k filtering into the merging process. Specifically, it processes the rule sets in parallel, picking the next rule according to the weight of the rules in descending order from a list that contains the top element of each rule subset. For merging the rule subsets, it first establishes an internal sorted list that contains at most one element of each rule subset, i.e., the current top rule of that specific rule subset. It then loops over this list, removing the first element and adding it to the final list L_f if it matches q_j. Otherwise, it is discarded. At the same time, the algorithm augments the empty space in L by adding the rule's successor to the list.

EXAMPLE 7 (LAZY MERGE). *Figure 7 shows an example where the* LMS *algorithm evaluates three rule subsets. Initially, its internal list L contains pointers to rules r_6, r_3, and r_1. If r_6 matches the current query, it is removed from L and its successor r_7 is added to L. As weight 1 is smaller than any other weight, it is pushed to the end of the queue, making r_3 the next rule to process.*

Strategy Discussion. Both of these strategies have their advantages and disadvantages. Obviously, SMS is inefficient a) if there exist a lot of rule subsets to merge and b) the top rules are found easily as they have a high weight and match the query term. On the other hand, LMS needs additional storage space and random accesses within the data structure as multiple rule sets need to be observed simultaneously. If the top rules are found late in the selection phase, our experiments show that a straightforward merging algorithm provides better performance (Section 7). Furthermore, SMS can be used in its current form for fuzzy rule selection. To allow multiple matches with the same weight, LMS is augmented to iterate over the current items of a rule set until all rules with the same weight have been processed.

7. EXPERIMENTS

In this section, we evaluate the different retrieval and selection techniques with three real-world and synthetic datasets. The evaluation is split into two parts, highlighting the advantages and disad-

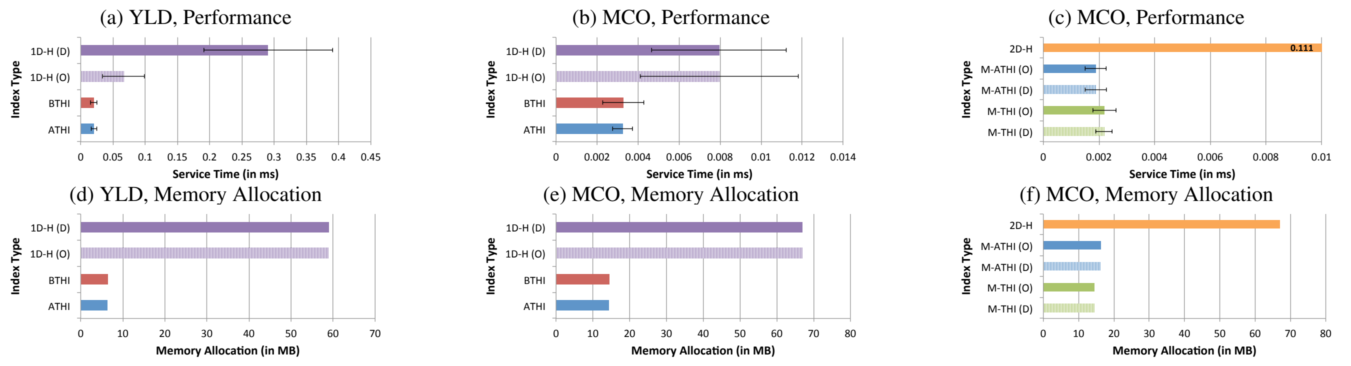

Figure 8: Comparison of (multi-dimensional) indexing structures for varying datasets.

vantages of the two components in business rule processing. First, the hierarchical indexing mechanisms are contrasted with existing hash-based techniques that are currently used by BRMS vendors. Afterwards, the proposed rule selection techniques are compared for different datasets to evaluate their robustness.

Datasets. Four different datasets are used in this evaluation, one synthetic dataset as well as three datasets obtained from the baseline BRMS. They contain a geographical hierarchy as described in this paper with a maximal hierarchy depth of $\delta = 5$ but they contain varying criteria and rule distribution characteristics (Table 3):

MCO This rule set contains 18,117 rules with two hierarchical criteria 'Origin' (O) and 'Destination' (D) and additional non-hierarchical criteria (flight numbers, dates, etc.). The hierarchy map for this rule set is described in Table 2 and contains 21,487 entries. Rules are distributed evenly across the nodes in the HI for both hierarchical criteria where approximately 6.5% of the rules can be found in the top and 60% in the lowest hierarchy level.

YLD This rule set has the same hierarchy map and rule set as MCO but has only two criteria, 'Origin' and 'Destination'. While all rule references for 'Origin' are on the third hierarchy level, they are found equally on the second or third hierachy level for 'Destination'.

GCA This rule set contains 2,594 rules that have three hierarchical criteria, adding a connection hub (C) as hierarchical criteria, as well as five non-hierarchical criteria. The hierarchy map used for this rule set is described in Table 2 and contains 21,520 markets. The majority of rule references are on the fourth hierarchy level for criteria 'Origin' and 'Destination'. For criterion 'Connection', about 62% of the rules are located on the first and the remaining 38% on the second hierarchy level (δ=2).

The rule weights of MCO and GCA vary according to the hierarchical level that the respective rule is associated with. In contrast, all rules in YLD have the same weight in our experimental evaluation.

Query Generation. To test the different approaches, the query workload per rule set is generated as the cross product of all distinct values of all hierarchical criteria. Only those queries are issued against the framework that have (multiple) matching rules. For the real-world datasets, we thus issue 502,681 (MCO), 578,483 (YLD), and 24,500 (GCA) queries to examine the performance of the tested indexes. Every query is run multiple times to ensure result robustness. The results documented in the following show the average, maximum, and minimum execution time of any query.

Table 3: Percentage of referenced rules for real-world datasets MCO, YLD, and GCA in their corresponding HI's.

Hierarchy Level	MCO		YLD		GCA		
	O	D	O	D	O	C	D
1	6.5	6.5	0	0	0	**62.1**	0
2	0.1	0.1	0	**53.6**	0.01	37.9	0.1
3	18.5	18.3	**100**	46.1	16.2	0	17.8
4	13.7	14.1	0	0.03	**70.8**	0	**68.9**
5	**60.3**	**60.2**	0	0	11.9	0	12.5

Indexing Techniques. For the evaluation of the hierarchical indexing techniques discussed in Section 5, three hierarchical indexes have been implemented:

THI	Range implementation of a tree-based hierarchy index.
BTHI	Variant of the THI using binary encoding.
ATHI	Array-based implementation of the THI.

Next to these single-dimension indexes, we test multi-level indexing as an alternative indexing technique marked by **M** in front of the index. The ordering of the levels of these indexes is indicated in the naming convention as follows: If the index type is **M-THI (OD)**, it is a multi-level THI where the first level criterion is 'Origin' and the second level criterion is 'Destination'. The baseline implementation is a combination of the hierarchy map and a hash index as described in Section 4. It is annotated with **H** and **1D** for a single dimension or **2D** and **3D** for indexes spanning multiple criteria.

Selection Techniques. For rule selection, both SMS and LMS have been implemented as described in Section 6.2. If not otherwise defined, LMS is selected by the processing engine as default rule selection mechanism.

All of these techniques have been implemented in Java. Furthermore, all experiments are executed on the same hardware, a Linux machine with two Intel Xeon L5520 processors and 24GB RAM.

7.1 Hierarchical Index Evaluation

In this section, the different hierarchical indexing approaches are compared to the hash indexing techniques that are currently implemented in our reference BRMS. First, the different approaches are evaluated for single-criterion indexes. Second, the indexes are layered thus making them multi-dimensional as described above.

Single-criterion indexing. For the YLD dataset, Figure 8 shows the performance and Figure 8d the memory allocation results of one-dimensional hash indexes compared to hierarchy indexes that

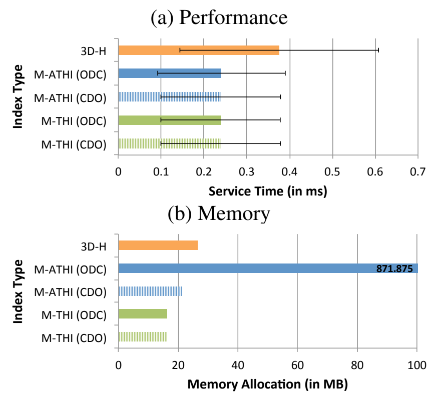

Figure 9: Multi-dimensional indexing for GCA dataset.

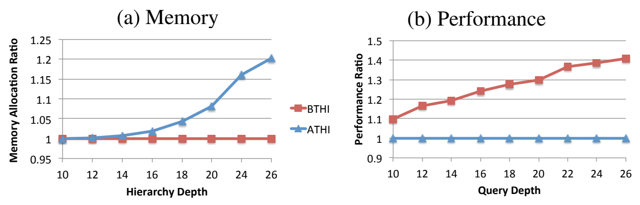

Figure 10: Memory allocation and performance comparison with varying hierarchy and query depth.

are composed of the hierarchical values in all available hierarchical criteria. Because the hierarchical index structure (a) can be used for two criteria and (b) the extensive baseline hierarchy map is replaced by a range map, memory requirements are reduced by at least 89.07%. Especially avoiding the use of a hierarchy map causes a large difference in memory cost: The large number of duplicate references can be efficiently pruned with the hierarchy indexing techniques while the whole map has to be stored to ensure correct processing for hash indexes. Furthermore, the performance of all hierarchy indexes is stable, varying within 0.005ms of the average execution time at most, while larger performance differences can be observed for 1D-H depending on the indexed criterion. Here, the performance varies as a hash index on 'Origin' profits more from the data skew in the rule distribution of YLD than a hash index on 'Destination': As the rules are deeper in the hierarchy for criterion 'Origin', they are more selective which increases overall performance comparatively.

If rules are more evenly distributed like in rule set MCO, the performance of one-dimensional hash indexes evens out as shown in Figure 8b. Furthermore, deep queries, i.e., queries that traverse the whole tree in depth, are comparatively more expensive for the THI-based approaches although they outperform any 1D-H index by at least 58.7%. The memory storage cost improvement per approach is similar in MCO as in YLD: Any hierarchy index requires at least 78.5% less memory than any of the hash indexes.

Multi-criterion indexing. Compared to single-criterion indexing, multi-criterion indexing directly incorporates multiple criteria into one index. They are realized for hash indexes by merging values to form a merged hash index. For multi-level indexing with hierarchical indexes, we follow the same approach as described in Section 5.4. For rule set MCO, we observe that multi-criterion indexing using any tree-based technique improves performance from an average of at least 0.0033 ms to 0.0022 ms as shown in Figure 8c. Similarly to the results shown in single-criterion indexing, ATHI outperforms THI slightly. In this specific setup, the performance improvement is at least 13.2%. Even the worst setup of THI is nevertheless 50 times faster than 2D-H. We furthermore observe that the access time of the merged hash index, 2D-H, is the average of its single-criterion counterparts which is reasonable as it does not fully benefit from the selectivity of 'Origin' but still maintains a better rule selectivity than 'Destination'. Note that it is possible but not efficient to implement BTHI for multi-level indexing which is why we omit its results from this evaluation because binary encoding is dependent on the (sub-) tree structure and cannot be pruned in the same way as a range encoding.

When comparing memory storage in this setup (Figure 8f), we observe that ATHI requires more storage space than THI. The reason is the fixed amount of memory that ATHI requires even in those criteria that form the later dimensions. For rule set GCA, the memory allocation cost as shown in Figure 9b is worse than in MCO for all approaches. The increased number of hierarchical criteria and their ordering especially influences the needed storage space of ATHI. For ATHI, we observe that it is best to have levels with shallow hierarchies like 'Connection' first to reduce the amount of duplicated (and empty but storage costly) internal array fields. Even in the best case scenario, ATHI needs 21.1 MB of memory while THI requires 16.2 MB in its worst case setup. Compared to the three-dimensional hash index, we observe that THI is always more memory-efficient than 3D-H. Furthermore, both hierarchical indexing techniques are at least 36% faster on average Figure 9a than 3D-H.

Synthetic datasets. To experiment with the trade-offs between ATHI and BTHI, we generated binary trees with varying hierarchy depths. In this set of experiments, we explore how increasing the hierarchy depth of the tree affects the storage cost of each of these approaches. Figure 10 shows the memory storage cost for this experiment where the applied metric is the memory allocation ratio of ATHI to BTHI. If the ratio is above 1, the memory allocated for ATHI is higher than the memory requirements for BTHI. As expected, this experiment shows that with an increase in hierarchy depth, the compression scheme of BTHI is more effective than the fixed space allocation scheme of ATHI which makes it the preferred solution for memory-conscious use cases.

Second, we evaluate the performance of ATHI compared to BTHI by varying the query depth. For example querying for value 'TXL' in our running example has a query depth of 5, while the query depth of 'World' is only 1. The results for this experiment are shown in Figure 10b where we observe the performance ratio of ATHI and BTHI. If the ratio is above 1, then BTHI is slower than ATHI. This graph shows the opposite development as observed for storage cost: The higher the query depth, the better the performance of ATHI compared to the performance of BTHI. For example for query level 26, we measured an average retrieval time of 0.001ms for ATHI and .0014ms for BTHI. As BTHI is implemented as a tree index, it takes additional processing time for traversing its internal structure. Thus, the longer the path that needs to be traversed, the more ATHI gains a comparative advantage.

7.2 Selection Evaluation

To evaluate the two selection processes introduced in Section 6.2, we describe results obtained when comparing SMS and LMS on rule sets YLD and MCO next. Note that the reported observations hold for all other rule sets that we used for this experimental evaluation as well. In this set of experiments, we compare the performance of the two rule selection strategies SMS and LMS. To evaluate their performance with new and old retrieval strategies, we chose

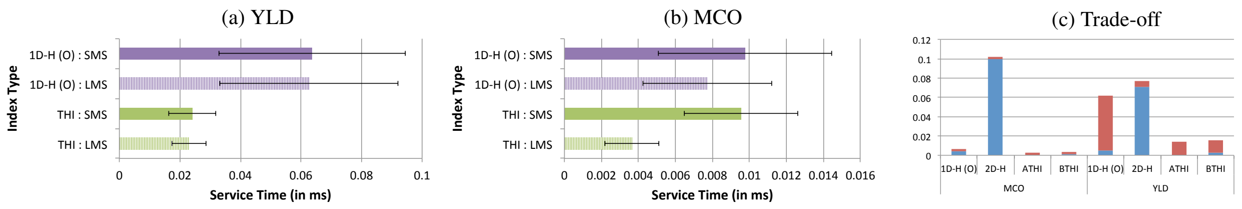

Figure 11: Rule selection trade-offs for varying datasets and in comparison to rule retrieval.

1D-H (O) (the better performing one-dimensional hash index) and THI as rule retrieval mechanisms. For YLD, we observe that using LMS has a stabilizing effect on the execution times but the observed average execution time is approximately the same for both selection techniques (Figure 11a). The reason for this behavior is that in this set of experiment, all rules in YLD are assigned the same weight to test the fuzzy matching capabilities of the selection strategy. As a result, the sorting is evenly expensive for SMS and LMS. In contrast, we observe an improvement when using LMS instead of SMS for the MCO dataset. Here, using LMS decreases the execution time of the rule selection process by 24.15% (1D-H (O)) resp. 69.34% (THI). The difference in the performance improvement can be attributed to the fact that THI stores multiple criteria in the hierarchical index if they use the same underlying hierarchy. Thus, it benefits from the performance improvement of LMS twice in this specific experiment.

To evaluate the importance of rule selection and rule retrieval in the larger context of query processing, Figure 11c shows the performance trade-off for varying indexing techniques and LMS as selection step. Comparing the baseline techniques, we observe that as indicated previously in Section 7.1, a one-dimensional hash index may outperform a multi-level hash index drastically. The second observation that we make is that rule selection in general dominates the time spent on rule retrieval for any THI variation. For these approaches, rule selection becomes the bottleneck as information retrieval is executed with one index access per stored criterion. Third, rule selection is more expensive in YLD than MCO, i.e., different dataset structures cause the trade-off between selection and retrieval to shift. In this specific case, YLD contains false positive matches that are discarded when filtering rules that match the whole query and not only the indexed values which makes this step more expensive in YLD.

8. CONLUSION

In this work, we introduced and addressed the problem of indexing hierarchical business logic. It might be a comparatively small part of a company's decision logic but it is difficult to handle with general indexing and data processing techniques as information accesses are hierarchy-oriented. As a result, retrieving hierarchical information can unnecessarily slow down query processing and thus decrease system performance. We therefore presented techniques to store client-specific hierarchies and adapted their intuitive graph structure to an efficient tree structure. Furthermore, we discussed and implemented two variations of the THI that either guarantee constant access time or improve memory consumption. Our work next showed how hierarchical knowledge can be used to define the specificity of a rule and how rule weights can be enforced during the update process which enables efficient rule selection. To handle multiple rule sets, we discussed different top-k selection mechanisms that suit hierarchical rule retrieval. All of our techniques have been evaluated extensively with real-world and synthetic datasets and their (dis-)advantages were explained in detail in an extensive experimental evaluation.

9. REFERENCES

[1] Bentley, J. L. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
[2] Calado, Pável and da Silva, Altigran Soares and Vieira, Rodrigo C. and Laender, Alberto H. F. and Ribeiro-Neto, Berthier A. Searching Web Databases by Structuring Keyword-based Queries. *CIKM*, pages 26–33, 2002.
[3] J. Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.
[4] Desislava Petkova and W. Bruce Croft and Yanlei Diao. Refining Keyword Queries for XML Retrieval by Combining Content and Structure. *ECIR*, pages 662–669, 2009.
[5] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *PODS*, pages 102–113, 2001.
[6] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *VLDB*, pages 436–445, 1997.
[7] Guo, Lin and Shao, Feng and Botev, Chavdar and Shanmugasundaram, Jayavel. XRANK: Ranked Keyword Search over XML Documents. *SIGMOD*, pages 16–27, 2003.
[8] B. V. Halle. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. John Wiley & Sons, Inc., 2001.
[9] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. *SIGMOD*, 19(2):271–280, May 1990.
[10] IBM Corporation. IBM Operational Decision Manager. *Whitepaper*, 2012.
[11] Jacobson, Guy and Krishnamurthy, Balachander and Srivastava, Divesh and Suciu, Dan. Focusing Search in Hierarchical Structures with Directory Sets. *CIKM*, pages 1–9, 1998.
[12] C. Kilger and G. Moerkotte. Indexing Multiple Sets. *VLDB*, pages 180–191, 1994.
[13] W. Kim, K. Kim, and A. G. Dale. Indexing Techniques for Object-Oriented Databases. *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, 1989.
[14] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index Structures for Structured Documents. *International Conference on Digital Libraries*, pages 91–99, 1996.
[15] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *VLDB*, pages 361–370, 2001.
[16] Liechti, Olivier and Sifer, Mark J. and Ichikawa, Tadao. Structured Graph Format: XML Metadata for Describing Web Site Structure. *WWW*, April 1998.
[17] C. C. Low, B. C. Ooi, and H. Lu. H-trees: A Dynamic Associative Search Index for OODB. *SIGMOD*, pages 134–143, 1992.
[18] Oracle. Oracle Business Rules. *Business Whitepaper*, 2005.
[19] T. Sellis and C.-C. Lin. A Geometric Approach to Indexing of Large Rule Bases. *Report No. UMIACS-TR-91-101*, 1991.
[20] Serge Abiteboul and Dallan Quass and Jason Mchugh and Jennifer Widom and Janet Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1:68–88, 1997.
[21] D. Shin, H. Jang, and H. Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. *Proceedings of the Third ACM Conference on Digital Libraries*, pages 235–243, 1998.
[22] G. Varghese. Prefix-Match Lookups. *Network Algorithmics*, pages 233–266, 2005.