# T U M

I N S T I T U T F Ü R I N F O R M A T I K

# Matching and Evaluation of Disjunctive Predicates for Data Stream Sharing

Richard Kuntschke Alfons Kemper

TUM-I0615
August 06

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-08-I0615-100/1.-FI


Druck: Institut für Informatik der
Technischen Universität München

# Matching and Evaluation of Disjunctive Predicates for Data Stream Sharing*

Richard Kuntschke Alfons Kemper

Lehrstuhl Informatik III: Datenbanksysteme
Fakultät für Informatik
Technische Universität München
Boltzmannstraße 3, D-85748 Garching bei München, Germany
{richard.kuntschke|alfons.kemper}@in.tum.de

**Abstract**

Traditional query optimization largely neglects the handling of disjunctive predicates. However, new and evolving applications and optimization techniques, e. g., in the domain of data stream management systems (DSMSs), make the treatment of disjunctive predicates a necessity.

In this paper, we introduce and discuss methods for matching and evaluating disjunctive predicates in the context of *data stream sharing* in a DSMS. Nevertheless, the presented techniques are generic and can be applied to other domains as well. Data stream sharing uses one data stream for satisfying multiple similar continuous queries in a network. Sharing an existing stream for answering a new query requires, among other things, the selection predicates of the new query to be matched with the predicates describing the contents of the shared stream. Predicate matching is a combination of predicate implication checking and predicate relaxation. If no match is found, sharing can be enabled by widening the stream, e. g., by relaxing a selection predicate, which can introduce additional disjunctions in the stream predicates. We propose heuristics as well as an exact algorithm for solving the predicate matching problem and discuss the use of multi-dimensional indexing for speeding up the matching and evaluation processes for interval-based disjunctive predicates. To the best of our knowledge, this is the first work to investigate the use of multi-dimensional indexes for matching and continuously evaluating disjunctive predicates. An extensive experimental study compares and evaluates the presented algorithms and reveals a performance gain of several orders of magnitude for predicate matching and evaluation through multi-dimensional indexing.

## 1 Introduction

Except for a few publications which have dealt with the issue in the database field [4, 6, 9, 20, 24], disjunctive predicates, which are known to be complex to handle, have largely been neglected in the context of query optimization for traditional database management systems (DBMSs) and also for domains like, e. g., active databases [30] and publish&subscribe systems. Instead, query optimization generally limits itself to considering conjunctive query predicates since well-known ways for efficiently managing such predicates exist. The main argument for justifying the neglect of disjunctive predicates has been that such predicates do not occur often in practice. While this argument might be true for traditional database systems and applications, it is not correct for new and evolving applications and optimization techniques, e. g., in the domains of semantic caching [10] and data stream management systems (DSMSs). Considering DSMSs for example, new network-aware optimization techniques that take into account the current

*This research is supported by the German Federal Ministry of Education and Research (BMBF) within the D-Grid initiative under contract 01AK804F and by Microsoft Research Cambridge (MSRC) under contract 2005-041.

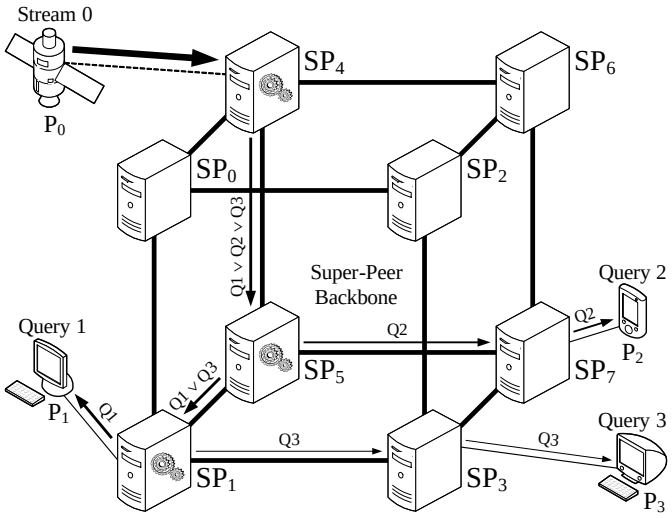



**Figure 1:** Example DSMS Scenario

network state for deciding how to distribute query processing operators among network nodes and how to route data streams through the network can introduce disjunctions in predicates. We focus on disjunctive predicates consisting of disjunctively combined conjunctive predicates. Each conjunctive predicate forms a multi-dimensional hyperrectangle with edges parallel to the coordinate axes in the data space. Our approaches can also be used as an approximation for more complex shaped predicates. However, this is not dealt with in this paper. Although we use a DSMS example scenario, it is worth noting that the techniques presented in this paper are generic and can be applied to other domains as well.

As an example for a DSMS that needs to handle disjunctive predicates during query optimization and evaluation consider Figure 1 and the following setting. The DSMS is based on a hierarchical P2P network where peers are classified into super-peers ($SP_0$ to $SP_7$ in Figure 1) and thin-peers ($P_0$ to $P_3$ in Figure 1). Super-peers usually are stationary powerful servers with extensive query processing capabilities that form a stable super-peer backbone network. Thin-peers can register themselves at any super-peer and deliver data streams to the network (e. g., Stream 0 delivered by $P_0$ in Figure 1) or register continuous queries—also simply called queries or subscriptions in the following—over data streams present in the network (e. g., Queries 1 to 3 registered by $P_1$ to $P_3$ in Figure 1). The super-peers are responsible for processing such queries and delivering the results back to the corresponding thin-peers. For each new query that is registered in the network, the optimizer has to decide where, i. e., at which super-peers, to execute the query processing operators and how to route the processing results to the receiving thin-peer. This can be done based on the following optimization techniques.

*Data stream sharing* reuses data streams already present in the network for answering newly registered continuous queries. Thereby, streams that have been computed for previously registered queries can be shared to satisfy new queries. This is enabled by *in-network query processing*, which offers the ability to execute query processing operators at any super-peer in the network, and *multi-subscription optimization*, which aims at identifying reusable data streams that can be shared to answer multiple queries. Considering the example depicted in Figure 1, the query processing operators for Query 1 are executed close to the data source at $SP_4$, e. g., to exploit early filtering. The result data stream of Query 1 is routed to $P_1$ via $SP_5$ and $SP_1$. It can be shared at $SP_5$ for satisfying Query 2, assuming that the result for Query 2 is completely contained in the result for Query 1. Any additional processing for obtaining the final result data stream of Query 2 is performed directly at $SP_5$.

Since sharing a data stream is only possible if the stream contains all the necessary information for

answering a new subscription, optimization quality depends on the registration sequence of queries. If queries selecting smaller parts of the data streams are registered first, their result data streams as produced by in-network query processing are not reusable for satisfying later registered queries that require larger parts of the streams. For example, registering Query 2 before Query 1 in the example network would prevent the sharing of the result data stream of Query 1 as described above. To alleviate this problem, *data stream widening* can be employed, e. g., by relaxing some selection predicate in the network. Thus, an existing result data stream is widened to deliver not only the result data for the query it was originally computed for, but also the data for the new query. Since, in its simplest form, relaxing a predicate in this way can be done by disjunctively combining the predicates of the two queries, data stream widening can introduce additional disjunctions in the predicates of selection operators in the network. To illustrate this, suppose that Query 3 in Figure 1 is registered after Query 1 but needs some more data of Stream 0 than Query 1. The processing operators at $SP_4$ can then be relaxed to produce a result stream containing the necessary information for both queries. This causes the selection predicate for producing the combined stream to become the disjunction of the selection predicates of the two individual queries ($Q_1 \vee Q_3$). Note that the additional disjunction can make future predicate matchings and evaluations more expensive. Further processing at $SP_1$ can then produce the final result data streams of Queries 1 and 3 and deliver them directly to $P_1$ and to $P_3$ via $SP_3$.

To be able to decide whether a certain data stream can be used to satisfy a newly registered query, among other things, the selection predicates of the new query and the query that produced the data stream considered for reuse have to be matched. The matching process consists of an implication check and, if the selection predicate of the new query does not imply the selection predicate of the stream producing query, a predicate relaxation that computes the relaxed predicate covering all the necessary information for both queries. To speed up the predicate matching and evaluation processes, predicates can be indexed using a multi-dimensional index structure. This corresponds to a change in perspective similar to predicate indexing in, e. g., active databases and publish & subscribe systems [18, 19, 31, 32]. In traditional DBMSs, data is relatively static and queries are dynamic, i. e., different queries are posed and answered using the already present data. Therefore, the data is indexed to support efficient answering of certain query types. In a DSMS, the set of registered continuous queries is relatively static (apart from registering new and deleting old queries) and the data arriving in the form of continuous, possibly infinite data streams is higly dynamic. Thus, the queries—or, in the context of this paper, the query predicates—are indexed for efficient predicate matching and evaluation.

In this paper, we describe and discuss various methods for matching and evaluating interval-based disjunctive predicates. We propose heuristics as well as an exact solution for the predicate matching problem and investigate the use of multi-dimensional indexing for speeding up the matching and evaluation processes for disjunctive predicates. To the best of our knowledge, this is the first work to examine the use of multi-dimensional indexes for matching and continuously evaluating disjunctive predicates. We have implemented all presented algorithms and show the results of an extensive experimental study that we have conducted for comparing and evaluating the various approaches. The study reveals that performance gains of several orders of magnitude are achievable for predicate matching and evaluation through our multi-dimensional indexing approach.

The paper is structured as follows. In Section 2, our notion of predicates in the context of this paper, the problems of predicate matching and evaluation, and some notation are introduced. Section 3 presents some heuristics and an exact solution for the matching problem. A standard and an index-based approach for the evaluation of disjunctive predicates are described in Section 4. In Section 5 we analyze the space and time complexities of the presented algorithms. Section 6 contains a description of our experimental studies and presents their results. Related work is discussed in Section 7. Section 8 concludes the paper and states some ideas for future work.

# 2 Preliminaries

Before we describe the algorithms for predicate matching and evaluation, we first introduce our restricted notion of predicates in the context of this paper and define the problems of predicate matching and evaluation.

## 2.1 Predicates

Predicates in our context are of the following three forms:

**Atomic predicate:** An atomic predicate is a comparison of the form $v\ \theta\ c$, where $v$ is a variable, $c$ is a constant, and $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

Example: $a \leq 5$

**Conjunctive predicate:** A conjunctive predicate is a conjunction of atomic predicates.

Example: $(a \leq 5) \wedge (b \geq 7)$

**Disjunctive predicate:** A disjunctive predicate is a disjunction of conjunctive predicates.

Example: $((a \leq 5) \wedge (b \geq 7)) \vee ((a \geq 0) \wedge (b < 9))$

We call the distinct variables referenced in a predicate the *dimensions* of the predicate. For example, the disjunctive predicate shown above has two dimensions named $a$ and $b$. Atomic predicates define intervals in the various dimensions of a predicate by setting lower and upper bounds that can be included in or excluded from the interval itself. For example, $a \geq 0$ defines an included lower bound of 0 for the interval in dimension $a$. In contrast, $b < 9$ defines an excluded upper bound of 9 for the interval in dimension $b$. Intervals can also be unbounded on one or both ends, i. e., the lower bound of an interval can be negative infinity and the upper bound can be positive infinity. Atomic predicates of the form $v = c$ can be replaced by $(v \leq c) \wedge (v \geq c)$ and atomic predicates of the form $v \neq c$ can be replaced by $(v < c) \vee (v > c)$. The disjunctively combined conjunctive predicates making up a disjunctive predicate are called the (conjunctive) *subpredicates* of the respective disjunctive predicate.

Note that the above definition defines a predicate hierarchy, i. e., any atomic predicate can be seen as a special case of a conjunctive predicate and any conjunctive predicate can in turn be seen as a special case of a disjunctive predicate but not vice versa. Note also that any conjunctive and disjunctive combination of atomic predicates can always be transformed into the form of a disjunctive predicate as defined above by transforming it into disjunctive normal form (DNF) with the atomic predicates being treated as literals.

## 2.2 Predicate Matching

Predicate matching, in our context, is the problem of deciding whether a predicate implies another and, if this is not the case, how the other predicate can be altered in order for the implication to be valid. More formally, given two predicates $p$ and $p'$, matching $p'$ with $p$ returns $(\text{true}, p)$, if $p' \Rightarrow p$, and $(\text{false}, \bar{p})$, where $\bar{p}$ is a relaxed version of $p$ such that $p' \Rightarrow \bar{p}$ (and of course also $p \Rightarrow \bar{p}$), otherwise.

While the implication problem can be solved easily for conjunctive predicates [25, 27], it is proven to be NP-hard for disjunctive predicates [27]. Since disjunctive predicates are inevitably created in the course of data stream widening in a DSMS, matching such predicates is a necessity during the optimization of stream processing in such a system.

In the following, we always consider the matching of the *query predicate* of a newly registered query with the *stream predicate* of a stream producing query that is already being executed in a DSMS. Nevertheless, the presented techniques are generic and can be applied to any predicates of the form described in Section 2.1 and in any domain.

| Variable | Description |
|---|---|
| $p$ | disjunctive stream predicate |
| $p'$ | disjunctive query predicate |
| $c$ | conjunctive subpredicate of stream predicate $p$ |
| $c'$ | conjunctive subpredicate of query predicate $p'$ |
| $d$ | dimension of a conjunctive subpredicate $c$ in $p$ |
| $d'$ | dimension of a conjunctive subpredicate $c'$ in $p'$ |
| $I_d$ | interval defined by $c$ in dimension $d$ |
| $I_{d'}$ | interval defined by $c'$ in dimension $d'$ |
| $D$ | data space |
| $\|D\|$ | number of dimensions in the data space |
| $\|D_c\|$ | number of dimensions referenced by $c$ |

**Table 1:** Variables used in algorithm descriptions

### 2.3 Predicate Evaluation

Predicate evaluation, in our context, is the problem of deciding whether a data item satisfies a predicate or not. More formally, given a predicate $p$ and a data item $i$, evaluating $p$ against $i$ returns true, if, for all dimensions referenced in $p$, the value of $i$ in the corresponding dimension lies within the interval defined for that dimension in $p$.

### 2.4 Notation

Before starting with the algorithm descriptions, we introduce some notation. In the pseudocode representations of the predicate matching and evaluation algorithms in Appendix A on page 43, assignments of the value of a variable $y$ to a variable $x$ are written $x \leftarrow y$. Assignments are supposed to assign a copy of the value of $y$ to $x$. Furthermore, function calls are supposed to use call-by-value. Unless explicitly stated, queues used in the algorithm descriptions can be either FIFO or LIFO queues. Important variables used during algorithm description are shown in Table 1. Note that the variable $d$, which denotes a dimension in the data space in the algorithm descriptions, is sometimes also used to denote $|D|$, i. e., the number of dimensions in the data space. In each case, the meaning of $d$ will be clear from the context.

## 3 Predicate Matching

In the following, three algorithms for matching disjunctive predicates are presented. The first two are heuristics that are very efficient, yet do not deliver exact results. The last one is an exact method whose worst case running time is exponential in the number of subpredicates contained in the predicates to be matched.

### 3.1 Example

Consider predicates $p_1$ and $p_2$ defined below as examples and suppose we want to match $p_2$ with $p_1$, i. e., determine whether $p_2$ implies $p_1$ or how $p_1$ could be modified in order for the implication to be valid.

$p_1$: $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5)) \vee$
$((a \geq 9) \wedge (a \leq 14) \wedge (b \geq 2) \wedge (b \leq 8)) \vee$
$((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6))$

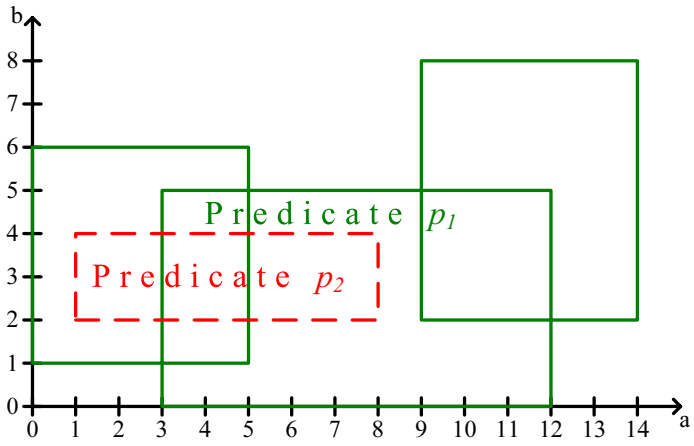



**Figure 2:** Graphical representation of predicates $p_1$ (solid boxes) and $p_2$ (dashed box)

$p_2$: $((a \geq 1) \wedge (a \leq 8) \wedge (b \geq 2) \wedge (b \leq 4))$

Note that $p_1$ is a disjunction of conjunctive predicates and that, for simplicity and a shorter presentation, $p_2$ consists of only one conjunctive subpredicate and does not contain any disjunctions. However, our algorithms are of course capable of handling the general case of more than one subpredicate in $p_2$. A graphical representation of predicates $p_1$ and $p_2$ is shown in Figure 2.

### 3.2 Quick Check

Before the actual predicate matching algorithms, we first introduce a simple *quick check* (QC) algorithm that can be combined with each of the matching algorithms. It is described in Algorithm 1 and tests for a conjunctive subpredicate $c'$, whether it implies at least one of the conjunctive subpredicates $c$ of a given stream predicate $p$. The implication check for conjunctive predicates can easily be done by checking the bounds of $c'$ for containment in the intervals defined by the atomic predicates in $c$ for all dimensions [25, 27]. If the quick check returns true, nothing more remains to be done for the tested subpredicate because it is clear that it already matches the stream predicate as is.

Concerning our running example, comparing the only conjunctive subpredicate of $p_2$ to each conjunctive subpredicate of $p_1$ obviously yields no match, i. e., the quick check returns false. This is due to the fact that none of the tested implications is valid which can easily be seen from Figure2. The dashed box of $p_2$ is not completely contained in any one of the three solid boxes of $p_1$.

Since the algorithm has to iterate once over all conjunctive subpredicates of $p$ and, for each subpredicate, over all dimensions referenced in that subpredicate, the worst case complexity of this algorithm is in $O(n \cdot d)$, where $n$ is the number of conjunctive subpredicates $c$ in $p$ and $d$ is the number of dimensions in the data space. Algorithm 6 on page 43 contains a pseudocode representation of the quick check.

### 3.3 Heuristics with Simple Relaxation

The easiest way to perform predicate matching is to completely skip the predicate implication checking and go directly to the relaxation part. This is the idea of the *heuristics with simple relaxation* (HSR) shown in Algorithm 2. When matching a predicate $p'$ with a predicate $p$, all conjunctive subpredicates of $p'$ are disjunctively added to $p$. Since this solution does not perform any implication checking at all, it will miss matches already present in the original predicates and perform unnecessary relaxations in general.

**Algorithm 1** Quick Check (QC)

**Input:** Stream predicate $p$ and a conjunctive subpredicate $c'$ of query predicate $p'$.
**Output:** true, if $c' \Rightarrow c$ for at least one conjunctive subpredicate $c$ in $p$; false, otherwise.

1. *Compare subpredicates.* Compare $c'$ to each conjunctive subpredicate $c$ in $p$, i. e., check if $c' \Rightarrow c$.
2. *Return result.* As soon as, for the current values of $c'$ and $c$, $c' \Rightarrow c$, return true. If no conjunctive subpredicate $c$ in $p$ with $c' \Rightarrow c$ exists, return false.

The situation can be improved by combining the approach with the quick check algorithm of Section 3.2. The matching problem for disjunctive predicates is thereby basically reduced to the implication problem for conjunctive predicates. In this solution, two nested loops compare each conjunctive subpredicate of the query predicate to each conjunctive subpredicate of the stream predicate, checking for implication. If, for each subpredicate of the query predicate, a matching subpredicate in the stream predicate is found, the matching succeeds, else it fails. Obviously, this approach might fail even though the query and the stream predicates do match. In the running example, the only subpredicate $((a \geq 1) \wedge (a \leq 8) \wedge (b \geq 2) \wedge (b \leq 4))$ of predicate $p_2$ does not match any of the three subpredicates $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5))$, $((a \geq 9) \wedge (a \leq 14) \wedge (b \geq 2) \wedge (b \leq 8))$, or $((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6))$ of predicate $p_1$ directly. However, it matches the whole predicate $p_1$, as can be seen from Figure 2, which this algorithm would not realize. Therefore, a mismatch would be reported although the predicates actually do match.

Predicate relaxation in the case of a mismatch is simply done by adding the concerned query subpredicate to the stream predicate using a disjunction. This yields $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5)) \vee ((a \geq 9) \wedge (a \leq 14) \wedge (b \geq 2) \wedge (b \leq 8)) \vee ((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6)) \vee ((a \geq 1) \wedge (a \leq 8) \wedge (b \geq 2) \wedge (b \leq 4))$ for $p_1$ in our example and clearly causes the number of disjunctions in the stream predicate to increase by one. In general, even more than one disjunction might be added—one for each conjunctive subpredicate of the query predicate in the worst case. Note that, if one or more subpredicates of the query predicate already matched the stream predicate before the relaxation and the algorithm just was not able to detect these matches, this strategy still adds unnecessary disjunctions to the stream predicate. This should be avoided since additional disjunctions can cause future predicate matchings and evaluations to become more expensive as the number of subpredicates has direct impact on algorithm complexities.

The worst case complexity of Algorithm 2 is in $O(m)$ without quick check and in $O((m \cdot n + m^2) \cdot d)$ with quick check, where $m$ is the number of conjunctive subpredicates $c'$ in $p'$, $n$ is the number of conjunctive subpredicates $c$ in $p$, and $d$ is the number of dimensions in the data space.

The advantages of the HSR algorithm without as well as with quick check are that it is fast and easy to implement. The disadvantages of the approach obviously are that it generally misses matches—actually all matches if it is used without the quick check—and that it can therefore cause unnecessary predicate relaxations which affects the performance of future predicate matching and evaluation processes. Algorithm 7 on page 43 shows a pseudocode representation of the HSR approach.

### 3.4 Heuristics with Complex Relaxation

The *heuristics with complex relaxation* (HCR) avoids the increase in the number of subpredicates in the stream predicate induced by HSR at the expense of potentially producing only approximate results. The approach is shown in Algorithm 3. For each conjunctive subpredicate in the query predicate, HCR relaxes one of the conjunctive subpredicates in the stream predicate in order for it to match the query subpredicate if no direct match between subpredicates has been found. Relaxing a subpredicate means

**Algorithm 2** Heuristics with Simple Relaxation (HSR)

**Input:** Stream predicate $p$ and query predicate $p'$.

**Output:** $(\text{true}, p)$, if the quick check of Algorithm 1 is activated and, for all conjunctive subpredicates $c'$ in $p'$, $c' \Rightarrow c$ for at least one conjunctive subpredicate $c$ in $p$; $(\text{false}, \overline{p})$, where $\overline{p}$ is a relaxed version of $p$ such that the above condition is satisfied, otherwise.

1. *Relax predicate.* Disjunctively add each conjunctive subpredicate $\acute{c}$ in $p'$ to $p$. Optionally, perform the quick check of Algorithm 1 for $p$ and each $c'$ in $p'$ and only append those conjunctive subpredicates $c'$ in $p'$ to $p$ for which the quick check returns false.
2. *Return result.* Return $(\text{true}, p)$, if no changes have been made to $p$, i. e., no conjunctive subpredicates $c'$ of $p'$ have been disjunctively added to $p$. Otherwise, return $(\text{false}, \overline{p})$, where $\overline{p}$ is the modified version of $p$ after the addition of one or more conjunctive subpredicates $\acute{c}$ of $p'$.

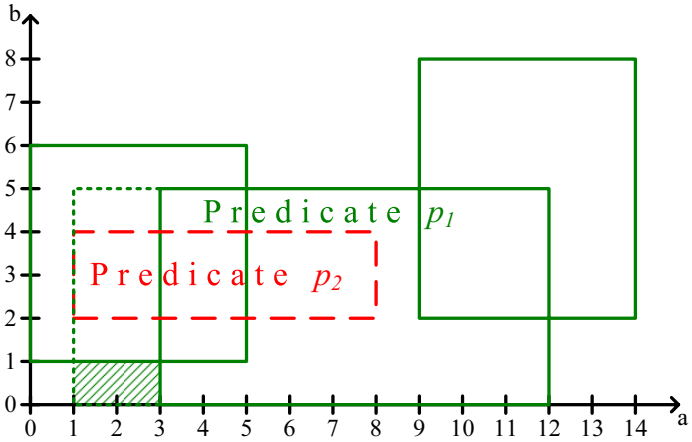



**Figure 3:** Relaxation of predicate $p_1$ (solid boxes) to match predicate $p_2$ (dashed box)

employing a less restrictive filter on the corresponding data stream, therefore increasing network traffic. Thus, the subpredicate of the stream predicate which needs the least amount of relaxation in order to match the query subpredicate should be relaxed. In our running example, this would be $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5))$ which becomes $((a \geq 1) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5))$. The situation is illustrated in Figure 3. In general, this kind of relaxation causes the data stream to contain unnecessary data, e. g., the data with $((a \geq 1) \wedge (a < 3) \wedge (b \geq 0) \wedge (b < 2)) \vee ((a \geq 1) \wedge (a < 3) \wedge (b > 4) \wedge (b \leq 5))$ in our example. However, in the example, parts of these areas are already covered by another conjunctive subpredicate of $p_1$ as can be seen from Figure 3. Therefore, additional unnecessary network traffic is only caused by the inclusion of the hatched area $((a \geq 1) \wedge (a < 3) \wedge (b \geq 0) \wedge (b < 1))$ in Figure 3 in this specific case.

Deciding which subpredicate should be chosen for relaxation is a complex issue. As the example indicates, minimizing the extensions that have to be made to the intervals covered by the subpredicate in the various dimensions of the data space is generally not enough in order to maximize the quality of the solution. This is because unnecessary data added by the relaxation of the subpredicate might or might not already be covered by other subpredicates and therefore might or might not cause additional unnecessary network traffic, i. e., lead to an approximation of the correct result. The decision whether or not the unnecessary parts of the intervals added to the relaxed subpredicate are covered by other subpredicates leads to the same kind of matching problem that we initially intended to solve—without the relaxation

**Algorithm 3** Heuristics with Complex Relaxation (HCR)

**Input:** Stream predicate $p$ and query predicate $p'$.

**Output:** $(\text{true}, p)$, if, for all conjunctive subpredicates $c'$ in $p'$, $c' \Rightarrow c$ for at least one conjunctive subpredicate $c$ in $p$; $(\text{false}, \overline{p})$, where $\overline{p}$ is a relaxed version of $p$ such that the above condition is satisfied, otherwise.

1. *Compare subpredicates.* Compare each conjunctive subpredicate $c'$ in $p'$ to each conjunctive subpredicate $c$ in $p$ in step 2. Optionally, perform the quick check of Algorithm 1 for $p$ and $c'$ and only consider $c'$ in the following if the quick check returns false.
2. *Compare dimensions.* Compare each dimension $d$ of $c$ to the corresponding dimension $d'$ of $c'$, i. e., in the following, $d = d'$ holds. For each pair of corresponding dimensions $d$ and $d'$, if the interval $I_{c'}^{d'}$ of $c'$ in $d'$ is not completely contained in the interval $I_c^d$ of $c$ in $d$, compute the amount $p$ by which $I_c^d$ has to be extended, i. e., the sum of the amounts by which its lower bound has to be decreased and its upper bound has to be increased in order for the containment to be valid. Multiply $p$ with the product of the non-zero extents of all finite intervals in all other dimensions of $c$ and add up the results for all dimensions, yielding an accumulated value $e$. Replace $I_c^d$ in $c$ with its extended version. If, after the comparison of all dimensions, the relaxed version of $c$ has less unbounded interval ends than the current best solution (the initial best solution has an infinite number of unbounded interval ends) or the same number of unbounded interval ends and a smaller value for $e$ (again, the initial value for $e$ is infinite), it is saved as the new current best solution. If, after the comparison of $c'$ with all $c$ in $p$, no match for $c'$ without relaxation has been found, replace the original version of the current best solution in $p$ with the relaxed version computed above.
3. *Return result.* Return $(\text{true}, p)$, if no changes have been made to $p$, i. e., no conjunctive subpredicates $c$ of $p$ have been replaced with relaxed versions. Otherwise, return $(\text{false}, \overline{p})$, where $\overline{p}$ is the modified version of $p$ after the relaxation of one or more conjunctive subpredicates $c$ in $p$.

aspect of course. We use heuristics to solve this problem and choose the subpredicate with the lowest number of infinite interval bounds for relaxation. If the number of infinite interval bounds is equal for two subpredicates, we choose the subpredicate that yields the lowest increase in the volume of the data space it covers, ignoring dimensions with infinite interval length.

Figure 4 illustrates a case where relaxation is actually necessary. In this example, the third subpredicate of predicate $p_1$ has been altered from $((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6))$ to $((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 3) \wedge (b \leq 6))$ to form predicate $p_1'$. This leads to the necessary inclusion of the hatched area described by $((a \geq 1) \wedge (a < 3) \wedge (b \geq 2) \wedge (b < 3))$ in Figure 4 whereas the other hatched area described by $((a \geq 1) \wedge (a < 3) \wedge (b \geq 0) \wedge (b < 2))$ is unnecessarily included additionally.

HCR, like HSR, can be combined with the quick check algorithm of Section 3.2 to detect obvious matches before starting the more complex relaxation algorithm. The worst case complexity of the algorithm is in $O(m \cdot n \cdot d^2)$ with and without quick check, with $m$, $n$, and $d$ as defined before.

The advantages of HCR are similar to those of HSR, i. e., the approach is relatively fast and easy to implement. Furthermore, it does not introduce any additional disjunctions in the stream predicate. The disadvantages are also similar since the approach still misses matches and therefore performs unnecessary predicate relaxations in general. Additionally, HCR, in contrast to HSR, can lead to the inclusion of unneeded parts of the data space in the relaxed predicate and therefore cause unnecessary network traffic through false drops. This necessitates additional filtering to obtain the exact result if approximate results are not acceptable. Algorithm 8 on page 45 shows a detailed pseudocode representation of the HCR algorithm.

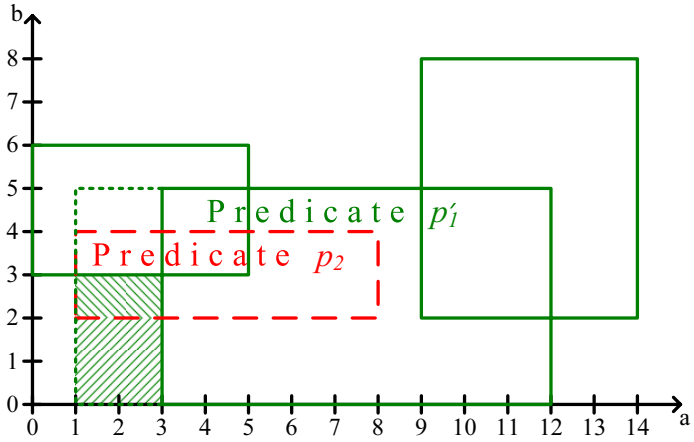



**Figure 4:** Partial match of predicates $p'_1$ (solid boxes) and $p_2$ (dashed box)

### 3.5 Exact Matching

The *exact matching* (EM) algorithm is a split algorithm that always correctly detects a match of a query predicate $p'$ with a stream predicate $p$. It does not miss matches like the heuristics above nor does it report false matches. The query predicate is split along its dimensions according to the boundaries of the overlapping intervals of the stream predicate. Only if, at the end of the matching process, all parts of the query predicate have been successfully matched, a match is reported. Otherwise, the stream predicate is relaxed. The approach is described in detail in Algorithm 4.

Concerning the two dimensions $a$ and $b$ of our running example, we first match the intervals $[3,12]$ for $a$ and $[0,5]$ for $b$ of the first subpredicate $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5))$ of $p_1$ with the intervals $[1,8]$ for $a$ and $[2,4]$ for $b$ corresponding to $((a \geq 1) \wedge (a \leq 8) \wedge (b \geq 2) \wedge (b \leq 4))$ of $p_2$. Since the interval for $b$ of $p_2$ is completely contained in the interval for $b$ of $p_1$, we do not have to split the interval for $b$ of $p_2$. We simply keep this interval and only split the interval for $a$ of $p_2$ into the two intervals $[1,3[$ and $[3,8]$. The second of these two intervals is covered by the first subpredicate of $p_1$ and therefore does not have to be considered any further. Thus, in the following, we only need to match the intervals $[1,3[$ for $a$ and $[2,4]$ for $b$ of $p_2$. This corresponds to a *rest predicate* of $((a \geq 1) \wedge (a < 3) \wedge (b \geq 2) \wedge (b \leq 4))$. Matching this with intervals $[9,14]$ and $[2,8]$ of the second subpredicate $((a \geq 9) \wedge (a \leq 14) \wedge (b \geq 2) \wedge (b \leq 8))$ of $p_1$ does not yield any additional matches. So we continue with the third and final subpredicate $((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6))$ of $p_1$, which yields intervals $[0,5]$ for $a$ and $[1,6]$ for $b$. These intervals completely contain the intervals for $a$ and $b$ of the rest predicate of $p_2$. Thus, the resulting rest predicate of $p_2$ is empty and the whole predicate has been matched. This yields the fact that $p_1$ is implied by $p_2$.

The above example illustrates the case of a complete match between predicates. In case of a mismatch, the resulting rest predicate will not be empty. In order to appropriately relax the query predicate, either the rest predicate or the original query predicate has to be disjunctively added to the stream predicate. While the former solution does not cause any additional overlap in the stream predicate, the latter always introduces only one additional subpredicate. Overlap is not that critical for predicate matching and evaluation as indicated by our performance tests in Section 6 and due to our short-circuit optimization introduced in Section 4.2. In contrast, the number of subpredicates has direct impact on the efficiency of matching and evaluation algorithms and should therefore be kept small. Consequently, we choose to add the original query predicate in case of a mismatch. Note that, in each case, no unnecessary parts of the data space are added to the predicate during relaxation, as opposed to the HCR algorithm.

**Algorithm 4** Exact Matching (EM)

**Input:** Stream predicate $p$ and query predicate $p'$.
**Output:** $(\text{true}, p)$, if $p' \Rightarrow p$; $(\text{false}, \overline{p})$, where $\overline{p}$ is a relaxed version of $p$ such that $p' \Rightarrow \overline{p}$, otherwise.

1. *Compare subpredicates.* Compare each conjunctive subpredicate $c'$ in $p'$ to each conjunctive subpredicate $c$ in $p$ in step 2. Optionally, perform the quick check of Algorithm1 for $p$ and $c'$ and only consider $c'$ in the following if the quick check returns false.
2. *Compare dimensions.* Compare each dimension $d$ of $c$ to the corresponding dimension $d'$ of $c'$, i. e., in the following, $d = d'$ holds. Let $I_c^d$ and $I_{c'}^{d'}$ be the intervals defined by $c$ and $c'$ in dimensions $d$ and $d'$, respectively. We distinguish four cases:
   (a) If $I_c^d$ and $I_{c'}^{d'}$ are disjoint, continue with the next conjunctive subpredicate $c$ in $p$ to be matched with $c'$.
   (b) If $I_{c'}^{d'}$ is completely contained in $I_c^d$, continue with the next pair of dimensions from $c$ and $c'$.
   (c) If $I_c^d$ is completely contained in $I_{c'}^{d'}$, split $c'$ along dimension $d'$ into the part $c'_i$ that is overlapping with $c$ in dimension $d'$ and the remaining parts $c'_{o1}$ and $c'_{o2}$. Enqueue $c'_{o1}$ and $c'_{o2}$ in a queue $Q'_c$.
   (d) If $I_c^d$ and $I_{c'}^{d'}$ overlap, split $c'$ along dimension $d'$ into the part $c'_i$ that is overlapping with $c$ in dimension $d'$ and the remaining part $c'_o$. Enqueue $c'_o$ in a queue $Q'_c$.

   Match the remaining parts of $c'$ contained in $Q'_c$ with the remaining conjunctive subpredicates $c$ in $p$ as above. As soon as $Q'_c$ does not contain any more unmatched parts of $c'$, continue with the next $c'$ in $p'$ from the beginning. If not all parts of $c'$ could be matched, disjunctively add $c'$ to $p$.
3. *Return result.* Return $(\text{true}, p)$, if no changes have been made to $p$, i. e., no conjunctive subpredicates $c'$ of $p'$ have been disjunctively added to $p$. Otherwise, return $(\text{false}, \overline{p})$, where $\overline{p}$ is the modified version of $p$ after the addition of one or more conjunctive subpredicates $c'$ of $p'$.

---

To demonstrate the case of a partial match, we introduce predicate $p_3$.

$p_3$: $((a \geq 4) \wedge (a \leq 13) \wedge (b \geq 7) \wedge (b \leq 10))$

We now want to match $p_3$ with $p_1$. The situation is illustrated in Figure 5. Considering the intervals $[4,13]$ for $a$ and $[7,10]$ for $b$ of $p_3$, we want to match these with intervals $[3,12]$ for $a$ and $[0,5]$ for $b$ of the first subpredicate $((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5))$ of $p_1$. Since the intervals for $b$ of both predicates are disjoint, no match is found. We therefore continue by matching the intervals of $p_3$ with intervals $[9,14]$ for $a$ and $[2,8]$ for $b$ of the second subpredicate of $p_1$. Since the intervals for $a$ of both predicates overlap, we have to split the interval for $a$ of $p_3$ into the two intervals $[4,9[$ and $[9,13]$. As the first interval does not overlap with the currently examined subpredicate of $p_1$, this part of $p_3$ is saved for future comparisons with other subpredicates of $p_1$. The second interval completely overlaps with the current subpredicate of $p_1$ and is therefore compared with this subpredicate in the remaining dimensions. Since there is also an overlap of the intervals for $b$, another split is made. The resulting intervals for $b$ are $[7,8]$ and $]8,10]$. After the decomposition, $p_3$ looks as follows:

$$((a \geq 4) \wedge (a < 9) \wedge (b \geq 7) \wedge (b \leq 10)) \vee$$
$$((a \geq 9) \wedge (a \leq 13) \wedge (b > 8) \wedge (b \leq 10)) \vee$$
$$((a \geq 9) \wedge (a \leq 13) \wedge (b \geq 7) \wedge (b \leq 8))$$

The three disjoint parts of this predicate are indicated by the dotted lines within the rectangle of $p_3$ in Figure 5. The third of the three disjoint subpredicates of the decomposed predicate $p_3$ above has

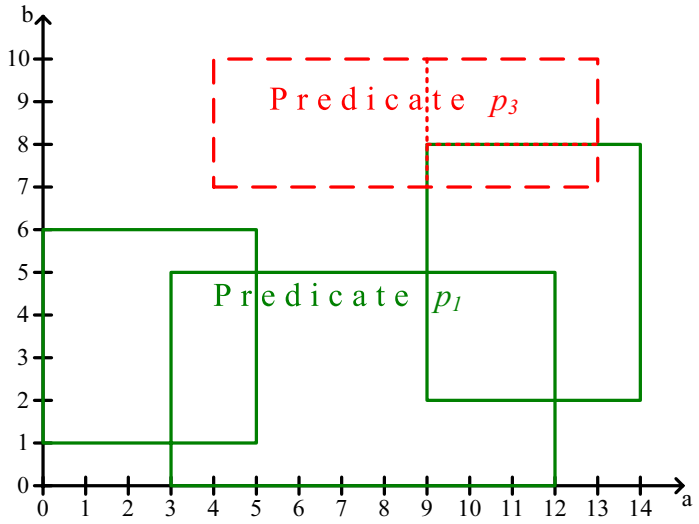



**Figure 5:** Partial match of predicates $p_1$ (solid boxes) and $p_3$ (dashed box)

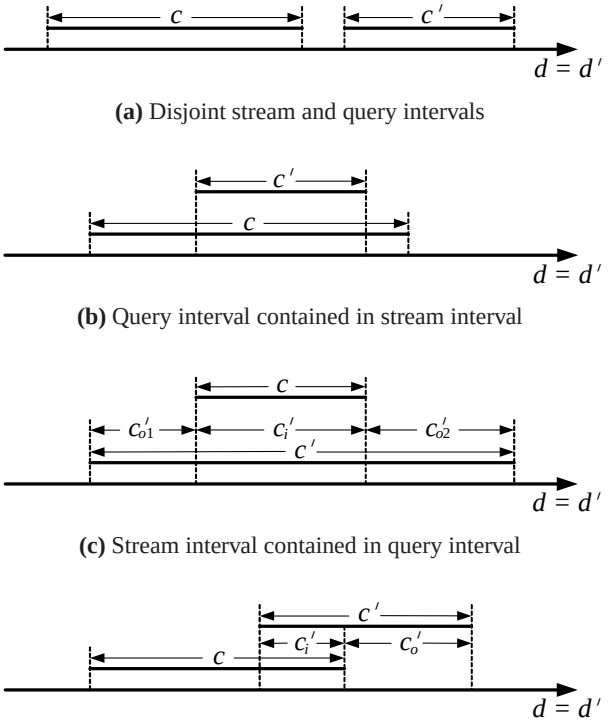
now been completely matched with the second subpredicate of $p_1$. The other two subpredicates of the decomposed predicate $p_3$ have to be matched with the remaining subpredicates of $p_1$. This yields no further matches. Therefore, $p_3$ does not match $p_1$. In order to force the match, $p_1$ has to be relaxed by disjunctively adding the whole subpredicate of $p_3$, which in our example happens to be $p_3$ itself. The relaxed predicate looks as follows:

$$((a \geq 3) \wedge (a \leq 12) \wedge (b \geq 0) \wedge (b \leq 5)) \vee$$
$$((a \geq 9) \wedge (a \leq 14) \wedge (b \geq 2) \wedge (b \leq 8)) \vee$$
$$((a \geq 0) \wedge (a \leq 5) \wedge (b \geq 1) \wedge (b \leq 6)) \vee$$
$$((a \geq 4) \wedge (a \leq 13) \wedge (b \geq 7) \wedge (b \leq 10))$$

Another example for a partial match is shown in Figure 4. In this case, the EM algorithm would split predicate $p_2$ into the three disjoint parts $(a \geq 3) \wedge (a \leq 8) \wedge (b \geq 2) \wedge (b \leq 4)$, $(a \geq 1) \wedge (a < 3) \wedge (b \geq 3) \wedge (b \leq 4)$, and $(a \geq 1) \wedge (a < 3) \wedge (b \geq 2) \wedge (b < 3)$. The third of these three parts remains unmatched.

The EM algorithm needs to compare the intervals defined by stream subpredicates in the various dimensions of the data space to the intervals defined by query subpredicates in the corresponding dimensions. This dimension comparison distinguishes four cases which are illustrated in Figure 6. The intervals defined by the stream and the query subpredicate may either be disjoint (Figure 6(a)), overlapping (Figure 6(d)), or contained in one another (Figures 6(b) and 6(c)). The dimension comparison is shown in detail in Algorithm 9 on page 46.

We have developed three split strategies for use with the EM algorithm which differ in the order in which they process unmatched parts of previously split subpredicates. These differences have impact on the average execution time—though not on the theoretical time complexity—and on the space complexity of the EM algorithm. We call the (unmatched) parts resulting from the splitting of a conjunctive subpredicate the (unmatched) *subparts* of the subpredicate and the original subpredicate itself the corresponding *superpart* of these subparts. Note that each subpart and each superpart by itself constitutes a conjunctive subpredicate. A schematic illustration of the splitting strategies for a small example with a query subpredicate $c'$, $n = 3$ stream subpredicates, and $d = 2$ dimensions in the data space is shown in Figure 7. The figure assumes the worst case of the query subpredicate and each of its resulting subparts in turn being split into $2d$ unmatched parts in the course of the matching. Note that this is a conservative approximation since, in reality, it is not possible that each of the subparts originating from the same superpart is split into $2d$ unmatched parts. This is due to the fact that all the subparts of the same superpart




**(a)** Disjoint stream and query intervals

**(b)** Query interval contained in stream interval

**(c)** Stream interval contained in query interval

**(d)** Overlapping stream and query intervals

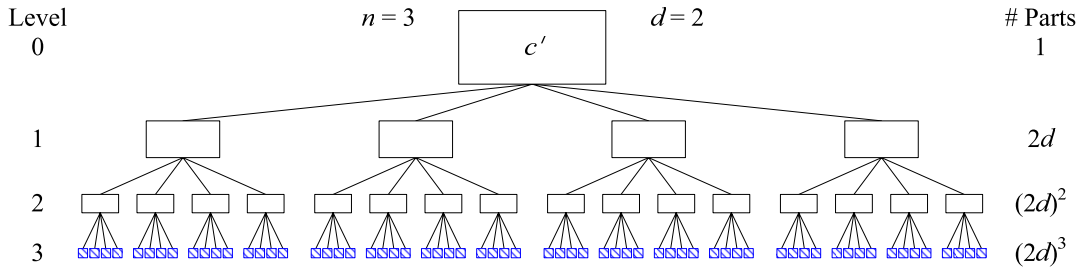
**Figure 6:** Cases distinguished during dimension comparison

are pairwise disjoint and $2d$ unmatched parts only result from the splitting if the stream subpredicate is completely contained in the query subpredicate to be split. However, a stream subpredicate cannot be completely contained in more than one of a set of disjoint subparts. Nevertheless, we use this approximation for the EM algorithm, yielding exponential worst case time complexity. Note that due to the fact that the general implication problem involving disjunctive predicates is proven to be NP-hard [27], the actual worst case time complexity, although lower than our approximation, will still be exponential.

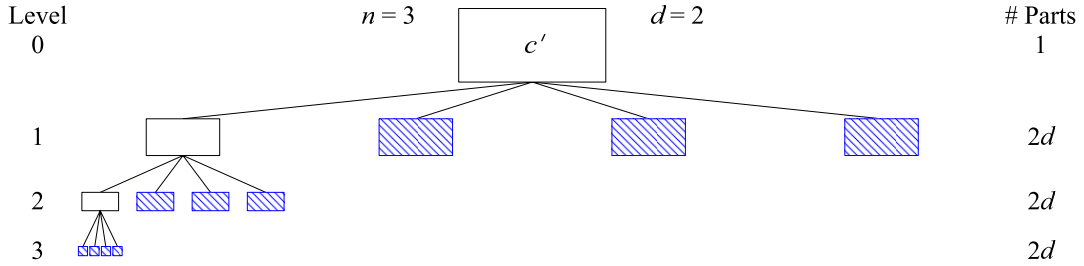
### Breadth-First Split Strategy (BFS)

The breadth-first split strategy (BFS) starts by splitting a query subpredicate $c'$ by comparing it to the first stream subpredicate $c$ of a stream predicate $p$. The remaining unmatched parts of $c'$ are then split in turn by comparing them to the next stream subpredicate in $p$ until no more unmatched parts remain, in which case the matching succeeds, or no more stream subpredicates remain, in which case the matching fails and the stream predicate needs to be relaxed. The resulting tree of unmatched parts for the worst case is shown in Figure 7(a). The tree has a maximum of $n+1$ levels, one for each stream subpredicate and one for the root, which is the original query subpredicate $c'$. On each level, every subpart on that level is split into $2d$ new subparts that belong to the next level of the tree. Thus, the leaf level contains a total of $(2d)^n$ subparts. The EM algorithm with BFS strategy is shown in detail in Algorithm 10 on page 47.

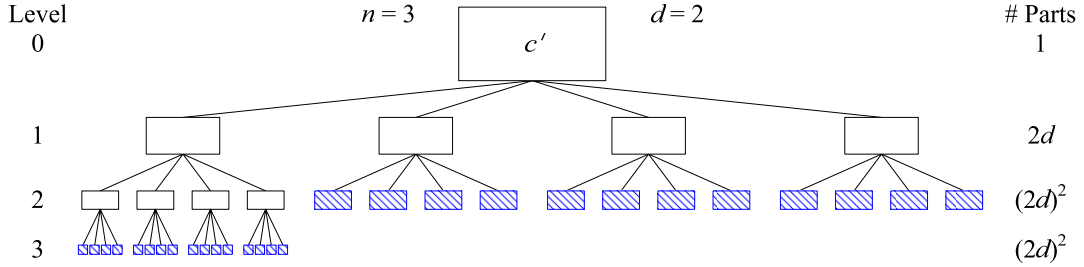
Using the BFS strategy leads to building up the aforementioned tree in a breadth-first manner, i. e., each level of the tree is completely filled with all subparts belonging to that level before the next level is built. This leads to a maximum in computational cost and memory usage, since every possible split is performed and the maximum possible number of subparts is created and needs to be stored.




**(a)** Breadth-first split strategy (BFS)




**(b)** Depth-first split strategy (DFS)




**(c)** Mixed split strategy (MIX)

**Figure 7:** Exact matching algorithm split strategies

### Depth-First Split Strategy (DFS)

The situation can be vastly improved by using the depth-first split strategy (DFS), which is illustrated in Figure 7(b). Using this strategy, only one of the subparts on each level of the tree is split further by comparing it to the next stream subpredicate. If all parts on one level of the tree have been matched, the matching continues with the next subpart on the parent level of the tree. In the worst case, each level of the tree contains $2d$ parts. Thus, the tree has a total of $(n-1)\cdot(2d-1)+2d$ leave nodes. The EM algorithm with DFS strategy is shown in detail in Algorithm 11 on page 48.

Using the DFS strategy leads to building up the aforementioned tree in a depth-first manner. Since matched parts are removed from the tree, each level contains at most $2d$ parts at each time, except for the root level which always contains at most one part—the original query subpredicate $\acute{c}$. This leads to a reduction of the space complexity to quadratic in $d$ and to linear in all other variables. Further, it reduces the average execution time of the algorithm since mismatches can be detected early and therefore—compared to the BFS strategy—many comparisons between subparts and stream subpredicates can be saved. On the other hand, the strategy is a little more difficult to implement and cannot be supported as well by a multi-dimensional index structure (see Section 3.6) as the BFS strategy. This is due to the fact that the index maintenance overhead is much larger since, on each level of the tree, the algorithm must keep track of the stream subpredicates that have not yet been used for matching. This can be done by

storing a copy of the index for each tree level with the stream subpredicates that have already been used for matching removed. Alternatively, using only one single index, subpredicates can be deleted from and reinserted into the stream predicate index as needed in the course of the algorithm. A subpredicate is deleted from the index when descending one level in the tree and reinserted when going back up to the parent level. A more sophisticated solution could be achieved by developing an index structure that allows marking index entries as active or inactive during runtime without causing any index reorganization. This would avoid the need for copying the index or deleting and reinserting index entries.

**Mixed Split Strategy (MIX)**

The mixed split strategy (MIX) is a compromise between the BFS and DFS strategies. While its memory consumption is higher than that of the DFS strategy and it might perform some unnecessary splits before detecting a mismatch, it still needs much less memory than the BFS strategy and can be implemented with less data structure maintenance overhead than the DFS strategy. The idea is to always compare all the subparts that belong to the same superpart against the next stream subpredicate at once, splitting them as needed. The approach is illustrated in Figure 7(c). Using this strategy, the tree of subparts has one node in level 0, $2d$ nodes in level 1, and $(2d)^2$ nodes in all remaining levels in the worst case. Therefore, the number of leave nodes is $(n-2)\cdot(2d-1)\cdot 2d+(2d)^2$ in the worst case. The EM algorithm with MIX strategy is shown in detail in Algorithm 12 on page 49.

The MIX strategy detects mismatches potentially later than the DFS strategy but still earlier than the BFS strategy. It furthermore has cubic space complexity in $d$, whereas the DFS and BFS strategies have quadratic and exponential space complexities in $d$, respectively. This will be detailed in the space complexity analyis in Section 5. The MIX strategy is supposed to be slower than the DFS strategy for large numbers of stream subpredicates $n$ and dimensions $d$ if many mismatches occur, since the DFS strategy then benefits from detecting mismatches earlier. If only few mismatches occur, however, the reduction in maintenance overhead achieved in the MIX strategy might cause it to outperform the DFS strategy. Like the DFS strategy, the MIX strategy is a little more difficult to implement than the BFS strategy and cannot be supported as well by a multi-dimensional index structure as the BFS variant since the index maintenance overhead is much higher. However, all three variants are comparatively easy to implement and the early detection of mismatches in the DFS and MIX strategies usually outweighs the lack of efficient index support. Performance could however still be improved by developing a specialized index structure that can be efficiently employed in the context of predicate matching as it is described here.

Again, the quick check presented in Section 3.2 can be executed in combination with the EM approach to check for matching subpredicates in advance before starting the more complex relaxation algorithm. The worst case time complexity of the EM algorithm is in $O(\sum_{i=0}^{m-1}\sum_{j=0}^{n+i-1}(2d)^j\cdot d)$ without quick check and therefore in $O(d\cdot(2d)^{m+n})$, with $m$, $n$, and $d$ as defined in Section 3.3. We will present all details on the complexity analysis in Section 5.

The advantages of the exact solution are that it determines matches between predicates in an exact way, i e., all existing matches are found—as opposed to the heuristics—and no false matches are reported. Therefore, the approach does not cause any unnecessary predicate relaxations. Also, the non-matching parts of the query predicate are exactly identified. The major disadvantage of the exact solution is its high algorithmic complexity, which is exponential in the number of subpredicates in the predicates to be matched. This might slow down the optimization process considerably for predicates with many disjunctions and makes the algorithm inapplicable for larger problem sizes. In such cases, the heuristics have to be used.

### 3.6 Multi-Dimensional Indexing

The previously described algorithms can be supported by multi-dimensional indexing as follows. The quick check of Algorithm 1 can use such an index to retrieve all conjunctive subpredicates of the stream predicate $p$ that overlap with the current conjunctive subpredicate $c'$ of the query predicate $p'$. Only the overlapping subpredicates instead of all subpredicates of the stream predicate have to be iterated and compared to $c'$ in the course of the algorithm.

Algorithm 2 without quick check offers no possibility for indexing. Algorithm 3 cannot be supported directly by an index either. It must always take all subpredicates of the stream predicate into consideration for relaxation since it is not clear—and cannot be decided using a multi-dimensional index—which of these subpredicates should be relaxed to minimize the cost. However, as with all matching algorithms presented in this paper, the quick check that can be combined with these algorithms can use an index.

The exact solution of Algorithm 4 can benefit the most from a multi-dimensional index. In addition to the indexed quick check, the index can also be used during the splitting step to quickly identify the subpredicates of the stream predicate that overlap with the current subpredicate of the query predicate. Only these overlapping subpredicates have to be considered during splitting instead of iterating over all subpredicates of the stream predicate. However, depending on the index used, index maintenance may outweigh its benefits for the exact matching algorithm when using the depth-first or mixed split strategy.

Various multi-dimensional index structures with different characteristics have been developed over the years [15]. In this paper, we use different variants of the R-tree [3, 17]. Since R-trees index boxes in multi-dimensional space, they can naturally index the multi-dimensional intervals described by our predicates without any additional post-processing. Arbitrary intervals that form polygons rather than rectangular boxes in the data space are not dealt with in this paper but could be approximated by minimum bounding boxes in an R-tree. This may necessitate a non-trivial post-processing step in addition to an index access. Note that the index does not need to comprise all dimensions of the data space. It only needs to contain the dimensions that are actually referenced by the indexed predicate. These are usually few, compared to the potentially many dimensions of the data space. If the dimensions referenced by predicates can change dynamically, e. g., due to predicate relaxation as in our DSMS scenario, a subset of the dimensions of the data space containing the most selective dimensions can be indexed. This yields a quick and efficient reduction of the data volume. If a predicate references non-indexed dimensions, these can be evaluated conventionally. Alternatively, the index can be rebuilt each time new dimensions are introduced.

## 4 Predicate Evaluation

Apart from predicate matching, efficient predicate evaluation is also important in a DSMS. The goal is to evaluate a given predicate against as many data items per time unit as possible, i. e., achieve a high throughput. In the following, two approaches for predicate evaluation are presented.

### 4.1 Standard Evaluation

We use the term *standard evaluation* (SE) to denote a simple sequential scan that is shown in Algorithm 5. It evaluates a given predicate $p$ against a given data item $i$ by iterating over the conjunctive subpredicates $c$ of $p$ and testing for each dimension, whether the value of $i$ in that dimension lies within the interval defined for the same dimension in $c$. As soon as a subpredicate containing the data item, i. e., containing the values of $i$ in each dimension, is found, the algorithm terminates and returns true. Only if, after inspecting all conjunctive subpredicates $c$ of $p$, no subpredicate containing $i$ could be found, it returns false.

**Algorithm 5** Standard Evaluation (SE)

**Input:** Predicate $p$ and data item $i$.
**Output:** true, if $i$ satisfies $p$; false, otherwise.

1. *Iterate subpredicates.* Compare $i$ to each conjunctive subpredicate $c$ in $p$. For each such subpredicate, compare each dimension $d_c$ in $c$ to the corresponding dimension $d_i$ in $i$ in step 2.
2. *Compare dimensions.* For each pair of corresponding dimensions $d_c$ and $d_i$, i. e., $d_c = d_i$, check if the value for $d_i$ in $i$ lies within the interval defined for $d_c$ in $c$. If so, continue with the next dimension in $c$. Otherwise, continue with the next conjunctive subpredicate $c$ in $p$.
3. *Return result.* As soon as, for a certain conjunctive subpredicate $c$ in $p$, the intervals of all dimensions $d_c$ in $c$ contain the values of all the corresponding dimensions $d_i$ in $i$, return true. If there is no conjunctive subpredicate $c$ in $p$ such that the above condition is satisfied, return false.

The worst case complexity of the standard evaluation algorithm is in $O(n \cdot d)$, where $n$ denotes the number of conjunctive subpredicates in the predicate $p$ to be evaluated and $d$ is the number of dimensions in the data space. Algorithm 13 on page 50 shows a pseudocode representation of the standard evaluation algorithm.

## 4.2 Index-based Evaluation

Considering the facts that the exact matching algorithm is only applicable for small problem sizes and the approximate results of the HCR algorithm are often not desirable, a switch to the HSR algorithm for larger problem sizes, i. e., larger numbers of dimensions and subpredicates, seems necessary in many cases. Since this algorithm—with as well as without quick check—can introduce a considerable number of additional disjunctions in predicates, the standard evaluation algorithm above will quickly become inefficient. Therefore, an optimized predicate evaluation strategy that better handles large numbers of subpredicates is needed.

Predicate evaluation can benefit even more from multi-dimensional indexing than the predicate matching algorithms of Section 3. We call the evaluation algorithm with index support *index-based evaluation* (IE). It differs from standard evaluation in that it does not iterate over the conjunctive subpredicates of the predicate to be evaluated. Instead, it uses a multi-dimensional index on the predicate, i. e., the predicate is represented by a multi-dimensional index structure. To evaluate the predicate against a data item, the algorithm simply executes the containment method of the index with the data item as its only parameter. The evaluation is then performed completely by the index, returning true if the predicate covers the data item and false otherwise. If a predicate references a very large number of dimensions, it is possible to index only a subset, i. e., the most selective of these dimensions to obtain a quick index-based prefiltering with only a small number of false drops in the result. The remaining dimensions can then be evaluated traditionally using standard evaluation. Furthermore, predicate evaluation can be dynamically adapted to available computing resources by limiting the index level to which the evaluation descends before deciding whether a data item satisfies the indexed predicate. This will in general lead to approximate results, i. e., the resulting data stream will contain data items that do not satisfy the original predicate. In a DSMS scenario, this can be corrected by an additional filtering step at another peer in the network. Note that this approach does not remove any qualifying data items from the stream. In the remainder of the paper, we always assume that all dimensions referenced in a predicate are indexed if using an index and that the index-based evaluation is exact, i. e., does not use the dynamic adaptation described above.

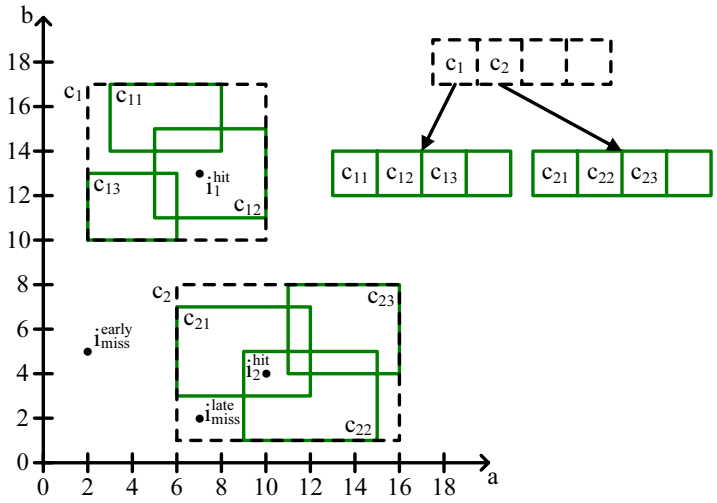



**Figure 8:** Index-based predicate evaluation

It has been noted several times in the literature that multi-dimensional index structures are not suitable for predicate indexing in active databases and publish&subscribe systems [19] because overlap between regions tends to be high and multi-dimensional index structures are prone to deteriorate under such circumstances. Searching the regions containing a certain multi-dimensional point in an R-tree with higly overlapping regions could, for example, lead to a full tree traversal in the worst case. However, this is only true if *all* containing regions for a data item have to be returned as in traditional use cases. Here, we index disjunctive predicates and the first hit determines the result, i. e., it suffices to determine whether there is *at least one* region containing the data item. If such a region is found, the search can be stopped and true can be returned as the evaluation result. If no containing region is present in the index, the mismatch is likely to be detected early on anyway. Using this *short-circuit* optimization on an R-tree proves to be a beneficial evaluation strategy as we will see in Section 6. Even better results could be achieved by using adaptive index-structures like the TV-tree [23] which—in contrast to R-trees—are able to dynamically adapt the set of indexed dimensions. This can help to avoid indexing unbounded dimensions in a predicate, which is desirable since unbounded dimensions in the index can cause excessive overlap between index regions and therefore degrade index performance. Examining the use of such advanced indexing techniques as well as the direct integration of application specific improvements and tuning into the index itself are the subject of future research.

Figure 8 illustrates an example where a disjunctive predicate consisting of 6 conjunctive subpredicates is represented by an R-tree and evaluated against 4 different data items. The figure shows the graphical representation of the predicate in the data space and the corresponding R-tree. For data item $i_{miss}^{early}$, the fact that the data item does not satisfy the predicate can already be determined by matching it with the root of the index tree. In contrast, for $i_{miss}^{late}$, the mismatch is not detected before the leaf level of the index tree. While $i_1^{hit}$ can be identified as a match by traversing one single path in the index tree, $i_2^{hit}$ would normally require to visit two different leaf nodes. However, by using our short-circuit optimization, the evaluation can be stopped and true can be returned after the first matching leaf node has been found.

| Variable | Description |
|---|---|
| $p$ | disjunctive stream predicate |
| $p'$ | disjunctive query predicate |
| $c$ | conjunctive subpredicate of stream predicate $p$ |
| $c'$ | conjunctive subpredicate of query predicate $p'$ |
| $n$ | number of conjunctive subpredicates $c$ in $p$ |
| $m$ | number of conjunctive subpredicates $c'$ in $p'$ |
| $d$ | number of dimensions in the data space |

**Table 2:** Variables used during complexity analysis

# 5 Complexity Analysis

In this section, we analyze the best, average, and worst case time and space complexities of the matching and evaluation algorithms introduced in Sections 3 and 4.

## 5.1 Prerequisites

Before starting with the complexity analysis, we introduce some general prerequisites. Table 2 lists the variables used throughout the complexity analysis together with their meaning. We use a single dimension in the data space as the most fine-grained unit for time and space complexity analysis. For time complexity, comparing two dimensions is the most fine-grained unit. Note that comparing two dimensions always consists of comparing the upper and lower bounds of the two dimensions, i. e., always leads to two value comparisons. Since this is the same for each comparison between two dimensions, we abstract from the actual value comparisons and choose the comparison between two dimensions as the most fine-grained unit for time complexity analysis. Equally, for space complexity analysis, the most fine-grained unit is the memory required to store the information associated with a dimension in the data space. Again, for each dimension, two values—its upper and its lower bound—have to be stored. However, again we abstract from these values and choose the memory needed for storing all the information of a single dimension as the most fine-grained unit for space complexity analysis.

Note that in the algorithm descriptions, $d$ denotes a single dimension of the data space $D$, whereas during complexity analysis, $d$ denotes the number of dimensions in the data space $D$, i. e., $d = |D|$.

## 5.2 Quick Check (QC)

### 5.2.1 Time Complexity

- Best Case:
  The best case for the QC algorithm occurs when the query subpredicate $c'$ already implies the first stream subpredicate $c$ of $p$, and $c$ only references one of the $d$ dimensions of the data space. Then, only one single comparison between two dimensions of the data space is necessary. In this case, the time complexity of the QC algorithm is constant and is in

$$\Omega(1)$$

- Worst Case:
  The worst case for the QC algorithm occurs when the query subpredicate $c'$ does not imply any of the $n$ stream subpredicates $c$ of $p$, and each stream subpredicate references all $d$ dimensions of the data space. Then, for each of the $n$ conjunctive subpredicates $c$ in $p$, all $d$ dimensions of the data

space need to be considered. In this case, the time complexity of the QC algorithm is linear in $n$ and $d$, and is in

$$O(n \cdot d)$$

- Average Case:
  The average case for the QC algorithm occurs when the query subpredicate $c'$ is found to imply a stream subpredicate $c$ in $p$ after checking half of the $n$ subpredicates $c$ in $p$, and each subpredicate $c$ in $p$ on average references half of the $d$ dimensions of the data space. In this case, the time complexity of the QC algorithm is linear in $n$ and $d$, and is in

$$\Theta\left(\frac{n}{2} \cdot \frac{d}{2}\right) = \Theta(n \cdot d)$$

### 5.2.2 Space Complexity

- Best Case:
  The best case for the QC algorithm occurs when each stream subpredicate $c$ in $p$ as well as the query subpredicate $c'$ only reference one of the $d$ dimensions of the data space. Then, for each of the $n$ stream subpredicates $c$ in $p$ as well as for the query subpredicate $c'$, the information for only one dimension needs to be stored. In this case, the space complexity of the QC algorithm is linear in $n$ and is in

$$\Omega(n+1)$$

- Worst Case:
  The worst case for the QC algorithm occurs when each stream subpredicate $c$ in $p$ as well as the query subpredicate $c'$ references all of the $d$ dimensions of the data space. Then, for each of the $n$ stream subpredicates $c$ in $p$ as well as for the query subpredicate $c'$, the information for all $d$ dimensions of the data space needs to be stored. In this case, the space complexity of the QC algorithm is linear in $n$ and $d$, and is in

$$O((n+1) \cdot d)$$

- Average Case:
  The average case for the QC algorithm occurs when each stream subpredicate $c$ in $p$ as well as the query subpredicate $c'$ references half of the $d$ dimensions of the data space. Then, for each of the $n$ stream subpredicates $c$ in $p$ as well as for the query subpredicate $c'$, the information for half of the $d$ dimensions of the data space needs to be stored. In this case, the space complexity of the QC algorithm is linear in $n$ and $d$, and is in

$$\Theta\left((n+1) \cdot \frac{d}{2}\right)$$

### 5.2.3 Summary

The QC algorithm is an efficient algorithm for quickly determining obvious matches of a query subpredicate with a stream predicate. Its time and space complexities are at most linear. The complexities of the index-based QC algorithm depend on the complexities of the employed index structure.

## 5.3 Heuristics with Simple Relaxation (HSR)

### 5.3.1 Time Complexity

For the HSR algorithm without quick check, the best, worst, and average time complexities are linear in $m$ and are in $\Omega(m)$, $O(m)$, and $\Theta(m)$, respectively. This is due to the fact that, with the quick check deactivated, the HSR algorithm simply iterates over all $m$ conjunctive subpredicates in the query predicate and disjunctively adds them to the stream predicate. The situation is different with the quick check activated as is described in the following.

- Best Case:
  The best case for the HSR algorithm with quick check occurs if, for each of the $m$ query subpredicates $c'$ in $p'$, the quick check finds a matching subpredicate in the first stream subpredicate $c$ in $p$, and $c$ only references one of the $d$ dimensions of the data space. Then, the algorithm only has to iterate over all $m$ subpredicates of the query predicate and to compare two dimensions for each query subpredicate. In this case, the time complexity of the HSR algorithm with quick check is linear in $m$ and is in
  $$\Omega(m)$$

- Worst Case:
  The worst case for the HSR algorithm with quick check occurs if, for each of the $m$ query subpredicates $c'$ in $p'$, the quick check iterates over all of the $n$ stream subpredicates $c$ in $p$ without finding a match, and each subpredicate in the stream predicate references all of the $d$ dimensions of the data space. Since each of the unmatched query subpredicates is disjunctively added to the stream predicate, the number of conjunctive subpredicates in the stream predicate increases by one each time the algorithm starts to consider the next conjunctive subpredicate of the query predicate. Therefore, the number of comparisons between dimensions can be estimated as
  $$\begin{aligned} \sum_{i=0}^{m-1} (n+i)\cdot d &= \left(m\cdot n + \sum_{i=0}^{m-1} i\right)\cdot d \\ &\overset{\substack{arith.\\series}}{=} \left(m\cdot n + \frac{(m-1)\cdot m}{2}\right)\cdot d \\ &\leq \left(m\cdot n + m^2\right)\cdot d \end{aligned}$$

  Consequently, the worst case time complexity of the HSR algorithm with quick check is quadratic in $m$, linear in $n$ and $d$, and is in
  $$O\left(\left(m\cdot n + m^2\right)\cdot d\right)$$

- Average Case:
  The average case for the HSR algorithm with quick check occurs if, for each of the $m$ query subpredicates $c'$ in $p'$, the quick check iterates over half of the $n$ stream subpredicates $c$ in $p$ before finding a match, and each subpredicate in the stream predicate references half of the $d$ dimensions of the data space. Furthermore, we assume that half of the $m$ query subpredicates remain unmatched and are therefore disjunctively added to the stream predicate, increasing the number of conjunctive subpredicates in the stream predicate by one for half of the query subpredicates.

Therefore, the number of comparisons between dimensions can be estimated as

$$\begin{aligned}\sum_{i=0}^{m-1}\left(\frac{n+\frac{i}{2}}{2}\right)\cdot\frac{d}{2} &= \left(m\cdot n+\frac{1}{2}\cdot\sum_{i=0}^{m-1} i\right)\cdot\frac{d}{4}\\ &\overset{\substack{arith.\\series}}{=} \left(m\cdot n+\frac{1}{2}\cdot\frac{(m-1)\cdot m}{2}\right)\cdot\frac{d}{4}\\ &\leq \left(m\cdot n+m^2\right)\cdot d\end{aligned}$$

Consequently, the average case time complexity of the HSR algorithm with quick check is quadratic in $m$, linear in $n$ and $d$, and is in

$$\Theta\left(\left(m\cdot n+m^2\right)\cdot d\right)$$

### 5.3.2 Space Complexity

The space complexities of the HSR algorithm are the same, whether the algorithm is executed with the quick check activated or deactivated.

- Best Case:
  The best case for the HSR algorithm occurs when each subpredicate only references one of the $d$ dimensions of the data space. Then, for each of the $m$ query subpredicates $c'$ in $p'$ as well as each of the $n$ stream subpredicates $c$ in $p$, only one dimension needs to be stored. In this case, the space complexity of the HSR algorithm is linear in $m$ and $n$, and is in

  $$\Omega(m+n)$$

- Worst Case:
  The worst case for the HSR algorithm occurs when each subpredicate references all of the $d$ dimensions of the data space. Then, for each of the $m$ query subpredicates $c'$ in $p'$ as well as each of the $n$ stream subpredicates $c$ in $p$, the information for all $d$ dimensions of the data space needs to be stored. In this case, the space complexity of the HSR algorithm is linear in $m$, $n$, and $d$, and is in

  $$O((m+n)\cdot d)$$

- Average Case:
  The average case for the HSR algorithm occurs when each subpredicate references half of the $d$ dimensions of the data space. Then, for each of the $m$ query subpredicates $c'$ in $p'$ as well as each of the $n$ stream subpredicates $c$ in $p$, the information for half of the $d$ dimensions of the data space needs to be stored. In this case, the space complexity of the HSR algorithm is linear in $m$, $n$, and $d$, and is in

  $$\Theta\left((m+n)\cdot\frac{d}{2}\right)$$

### 5.3.3 Summary

The HSR algorithm without quick check is a simple relaxation algorithm with time complexity linear in $m$ and linear space complexity. Combining the HSR algorithm with the QC algorithm does not affect space complexity but leads to quadratic time complexity in $m$ in the worst and average case. It can be noticed that the worst and average case time complexities are the same. Still, the HSR algorithm is a

simple and relatively fast algorithm. The HSR algorithm can only indirectly be supported by an index when using an index-supported quick check. In this case, the complexities of the algorithm depend on the complexities of the employed index structure.

## 5.4 Heuristics with Complex Relaxation (HCR)

### 5.4.1 Time Complexity

- Best Case:
  The best case for the HCR algorithm without quick check occurs if, for each of the $m$ query subpredicates $c'$ of $p'$, the algorithm finds a matching subpredicate in the first stream subpredicate $c$ of $p$, and $c$ only references one of the $d$ dimensions of the data space. Then, the algorithm only has to iterate over all $m$ subpredicates of the query predicate and to compare two dimensions for each query subpredicate. In this case, the time complexity of the HCR algorithm without quick check is linear in $m$ and is in
  $$\Omega(m)$$
  The best case complexity of the HCR algorithm remains the same with the quick check activated. The best case then occurs when the quick check finds a match for each query subpredicate $\acute{c}$ in $p'$ when comparing it to the first stream subpredicate $c$ in $p$ and furthermore, $c$ references only one of the $d$ dimensions of the data space. Therefore, the best case complexity of the HCR algorithm with quick check is the same as the best case complexity of the QC algorithm.

- Worst Case:
  The worst case for the HCR algorithm without quick check occurs if, for each of the $m$ query subpredicates $c'$ of $p'$, the algorithm iterates over all of the $n$ stream subpredicates $c$ of $p$ without finding a match, and each subpredicate in the stream predicate references all of the $d$ dimensions of the data space. Since, for each pair of subpredicates $\acute{c}$ in $p'$ and $c$ in $p$, and for each dimension in the data space, the list of all $d$ dimensions has to be iterated to compute $v$ in line 20 of Algorithm8 on page 45, the worst case time complexity of the HCR algorithm without quick check is linear in $m$ and $n$, quadratic in $d$, and is in
  $$O\left(m \cdot n \cdot d^2\right)$$
  The worst case complexity of the HCR algorithm with quick check has to additionally take into account the worst case complexity of the QC algorithm. Since the quick check is executed before the actual HCR algorithm, and in case the quick check does not yield a match, the normal HCR algorithm is executed, the number of dimension comparisons of the QC algorithm and the HCR algorithm have to be added. This yields
  $$\begin{aligned} m \cdot n \cdot d + m \cdot n \cdot d^2 &= m \cdot n \cdot \left(d + d^2\right) \\ &\leq 2 \cdot m \cdot n \cdot d^2 \end{aligned}$$
  Therefore, the worst case complexity of the HCR algorithm with quick check is still linear in $m$ and $n$, quadratic in $d$, and is in
  $$O\left(m \cdot n \cdot d^2\right)$$

- Average Case:
  The average case for the HCR algorithm without quick check occurs if, for each of the $m$ query subpredicates $c'$ of $p'$, the algorithm iterates over half of the $n$ stream subpredicates $c$ of $p$ before

finding a match, and each subpredicate in the stream predicate references half of the $d$ dimensions of the data space. Furthermore, we assume that half of the $m$ query subpredicates remain unmatched and are therefore relaxed during the execution of the algorithm. Therefore, the average case time complexity of the HCR algorithm without quick check is linear in $m$ and $n$, quadratic in $d$, and is in

$$\Theta\left(m \cdot \frac{n}{2} \cdot \left(\frac{d}{2}\right)^2\right) = \Theta\left(m \cdot n \cdot d^2\right)$$

The average case complexity of the HCR algorithm with quick check is determined analogously to the worst case. For the number of dimension comparisons, this yields

$$\begin{aligned} m \cdot \frac{n}{2} \cdot \frac{d}{2} + \frac{m}{2} \cdot n \cdot \left(\frac{d}{2}\right)^2 &\leq m \cdot n \cdot \left(d + d^2\right) \\ &\leq 2 \cdot m \cdot n \cdot d^2 \end{aligned}$$

Therefore, the average case complexity of the HCR algorithm with quick check is still linear in $m$ and $n$, quadratic in $d$, and is in

$$\Theta\left(m \cdot n \cdot d^2\right)$$

#### 5.4.2 Space Complexity

The space complexities of the HCR algorithm are the same as for the HSR algorithm since both algorithms only need to store the query and the stream predicate but, in contrast to the exact matching algorithm introduced below, do not split and therefore create additional subpredicates.

#### 5.4.3 Summary

The time and space complexities of the HCR algorithm are the same whether the quick check is activated or deactivated. It can also be noticed that the worst and average case time complexities are the same. Although they are quadratic in $d$, the algorithm is still relatively fast compared to the exact solution analyzed below. Like the HSR algorithm, the HCR algorithm can only indirectly be supported by an index when using an index-supported quick check. In this case, the complexities of the algorithm depend on the complexities of the employed index structure.

### 5.5 Exact Matching (EM)

#### 5.5.1 Time Complexity

The time complexity of the EM algorithm is the same for each of the three split strategies introduced in Section 3.5. This is due to the fact that all strategies have to examine the same number of subparts in the best, worst, and average case. They only do so in different order. The time complexity is therefore only considered for the EM algorithm with and without quick check in general, without distinguishing the different split strategies.

- Best Case:
  The best case for the EM algorithm without quick check occurs if, for each of the $m$ query subpredicates $c'$ of $p'$, the algorithm finds a matching subpredicate in the first stream subpredicate $c$ of $p$, and $c$ only references one of the $d$ dimensions of the data space. Then, the algorithm only has to iterate over all $m$ subpredicates of the query predicate and to compare two dimensions for each

query subpredicate. In this case, the time complexity of the EM algorithm without quick check is linear in $m$ and is in

$$\Omega(m)$$

The best case complexity of the EM algorithm remains the same with the quick check activated since the best case then occurs when the quick check finds a match for each query subpredicate $c'$ in $p'$ when comparing it to the first stream subpredicate $c$ in $p$ and furthermore, $c$ references only one of the $d$ dimensions of the data space. Therefore, the best case complexity of the EM algorithm with quick check is the same as the best case complexity of the QC algorithm.

- Worst Case:
  The worst case for the EM algorithm without quick check occurs if, for each of the $m$ query subpredicates $c'$ of $p'$, the algorithm iterates over all of the $n$ stream subpredicates $c$ of $p$ without finding a match, each subpredicate in the stream predicate references all of the $d$ dimensions of the data space, and the intervals defined by the stream subpredicate in each dimension are completely contained in the respective intervals defined by the query subpredicate in the corresponding dimensions. Then, the algorithm has to split the query subpredicate into three parts during each comparison (lines 7–10 in Algorithm 9 on page 46), two of which have to be taken into account in future comparisons. Since each of the unmatched query subpredicates is disjunctively added to the stream predicate, the number of conjunctive subpredicates in the stream predicate increases by one each time the algorithm starts to consider the next conjunctive subpredicate of the query predicate. Therefore, with $d > 0$, the number of comparisons between dimensions can be estimated as

$$\begin{aligned}
\sum_{i=0}^{m-1}\sum_{j=0}^{n+i-1}\left((2d)^j \cdot d\right) &\overset{\text{geom. series}}{=} \sum_{i=0}^{m-1}\left(\frac{(2d)^{n+i}-1}{2d-1}\cdot d\right)\\
&= \frac{d}{2d-1}\cdot\left(\sum_{i=0}^{m-1}\left((2d)^{n+i}-1\right)\right)\\
&= \frac{d\cdot(2d)^n}{2d-1}\cdot\left(\sum_{i=0}^{m-1}(2d)^i - m\right)\\
&\overset{\text{geom. series}}{=} \frac{d\cdot(2d)^n}{2d-1}\cdot\left(\frac{(2d)^m-1}{2d-1} - m\right)\\
&\leq d\cdot(2d)^{m+n}
\end{aligned}$$

Therefore, the worst case time complexity of the EM algorithm without quick check is polynomial in $d$, exponential in $m$ and $n$, and is in

$$O\left(d\cdot(2d)^{m+n}\right)$$

The worst case time complexity of the EM algorithm with the quick check activated has to additionally take into account the worst case time complexity of the quick check algorithm. Since the quick check is executed before the actual EM algorithm and in case the quick check does not yield a match, the normal EM algorithm is executed, the number of dimension comparisons of the quick

check algorithm and the EM algorithm have to be added. This yields

$$\begin{aligned}\sum_{i=0}^{m-1}\left((n+i)\cdot d+\sum_{j=0}^{n+i-1}(2d)^j\cdot d\right) &= \sum_{i=0}^{m-1}(n+i)\cdot d+\sum_{i=0}^{m-1}\sum_{j=0}^{n+i-1}(2d)^j\cdot d\\ &\leq \left(m\cdot n+m^2\right)\cdot d+d\cdot(2d)^{m+n}\\ &= \left(m\cdot n+m^2+(2d)^{m+n}\right)\cdot d\end{aligned}$$

Therefore, the worst case time complexity of the EM algorithm with quick check is polynomial in $d$, exponential in $m$ and $n$, and is in

$$O\left(\left(m\cdot n+m^2+(2d)^{m+n}\right)\cdot d\right)$$

- Average Case:
  The average case for the EM algorithm without quick check occurs if, for each of the $m$ query subpredicates $c'$ of $p'$, the algorithm iterates over half of the $n$ stream subpredicates $c$ of $p$ before finding a match, each subpredicate in the stream predicate references half of the $d$ dimensions of the data space, and the intervals defined by the stream subpredicate in each dimension overlap with the respective intervals defined by the query subpredicate in the corresponding dimensions. Then, the algorithm has to split the query subpredicate into two parts during each comparison (lines 11–15 in Algorithm 9 on page 46), one of which has to be taken into account in future comparisons. Furthermore, we assume that half of the $m$ query subpredicates remain unmatched and are therefore disjunctively added to the stream predicate, increasing the number of conjunctive subpredicates in the stream predicate by one for half of the query subpredicates. Therefore, assuming $d > 4$, the number of comparisons between dimensions can be estimated as

$$\begin{aligned}\sum_{i=0}^{m-1}\sum_{j=0}^{\left\lceil\frac{n+\frac{i}{2}}{2}\right\rceil-1}\left(\frac{d}{2}\right)^j\cdot\frac{d}{2} &\overset{\substack{geom.\\series}}{=} \sum_{i=0}^{m-1}\frac{\left(\frac{d}{2}\right)^{\left\lceil\frac{n+\frac{i}{2}}{2}\right\rceil}-1}{\frac{d}{2}-1}\cdot\frac{d}{2}\\ &= \frac{d}{d-2}\cdot\left(\sum_{i=0}^{m-1}\left(\frac{d}{2}\right)^{\left\lceil\frac{n+\frac{i}{2}}{2}\right\rceil}-m\right)\\ &\leq \frac{d}{d-2}\cdot\left(\sum_{i=0}^{m-1}\left(\frac{d}{2}\right)^{\lceil\frac{n}{2}\rceil}\cdot\left(\frac{d}{2}\right)^{\lceil\frac{i}{4}\rceil}-m\right)\\ &\leq \frac{d\cdot\left(\frac{d}{2}\right)^n}{d-2}\cdot\sum_{i=0}^{m-1}\left(\frac{d}{2}\right)^i\\ &\overset{\substack{geom.\\series}}{=} \frac{d\cdot\left(\frac{d}{2}\right)^n}{d-2}\cdot\frac{\left(\frac{d}{2}\right)^m-1}{\frac{d}{2}-1}\\ &\leq d\cdot\left(\frac{d}{2}\right)^{m+n}\end{aligned}$$

Therefore, the average case time complexity of the EM algorithm without quick check is polynomial in $d$, exponential in $m$ and $n$, and is in

$$\Theta\left(d\cdot\left(\frac{d}{2}\right)^{m+n}\right)$$

The average case complexity of the EM algorithm with quick check is determined analogously to the worst case. For the number of dimension comparisons, this yields

$$\begin{aligned}
\sum_{i=0}^{m-1}\left(\left(\frac{n+\frac{i}{2}}{2}\right)\cdot\frac{d}{2}+\sum_{j=0}^{\left\lceil\frac{n+\frac{i}{2}}{2}\right\rceil-1}\left(\frac{d}{2}\right)^{j}\cdot\frac{d}{2}\right) &= \sum_{i=0}^{m-1}\left(\frac{n+\frac{i}{2}}{2}\right)\cdot\frac{d}{2}+\sum_{i=0}^{m-1}\sum_{j=0}^{\left\lceil\frac{n+\frac{i}{2}}{2}\right\rceil-1}\left(\frac{d}{2}\right)^{j}\cdot\frac{d}{2} \\
&\leq \left(m\cdot n+m^2\right)\cdot d+d\cdot\left(\frac{d}{2}\right)^{m+n} \\
&= \left(m\cdot n+m^2+\left(\frac{d}{2}\right)^{m+n}\right)\cdot d
\end{aligned}$$

Therefore, the average case time complexity of the EM algorithm with quick check is polynomial in $d$, exponential in $m$ and $n$, and is in

$$\Theta\left(\left(m\cdot n+m^2+\left(\frac{d}{2}\right)^{m+n}\right)\cdot d\right)$$

### 5.5.2 Space Complexity

The space complexities of the EM algorithm are the same, whether the algorithm is executed with the quick check activated or deactivated.

- Best Case:
  The best case for all variants of the EM algorithm occurs when each subpredicate only references one of the $d$ dimensions of the data space and no splitting of subpredicates occurs in the course of the algorithm. Then, for each of the $m$ query subpredicates $c'$ of $p'$ as well as each of the $n$ stream subpredicates $c$ of $p$, the information for only one dimension of the data space needs to be stored. In this case, the space complexity of the EM algorithm is linear in $m$ and $n$, and is in

  $$\Omega(m+n)$$

- Worst Case:
  The worst case for the EM algorithm occurs when each subpredicate references all of the $d$ dimensions of the data space, and each query subpredicate $c'$ of $p'$ needs to be split in three parts in each dimension (lines 7–10 of Algorithm 9 on page 46), two of which need to be stored in a queue for later matching. In the worst case, during each split, two additional conjunctive subpredicates are created. Using the BFS strategy, this leads to $2d$ additional subpredicates after comparing all dimensions and $(2d)^n$ additional subpredicates after comparing a query subpredicate $c'$ to all $n$ stream subpredicates $c$ of $p$ in the worst case. This is indicated by the hatched subparts in Figure 7(a). All subparts at the leaf level of the tree need to be stored in memory. Then, for each of the $m$ query subpredicates $c'$ of $p'$ as well as each of the $n$ stream subpredicates $c$ of $p$, the information for all $d$ dimensions of the data space needs to be stored. This is also true for all the conjunctive subpredicates that were newly created in the course of the algorithm due to splitting existing query subpredicates. Therefore, in this case, the space complexity of the EM algorithm is linear in $m$, polynomial in $d$, exponential in $n$, and is in

  $$O\left((m+n+(2d)^n)\cdot d\right)$$

The DFS strategy only produces up to $(n-1)\cdot(2d-1)+2d$ subparts during matching as indicated by the hatched subparts in Figure 7(b). Therefore, its worst case space complexity is linear in $m$ and $n$, quadratic in $d$, and is in

$$O((m+n+(n-1)\cdot(2d-1)+2d)\cdot d)$$

The MIX strategy produces up to $(n-2)\cdot(2d-1)\cdot 2d+(2d)^2$ subparts during matching as indicated by the hatched subparts in Figure 7(c). Therefore, its worst case space complexity is linear in $m$ and $n$, cubic in $d$, and is in

$$O\left(\left(m+n+(n-2)\cdot(2d-1)\cdot 2d+(2d)^2\right)\cdot d\right)$$

- Average Case:
  The average case for the EM algorithm occurs when each subpredicate references half of the $d$ dimensions of the data space and half of the query subpredicates $c'$ of $p'$ need to be split in two parts in half of the dimensions (lines 23–27 of Algorithm 10), one of which needs to be stored in a queue for later matching. In the average case, during each split, one additional conjunctive subpredicate is created. Using the BFS strategy, this leads to $d/2$ additional subpredicates after comparing half of the dimensions and $(d/2)^{n/2}=\sqrt{(d/2)^n}$ additional subpredicates after comparing a query subpredicate to half of the $n$ stream subpredicates on average. Then, for each of the $m$ query subpredicates $c'$ of $p'$ as well as each of the $n$ stream subpredicates $c$ of $p$, the information for half of the $d$ dimensions of the data space needs to be stored. This is also true for all the conjunctive subpredicates that were newly created in the course of the algorithm. Therefore, in this case, the space complexity of the EM algorithm is linear in $m$, polynomial in $d$, exponential in $n$, and is in

$$\Theta\left(\left(m+n+\sqrt{(d/2)^n}\right)\cdot\frac{d}{2}\right)$$

  The DFS strategy only produces $((n-1)/2)\cdot(d/2-1)+d/2$ subparts on average during matching. Therefore, its average case space complexity is linear in $m$ and $n$, quadratic in $d$, and is in

$$\Theta\left(\left(m+n+\frac{n-1}{2}\cdot\left(\frac{d}{2}-1\right)+\frac{d}{2}\right)\cdot\frac{d}{2}\right)$$

  The MIX strategy produces $((n-2)/2)\cdot(d/2-1)\cdot(d/2)+(d/2)^2$ subparts on average during matching. Therefore, its average case space complexity is linear in $m$ and $n$, cubic in $d$, and is in

$$\Theta\left(\left(m+n+\frac{n-2}{2}\cdot\left(\frac{d}{2}-1\right)\cdot\frac{d}{2}+\left(\frac{d}{2}\right)^2\right)\cdot\frac{d}{2}\right)$$

### 5.5.3 Summary

The EM algorithm with BFS strategy shows exponential time and space complexity in the worst and average case. Since the implication problem for disjunctive predicates has been proven to be NP-hard [27], the exponential time complexity cannot be improved substantially. However, it is possible to improve the space complexity to cubic in the number of dimensions in the data space and linear in all other parameters using the MIX instead of the BFS strategy, and even to quadratic in the number of dimensions in the data space and linear in all other parameters using the DFS strategy. The complexities of the index-based EM algorithm depend on the complexities of the employed index structure.

## 5.6 Standard Evaluation (SE)

### 5.6.1 Time Complexity

- Best Case:
  The best case for the SE algorithm occurs when the first conjunctive subpredicate $c$ of the predicate $p$ to be evaluated references only one of the $d$ dimensions of the data space, and the interval defined in $c$ for that dimension contains the value of the corresponding dimension in $i$. Then, only one value to interval comparison is necessary to evaluate the predicate to true. In this case, the time complexity of the SE algorithm is constant and is in

$$\Omega(1)$$

- Worst Case:
  The worst case for the SE algorithm occurs when every conjunctive subpredicate $c$ of the predicate $p$ to be evaluated references all $d$ dimensions of the data space, but none of them matches the data item $i$ and the mismatch in each case is only detected after all dimensions have been considered. Then, all $n$ subpredicates of predicate $p$ and, for each subpredicate, all $d$ dimensions of the data space need to be iterated and compared to the values in $i$. In this case, the time complexity of the SE algorithm is linear in $n$ and $d$, and is in

$$O(n \cdot d)$$

- Average Case:
  The average case for the SE algorithm occurs when every conjunctive subpredicate of the predicate $p$ to be evaluated references half of the $d$ dimensions of the data space, a match is found after considering half of the $n$ subpredicates of predicate $p$, and mismatches are detected after considering half of the dimensions referenced in the corresponding subpredicate on average. Then, half of the $n$ subpredicates of predicate $p$ and, for each subpredicate, one quarter of the $d$ dimensions of the data space need to be iterated and compared to the values in $i$. In this case, the time complexity of the SE algorithm is linear in $n$ and $d$, and is in

$$\Theta\left(\frac{n}{2} \cdot \frac{d}{4}\right) = \Theta(n \cdot d)$$

### 5.6.2 Space Complexity

- Best Case:
  The best case for the SE algorithm occurs when each conjunctive subpredicate $c$ in the predicate $p$ to be evaluated references only one of the $d$ dimensions of the data space. Additionally, the $d$ values of the data item $i$ have to be stored. In this case, the space complexity of the SE algorithm is linear in $n$ and $d$, and is in

$$\Omega(n+d)$$

- Worst Case:
  The worst case for the SE algoritm occurs when each conjunctive subpredicate $c$ in the predicate $p$ to be evaluated references all of the $d$ dimensions of the data space. Additionally, the $d$ values of the data item $i$ have to be stored. In this case, the space complexity of the SE algorithm is linear in $n$ and $d$, and is in

$$O(n \cdot d + d) = O((n+1) \cdot d)$$

| | Time | | |
|---|---|---|---|
| | Best Case | Average Case | Worst Case |
| QC | $\Omega(1)$ | $\Theta(n \cdot d)$ | $O(n \cdot d)$ |
| HSR | $\Omega(m)$ | $\Theta(m)$ | $O(m)$ |
| HSR+QC | $\Omega(m)$ | $\Theta\left(\left(m \cdot n + m^2\right) \cdot d\right)$ | $O\left(\left(m \cdot n + m^2\right) \cdot d\right)$ |
| HCR | $\Omega(m)$ | $\Theta\left(m \cdot n \cdot d^2\right)$ | $O\left(m \cdot n \cdot d^2\right)$ |
| HCR+QC | $\Omega(m)$ | $\Theta\left(m \cdot n \cdot d^2\right)$ | $O\left(m \cdot n \cdot d^2\right)$ |
| EM | $\Omega(m)$ | $\Theta\left(d \cdot (d/2)^{m+n}\right)$ | $O\left(d \cdot (2d)^{m+n}\right)$ |
| EM+QC | $\Omega(m)$ | $\Theta\left(\left(m \cdot n + m^2 + (d/2)^{m+n}\right) \cdot d\right)$ | $O\left(\left(m \cdot n + m^2 + (2d)^{m+n}\right) \cdot d\right)$ |

**Table 3:** Time complexities of predicate matching algorithms

- Average Case:
  The average case for the SE algorithm occurs when each conjunctive subpredicate $c$ in the predicate $p$ to be evaluated references half of the $d$ dimensions of the data space. Additionally, the $d$ values of the data item $i$ have to be stored. In this case, the space complexity of the SE algorithm is linear in $n$ and $d$, and is in

$$\Theta\left(n \cdot \frac{d}{2} + d\right) = \Theta\left(\left(\frac{n}{2} + 1\right) \cdot d\right)$$

#### 5.6.3 Summary

The SE algorithm has linear time and space complexities in the worst and average case. This causes it to slow down for large numbers of subpredicates $n$ and dimensions $d$. Therefore, we have introduced the index-based evaluation strategy alleviating this problem.

### 5.7 Index-based Evaluation (IE)

The best, worst, and average time and space complexities of the index-based evaluation depend on the respective complexities of the employed index structure.

### 5.8 Summary

The time and space complexities of the predicate matching algorithms are summarized in Tables3 and 4, respectively. Since the time complexity of the EM algorithm is the same for all three splitting strategies, it is only shown once for the generic EM algorithm in Table3. Also, in Table 4, the space complexities for the algorithms with quick check are omitted since they are the same as for the corresponding algorithms without quick check.

The time and space complexities of the predicate evaluation algorithms are summarized in Tables5 and 6, respectively. In Table 5, $I_{\Omega}^{point}(n,d)$, $I_{\Theta}^{point}(n,d)$, and $I_{O}^{point}(n,d)$ denote the index structure dependent best case, average case, and worst case time complexity of a point containment query on the corresponding index structure, respectively. Analogously, in Table6, $I_{\Omega}^{space}(n,d)$, $I_{\Theta}^{space}(n,d)$, and $I_{O}^{space}(n,d)$ denote the index structure dependent best case, average case, and worst case space complexity of the corresponding index structure, respectively. In addition to the index, the values in the $d$ dimensions of the data item $i$ need to be stored.

| | Space | |
|---|---|---|
| | Best Case | Worst Case |
| QC | $\Omega(n+1)$ | $O((n+1)\cdot d)$ |
| HSR | $\Omega(m+n)$ | $O((m+n)\cdot d)$ |
| HCR | $\Omega(m+n)$ | $O((m+n)\cdot d)$ |
| EM-BFS | $\Omega(m+n)$ | $O\left((m+n+(2d)^n)\cdot d\right)$ |
| EM-DFS | $\Omega(m+n)$ | $O((m+n+(n-1)\cdot(2d-1)+2d)\cdot d)$ |
| EM-MIX | $\Omega(m+n)$ | $O\left(\left(m+n+(n-2)\cdot(2d-1)\cdot 2d+(2d)^2\right)\cdot d\right)$ |

| | Space |
|---|---|
| | Average Case |
| QC | $\Theta((n+1)\cdot d/2)$ |
| HSR | $\Theta((m+n)\cdot d/2)$ |
| HCR | $\Theta((m+n)\cdot d/2)$ |
| EM-BFS | $\Theta\left(\left(m+n+\sqrt{(d/2)^n}\right)\cdot(d/2)\right)$ |
| EM-DFS | $\Theta((m+n+((n-1)/2)\cdot((d/2)-1)+(d/2))\cdot(d/2))$ |
| EM-MIX | $\Theta\left(\left(m+n+((n-2)/2)\cdot((d/2)-1)\cdot(d/2)+((d/2))^2\right)\cdot(d/2)\right)$ |

**Table 4:** Space complexities of predicate matching algorithms

| | Time | | |
|---|---|---|---|
| | Best Case | Average Case | Worst Case |
| SE | $\Omega(1)$ | $\Theta(n\cdot d)$ | $O(n\cdot d)$ |
| IE | $\Omega(I_\Omega^{point}(n,d))$ | $\Theta(I_\Theta^{point}(n,d))$ | $O(I_O^{point}(n,d))$ |

**Table 5:** Time complexities of predicate evaluation algorithms

# 6 Benchmarks

In this section, we present some of our experimental evaluation results. First, we describe the implementation of the algorithms presented in this paper and the benchmark setting. Second, we show some comparative benchmark results for predicate matching and evaluation.

## 6.1 Implementation and Setting

All algorithms presented in this paper have been implemented in Java 5.0. The internal representation of a disjunctive predicate contains a set of conjunctive subpredicates. The entries in this set are automatically kept sorted in decreasing order in terms of the volume of the hyperrectangle representing the predicate in multi-dimensional space. If predicates have unbounded interval ends, a predicate is "smaller" than

| | Space | | |
|---|---|---|---|
| | Best Case | Average Case | Worst Case |
| SE | $\Omega(n+d)$ | $\Theta((n/2+1)\cdot d)$ | $O((n+1)\cdot d)$ |
| IE | $\Omega(I_\Omega^{space}(n,d)+d)$ | $\Theta(I_\Theta^{space}(n,d)+d)$ | $O(I_O^{space}(n,d)+d)$ |

**Table 6:** Space complexities of predicate evaluation algorithms

another one in our terminology if it has less unbounded interval ends or an equal number of unbounded interval ends and less volume when restricting the volume computation to the finite dimensions. The purpose of the sorting is to begin comparing the larger subpredicates when iterating the subpredicate list of a disjunctive predicate. This helps to find matches earlier or match as much of a subpredicate as possible early on to potentially reduce the number of matching steps. Also, the intervals for each dimension within a conjunctive predicate are sorted in increasing order according to their length. Considering shorter intervals first increases the probability of comparing two disjoint intervals early on, therefore being able to break the comparison with the current subpredicate and go on to the next without having to consider the remaining dimensions. These optimizations are used for all matching and evaluation algorithms throughout. Further, constants in atomic predicates are implemented as double values.

Various index structures can be used in our implementation via a generic interface. For each index, the interface is implemented by an adapter class that delegates the interface method calls to the appropriate method calls of the underlying index structure and performs any necessary conversions. We have compared various implementations of R-tree variants for our benchmarks. For the matching benchmark, we have decided to use an efficient lightweight main memory implementation of a standard R-tree with quadratic split strategy [17]. This specific implementation turned out to be the fastest of all the index structures that we have tested for predicate matching and evaluation. For the evaluation benchmark, however, we have switched to a—in our tests—slightly slower but more generic and flexible main memory implementation of an R*-tree since this implementation already supports our short-circuit evaluation optimization without having to edit the index source code [28]. We have also repeated our matching benchmarks using this R*-tree. In our tests, all index structures use a minimum node capacity of 5 and a maximum node capacity of 10.

All tests ran on a single server blade with two 2.8 Ghz Intel Xeon processors, 4 GB of main memory, and SuSE Linux Enterprise Server 9.

## 6.2 Predicate Matching

For the predicate matching benchmarks, we randomly generated a set of query predicates and a set of stream predicates by setting the number of dimensions and subpredicates and randomly choosing the constant values for the interval bounds of each dimension in each subpredicate. The values were chosen from a list of 21 distinct values between 0 and 100 using a normal distribution for query predicates and a uniform distribution for stream predicates. Each subpredicate defines a finite interval in each dimension of the data space. For the tests shown in this paper, we used a set of 60 query predicates that were matched against a set of 20 stream predicates. We matched each query predicate with each stream predicate, i. e., 1200 predicate pairs were matched in total. Figures 9 to 11 show the average matching time per predicate pair in milliseconds for the heuristics with simple relaxation (HSR+QC), the heuristics with complex relaxation (HCR+QC), and the exact matching algorithm with breadth-first split strategy without (EM+QC) as well as with index support (EM+QC+I). All algorithms use the quick check (QC). Note that the matching time is scaled logarithmically on each of the three diagrams. If nothing else is stated, the default settings used for the tests were 6 dimensions, 2 subpredicates per query predicate, and 20 subpredicates per stream predicate. We use relatively low values for the numbers of dimensions and subpredicates in the matching tests to be able to include the EM approach, which is not feasible for large problem sizes, in the comparison. Note that, in our DSMS scenario, matching time is a part of query compilation time and is therefore less important if dealing with long-running continuous queries.

In Figure 9, we varied the number of distinct dimensions in the data space that are referenced in each predicate. It can clearly be seen that the matching time of the EM algorithm without index support grows exponentially with increasing number of dimensions. Using the index approach significantly reduces matching time and can therefore keep the approach feasible for larger problem sizes. It does not even

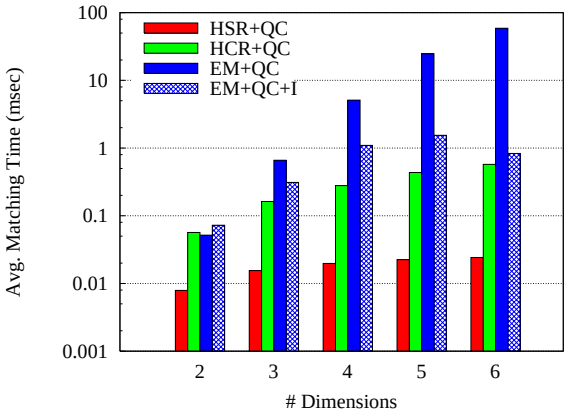



**Figure 9:** Varying number of dimensions

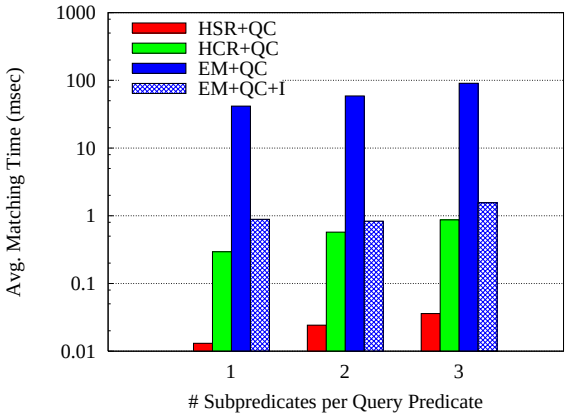



**Figure 10:** Varying query predicate size

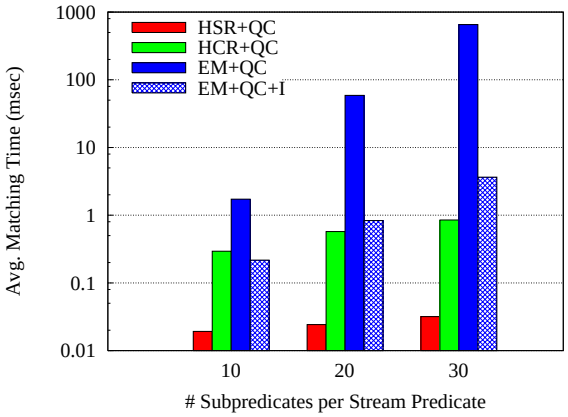



**Figure 11:** Varying stream predicate size

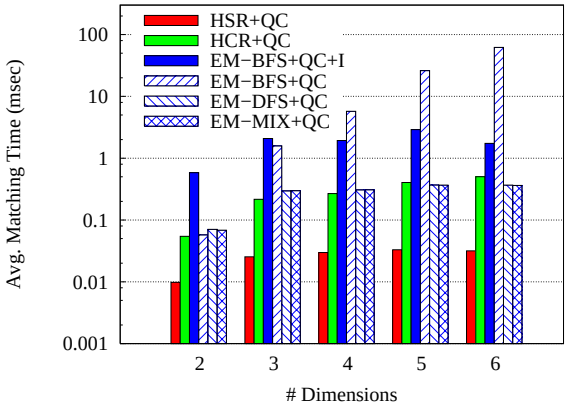



**Figure 12:** Varying number of dimensions

differ that much from the HCR approach which has polynomial complexity and whose running time increases linearly. The matching time of the HSR approach also increases linearly but is generally lower than for the other algorithms since less complex computations have to be performed.

The effects of a varying number of subpredicates in the query and the stream predicate are shown in Figures 10 and 11, respectively. Again, the index-based version of the EM algorithm clearly outperforms the version without index and can compete roughly with the HCR approach. In each case, the performance gain achieved through multi-dimensional indexing is about two orders of magnitude.

We repeated the above tests with predicates containing infinite intervals. The predicates were generated by randomly choosing the finite dimensions of the data space for each subpredicate using a uniform distribution. The remaining dimensions were not referenced by the subpredicate and were therefore unbounded. In this setting, the performance gain of the index was about an order of magnitude less. However, the index-based EM algorithm still was superior to the version without index, especially for larger numbers of dimensions and subpredicates.

We furthermore repeated the matching tests using the R*-tree index that we used for the evaluation benchmarks below instead of the lightweight R-tree index implementation. Also, we included the exact matching algorithms with depth-first (EM-DFS) and mixed split strategy (EM-MIX) in addition to the exact matching algorithm with breadth-first split strategy (EM-BFS). The results shown in Figures 12 to 14 indicate that the R*-tree is a little bit slower in our tests than the R-tree index of the previous

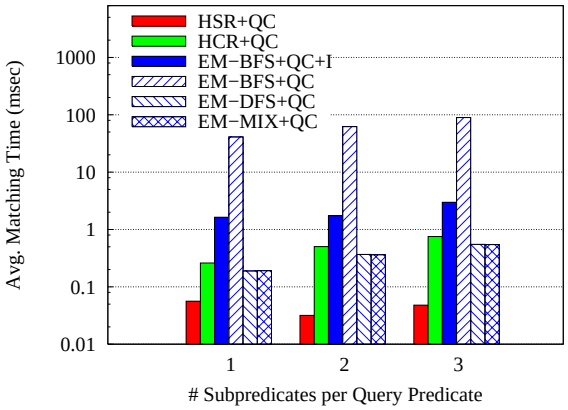



**Figure 13:** Varying query predicate size

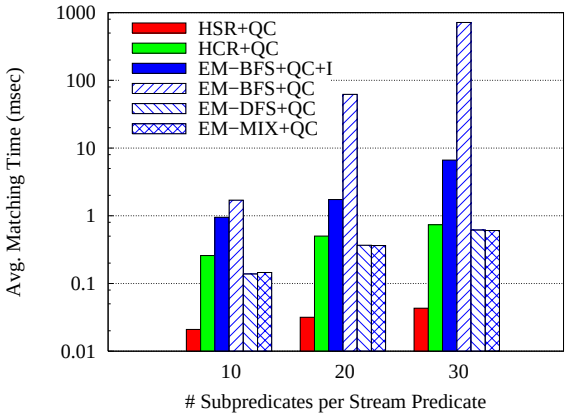



**Figure 14:** Varying stream predicate size

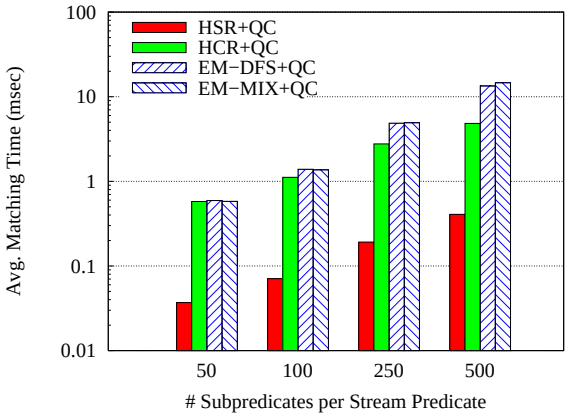



**Figure 15:** Matching large predicates

benchmarks. Also, with growing number of dimensions and subpredicates in the query and stream predicates, the EM-DFS and EM-MIX algorithm variants increasingly outperform the EM-BFS variant without as well as with index support. The difference in matching time between the EM-DFS and EM-MIX variants is, however, marginal in our tests. As already mentioned in Section 3.6, index support for the EM-DFS and EM-MIX algorithm variants is not beneficial using the R-tree indexes we considered since the overhead for updating the index structure by far outweighs its benefits.

In another test, we investigated how the matching algorithms perform for larger predicates, i. e., stream predicates with a larger number of subpredicates. Figure 15 shows the results. The EM-BFS algorithm variant is missing in the figure since it was not able to process stream predicates with more than 30 subpredicates without running out of main memory. As the figure shows, the performance of the EM-DFS and EM-MIX algorithm variants is comparable to that of the HCR algorithm for up to 50 subpredicates in the stream predicate. For 500 subpredicates, HCR is about a factor of 3 faster than the exact matching variants.

To illustrate the differences in predicate relaxation during predicate matching between the matching algorithms, we randomly generated a predicate with 6 dimensions and 20 subpredicates representing a stream predicate. We also generated 4 predicates with 6 dimensions and 2 subpredicates each, which represent query predicates. Predicate generation was done as described above except that the values for the interval bounds were chosen among 6 distinct values between 0 and 100 for the query predicates and

| | $p_A$ | $p_B$ | $p_C$ | $p_D$ |
|---|---|---|---|---|
| HSR | 66.31 | 66.37 | 66.37 | 66.44 |
| HSR+QC | 66.31 | 66.37 | 66.37 | 66.44 |
| HCR | 66.31 | 67.93 | 69.26 | 69.82 |
| EM | 66.31 | 66.37 | 66.37 | 66.44 |

**Table 7:** Selectivities for combined test (%)

among 4 distinct values for the stream predicate. We then matched the stream predicate with the first query predicate to obtain predicate $p_A$. Predicate $p_A$ was then matched with the second query predicate to obtain predicate $p_B$. Predicate $p_B$ was again matched with the third query predicate to obtain predicate $p_C$ which was in turn matched with the fourth query predicate to obtain predicate $p_D$. Matching was performed with the HSR, the HSR+QC, the HCR, and the EM algorithms. Note that the use of the quick check for HCR and EM and the use of index-based matching for all algorithms has no influence on the structure of the resulting predicates. Predicates $p_A$ to $p_D$ were then evaluated using one million uniformly distributed data items. Table 7 shows the results for the observed selectivities of all four predicates depending on the matching algorithm that produced them. As expected, HSR, HSR+QC, and EM yield predicates with identical selectivities because these algorithms produce exact predicates that do not cause any false drops. Also, it can be seen that predicates $p_A$ and $p_C$ need to be relaxed to obtain predicates $p_B$ and $p_D$, respectively. The HCR algorithm relaxes the predicate during each matching step and yields higher selectivity values which shows that this algorithm causes false drops. However, the increase in selectivity induced by the false drops never exceeds 3.5 % which indicates that the approximation made by the HCR algorithm stays close to the exact solution in this test.

The difference between the various matching algorithms in the ability to detect matching predicates that do not need to be relaxed is illustrated by the following test. Using a randomly generated set of 60 query and 20 stream predicates with 3 possibly unbounded dimensions, 3 subpredicates per query predicate, and 30 subpredicates per stream predicate, the EM algorithm successfully matched 933 out of the 1200 predicate pairs without relaxation. The remaining algorithms, except the HSR algorithm without QC, detected only 660 matching pairs. This means that the heuristics missed 273 matches in this example. The HSR algorithm without QC of course never detects any matches at all.

Summarizing, we can state that the EM algorithm is applicable in practice as long as the problem size, i. e., the number of dimensions in the data space and the number of subpredicates in the stream predicate, is reasonably low. However, performance quickly degrades with increasing problem size. This effect can be alleviated by combining the EM algorithm with a multi-dimensional index on the subpredicates of the stream predicate. Admittedly, the R-tree index structures used by us only yield a benefit when used with the EM algorithm with BFS split strategy. For the other split strategies, the maintenance overhead of these index structures is too high. Since the DFS and MIX split strategies outperform the BFS split strategy by far in all tests and are applicable for larger problem sizes due to much less memory consumption, they are the preferred choice. Combining them with less maintenance-intensive index structures could yield a further performance benefit. For very large problem sizes, the exponential time complexity of the EM algorithm becomes prohibitive. In this case, the heuristics have to be used instead. Both heuristics, HSR as well as HCR, perform and scale well with increasing problem size. Also, the increase in predicate selectivity induced by the approximation made when using HCR proves to be low in our tests.

### 6.3 Predicate Evaluation

In contrast to predicate matching time, predicate evaluation time is performance critical in our DSMS scenario. The predicate evaluation benchmarks were carried out by evaluating a given predicate against

one million randomly generated data items with values distributed uniformly between 0 and 100 for each dimension. Predicates were also again generated randomly using a certain number of subpredicates for disjunctively covering an area of the data space. The remaining subpredicates were placed in the middle of that area using a normal distribution for placing the center point of each subpredicate. Since data items are distributed uniformly in the data space, the percentage of the data space covered by the predicate yields the predicate's selectivity. Unless otherwise stated, all parameters take default values which means 3 dimensions, 100 subpredicates in the predicate to be evaluated, and a predicate selectivity of 1 %. We choose a larger number of subpredicates for the evaluation benchmark than for the matching benchmark since we expect many subpredicates to be introduced by predicate relaxation in the EM and especially the HSR matching approaches. In our tests, predicates have an overlap of 50 %. This is achieved by using half of the subpredicates for disjunctively covering the area of the data space needed to obtain the desired selectivity and placing the remaining subpredicates within this area as described above. Figures 16 to 18 show the throughput of predicate evaluation in data items per millisecond for the standard evaluation (SE), the index-based evaluation (IE), and the index-based evaluation with short-circuit optimization (IE+SC).

Figure 16 shows the throughput for a varying number of dimensions in the data space. Clearly, index-based evaluation is superior for the given settings. Short-circuiting the evaluation yields an additional performance gain of about 20 %. For all three algorithms, throughput decreases only moderately for an increasing number of dimensions.

The number of subpredicates in the predicate to be evaluated is varied in Figure 17. Again, index-based evaluation is clearly superior to standard evaluation and performance for the index-based approach degrades slower with increasing number of subpredicates than for the standard approach. For 5000 subpredicates, the index-based approach is still better than the standard evaluation for 10 subpredicates.

In another test, we varied the selectivity of the predicate to be evaluated between 1 %, meaning that only one in a hundred data items satisfies the predicate, and 100 %, which means that all data items satisfy the predicate. The results are shown in Figure 18. Obviously, the lower the selectivity, the better the index-based solutions perform. In contrast, the standard evaluation performs better for higher selectivities. This is due to the fact that the index can identify a non-qualifying data item early on, for example when comparing it to the root of the index tree if the data item lies outside the root box of the index. In contrast, it has to descend to the leaves in the tree if the data item qualifies. For the standard evaluation it is vice versa. A qualifying data item can be identified early on because as soon as it lies within the current subpredicate, no further comparisons with the remaining subpredicates are necessary. If a data item does not qualify, all subpredicates need to be considered in order to be sure that no matching subpredicate exists.

It can roughly be expected from Figure 18 that for our default settings and for 50 % subpredicate overlap, standard evaluation is better than index-based evaluation for selectivity values above about 70 % and better than index-based evaluation with short-circuit optimization for selectivity values above about 90 %. With less overlap between subpredicates the situation changes in favor of the index-based solutions. For overlap ratios up to 40 %, the index-based evaluation with short-circuit optimization is superior for all selectivities when using default values for the other parameters. Higher overlap hurts the index-based solutions as more overlap means that more paths need to be traversed in the tree. This effect is worse for the index-based evaluation without short-circuit optimization. However, selectivities are expected to be low in practice since users are mostly interested in very specific parts of the available information. Also, the number of dimensions in our DSMS scenario is supposed to be moderate since, e. g., sensor data streams rarely contain more than about 10 to 20 dimensions per data item. Further, often only a small subset of the available dimensions is actually referenced by queries. In contrast, the number of subpredicates can become large (in the order of many thousand) if many queries are registered. This is due to the effects of data stream widening, especially if the HSR approach is used.

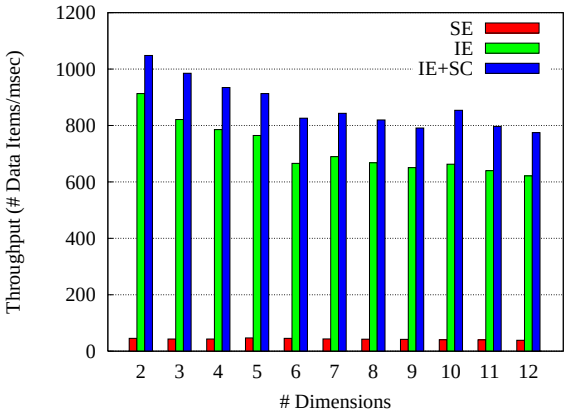



**Figure 16:** Varying Number of Dimensions

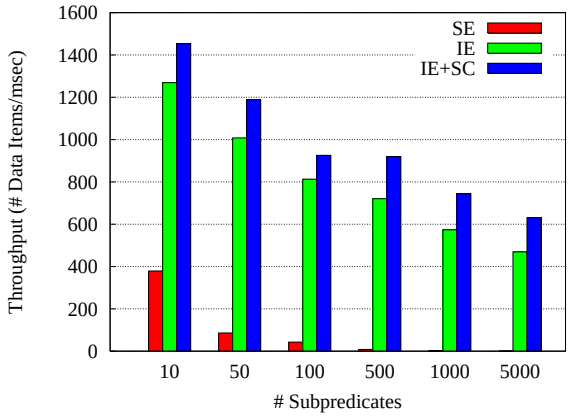



**Figure 17:** Varying Predicate Size

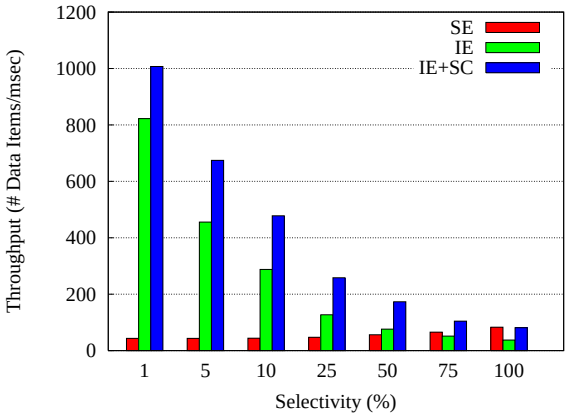



**Figure 18:** Varying Predicate Selectivity

Table 8 shows the relative throughput of the three predicate evaluation algorithms for varying sub-predicate overlap ratios of the evaluated predicate with low ($\sigma = 10\,\%$) and high ($\sigma = 75\,\%$) selectivities. The baseline is the throughput of the standard evaluation (SE) which is set to 100 % in each case. For low selectivity values, the index-based evaluation (IE) and the index-based evaluation with short-circuit optimization yield a performance gain of about factor 6 and 11, respectively. For high selectivity values, the index-based evaluation is already inferior to the standard evaluation. The index-based evaluation with short-circuit optimization is however still superior by about 50 % to 85 %. This shows that the short-circuit optimization yields a major improvement over the non-optimized index approach and makes index use beneficial even for high predicate overlap ratios and selectivity values.

| Selectivity | $\sigma = 10\,\%$ | | | | | $\sigma = 75\,\%$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Overlap | 0 % | 15 % | 25 % | 30 % | 50 % | 0 % | 15 % | 25 % | 30 % | 50 % |
| SE | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| IE | 644 | 690 | 670 | 633 | 655 | 74 | 85 | 89 | 84 | 79 |
| IE+SC | 1215 | 1120 | 1114 | 1015 | 1088 | 160 | 156 | 185 | 181 | 160 |

**Table 8:** Relative throughput for varying predicate overlap (%)

The fact that the throughput for each algorithm does not show a stable trend for increasing overlap ratios in Table 8 is due to the fact that, for each overlap ratio, a new predicate has been randomly generated and the characteristics of the different predicates can therefore not be fully controlled. However, variations are about the same in relation for all three algorithms.

Summarizing, we can say that the index-based evaluation is beneficial especially for predicates with many subpredicates and realistically low selectivity values. It yields performance gains of up to three orders of magnitude for predicates with 50 % overlap and even more if overlap is less. The disadvantages of multi-dimensional index structures like the R-tree when indexing higly overlapping regions can largely be alleviated by the short-circuit optimization.

# 7 Related Work

In this section, we present an overview of related work in the fields of predicate handling, multi-dimensional indexing, and data stream management.

## 7.1 Predicate Handling

Efficient handling of predicates has been a research topic for many years. Handling of conjunctive predicates has already been examined in the early 1980s [25] and deals with problems like predicate representation and minimization as well as equivalence and satisfiability checking. Implication checking for conjunctive predicates has also been dealt with [27]. Also, detailed studies for solving satisfiability, implication, and equivalence problems for conjunctive predicates concerning different domains and operator sets have been conducted [16]. All of these works are restricted to conjunctive predicates.

Predicate indexing, as opposed to indexing data, has been an active research area, especially in the domains of active databases [30] and publish&subscribe systems like Le Subscribe [13] and MDV [21]. Two index structures that have been proposed for predicate indexing in active databases are the IBS-tree [18] and interval skip lists [19]. These are 1-dimensional index structures for indexing a set of independent intervals on one attribute. Another approach for indexing a set of independent 1-dimensional intervals are virtual construct intervals (VCIs) [32]. There also exists a 2-dimensional variant, the virtual construct rectangles (VCRs) [31], for indexing a set of independent 2-dimensional intervals. In contrast, we propose using a multi-dimensional index structure for indexing a set of multi-dimensional conjunctive predicates that are all part of the same disjunctive predicate. Multi-dimensional predicate indexing for event filtering in publish&subscribe systems has also been studied using the UB-tree as an index structure [29]. The approach accordingly transforms the dimensions using a space filling curve to map the multi-dimensional universe to one dimensional space. Further, index support for the evaluation of queries over data sets containing interval-valued attributes has also been examined [12].

There are only few works on predicate handling for disjunctive predicates in the database field. They deal with the efficient evaluation of disjunctive predicates by merging disjuncts [24] or by using a special form of relational algebra translation [4]. Other work focusses on bypassing the evaluation of expensive predicate terms if possible [9] and on union pushdown techniques [6] to optimize the processing of disjunctive predicates. Work on optimizing query evaluation by appropriately moving expensive predicates in the query plan has also considered disjunctive predicates [20].

## 7.2 Multi-Dimensional Indexing

Multi-dimensional indexing has originally been motivated by the needs of spatial databases. One of the most well known spatial index structures is the R-tree [17]. It uses minimum bounding boxes to index spatial objects and stores multi-dimensional rectangles, such as our conjunctive subpredicates, without

any further transformation or clipping. The R*-tree [3] is an advanced version of the R-tree aiming at improved performance by reducing the area, margin, and overlap of the rectangles stored in the index. These goals are achieved by a modified insertion strategy that uses a forced reinsert policy. To completely eliminate any overlap between regions, which is known to be responsible for performance degradation in an R-tree due to the necessity to traverse all paths covering or intersecting the searched data point or region during a search, the R+-tree was developed [26]. It uses clipping to distribute non-overlapping parts of rectangles over different index regions. While the problem of overlapping regions is thus avoided, the clipping approach may lead to a high fragmentation of indexed regions. The TV-tree [23] indexes high-dimensional data by dynamically choosing an appropriate subset of dimensions for indexing on each index level. An extensive survey on multi-dimensional access methods including a classification and comparative studies has also been published [15].

### 7.3 Data Stream Management

In the domain of data stream management systems, which is the main—albeit not the only possible—application scenario that we target with our predicate matching and evaluation approaches, various systems have been proposed in recent years. STREAM [1] processes data streams using the continuous query language (CQL) [2] by transforming them into relations and the query results back into streams again. TelegraphCQ [5] adaptively processes data streams using, among other things, the Eddy approach for adaptive tuple routing. NiagaraCQ [7] optimizes query processing by sharing common computations among continuous queries. This is achieved by appropriately grouping queries according to similar structures. In the field of XML data streams, ONYX [11] has been proposed as a scalable system for content-based data dissemination in large-scale networks. Aurora is basically a centralized data flow system that processes tuple streams. A decentralized version of Aurora is Aurora*. Finally, Medusa is a distributed infrastructure that supports federated operation of nodes [8]. PIPES [22] is a public infrastructure for processing and exploring data streams. The needs of high fan-in systems are being addressed in the HiFi [14] system. The Cougar [33] approach deals with in-network query processing in sensor networks.

## 8 Conclusion

In this paper, we have presented various methods for matching and evaluating interval-based disjunctive predicates. Matching involves deciding whether a predicate implies another and, if this is not the case, how the other predicate can be altered in order for the implication to be valid. We have concentrated on disjunctive predicates consisting of conjunctive subpredicates that form multi-dimensional hyperrectangles with edges parallel to the coordinate axes in the data space. The approach can also be used as an approximation for more complex shaped predicates, which affords non-trivial post-processing that we do not elaborate on in this paper. We have introduced two heuristics that can be executed efficiently but either cause the number of subpredicates of a disjunctive predicate to increase or deliver only approximate results. An exact solution that is applicable for small input sizes, i. e., small numbers of dimensions and subpredicates, has also been shown. Achieving high throughput during predicate evaluation is a major goal in most application scenarios. We therefore have further dealt with the evaluation of disjunctive predicates and examined the use of multi-dimensional indexing for speeding up predicate matching and evaluation. To the best of our knowledge, this is the first work to examine the use of multi-dimensional indexes for matching and continuously evaluating disjunctive predicates. All algorithms have been implemented and evaluated in a comparative experimental study asserting the effectiveness of the index-based approach which yields a performance gain of up to several orders of magnitude compared to the solution without indexing.

There are numerous opportunities for future work. First, the applicability and efficiency of other multi-dimensional index structures in the context of predicate matching and evaluation could be examined. Second, we think about implementing a specialized index structure that is based on existing index techniques but especially fits the needs of indexing disjunctive predicates. In this course, the functionality of predicate matching could be fully or partially incorporated into the index itself.

## References


[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, Mar. 2003.

[2] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proc. of the Intl. Workshop on Database Programming Languages*, pages 1–19, Potsdam, Germany, Sept. 2003.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, USA, May 1990.

[4] F. Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 193–204, Portland, OR, USA, May 1989.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, Jan. 2003.

[6] J.-Y. Chang and S.-G. Lee. An Optimization of Disjunctive Queries: Union-Pushdown. In *Proc. of the Intl. Computer Software and Applications Conf.*, pages 356–361, Washington, DC, USA, Aug. 1997.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, Dallas, TX, USA, May 2000.

[8] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable Distributed Stream Processing. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, Jan. 2003.

[9] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and Evaluation of Disjunctive Queries. *IEEE Trans. on Knowledge and Data Engineering*, 12(2):238–260, Mar. 2000.

[10] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 330–341, Mumbai (Bombay), India, Sept. 1996.

[11] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 612–623, Toronto, Canada, Aug. 2004.

[12] J. Enderle, N. Schneider, and T. Seidl. Efficiently Processing Queries on Interval-and-Value Tuples in Relational Databases. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 385–396, Trondheim, Norway, Aug. 2005.

[13] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 115–126, Santa Barbara, CA, USA, May 2001.

[14] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-in Systems: The HiFi Approach. In *Proc. of the Conf. on Innovative Data Systems Research*, pages 290–304, Asilomar, CA, USA, Jan. 2005.

[15] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[16] S. Guo, W. Sun, and M. A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM Trans. on Database Systems*, 21(2):270–293, June 1996.

[17] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, Boston, MA, USA, June 1984.

[18] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 271–280, Atlantic City, NJ, USA, May 1990.

[19] E. N. Hanson and T. Johnson. Selection Predicate Indexing for Active Databases Using Interval Skip Lists. *Information Systems*, 21(3):269–298, May 1996.

[20] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 267–276, Washington, D.C., USA, May 1993.

[21] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A Publish & Subscribe Architecture for Distributed Metadata Management. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 309–320, San José, CA, USA, Feb. 2002.

[22] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 925–926, Paris, France, June 2004.

[23] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree – an index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, Oct. 1994.

[24] M. Muralikrishna and D. J. DeWitt. Optimization of Multiple-Relation Multiple-Disjunct Queries. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 263–275, Austin, TX, USA, Mar. 1988.

[25] D. J. Rosenkrantz and H. B. Hunt. Processing Conjunctive Predicates and Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 64–72, Montreal, Canada, Oct. 1980.

[26] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 507–518, Brighton, England, Sept. 1987.

[27] X.-H. Sun, N. Kamel, and L. M. Ni. Solving Implication Problems in Database Applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 185–192, Portland, OR, USA, May 1989.

[28] J. van der Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL – A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 39–48, Roma, Italy, Sept. 2001.

[29] B. Wang, W. Zhang, and M. Kitsuregawa. UB-tree Based Efficient Predicate Index with Dimension Transform for Pub/Sub System. In *Proc. of the Intl. Conf. on Database Systems for Advanced Applications*, pages 63–74, Jeju Island, Korea, Mar. 2004.

[30] J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1995.

[31] K.-L. Wu, S.-K. Chen, and P. S. Yu. VCR Indexing for Fast Event Matching for Highly-Overlapping Range Predicates. In *ACM Symp. on Applied Computing*, pages 740–747, Nicosia, Cyprus, Mar. 2004.

[32] K.-L. Wu, S.-K. Chen, P. S. Yu, and M. Mei. Efficient Interval Indexing for Content-Based Subscription E-Commerce and E-Service. In *Proc. of the IEEE Intl. Conf. on E-Commerce Technology for Dynamic E-Business*, pages 22–29, Beijing, China, Sept. 2004.

[33] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, Sept. 2002.

# A Predicate Matching and Evaluation Algorithms

All algorithms assume that each stream predicate $p$ and each query predicate $p'$ contains at least one conjunctive subpredicate $c$ and $c'$, respectively. Extensions for the handling of special cases like empty predicates and predicates that constitute tautologies or contradictions are straightforward but would clutter the algorithm presentations. It is also possible to deal with these special cases beforehand. In our DSMS scenario, these cases are handled within the data stream sharing optimizer.

## A.1 Quick Check (QC)

**Algorithm 6** Quick Check (QC)

**Input:** Stream predicate $p$ and a conjunctive subpredicate $c'$ of query predicate $p'$.

**Output:** 1, if $c' \Rightarrow c$ for at least one conjunctive subpredicate $c$ in $p$; 0, if $c'$ overlaps with at least one conjunctive subpredicate $c$ in $p$, $-1$ if $c'$ does not overlap with any conjunctive subpredicate $c$ in $p$, otherwise.

```
1:  overlap ← −1;
2:  for all conjunctive subpredicates c in p do
3:     if c′ ⇒ c then
4:        return 1;
5:     end if
6:     if c′ overlaps with c then
7:        overlap ← 0;
8:     end if
9:  end for
10: return overlap;
```

## A.2 Heuristics with Simple Relaxation (HSR)

**Algorithm 7** Heuristics with Simple Relaxation (HSR)

**Input:** Stream predicate $p$ and query predicate $p'$.

**Output:** $(\text{true}, p)$, if the quick check of Section 3.2 is activated and, for all conjunctive subpredicates $c'$ in $p'$, $c' \Rightarrow c$ for at least one conjunctive subpredicate $c$ in $p$; $(\text{false}, \bar{p})$, where $\bar{p}$ is a relaxed version of $p$ such that the above condition is satisfied, otherwise.

```
1:  match ← true;
2:  for all conjunctive subpredicates c′ in p′ do
3:     if quick check is activated ∧ QC(p, c′) = 1 then
4:        continue;
5:     else
6:        match ← false;
7:        disjunctively add c′ to p;
8:     end if
9:  end for
10: return (match, p);
```

## A.3 Heuristics with Complex Relaxation (HCR)

$$f^{\geq 0}(x) := \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

**Algorithm 8** Heuristics with Complex Relaxation (HCR)

**Input:** Stream predicate $p$ and query predicate $p'$.

**Output:** $(\text{true}, p)$, if, for all conjunctive subpredicates $c'$ in $p'$, $c' \Rightarrow c$ for at least one conjunctive subpredicate $c$ in $p$; $(\text{false}, \overline{p})$, where $\overline{p}$ is a relaxed version of $p$ such that the above condition is satisfied, otherwise.

$match \leftarrow$ true;
**for all** conjunctive subpredicates $c'$ in $p'$ **do**
  **if** quick check is activated $\wedge\ QC(p, c') = 1$ **then**
    **continue**;
  **else**
    $i_b \leftarrow +\infty$; $e_b \leftarrow +\infty$; $c_b \leftarrow$ **null**; $c_{orig} \leftarrow$ **null**; $m \leftarrow$ true;
    **for all** conjunctive subpredicates $c$ in $p$ **do**
      $i \leftarrow 2(|D| - |D_c|)$; $e \leftarrow 0$; $c_c \leftarrow c$; $m \leftarrow$ true;
      **for all** pairs of corresponding dimensions $d, d'$ in $c_c, c'$ with $d = d'$ **do**
        **if** $(lowerBound(I_d) = -\infty) \vee (lowerBound(I_{d'} = -\infty)$ **then**
          $i \leftarrow i + 1$;
        **end if**
        **if** $(upperBound(I_d) = +\infty) \vee (upperBound(I_{d'} = +\infty)$ **then**
          $i \leftarrow i + 1$;
        **end if**
        **if** $I_d \cap I_{d'} \neq I_{d'}$ **then**
          $m \leftarrow$ false;
          $a \leftarrow f^{\geq 0}(lowerBound(I_d) - lowerBound(I_{d'})) +$
            $f^{\geq 0}(upperBound(I_{d'}) - upperBound(I_d))$;
          $v \leftarrow \prod_{d^* \in D:((d^* \neq d) \wedge (0 < \overline{I_{d^*}} < +\infty))} \overline{I_{d^*}}$;
          $e \leftarrow e + (a \cdot v)$;
          replace $I_d$ in $c_c$ with $I_{d_c} := [min(lowerBound(I_d), lowerBound(I_{d'})),$
            $max(upperBound(I_d), upperBound(I_{d'}))]$;
        **end if**
      **end for**
      **if** $m =$ true **then**
        **break**;
      **else if** $(i < i_b) \vee ((i = i_b) \wedge (e < e_b))$ **then**
        $i_b \leftarrow i$; $e_b \leftarrow e$; $c_b \leftarrow c_c$; $c_{orig} \leftarrow c$;
      **end if**
    **end for**
    **if** $m =$ false **then**
      $match \leftarrow$ false;
      replace $c_{orig}$ in $p$ with $c_b$;
    **end if**
  **end if**
**end for**
**return** $(match, p)$;

## A.4 Exact Matching (EM)

**Algorithm 9** Compare Dimensions (CD)

**Input:** Conjunctive stream subpredicate $c$ and conjunctive query subpredicate $c'_c$.
**Output:** Queue $Q'_c$ of unmatched parts of query subpredicate $c'_c$.

```
1:  Q'_c ← ∅; c'_orig ← c'_c;
2:  for all pairs of corresponding dimensions d, d' in c, c'_c with d = d' do
3:     if I_d ∩ I_d' = ∅ then
4:        Q'_c ← ∅; enqueue(Q'_c, c'_orig); break;
5:     else if I_d ∩ I_d' = I_d' then
6:        continue;
7:     else if I_d ∩ I_d' = I_d then
8:        divide c'_c along dimension d' into the part c'_i that is
9:        overlapping with c in dimension d' and the remaining parts c'_o1 and c'_o2;
10:       enqueue(Q'_c, c'_o1); enqueue(Q'_c, c'_o2); c'_c ← c'_i;
11:    else
12:       /* I_d and I_d' overlap */
13:       divide c'_c along dimension d' into the part c'_i that is
14:       overlapping with c in dimension d' and the remaining part c'_o;
15:       enqueue(Q'_c, c'_o); c'_c ← c'_i;
16:    end if
17: end for
18: return Q'_c;
```

**Algorithm 10** Exact Matching with Breadth-First Split Strategy (EM-BFS)

**Input:** Stream predicate $p$ and query predicate $p'$.
**Output:** $(\text{true}, p)$, if $p' \Rightarrow p$; $(\text{false}, \bar{p})$, where $\bar{p}$ is a relaxed version of $p$ such that $p' \Rightarrow \bar{p}$, otherwise.

```
1:  match ← true;
2:  for all conjunctive subpredicates c' in p' do
3:    if quick check is activated ∧ QC(p, c') = 1 then
4:      continue;
5:    else if quick check is activated ∧ QC(p, c') = −1 then
6:      disjunctively add c' to p;
7:      continue;
8:    else
9:      /* quick check is deactivated or returns 0 */
10:     Q'_1 ← ∅; Q'_2 ← ∅; enqeue(Q'_1, c');
11:     for all conjunctive subpredicates c in p do
12:       Q'_2 ← Q'_1; Q'_1 ← ∅;
13:       while Q'_2 ≠ ∅ do
14:         c'_c ← deqeue(Q'_2);
15:         /* compare dimensions using Algorithm 9 */
16:         Q'_c ← CD(c, c'_c);
17:         append(Q'_1, Q'_c);
18:       end while
19:       if Q'_1 = ∅ then
20:         break;
21:       end if
22:     end for
23:     if Q'_1 ≠ ∅ then
24:       match ← false;
25:       disjunctively add c' to p;
26:     end if
27:   end if
28: end for
29: return (match, p);
```

**Algorithm 11** Exact Matching with Depth-First Split Strategy (EM-DFS)

**Input:** Stream predicate $p$ and query predicate $p'$.
**Output:** $(\text{true}, p)$, if $p' \Rightarrow p$; $(\text{false}, \bar{p})$, where $\bar{p}$ is a relaxed version of $p$ such that $p' \Rightarrow \bar{p}$, otherwise.

1: $match \leftarrow$ true;
2: **for all** conjunctive subpredicates $c'$ in $p'$ **do**
3: **if** quick check is activated $\wedge\ QC(p, c') = 1$ **then**
4: **continue**;
5: **else if** quick check is activated $\wedge\ QC(p, c') = -1$ **then**
6: disjunctively add $c'$ to $p$;
7: **continue**;
8: **else**
9: /* quick check is deactivated or returns 0 */
10: $Q_{LIFO} \leftarrow \emptyset$; $Q_i \leftarrow \emptyset$; $Q_p \leftarrow \emptyset$; $enqeue(Q_i, c')$; $enqeue(Q_{LIFO}, Q_i)$; $enqeue(Q_p, p)$;
11: **while** $Q_{LIFO} \neq \emptyset$ **do**
12: $Q_n \leftarrow deqeue(Q_{LIFO})$; $c'_c \leftarrow deqeue(Q_n)$; $p^- \leftarrow deqeue(Q_p)$;
13: **if** $Q_n \neq \emptyset$ **then**
14: $enqeue(Q_{LIFO}, Q_n)$; $enqeue(Q_p, p^-)$;
15: **end if**
16: let $c$ be the first conjunctive subpredicate in $p^-$; remove $c$ from $p^-$;
17: /* compare dimensions using Algorithm 9 */
18: $Q'_c \leftarrow CD(c, c'_c)$;
19: **if** $Q'_c \neq \emptyset$ **then**
20: **if** $p^-$ is not the empty predicate **then**
21: $enqeue(Q_{LIFO}, Q'_c)$; $enqeue(Q_p, p^-)$;
22: **else**
23: $match \leftarrow$ false;
24: disjunctively add $c'$ to $p$;
25: **break**;
26: **end if**
27: **end if**
28: **end while**
29: **end if**
30: **end for**
31: **return** $(match, p)$;

**Algorithm 12** Exact Matching with Mixed Split Strategy (EM-MIX)

**Input:** Stream predicate $p$ and query predicate $p'$.
**Output:** $(\text{true}, p)$, if $p' \Rightarrow p$; $(\text{false}, \bar{p})$, where $\bar{p}$ is a relaxed version of $p$ such that $p' \Rightarrow \bar{p}$, otherwise.

1: $match \leftarrow$ true;
2: **for all** conjunctive subpredicates $c'$ in $p'$ **do**
3: **if** quick check is activated $\wedge\ QC(p, c') = 1$ **then**
4: **continue**;
5: **else if** quick check is activated $\wedge\ QC(p, c') = -1$ **then**
6: disjunctively add $c'$ to $p$;
7: **continue**;
8: **else**
9: /* quick check is deactivated or returns 0 */
10: $Q_{LIFO} \leftarrow \emptyset$; $Q_i \leftarrow \emptyset$; $Q_p \leftarrow \emptyset$; $enqeue(Q_i, c')$; $enqeue(Q_{LIFO}, Q_i)$; $enqeue(Q_p, p)$; $m \leftarrow$ true;
11: **while** $Q_{LIFO} \neq \emptyset$ **do**
12: $Q_n \leftarrow deqeue(Q_{LIFO})$; $p^- \leftarrow deqeue(Q_p)$;
13: let $c$ be the first conjunctive subpredicate in $p^-$; remove $c$ from $p^-$;
14: **while** $Q_n \neq \emptyset$ **do**
15: $c'_c \leftarrow deqeue(Q_n)$;
16: /* compare dimensions using Algorithm 9 */
17: $Q'_c \leftarrow CD(c, c'_c)$;
18: **if** $Q'_c \neq \emptyset$ **then**
19: **if** $p^-$ is not the empty predicate **then**
20: $enqeue(Q_{LIFO}, Q'_c)$; $enqeue(Q_p, p^-)$;
21: **else**
22: $m \leftarrow$ false; $match \leftarrow$ false;
23: disjunctively add $c'$ to $p$;
24: **break**;
25: **end if**
26: **end if**
27: **end while**
28: **if** $m =$ false **then**
29: **break**;
30: **end if**
31: **end while**
32: **end if**
33: **end for**
34: **return** $(match, p)$;

## A.5 Standard Evaluation (SE)

**Algorithm 13** Standard Evaluation (SE)

**Input:** Predicate $p$ and data item $i$.
**Output:** true, if $i$ satisfies $p$; false, otherwise.

```
1:  for all conjunctive subpredicates c in p do
2:      match ← true;
3:      for all pairs of corresponding dimensions d_c, d_i in c, i with d_c = d_i do
4:          if the value for d_i in i lies outside the interval defined for d_c in c then
5:              match ← false;
6:              break;
7:          end if
8:      end for
9:      if match = true then
10:         return true;
11:     end if
12: end for
13: return false;
```