Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 104

Systems Group, Department of Computer Science, ETH Zurich

Business Rules Retrieval and Processing

by
Alessandra Loro

Supervised by
Anja Gruenheid, Lucas Braun, Donald Kossmann

01.08.2013 - 01.02.2014

inf | Informatik
Computer Science

Contents

Abstract

Business Rules Engines are systems created to deliver relevant business information to applications and business processes that depend on them. Given their critical position, efficient indexing to minimize time and memory consumption is essential. Unfortunately, the usage of standard hashing indexing techniques has proved not to be an efficient solution to the problem, both in term of performance and memory consumption. In this thesis we propose an alternative hierarchical indexing strategy that stems from the analysis of domain-specific business data. We first explain how to represent the business data in a more efficient manner, then we suggest three different strategies to use this representation for indexing. To prove the efficacy of our technique, we have based our analysis on the Amadeus Business Rules Engine, and benchmarked our solution against a standard approach based on a merged index currently deployed in the company.

1 Introduction

Companies of all sizes nowadays rely heavily on internal information and business rules to govern and define the strategies and the processes necessary to obtain a good performance. Business rules can cover any work flow aspect inside any kind of firm: they can deal with work regulations and practices common to many industries, or bear information that is strictly related to the domain in which the company is operating. For example they may define what kind of credit should be applied to different types of clients, or what company department should be assigned to manage particular events. In this regard a good broad definition has been provided by the Business Rule Group [6], an organization with the goal of formulating statements and supporting standards about the nature and structure of business rules: *a business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business.*

An important characteristic of business rules is that they need to be always available, since business processes and applications heavily rely on them. As an example we can think about how the rules defining conditions for credit access are at the core of all the banking activities dealing with credit cards, mortgage requests, or money lending. For all of these sectors to work in a manner compliant with bank policies, their applications need to have constant access to business rules. Business Rules Engines (BREs) are software systems built and designed exactly to organize and store business rules. Given their critical position BREs needs are especially focused on:

- **Availability and correctness**. They need to ensure that for any given context in any given moment the best applicable rule(s) is(are) always returned.
- **Performance**. BREs should be efficient and not be a bottle neck for the system built on top of them.
- **Customisation**. The BREs should be data-agnostic and all business information should be easily internally updatable. A strict separation between the application code and the business logic is therefore absolutely

necessary. This generally require the creation and usage of a special ad-hoc language to formalize the rule definitions.

1.1 Problem description

In this work we are going to focus on a particular BRE, the Amadeus Business Rules Engine (ABR), that may serve as an inspiration to explain and evaluate the functioning of real engines currently deployed in industry. Amadeus is an international company operating as a transaction processor in the global travel and tourism industry [1]. The firm operates both as a connector between travel providers and travel agencies and as a provider of IT solutions for the field. The ABR is part of the solution framework maintained by Amadeus to support transaction processes on behalf of its clients, and it manages mainly business rules used by airlines or other transport service providers. Its main responsibility is to deliver appropriate answers regarding flights (or other transportation, but for the scope of this work we are going to assume to be working with flights) specifics and logistic to queries by applications using its services, usually software implementing procedures or visualizing information regarding the set up of the flights. As an example, the ABR engine may be asked by an application managing logistics: *What type of food is served on the flight LX16 from Zurich to New York, on November 28th 2013?*

To answer this query the engine needs to check against all the business rules defining policies for meals previously defined by the company and have detailed knowledge about information specific to the travel industry such as flights and geographical locations. Sometimes more than one rule may be applicable to the provided query, or rules may be underspecified compared to the provided rules. The ABR task is to address all of this issues and return to the above query a unique appropriate answer in line with the airline policies.

Flight Group	**Market origin**	**Market destination**	**Start Date**	**End Date**	**Meal**
LX01-20	Switzerland	Europe	*	*	*Drinks*
*	Switzerland	US	01.01.2013	31.12.2013	*Sandwiches*
*	Switzerland	New York	01.11.2013	03.01.2014	*Rösti*

Table 1: Sample Rules

Example 1.1.1. In Table 1 are shown few sample rules that may be used to answer the previously specified query. While the first rule can be excluded because the destination "*Europe*" is not a match for destination "*New York*", the second and the third rule are both applicable. In this situation, the system needs to break the tie by returning the most specific rule. The correct answer will therefore be given by the third rule: Rösti, because destination "*New York*" is more specific than destination "*US*". This is an example that shows how the system needs to be aware of geographical location: it needs to know that New York is in the US, and that a city is a more specific concept than a country.

1.2 Problem Statement

Unlike the previously presented small example, the ABR needs to store for each possible type of query thousands of rules, and to answer within a SLA of 30 ms to all the different applications located worldwide that store their business rules in the system. Performance is therefore especially crucial. In this work, we aim to :

1. Evaluate the current system and find its eventual weaknesses
2. Propose better solutions
3. Provide an implementation that shows how these solutions can improve the performance of the current system

1.2.1 Formal Definition

The ABR engine contains as many rule types as defined by the customers, but the algorithm used to evaluate the rules is universal. Since the contents of different rule types do not effect each other, different rule types can be considered as independent environments. The dataset for a rule type consists of a rule set R, containing several rules r_i, a set of criteria c_j specifying the rules' structure, and a domain D. Each rule $r_i \in R$ is defined by a set of mappings from criterion to attribute $c_j \rightarrow a_i^j \in r_i$ and a weight w_i.

In order to extract information, the rule engine can be queried through another collection of mappings $c_j \rightarrow q_j$. Each value q_j refers to a specific attribute a_j identified by the same criterion c_j, but neither the query nor the rules need to specify all the possible attributes. When checking if a rule is valid against a specific query, elements that are not defined by either the query or the rule are ignored. Correspondence $q_j \rightarrow a_j$ is usually direct, except for a few particular cases, such as domain specific query values, and SET and PAIR fields. Domain-specific query values (i.e. $Market$, $FlightGroups$ and $PointofSales$) need to be translated through the definitions contained in D, operation that normally results in $q_j \rightarrow \{q_{j1}, q_{j2}, ..., q_{jm}\}$. This means that after translation a single q_j is usually turned into a list of values. Elements $\{q_{j1}, q_{j2}, ..., q_{jm}\}$ for Market Criteria (MC) also include a second weighting criterion called lom.
Other particular cases are represented by SET fields, where a rule can define more values for a single a^j, and $PAIR$s made of dates and integers, fields that do not a single value but a range in the format $[a^{jstart}, a^{jend}]$. Pairs of other criteria, such as markets, can instead be treated logically as if the two elements were separately defined criteria.

To consider a rule r_i *matching* to the given query, it needs to fit the following criteria:

- For domain-specific query values, $a_i^j \in \{q_{j1}, q_{j2}, ..., q_{jm}\}$
- For SET elements, $q_j \in \{a_i^{j1}, a_i^{j2}, ..., a_i^{jm}\}$
- For $PAIR$s of ranges and integers, $q_j \in [a_i^{jstart}, a_i^{jend}]$
- For all the other types of elements, $q_j = a_i^j$

Among all the rules matching the previous specifications, a single rule r^* with the following characteristics is chosen:

- no other matching rule $r_i \in R$ has $w_i > w^*$
- if $\exists r_i \in R | w_i = w^*$, then: $\sum_{j=1}^{\#MC} lom_j^* < \sum_{j=1}^{\#MC} lom_j^i$. In case of rules with equal weights the chosen rule needs to have the smallest sum of *loms*.

1.2.2 Metrics

The different techniques examined and presented in this work are evaluated against three types of metrics that are considered relevant for deployment in Amadeus:

Execution Time Time it takes for the engine to return r^*. Time is measured as the interval between the moment all the values q_j are fed into the system and the choice of r^* over all the other candidates is finalized.

Memory Usage Space taken in-memory by the dataset. This includes all the necessary information to correctly answer the query, i.e. $space(R + D + index)$

CPU Usage What CPU percentage is used on average during the query execution. Execution is defined in the same way as for execution time.

2 Amadeus Business Rule Engine

In this section we describe more in detail how the ABR engine is made and how the currently deployed strategy for indexing works.

2.1 Rule Set

As previously mentioned, the ABR contains several different rule types that identify and define different rule sets. A rule type is identified by a code (a string) and a release (a number) and declares a collection of different criteria that can or must appear in its associated rule set. A *criterion* is a formal way to define the rules' expected content. Each criterion consists of: an identifying code, a boolean determining if it is mandatory for each rule, a supertag specifying what data type it contains, and its weight, the "importance" assigned to the specific criterion.

The supertag types include primitive and domain-specific elements. Both of them can be found in the rules on their own or grouped into compound elements such as sets and pairs. In Table 2 are shown all the possible elements with their associated supertag code.

Type	**Supertag**
Boolean	B
Date	D
Integer	I
String	S
Market	M
Flight Group	F
Point of Sale	P

(a) Simple Types

Type	**Supertag**
Set	E
Pair	A

(b) Compound Types

Table 2: Different data types with supertag codes

Criteria can be divided between optional and mandatory through the *isMandatory* boolean field. Mandatory fields need to be always specified both in the rules and in the query and carry $weight = 0$. This is because the presence of these fields is always granted and cannot be used to differentiate between rules. Optional fields instead (fields in which $isMandatory = false$) need to specify an appropriate weight to quantify the extra-specificity acquired by the rule if it defines them. The weight associated to each optional criterion is determined by companies when they define the rule types structure.

Example 2.1.1. Table 3 shows a potential definition for the example in Table 1.

code	**isMandatory**	**supertag**	**weight**
FG	false	F	2
MKT_PAIR	true	AMM	0
DATE_PAIR	false	ADD	4

Table 3: Criteria defined in an example rule type

Although a formal definition for the usage of different types of pairs is already provided in Section 1.2.1, this example shows now how it is applied. In this description both MKT_PAIR (combination of previous market origin and destination) and DATE_PAIR (combination of previous start and end date) are haven been coded as a pair, but they hold different meanings. When dealing with a pair of markets, the origin and the destination are two distinct values that need to be considered independently. In contrast, a pair of dates (or integers) are to be treated as a range. For example in Table 1, when querying for a flight on November 28$^{\text{th}}$, we considered the second and the third rule as valid matches because the date was included between the lower and upper bound, not because it perfectly matched any of the two numbers.

Finally, rules are stored in the ABR using an id, a weight and a collection of mappings from criterion to attribute value, $c_i \rightarrow a_i$. The weight for each rule is given by the sum of all the weights belonging to the specified optional criteria.

Example 2.1.2. Rules for our running example are represented in the system as shown in Table 4, using definitions from Table 3. The weight for rule 1 is given by the weight of criterion DATE_PAIR, while the weights for rules 2 and 3 are given by criterion FG.

Id	Weight	Values	
1	2	FG	1,2
		MKT_PAIR	404,501
2	4	MKT_PAIR	404,411
		DATE_PAIR	01.01.2013,31.12.2013
3	4	MKT_PAIR	404,211
		DATE_PAIR	01.01.2013,01.01.2014

Table 4: Rules

In the next section we explain how the definitions for markets and flight groups work. A description for point of sales is not provided, since its logic is the same as for markets and is thus considered redundant for the scope of this work.

2.1.1 Markets

Markets are represented in the current deployment as a dictionary, where each possible query value is translated into a list of elements, each one a pair of two numbers: a market id and a level-of-matching ($id : lom$). In figure 1 are shown example definitions for six markets.

Market ids are used to determine if a rule is a specific match for a query. For example if a query specifies for criterion c_i value *COUNTRY:US*, we can deduce from Figure 1 that valid rules are those having for c_i either value 601 or 411.

Loms are used instead to associate to the market definitions a weight related to specificity. The markets' *loms* represent a geographical hierarchy based on different area granularities. For ABR these are shown through the 6 layers defined in Table 5. Smaller *loms* are associated with more specific areas, while

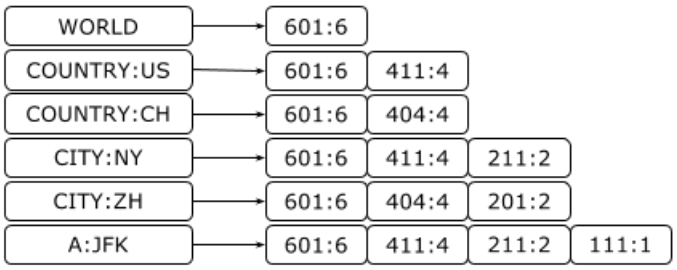


Figure 1: Markets

greater *loms* refer to more general zones. Since more specific area definitions are preferred, rules with market ids associated to lower *loms* are chosen over the others.

Area Type	**lom**
WORLD	6
REGION	5
COUNTRY	4
STATE	3
CITY	2
AIRPORT	1

Table 5: Lom specifications for area types

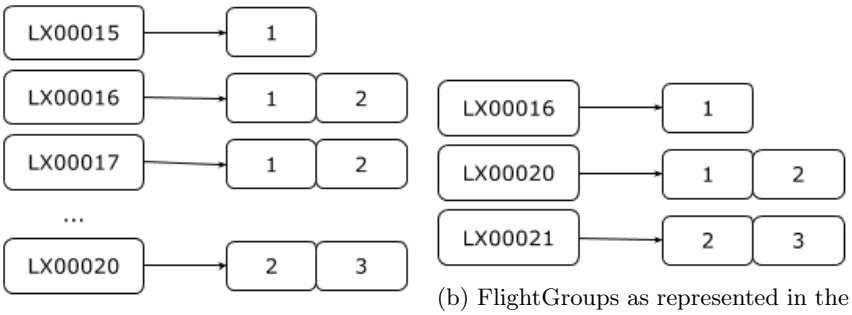
Example 2.1.3. To explain the functioning of markets, we can take again the query in Section 1.1, having market origin "Zurich" and as market destination New York, and match it against the rules in Table 4. Using the definitions provided in Figure 1, we have:

MKT_PAIR = [CITY:ZH → { 601:6, 404:4, 201:2 },
CITY:NY → { 601:6, 411:4, 211:2 }].

MKT_PAIR in the 2nd rule matches against CITY:ZH's 404:4 and CITY:NY's 411:4. With the same reasoning rule 3 matches against 404:4 and 211:2. Being both valid, we consider now the sum of the *loms*, that is 8 for rule 2 and 6 for rule 3. As 6 is smaller, the 3rd rule is returned as a result.

2.1.2 Flight Groups

Flight groups query values are resolved with a logic similar to the one used by markets but with some differences. Flight group data also consists of a translation from a query string to a set of numeric id as shown in the example in Figure 2a.



(a) FlightGroups direct translation

(b) FlightGroups as represented in the engine

Figure 2: Different FlightGroups representations

From the original dataset, it is easy to notice that many flight groups are defined by the same numerical ids. Therefore, Amadeus grouped all the ranges that appear both having equal numeric definitions and contiguous flight group numbers into the same entry, as shown in Figure 2b. This alternative representation has the same information content but it needs to be stored and queried in a specific manner. The dictionary keys need to be kept sorted (we are assuming ascending order for this implementation) and the key to be looked up for any flight group query value is the one that is next in the sorting order.

Example 2.1.4. To resolve our flight group query *LX00016* against the dictionary in Figure 2b, we need to look up the next strictly bigger stored value, i.e. *LX00020*. For the same logic all values from *LX00016* to *LX00019* refer to key *LX00020*.

This approach greatly reduces the number of definitions in our dictionary but it is not necessarily optimal, as it does not optimize for non-contiguous flight numbers having the same definition and it still does not eliminate most of the id replications between similar but not equal definitions.

2.2 Merged Index

The currently deployed version of the ABR engine mainly uses a merged index on its mandatory fields, or subsets of those. To obtain the index keys, all of the mandatory fields defined in the query except for ranges of integers or dates are hashed together in a single value.

Example 2.2.1. For the rule set defined in Table 3 its only mandatory criterion MKT_PAIR is used as an indexing field. However, market pairs are actually a compound field made of two distinct fields *origin* and *destination*, and the different markets are considered as separate values to be hashed to a single value using function:

$hash(origin, hash(destination, ""))$

In this example the rules happen to have all different market pair values, therefore they will be indexed to different keys as shown in Figure 3. This is an exception, we generally observe that there exist multiple rules with the same market pair that would then be hashed as list to the same value.

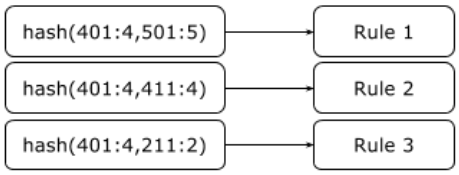


Figure 3: Merged Index

Although it does not happen frequently, rules can also indexed to more than one key, if one of their mandatory criteria is a set. In this case, the rule will be associated to as many hashes as the number of values the set field contains.

2.3 Query Processing and Index Retrieval

Before being able to directly access the index, query values need to be hashed using the same technique previously seen for indexing. However, before any hashing can be applied, query values for domain data need to go through a preprocessing phase where they are appropriately translated into numeric ids. These ids are generally more than one for each domain value, therefore we are not dealing with a single query anymore but with several queries that require multiple index accesses. Moreover, when pairs of domain criteria or more than a single domain criterion occurs in the rule type, all possible id combinations need to be checked against the index. The total number of such checks it the Cartesian product of all pair values.

Example 2.3.1. Figure 4 shows processing before index access for the initial query in Section 1.1. The number of hashes to be checked in the index is the number of market ids for the origin value times the number of market ids matching for destination value, i.e. 9.

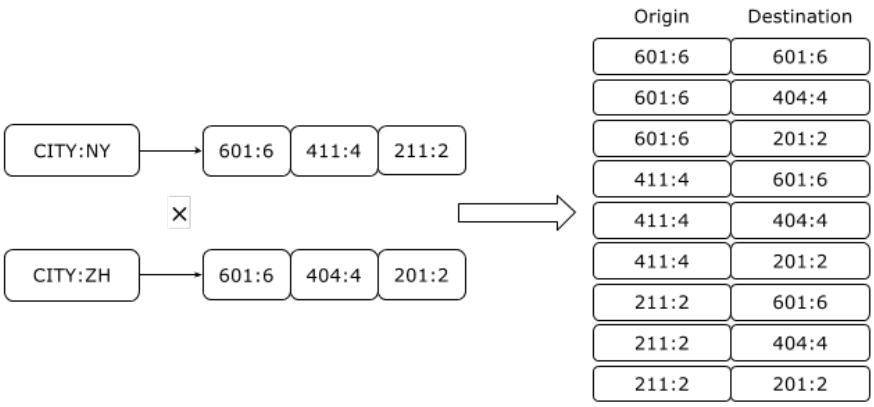


Figure 4: Market query processing for index retrieval

2.4 Rules Processing

After the rules have been retrieved through the index, the rules processing approach is simple: once retrieved, the rules are merged into a single list sorted

by weight in decreasing order and then checked one by one starting from top against all the criteria of the original query. When a first matching rule is found, the engine keeps parsing in the list all the other rules with the same weight before stopping. If more than one matching rule with the same weight is found, the *lom* coefficient for all market criteria is computed and the rule with the lowest *lom* is returned, as shown in Example 2.1.3.

2.5 Considerations

The current implementation is considered quite effective for regular primitive criteria with a high selectivity but shows its weakness when there are multiple domain fields or non-selective mandatory fields, as it can happen that mandatory criteria hold very few different values and offer no significant help in reducing the size of the rules to be checked. When multiple domain fields occur as in our example, the number of index accesses increases since whenever a query is issued the Cartesian product needs to be compiled. All the different retrieved rules need afterwards to be merged together maintaining order before any processing can be performed, adding a potentially expensive sorting phase to the algorithm. The lacking performance of the merged index for domain criteria is one the main motivations behind the development of our new hierarchical index in the following sections that tackles the performance disadvantage described above.

3 Hierarchical Index

Hierarchical indexing focuses on performance improvement for indexes based on the domain criteria. In our specific case we have chosen to pay particular attention to these fields because:

1. They are often specified as mandatory criteria in the rule type definition.
2. They have high selectivity, increasing therefore the chance to be considered as index fields independently from their optionality.
3. Their performance is critical for the currently deployed merged index, both for execution time and memory usage. In particular, the market definitions occupy several times more space in memory than the remaining rule set.

Hierarchical indexing leverages specific hierarchies implicitly defined in the market and point of sales domains. Its potential benefits can be described as follows:

Reduce memory consumption - By getting rid of the redundancy currently used to store the market definitions. For example, Figure 1 showed how most of the definitions kept repeating shared ids, which could be stored only once.

Speed up the index retrieval phase - By avoiding multiple accesses to the index for domain-defined criteria. This is especially important since markets mostly appear in pairs, and therefore the number of index accesses increases quadratically.

We will first illustrate the new logical representation using markets as an example, followed by an explanation of different strategies for hierarchy detection and implementation in Section 3.2. We will then present three alternative index strategies. Finally, in Section 3.6, we will discuss a strategy to improve the rule processing phase, i.e. how to return efficiently the best matching rule after preliminary results have been retrieved from the index.

3.1 Markets Representation

The main concept behind the hierarchical index is to identify hierarchies hidden into domain specific criteria and use them for indexing. In out case the query values defined in the market domain can be naturally interpreted as elements of a geographical hierarchy. Each layer of this hierarchy represents a geographical granularity and is identified by a *lom* as described previously. We can take for example Table 5, where WORLD and market ids with $lom = 6$ are the root of this tree, REGION and market ids with $lom = 5$ are its direct children and so on, following the rule that a region with a bigger *lom* is a parent for regions with the next smaller *lom*. Any market query value can only be described by market ids associated with a *lom* equal or bigger than the one defined for their prefix, therefore it results naturally that definitions for smaller *loms* can be placed at the lower levels of the tree because they introduce specificity.

Example 3.1.1. As the *lom* for *COUNTRY* is 4, the query *COUNTRY:US* can only be defined by market ids having lom ≥ 4, i.e. belonging to *REGION* and *WORLD*. This follows the logic that if we are seeking for a rule valid for

the whole country USA, the rules specified for North America or worldwide are also valid, but the ones defined for more specific American areas, such as San Francisco or New York, are not. We have therefore that *WORLD* is a shared ancestor for all nodes, while *COUNTRY:US* is a shared ancestor for all of the US subregions, either states, countries or airports. By identifying *CITY:NY* as the child of *COUNTRY:US* and *WORLD*, we can omit from it the market ids that are already specified by its parents, i.e. all of the market ids with $lom \neq 2$, *CITY*'s lom.

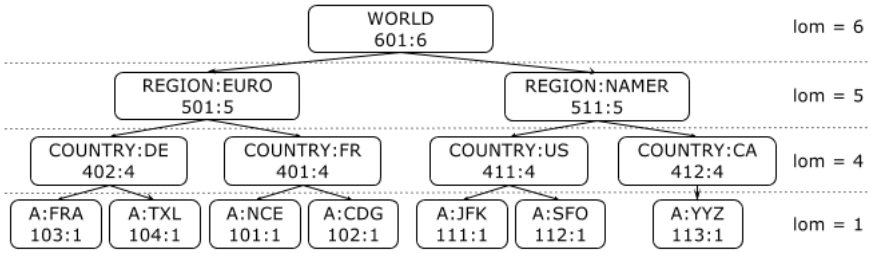
Figure 5 shows a translation from the standard dictionary model to the hierarchical model. For simplicity, we have considered only four layers out of the total six used in the original dataset, omitting CITY and STATE. This kind of simplification will also occur in future examples, but it does not involve any loss of information for the functioning of the indexing technique. The same definitions contained in Table 5a can be obtained by navigating the tree shown in Figure 5b.

Example 3.1.2. To reach node *A:CDG* we have to cross nodes *WORLD, REGION:EURO* and *COUNTRY:FR*, collecting meanwhile all the market id values that are part of its original definition.

The information content of the two representations is therefore the same. However, it is immediate by looking at the tree representation that it has eliminated all the redundant definitions belonging to the original table. Each market id appears in the tree only once in the layer associated with its lom.

Queries	**Market ids**	**Queries**	**Market ids**
WORLD	601:6	A:FRA	601:6, 501:5, 402:4, 103:1
REGION:EURO	601:6, 501:5	A:TXL	601:6, 501:5, 402:4, 104:1
REGION:NAMER	601:6, 511:5	A:NCE	601:6, 501:5, 401:4, 101:1
COUNTRY:DE	601:6, 501:5, 402:4	A:CDG	601:6, 501:5, 401:4, 102:1
COUNTRY:FR	601:6, 501:5, 401:4	A:JFK	601:6, 511:5, 411:4, 111:1
COUNTRY:US	601:6, 511:5, 411:4	A:SFO	601:6, 511:5, 411:4, 112:1
COUNTRY:CA	601:6, 511:5, 412:4	A:YYZ	601:6, 511:5, 412:4, 113:1

(a) Market Definitions



(b) Market Hierarchy

Figure 5: Ideal Market Definitions to Market Hierarchy

Given our sample data this representation can lead to consistent memory benefits, as it removes all the market id redundancy presented int the original definitions. It is also especially effective for all the zones that share an identical

definition. In the real data a lot of query values are in fact not considered important enough to be associated with their own market ids and share a common definition with other queries. In the original representation all of this information is stored independently and therefore multiple times, while with this technique we just need to add another query value in the appropriate node. For example, if another airport in Canada has the same definition as *A:YYZ*, the key value for the new airport value is simply placed in the same node.

3.2 Hierarchy detection

The data we received from Amadeus regarding markets consists only of a list of definitions in the form $query \rightarrow \{id_1, id_2, ..., id_n\}$, as the ones shown in Table 5a. Therefore the hierarchy described in Table 5b needs to be completely derived from the actual data set. Unfortunately our initial idea of deriving a perfectly shaped tree based on pure geography turned out to be wrong. ABR gives clients the possibility to define very flexibly their own market zones, including or excluding geographical sub-areas as they please without imposing mutual exclusivity between zones within the same lom layer. This obviously leads to the definition of overlapping areas.

Example 3.2.1. Clients may feel the need to define rules for both a region Europe including Switzerland and one excluding it. This does not imply that we are also interested in having a term to query both regions, but just that for correctness there has to be a way to specify rules for both cases. Table 6 shows a way the ABR would handle this situation. While a comprehensive *REGION:EURO* is explicitly presented in the dictionary, the new region "Europe minus Switzerland" is defined implicitly through market id *502:5*. This new market id will occur in all of the European sub-zones but Switzerland.

Query	**Market Ids**
REGION:EURO	501:5
COUNTRY:FR	501:5, 502:5, 401:4
COUNTRY:DE	501:5, 502:5, 402:4
COUNTRY:CH	501:5, 404:4

Table 6: Example of Hierarchical Non-Idealities

We next describe two different approaches to deal with the overlapping zones in the hierarchy.

3.2.1 Directed Acyclic Graph

A first idea we explored was using a Direct Acyclic Graph (DAG) to keep track of anomalies. The main reason behind this approach was to keep avoiding completely redundant market id definitions and ensuring that any id appears in the DAG exactly once. We tried therefore to identify what was the minimum number of nodes we could use to avoid replication and create a correct representation of the market data.

Example 3.2.2. Figure 6 shows a DAG representation for the data in Table 6. In this very simple case we can notice how *COUNTRY:FR* and *COUNTRY:DE*

are both reached by two different paths: one going through market id *501:5* and the other through *501:5*. To be able to reconstruct their original definition, both paths need to be explored.

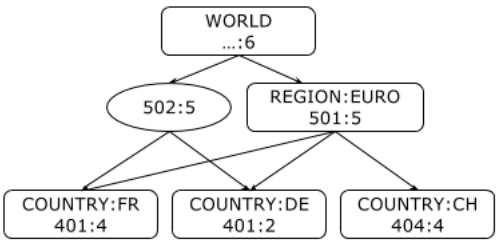


Figure 6: DAG representation for hierarchical anomalies

This approach was used in our first experiments and abandoned after experimental evaluation due to its lacking performance and excessive memory consumption. Given the amount of personalization the users are allowed, the ABR data is complex and it can include within the same layer a lot of different market ids combinations. This leads to the creation of many separate nodes on the same level that need to be aggregated according to the specific query. The DAG representing the dataset in this form has therefore many connections to be stored and each query results in a collection of multi-level paths to be checked. In fact, it is important to notice how in a bigger and more complicated dataset each DAG node is addressed via a multi-path, the number of which increasing exponentially with the size of the graph. In practice, this greatly slows down the potential performance of this approach, rendering the number of nodes to be crossed often directly comparable to the number of ids defining each query value. In the original dataset this number is in most cases at least more than twice the levels represented by the *Level-Of-Matching* attribute. This observation led us to develop the next technique that aims to use the number of *lom* as an upper bound for the number of nodes to be checked.

3.2.2 Redundant Tree

The Redundant Tree representation is a technique for storing market data that makes use of the meta extra information provided by *loms* in an effective manner. Through the *lom* information we can divide market ids in six mutually exclusive sets, representing six separate layers of our market tree. For each *lom* layer, we create a node for each for each of the possible market id combinations that appear in the query definitions with that specific *lom*. In this way, we introduce redundancy in market nodes, but we reduce the number of connections between nodes and reduce the number of nodes to be examined for each query to a maximum of six, all of them on the same path.

Example 3.2.3. In Figure 7, we report the tree version for Table 6. We can see that our data structure is once again a tree divided strictly in layers. Layer $lom = 5$ shows the way we have introduced replication to minimize the number of connections between nodes. Market ids for this level appear in the definitions in two possible combinations: one made just by {501:5} and the other made by {501:5,502:5}. Both combinations are associated with a different node that takes

care of a different path. All the nodes including market-definition *502:5* are now attached to a different region node than the ones that are not. Compared to the previous approach, this technique has introduced replication for *501:5*, but simplified query for *COUNTRY:FR* and *COUNTRY:DE*.

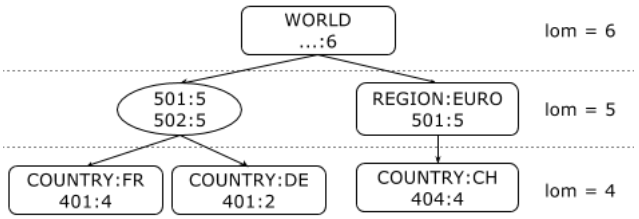


Figure 7: Redundant tree representation of hierarchical anomalies

We now discuss the advantages of this approach. First, each query is defined by a single path in the graph. Secondly, the maximum number of levels to be crossed in the tree is well-defined: six for any given data with the current *lom* definitions used by Amadeus.

We note that obviously this technique has a drawback: it introduces redundancy for the market ids. However, in the index the market ids will not be stored directly. The only information to be stored in each node are pointers to the rules indexed to that node's market id. Therefore by introducing redundancy for market ids we only replicate pointers to rules, as explained in Section 3.2.3. In practice, the size of these pointers is still much smaller than the size needed by the DAG to keep track of all the extra connections used. All strategies for hierarchical indexing discussed next, will therefore use this market representation without any further explanation. For clarity and simplicity each node will be represented in the example pictures by its key value, even if we are aware that some nodes may not possess one or have more than one. This is not relevant from a logical point of view since we store the query values separately from the tree.

3.2.3 Rule Indexing

Now that we have defined a strategy to create a tree hierarchy based on markets, we need to adapt our data structure to be used as an index. All the information currently shown in the nodes, such as query values and market ids is useful for building the hierarchy and understanding how it works but is unnecessary in the working index and will not be kept in memory once the building phase is finished. Each node will contain instead only the following information:

- Pointers to the rules matching the market ids representing it, sorted by weight in decreasing order.
- Pointer to its children. The logic for navigating the tree will be described in Sections 3.3 and 3.4.

It is important to notice that all the rules are stored only once in the system and accessed by reference from the index. Therefore having redundant ids simply mean to have multiple pointers pointing to the same rules from different

nodes of the same tree level. Each node contains as many buckets as indexed market criteria and keeps track of the rules that contain its associated value in the correspondent bucket. For example, in a tree used to index a market pair each node contains two buckets: one for origin and one for destination. Each bucket then points to a separate list of rules sorted decreasingly by weight.

3.3 Range Index

Now that we have defined how we are going to index the rules using a tree, we need an efficient way to navigate it. Given an arbitrary market query, we need to know exactly what path it should follow in the hierarchy. We next present our initial proposition, a range index.

3.3.1 Definition

Range Indexes are a well-known technique introduced by Bentley [2] to efficiently perform search queries for ranges in space coordinates. Originally, range indexes are usually represented as trees. More specifically, they are *balanced binary search trees where the data points are stored in the leaf nodes and the leaf nodes are linked in sorted order by use of a doubly linked list* [9]. However, the implementation of this data structure has to be adapted to the needs of our index. In our hierarchy, we are not interested in reaching all the leaf nodes belonging to the range (there should be just one), but we need to collect specific information from the nodes we use to reach our destination. Moreover, our market hierarchy is most definitely not a binary tree. We need therefore some consistent changes in how we address the ranges in the data structure.

3.3.2 Building the Index

The first required step towards the construction of a range tree is to associate ranges to all the tree nodes. Ranges are associated recursively starting from the tree root: the range associated with any given node is given by the cumulative ranges of all of its children plus a chosen offset α. If the node does not have any children, its range is given by a single number and does not include any offset. Algorithm 1 shows pseudo-code for range assignment. Range assignment is not limited to just the nodes of the tree but it involves also the query values. A separate dictionary $Query \rightarrow Range$ is then also necessary to be able to translate a query into useful directions to navigate the tree. Each query value is associated to the same range as the node that represents it in the tree.

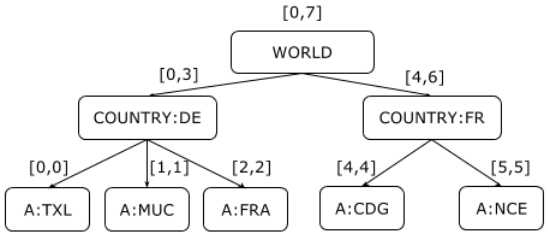
Algorithm 1 Algorithm to assign range values to hierarchy

```
function assignRange(Node current, int start, int offset )
    current.start = start
    if !current.hasChildren() then
        current.end = end
    else
        for all Node child ∈ current.children() do
            assignRange(child, start, offset)
            childEnd = child.end
        end for
        current.end = childEnd + offset
    end if
end function
```

3.3.3 Querying the Index

The execution of each query includes a look-up in the query-to-range dictionary and a tree exploration. First of all, the query is translated into a range made of two numbers that is necessary to navigate the tree. The dictionary can be implemented as a hash table and therefore be queried in constant time. The second step is to cross the tree looking for a specific range. For each node we traverse, we collect all the rules that have been indexed to that specific node. The node to be parsed next is the node whose range includes our query range. If no node matches this criterion, the tree parsing stops. When we deal with multiple indexed criteria, we can still parse the tree a single time to avoid as much as possible to visit the same nodes multiple times. We can parse only once all the nodes that the different fields have in common and split only when the paths diverge.

Example 3.3.1. Both parts of an example range index with offset $\alpha = 1$ are shown in Figure 8. Imagining a query A:CDG, the first thing to do is to look up its range in the dictionary in Figure 8b. Then we can proceed to explore the tree in Figure 8a starting from the top, seeking for all the nodes that include [4,4]. Our query will therefore return rules from node WORLD [0,7], COUNTRY:FR[4,6] and A:CDG[4,4]. In case the query is actually a market pair (*A:CDG,A:NCE*) the algorithm goes only once through nodes WORLD and COUNTRY:FR, splitting paths only for the last step. The total number of nodes nodes to be parsed would be therefore 4 instead of 6.



(a) Range Tree

Query Value	Range
WWORLD	0,7
COUNTRY:DE	0,3
COUNTRY:FR	4,6
A:TXL	0,0
A:MUC	1,1
A:FRA	2,2
A:CDG	4,4
A:NCE	5,5

(b) Query Range Dictionary

Figure 8: Range Index representation

3.3.4 Considerations

Compared to the original index criteria, the range index does not suffer from multiple index accesses depending on the number of market id defined for the query. On the contrary, the number of nodes to be accessed is bound by the number of layers of the tree. The range index scales also remarkably well as we increase the number of indexed criteria. The number of node accesses for each extra criterion added increases sublinearly, while in the current Amadeus index the number of index accesses increase quadratically.

The memory footprint for range storage scales with the number of leaves contained in the tree. The size of the ranges to be stored is in fact proportional to the number of leaves plus an offset based on the granularity of the fan out.

3.4 Multibit Trie

A second technique to explore the hierarchical tree was inspired and adapted from routing algorithms.

3.4.1 Routing Address Look-Up

Address look-ups play an important part when routing IP packets through the Internet network. It is impossible to store all the network information in every router. Therefore, in order to allow each router to keep information only about his neighbouring nodes, IP addresses are defined with hierarchical semantics. As a result, each hub of the network contains only a reduced set of rules matching prefixes of the packet IP addresses. The chosen rule for forwarding the packet will be the rule that matches the specific packet address best. Algorithms for prefix-match lookup used for this purpose,focus on how to store and compare IP address to minimize efficiently the number of bit comparison to effectuate for each look-up. As routers are often cheap resources with low memory and processing power, these algorithms put a strong emphasis on maximizing performance while minimizing memory requirements. Modern approaches to tackle this problem are proposed in [5].

3.4.2 Similarities

Our hierarchical indexing approach has many similarities to the routing look-up issue. In both cases, we have hierarchical information that we need to parse

until we get to the most specific match. The geographical parallelism within routing can also be transported in our market domain: we could see our queries in term of specific addresses.

Example 3.4.1. Germany can be seen as an hypothetical address 0, opposed to France with prefix 1. The airports situated in Germany will have therefore prefix 0, while the ones in France will have prefix 1. We add more suffixes to the address as we increase the geographical specificity (determined in out domain by the *lom* attributes).

Nevertheless, it is important to notice a goal difference between our index and the routing address look-up: for the routing look up the path taken is not relevant, because the goal is to find a single "best"rule. As for the Range Index instead, the goal of this tree exploration is not to find the most specific node itself, but the path to get there, since we need to collect all the possible matching rules.

3.4.3 Strategy and addressing

In the context of routing, Varghese [5] presents a data structure that can be easily fitted to our market hierarchy: the Multibit Trie. A trie is *a tree-based data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching* [8].
Onebit tries are simply binary trees where each branch is represented by a binary cipher, while multibit tries allow for nodes to have multiple branches represented by binary addresses with lengths ≥ 1.

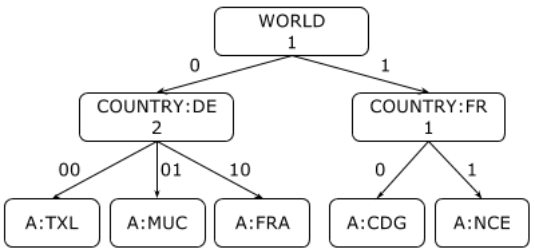
The first step then is to determine how to address our tree nodes to fit the trie. Multibit trie data structures come in two different flavours: fixed-stride and variable-stride tries [5]. With fixed-stried tries all children are addressed with the same amount of bits, while variable-stride tries take advantage of variable-length prefixes to make less comparisons and potentially save bits of address space. The variable-strides option however needs to store not only addresses, but also the size of the used stride for each node. For our market index we chose a variable-stride implementation, where each nodes compute the size of the suffix of its children based on its fanout.

As the only difference is given by addressing, the querying strategy looks very similar to the previous range index. It consists of two phases:

1. **Query to binary address look-up**. In the same way as for the range index, a hash table is used to retrieve the binary address associated with the query.
2. **Trie traversing**. The tree is is parsed top down, each time choosing the node that matches the address prefix. Each node keeps track of how many bits are necessary to identify its children, and once the decision of which node to follow has been taken, it shifts left the address with that number before passing it on to the child.

Example 3.4.2. In Figure 9 we can see the multibit trie approach on the same data used before for the range tree in Figure 8. Binary address are expressed in the format *address/#relevant bits*. The size of the relevant bits is stored once in each parent node and it is common for all of its children. Taking the market pair (A:TXL,A:FRA) as an example, we are going to parse the tree in Figure 9a

for 000/3 and 010/2, as defined in Table 9b. After the node WORLD our query takes follows the node COUNTRY:DE for both addresses and drops the part of the prefix just analysed, thus obtaining 00/2 and 10/2. For the final steps the queries split path and match against the respective addresses.



(a) Multibit Trie

Query	Address
WWORLD	0/0
COUNTRY:DE	0/1
COUNTRY:FR	1/1
A:TXL	000/3
A:MUC	001/3
A:FRA	010/3
A:CDG	10/2
A:NCE	11/2

(b) Query to binary address dictionary

Figure 9: Multibit Trie index

3.4.4 Considerations

The multibit trie presents the same feature for memory usage as the range tree regarding the number of nodes and connections. It differs for the size of the addresses that are stored in the query dictionary and in each node. The multibit trie can offer advantages in memory consumption, as it requires less space to store a single address. Any link in the range tree is represented by two integers representing a range, while the multibit trie technically only need few bits. Even assuming the use of integers to store bits in the actual implementation, an integer for each connection would still allow the multibit trie to have 2^k nodes as fan out in the worst case, where k = *number of bits for int.* To have such a fan out would mean for the range tree to have at least that amount of leaves, and it represents the number of leaves normally supported by simple integer range. The range tree would use therefore, even for the multibit trie's worst case, twice as much memory, since it needs to store both lower and upper bounds. As a disadvantage instead, in practice bit manipulation can be trickier and more expensive that range manipulation, especially for bigger fan outs.

3.5 Array Index

Finally, we propose one the last strategy that proposes a different way to address hierarchical data. It is still based on the market representation introduced in Section 3.1, but it stores the nodes of the market hierarchy in arrays instead of a tree. For our specific dataset we have seen that the market hierarchy is always given by a fixed number of layers determined by the *lom* definitions, six in our case. Therefore we can see each query as a collection of six nodes, each one of them representing one of the six *lom* layers. Not all the queries define all the six *lom* nodes (e.g. queries for COUNTRY values only have a node for *lom* six, five, and four), but the size of the collection is bound above by the maximum value of the *loms*. The most specific queries, such as airports, may instead specify all of the nodes. Therefore, we do not necessarily need to parse a hierarchy for each

query, but we just need a fast and simple access to the collection of nodes the query refers to. The array index takes implements this insight by substituting the tree data structure with an array the size of $max(lom)$ for each query value.

3.5.1 Data representation

The array index consists of an hash table that associates each possible query with an array of the size of $max(lom)$. Each element of the array represents a lom level and contains a pointer to the node where all the rules for its market ids are indexed.

Example 3.5.1. The array index in Figure 10 uses the same query definitions previously shown with the other indexes. Being a simplified example, the number of levels we are considering is three and each query is made of an array of size 3, one bucket for each of lom 6, 4, and 1. The elements of the array point to external nodes where the rules are indexed and represented for simplification by the market ids. We can see that for example A:TXL contains three pointers: one to the node where the rules containing 601:6 are indexed, one for 402:4 (market id generally referring to COUNTRY:DE), and one for 104:1 that is its specific airport code.

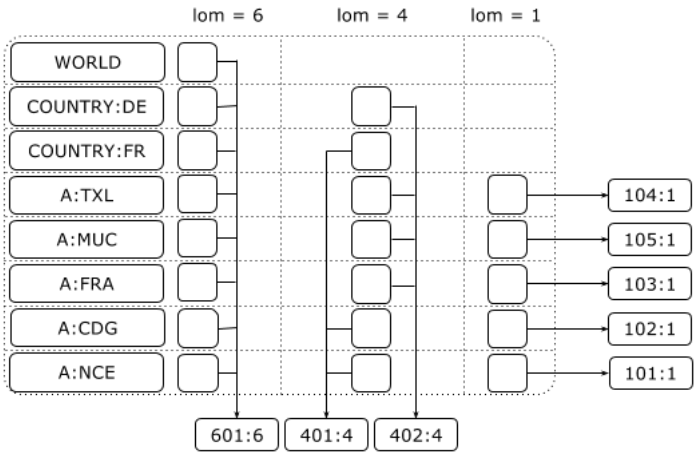


Figure 10: Array Index

3.5.2 Index Querying

Contrary to the previously tree-based indexes, the Index Array can be queried directly without needing any dictionary. For each value of the query the hash table $Query \rightarrow Array$ is accessed and all the rules pointed by its elements are retrieved.

3.5.3 Considerations

The array index presents performance advantages on the other indexes concerning: not having a 2-step approach (dictionary and tree parsing) and needing a single access to hash table to obtain all the rules, without needing to traverse any tree and be concerned about fan out performance. However, this approach

scales worse with the increment of the number of criteria to be indexed. It does not matter in this case if the queries values for the different attributes share part of the definitions or need to access the same nodes. This approach will keep the two queries strictly separate, scaling linearly with the number of criteria. Memory-wise, this approach will save space for datasets with a low number of *loms*, like ours. As usual the general space required for nodes or links is the same as before, but the extra-space for addressing is computed differently. For the array index the space of addressing strictly depends on the maximum *lom* and can be resumed as $space = (max(lom) * sizeof(pointer)) * queries$. Therefore, the memory performance scales with the depth of the market hierarchy.

3.6 Rule Processing

In all of the previous sections, the index development focused on the retrieval of rules from index, excluding the rule processing and matching phase. This part of the engine is the same for all of the indexes and is very similar to the strategy already implemented by Amadeus and explained in Section 2.4. For all of the newly introduced indexes the primary rule processing approach consists of the following phases:

1. The rules are retrieved in sorted buckets from the nodes of the hierarchical representation.
2. For each criterion retrieved, the rules merged into a single sorted list
3. In case of multiple indexed criteria the intersection between the lists is computed and only rules found for all the criteria are kept.
4. The list is parse top down (starting from the rules with the highest weight) and checked against all the query criteria until a match is found.

An alternative strategy just for rule processing is however proposed in Section 3.6.1.

3.6.1 Bucket Rule Processing

A bucket sorting rule processing approach consists in avoiding aggressive sorting (taking $\Theta(N)$ in the number of rules) and lazily process all the rules from the different buckets. Instead of getting sorted immediately, all the retrieved buckets are examined at the same time, iterating over all the rules with the highest weight to check if they include a possible match. Once a matching rule is found, the process is interrupted, before looking and ordering the remaining rules.

Example 3.6.1. We consider the familiar case where a market pair has been queried and we have therefore retrieved rules for two criteria (the origin and the destination). In this case two buckets for each criterion have been retrieved. In Figure 11 the buckets contain rules in the format *rule id, weight* and are sorted top down by weight. The bucket rule processing starts from the top and processes all of the rules with $weight = 8$, computing the intersection between the rules retrieved for origin destination. Since no rule matches both criteria, all rules with $weight = 8$ are removed from the buckets, and rules with $weight = 4$ are processed next. In this case rule *R2* is found to be matching both criteria and evaluated against all the query criteria. If *R2* is match then the processing

is over, otherwise we continue with rules with $weight = 2$ and so on, until a match is found or the buckets have been emptied.

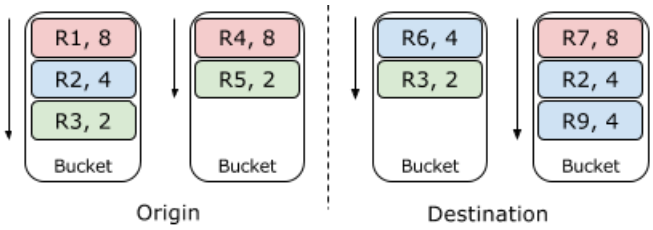


Figure 11: Bucket Sorting with two different criteria

4 Experimental evaluation

Experimental evaluation is made on a synthetic data set provided by Amadeus for testing. Amadeus gave us seven rule types for this purpose featuring different combinations of criteria. Two of them do not include any domain specific criteria, while the other five use them prominently. Among these 5 rule types all but one include market fields, and three of them also include points of sale as an important and/or selective field, criteria that for the moment we do not feature an implementation of. Given that our research was prototyped on markets, we have chosen therefore to use for evaluation the two dataset presenting only markets and flight groups, and not points of sale. These data sets are called YLD and MCO.

4.1 Data Sets

YLD is an especially simple data set, consisting of a single criterion made of markets alone. The YLD structure is defined in Table 7. Column *Selectivity* shows the average percentage of matching rules for all the possible query values on the total size of the dataset. It is formally defined as:

$$Selectivity = \frac{\#\, matching\, rules}{\#\, rules} * 100\ \%$$

For example MKT_PAIR selectivity 0.0375% means that on a data set with 3711 rules on average a query has 1.39 matches. In practice the MKT_PAIR criterion can be described as extremely selective: each query available in the dataset matches a maximum 2 of 3711 available rules.

Criterion	**isMandatory**	**Supertag**	**Weight**	**Selectivity**
MKT_PAIR	true	AMM	0	0.0375%

Table 7: YLD RuleType

Given the lack of extra criteria that could influence it, the YLD rule type shows the pure performance of our hierarchical indexing technique without any external influence. This case can be considered peculiar also because the weight does not influence the matching algorithm. As the only available criterion is mandatory, all the rules have $weight = 0$. The choice between eventual multiple matching rules is based exclusively on the loms of the fields.

MCO instead is a more heterogeneous rule type, enclosing most of the element types available in the engine, including flight groups and pairs of dates. The structure for MCO is shown in Table 8.

Criterion	isMandatory	Supertag	Weight	Selectivity
MKT_PAIR	true	AMM	0	0.57%
S_OPAIR	true	S	0	100.00%
FG1	false	F	64	67.94%
CSH	false	S	32	100.00%
DATE_PAIR	false	ADD	16	5.29%
DAYS	false	EI	8	96.88%
EQP	false	S	4	100.00%

Table 8: MCO RuleType

The mandatory market pair criterion (MKT_PAIR) is proven once again the most selective of the rule type's criteria. The majority of its queries return in fact 18 out of the 3166 rules the dataset is made of. Criterion S_OPAIR is shown instead as completely useless for selectivity in spite of its being mandatory, and therefore a totally unhelpful choice as an indexing field in the original index. In Figure 12 we show an analysis of the criteria selectivity introducing another level of detail relevant for the optional criteria. We take into consideration what percentage of retrieved rules is returned for a criterion just because it has not been defined in the rules. Flight group criteria FG1 is for example very selective for the rules which define it, but its efficacy is mined by the half of the rule set which does not. From the analysis we can see however that it is entirely reasonable to base our index on MKT_PAIR. Future works may consider DATE_PAIR as secondary index criteria.

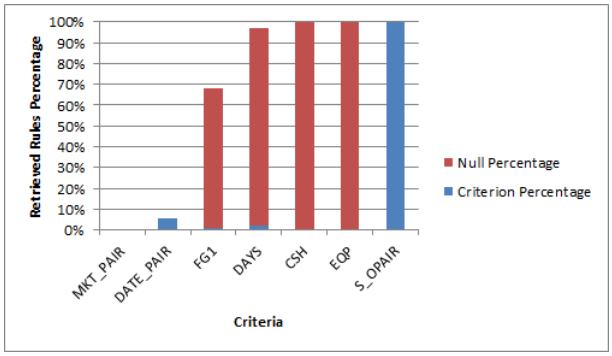


Figure 12: MCO Criteria Analysis

4.2 Queries

The queries used for testing have been chosen in order to mimic different contexts and *lom* granularities that may result in different performances of the index. We have therefore included queries with very specific granularities (i.e. airports) as well as queries including bigger zones as region, or both. A middle level of granularity is represented by countries. Queries with regions as origin or world in any position were not included because they would not yield any result. This detects also an asymmetry in the origin/destination definitions.

Content and statistics for queries used for rule type *YLD* are reported in Table 9. In the table are reported the numbers of rules matching the complete

query (column *Match*), just the origin criterion (column *Origin*) or just the destination (column *Dest.*).

Query	**Content**	**Match**	**Origin**	**Dest.**
ACDG_ATXL	A:CDG,A:TXL	2	86	188
ABLQ_AZRH	A:BLQ,A:ZRH	2	82	169
COCH_AJFK	COUNTRY:CH,A:JFK	2	79	159
COIT_CODE	COUNTRY:IT,COUNTRY:DE	2	82	188
COUS_COFR	COUNTRY:US,COUNTRY:FR	2	72	173
AZRH_RNAMER	A:ZRH,REGION:NAMER	1	79	94
AJFK_REEURO	A:JFK,REGION:EEURO	1	72	95

Table 9: YLD queries

From the table we can see how the criteria are not extremely selective per se, but they get more specific through their intersection. Also, the granularity used to define origins and destination is different, as destinations are generally kept broader than origins.

For *MCO* we have chosen a more extended testing set, including queries with different optional parameters and retrieved rules. The optional parameters include flight groups, pair of dates and sets of integers. The queries are fully described in Table 10. In column *Match* is reported the number of matching rules that specific query has, while in column *Market* are reported the number of matching rules just for the market pair, that is equivalent to the mandatory criteria.

Query	Content	Match	Market
0M_18	S_OPAIR LH MKT_PAIR REGION:ASIA,REGION:NAMER	18	18
0M_24	S_OPAIR LH MKT_PAIR A:HAM,CITY:KSZ	24	24
0M_44	S_OPAIR LH MKT_PAIR A:DUS,A:LHR	44	44
0M_50	S_OPAIR LH MKT_PAIR A:CDG,A:TXL	50	50
0M_85	S_OPAIR LH MKT_PAIR A:HAJ,A:FRA	85	85
3_12	S_OPAIR LH MKT_PAIR A:MUC,A:SFO FG1 LH440	12	33
3_106	S_OPAIR LH MKT_PAIR A:FRA,A:HAJ DAYS 1	106	107
4_16	S_OPAIR LH MKT_PAIR CITY:CIT,A:FRA DAYS 3 FG1 LH1985	16	34
4_33	S_OPAIR LH MKT_PAIR A:TXL,A:CDG DAYS 5 FG1 LH687	33	51
5_1	S_OPAIR LH MKT_PAIR A:BQG,A:DUS DAYS 3 DATE_PAIR 25Dec2012,26Dec2012 FG1 LH1985	1	28

Table 10: MCO queries

4.3 Experimental Results

Our tests were made running different index implementations for the same dataset and the same queries and evaluating following the metrics described in Section 1.2.2.

4.3.1 System Setup

Given the impossibility to test directly into the Amadeus environment all indexing strategies were implemented in a comprehensive Java framework that consented to compare results using the same environment and the same machine. All tests were performed on a server with the following hardware and software configuration:

Architecture	x86_64
CPU	Intel Xeon L5520 2.26 Ghz (Nehalem, with HT enabled)
CPUs	16
Thread(s) per core	2
Core(s) per socket	4
Socket(s)	2
RAM	24 GiB (6x4, 3 per cpu, 1 per channel)
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	8192K
Linux version	Debian 4.7.2-5
Java version	1.7.0_25

4.3.2 Hierarchical index

The evaluation of the hierarchical index consists in comparing the performance of the new different implementations against each other and against the baseline, that is the index currently used and deployed by Amadeus. The featured implementations are described as following:

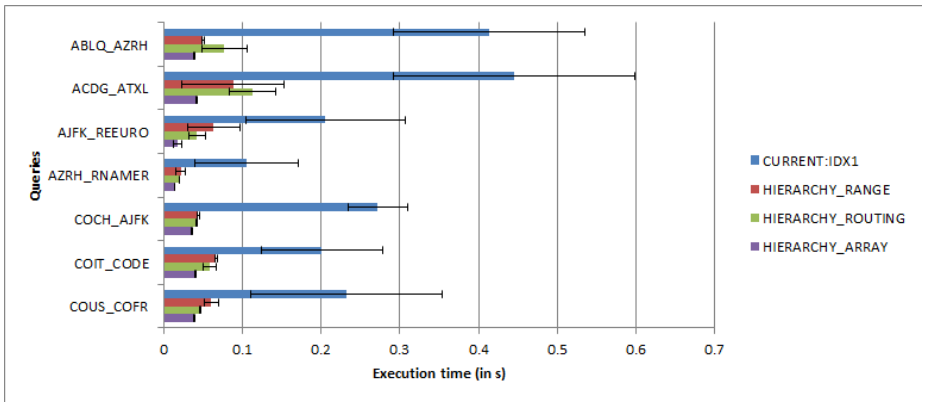
CURRENT:IDX1 The index currently implemented by Amadeus and described in Section 2.2. We have reproduced it locally and we consider it our baseline. IDX1 stands for the mandatory criteria chosen as an index by Amadeus. For the MCO and YLD rule types it always consisted of all the mandatory criteria.

HIERARCHY_RANGE The index based on a range tree described in Section 3.3.

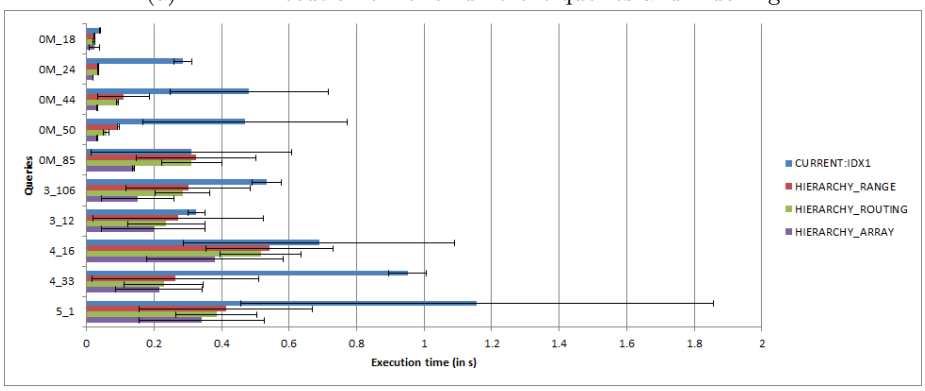
HIERARCHY_ROUTING The index inspired by routing structures based on a multibit trie described in Section 3.4.

HIERARCHY_ARRAY The index based on a hash table and array described in Section 3.5.

Execution Time
Plots for execution time for all the engines and all of the previously mentioned queries are shown in Figure 13.



(a) YLD: Execution time for different queries and indexing



(b) MCO: Execution time for different queries and indexing

Figure 13: Execution time for varying rule types

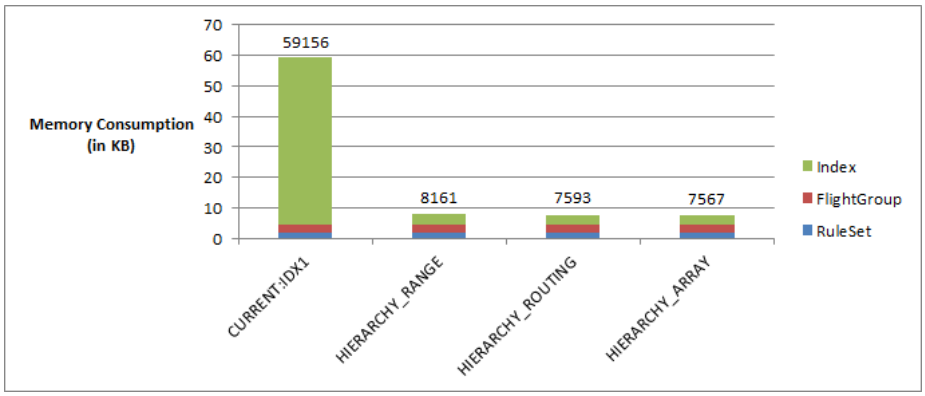
The graphs show how on average the original merged index is outperformed by the hierarchical index in all of its forms. In line with our expectations, the array index holds the most efficient performance, as it eliminates the two phase index parsing (dictionary lookup and tree traversing) and provides direct array access instead of tree explorations. The differences in performance between the range and the routing index are debatable. The routing index for our dataset normally outperforms the range index, except for cases in YLD where the queries are the most specific. The reasons behind this special case will be more thoroughly examined in the future. However, a possible factor that may influence them could be the node fan out. A bigger fan out is generally shown by the need of a bigger binary address, therefore in Table 11 are shown the binary addresses for the YLD queries in the format *integer binary address / number of relevant bits*. The queries in which the routing address performs worse are generally the ones where the binary addresses are made of a longer bit string, therefore where nodes tend to have on average a bigger fanout. This calls for a special attention in the structuring of the routing index for future developments.

Query	Binary Address	Query	Binary Address
A:BLQ	13718/14	COUNTRY:DE	3353/12
A:CDG	6712/13	COUNTRY:FR	6845/13
A:JFK	1808/12	COUNTRY:IT	3462/12
A:TXL	7024/13	COUNTRY:US	28/6
A:ZRH	1892/11	REGION:EEURO	296/9
COUNTRY:CH	586/10	REGION:NAMER	355/9

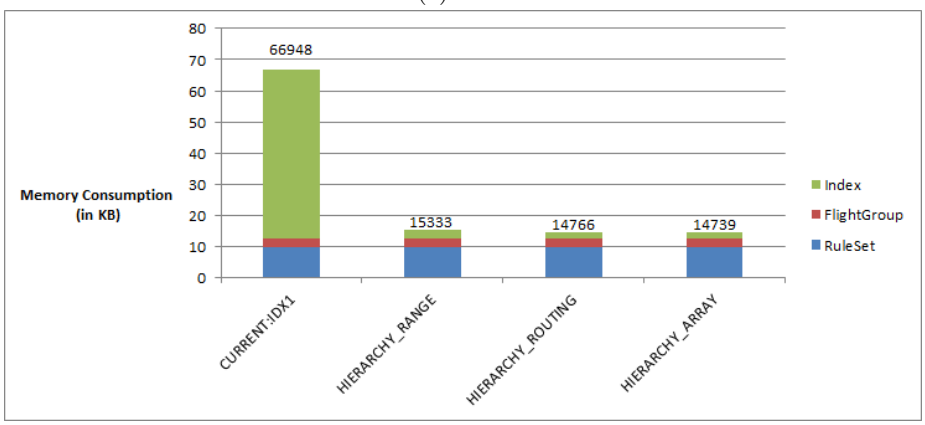
Table 11: Queries and binary addresses for YLD

For MCO this effect is not evident because queries are unintentionally more balanced. For example in query *5_1* contains both *A:DUS* represented by 18 bits, and *A:BQG* represented by only 7 bits. This shows how performance and the size of bit address is simply correlated to the specificity of the query, and not strictly dependant. The difference in performance depends mostly on the structuring of the tree and on the fan out of the crossed nodes.

Memory Consumption
The measurement of memory consumption takes into consideration for both datasets the space occupied by the pure rule set, the size of the flight group information and the size of the index including the market definitions for all the compared techniques. The market definitions are built into the index for all the hierarchical approaches, while they are kept independently for the current Amadeus index. Statistics for all the data sets are shown in Figure 14.
The results predict what has already been estimated in our previous considerations. The elimination of most of definition redundancy causes a sharp decrement in the space occupation between the current implementation and the hierarchical indexes. Differences between the hierarchical indexes are less noticeable but consistent with our prediction. For our number of *loms* the array index seems to be the more effective, but the its memory occupations strongly depends on the number of layers. Increasing the number of layers should improve the performance of the range and routing indexes versus the array one and will be addressed in future work.



(a) YLD



(b) MCO

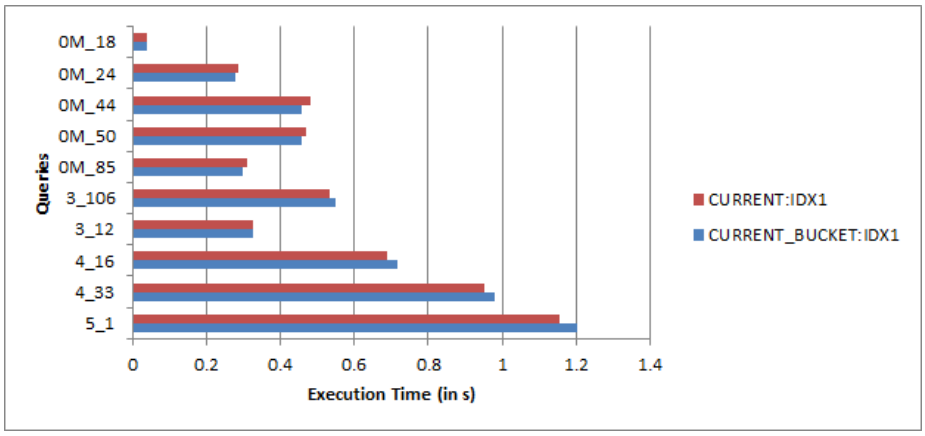
Figure 14: Memory consumption for varying rule types

CPU Usage

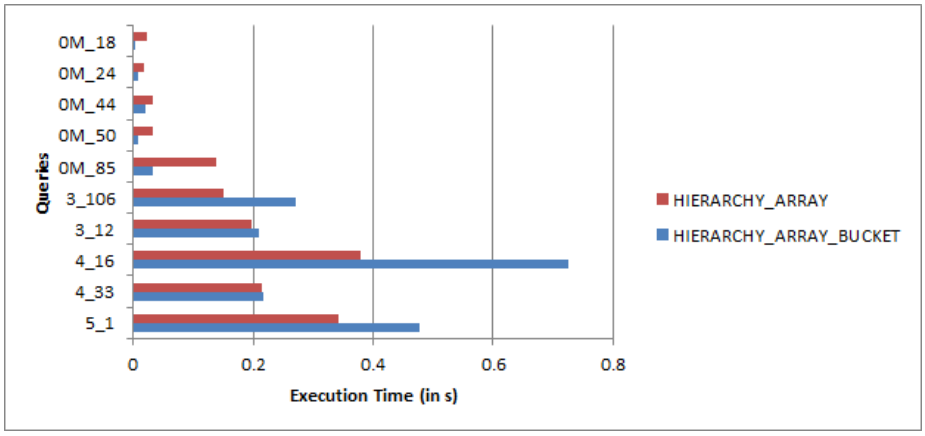
To measure CPU usage, we monitored the CPU parameters of the system during the whole execution of the experiments for all the three datasets. During the monitoring we could not detect any particular difference between the various approaches. All the index techniques are equivalent for CPU utilization and therefore we cannot use this parameter as a discriminant.

4.3.3 Bucket Rule Processing

We also tested the alternative bucket method for rule processing. Bucket rule processing was implemented for all the different index types and yielded similar results. Results for the Amadeus implementation and the Array Index applied to the MCO rule type are reported in Table 15. It is obviously useless to report data for the YLD dataset, as all of its rules have the same weight and a lazy sorting approach is therefore highly inefficient. Our expectations were that this type of processing should improve results in cases where the index retrieves a lot of rules and a match is found within the rules with a highest weight.



(a) Execution Times for the Current Amadeus Index



(b) Execution Times for the Array Index

Figure 15: MCO Execution Times for normal and bucket rule processing applied to different indexes

The results proved that a bucket approach is undoubtedly more convenient when only mandatory criteria from the index are used, but shows inefficiency when most complex queries are issued. In mandatory criteria-only queries, we already know that all the rules we have retrieved from the index are matching, or at least most of them (in case of secondary criteria other than markets used) and therefore avoiding the sorting of all the rules delivers a better performance because we know that a match will be found in the first buckets with the highest weight that we examine. The more matching rules are retrieved from the index, the more this difference is emphasized, as shown for query "0M_85", that bears the highest number of matching results for mandatory criteria only. On the contrary, by adding more optional criteria we increase the chance that our result may not be in the retrieved rules with the highest weight (that tends to be the most specific), but in the middle or the bottom of the buckets. In this case, the extra-effort to implement bucket sorting does not pay off and we have therefore a disadvantageous performance. An hybrid approach that may benefit from both techniques could use bucket rule processing for queries under a certain

threshold of specificity and normal sorting above.

4.4 Statistics

Finally, we provide some statistics for the Amadeus market data we used. Table 12 first reports data that is independent from our index: the number of different queries supported by the system and the number of market ids used to support them. Then it reports the features specific to our hierarchical indexes, in the specific: the total number of nodes, how many of them are leaves and statistics for the nodes' fanout. Obviously, only the number of nodes is partially relevant for the array index, as its memory allocation depends only on this and on the number of level-of-matching attributes.

Queries	21487
Markets	912
Nodes	993
Leaves	521
Fanout Avg	2.10
Fanout Min	1
Fanout Max	49

(a) YLD and MCO

Queries	21491
Markets	3468
Nodes	1298
Leaves	784
Fanout Avg	2.52
Fanout Min	1
Fanout Max	90

(b) AAS

Table 12: Statistics for hierarchical market information for datasets with different sizes

Unfortunately both YLD and MCO made use of the same market data, so we decided to compare their statistics with the ones of the only bigger dataset given to us by Amadeus, AAS (we were not able to test it for its featuring of points of sale). The tables show how the structuring of the information scales up when we increase the number of market ids used while keeping the number of queries almost constant. The memory occupation and the complexity of the tree scale remarkably well for the Amadeus data: incrementing 3.8 times the number of market ids, the number of nodes in the system does not even double, showing a better ratio *#market ids/#nodes* than our tested data sets. Statistics for other hierarchical parameters also scale sublinearly.

5 Related Works

Hierarchical indexing is not certainly a new idea and is already widely used in industry in many different forms. While we have already examined and adapted some of them to our problem, others have been proven less well-fitting to the task.
A widespread indexing technique is for example represented by the family of the B Trees. A good description of all of their possible varieties is given by Comer [3]. Commonly adopted in the form of B+ Trees, these data structures support very efficiently point and range queries with integer values. However, we cannot represent fit our data in ranges using B Trees as we did for the range index. Querying a large range would mean in the B Tree to retrieve all the elements contained in it and therefore would be logically wrong. For our hierarchical data, a bigger range represents a less specific query that does not need to retrieve values from its subranges. A possible adaptation of the B Tree would be hardly implemented and, above all, would lose all of the advantages of the original. The B Tree guarantees performance at the expense of a smaller control over the specific values contained in the nodes, that we would have to enforce to keep the logic of our hierarchy.

Another relevant technique is represented by the Bitmap Index, a widespread industry technique that was firstly formalized in [7]. A good definition is provided in [4]: *a bitmap index for a field F is a collection of bit-vectors of length n, one for each possible value that may appear in the field F. The vector for value v has 1 in position i if the i^{th} record has v in filed F, and it has 0 there if not.* This approach is useful especially in case of multiple index criteria, because it allows to compute the intersection of results on simple AND on bit operations. Its drawback is unfortunately high memory consumption. In our domain this would mean to associate to each query value a bit vector the size of our rule set. This strategy has therefore not been considered for memory-related performance issues.

6 Conclusion and Future Work

In this work we discussed alternative index strategies to optimize the performance of indexes applied to business rules engines. In particular our work took the Amadeus business rule engine, an engine dealing with domain specific data related to the travel industry, as inspiration and focus of experimental evaluation. We thoroughly analysed the functioning and indexing of the original index and decided to focus our work on the indexing of domain specific data, that proved to be a critical pain point. Therefore, we developed a new hierarchical representation to describe market data and three different techniques to effectively use it for indexing: the Range Index, the Multi Bit Trie Index (or Routing Index) and the Array Index. Finally we developed an alternative approach for rule processing after index retrieval, the bucket rule processing. All of the three different indexing strategies proved to improve consistently the baseline in experimental evaluation. For our tested data set the array index delivered the best performance, but the other techniques, including bucket rule processing, might also have good applications depending on the size of specific parameters in the data set. Future work will revolve first around studying more into depth different datasets. Scale up testing needs to be considered for different dimensions: number of rules, number of level-of-matching and number of queries / market ids. Another important point will be to create a multi-dimensional index based on our current data structure. Finally, we would like to turn the choice of the indexing criteria from static to dynamic, creating an index that picks the most appropriate criterion base on the selectivity of the available criteria.

References

[1] Amadeus IT. About Amadeus, 2013.

[2] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[3] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[4] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[5] George Varghese. Prefix-Match Lookups. In *Network Algorithmics*, chapter 11, pages 233–266. Elsevier/Morgan Kaufmann, 2005.

[6] David Hay and Keri Anderson Healy. Defining business rules: what are they really? Technical report, The Business Rule Group, 2000.

[7] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes, 1997.

[8] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15, 2001.

[9] Hanan Samet. Range Trees. In Morgan Kaufmann, editor, *Foundations of Multidimensional and Metric Data Structures*, chapter 1.2, pages 14–18. Kaufmann, Morgan, 2006.