Shuping Ji

Efficient Content-based Publish/Subscribe Routing and Boolean Expression Matching Algorithms

Technische
Universität
München

Technische Universität München

Fakultät für Informatik

Efficient Content-based Publish/Subscribe Routing and Boolean Expression Matching Algorithms

Shuping Ji

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

Vorsitzender: Prof. Dr. George Carle

Prüfer der Dissertation:

1. Prof. Dr. Hans-Arno Jacobsen
2. Prof. Dr. David E. Bakken

Die Dissertation wurde am 08.08.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15.09.2018 angenommen.

Abstract

Event routing and Boolean expression matching are two key functions for content-based publish/subscribe (`pub/sub`) systems. Moreover, Boolean expression matching plays an important role in a growing number of other applications, such as online advertising, online news dissemination and workflow management. However, the existing routing and Boolean expression matching solutions present limitations on flexibility, performance, expressiveness and applicability. To overcome these limitations, we propose new efficient content-based `pub/sub` routing and Boolean expression matching algorithms.

Despite suffering from inefficiency and flexibility limitations, the filter-based routing (FBR) algorithm is widely used in content-based `pub/sub` systems. To address its limitations, we propose a dynamic destination-based routing algorithm called D-DBR, which decomposes `pub/sub` into two independent parts: Content-based matching and destination-based multicasting. D-DBR exhibits low event matching cost and high efficiency, flexibility, and robustness for event routing in small scale overlays. To boost scalability, we further complement D-DBR with a new routing algorithm called MERC. MERC divides the overlay into interconnected clusters and applies content-based and destination-based mechanisms to route events inter- and intra-cluster, respectively. We implemented all algorithms in the PADRES `pub/sub` system. Experimental results show that our algorithms outperform FBR in terms of improving event dissemination throughput by up to 700% and reducing the end-to-end latency by up to 55%.

Conjunctive Boolean expression matching is an important function for many applications, including content-based `pub/sub` systems. However, existing solutions still suffer from limitations when applied to high-dimensional and dense workloads. To overcome these limitations, we design a novel data structure called `PS-Tree` that can efficiently index predicates. By dividing predicates into disjoint predicate spaces, `PS-Tree` achieves high matching performance and good expressiveness. Based on `PS-Tree`, we first propose a conjunctive Boolean expression matching algorithm `PSTBloom`. By efficiently filtering out a large proportion of unmatching expressions, `PSTBloom` achieves high matching performance, especially for high-dimensional workloads. `PSTBloom` also achieves fast index construction and a small memory footprint. Compared with state-of-the-art methods, comprehensive experiments show that PSTBloom reduces the matching time, index construction time and memory usage by up to 84%, 78% and 94%, respectively. Although PSTBloom is effective for many workload distributions, dense workloads represent new challenges to PSTBloom and other algorithms. To effectively handle dense workloads, we further propose the PSTHash algorithm, which divides conjunctive Boolean expressions into disjoint multi-dimensional predicate spaces. This organization prunes partially matching expressions efficiently. Comprehensive experiments on both synthetic and real-world datasets show that PSTHash improves the matching performance by up to 92% for dense workloads.

Conjunctive Boolean expression matching presents expressiveness limitation. A growing number of applications need the supporting of arbitrary Boolean expressions matching, such as advertising exchanges, automatic targeting and some pub/sub systems. However, most of the existing solutions can only support conjunctive Boolean expression matching. The limited number of solutions that can work directly on arbitrary Boolean expressions present applicability and performance limitations. Moreover, it is not always effective to normalize arbitrary Boolean expressions into conjunctive form and then use existing methods for evaluating such expressions because of the exponential increase in the size of the expressions. For these reasons, we propose a novel A-Tree data structure to efficiently index arbitrary Boolean expressions. A-Tree is a dynamic multiroot tree, in which predicates and subexpressions from different arbitrary Boolean expressions are aggregated and shared. A-Tree employs dynamic self-adjustment policies to adapt the index as the workload changes. Our comprehensive experiments show that the A-Tree-based matching solution outperforms existing arbitrary Boolean expression matching algorithms in terms of matching performance, index construction time and memory consumption by up to 96%, 71% and 65%, respectively. Moreover, on some conjunctive expression workloads, A-Tree achieves even better matching performance than *state-of-the-art* conjunctive Boolean expression matching algorithms.

Zusammenfassung

Event Routing und Boolean Expression Matching sind zwei Schlüsselfunktionen für Content-basierte Publish/Subscribe-System. Darüber hinaus spielt das Boolesche Expression-Matching eine wichtige Rolle in anderen stark wachsenden Anwendungsbereichen, wie der Online-Werbung, der Online-Nachrichtenverbreitung und dem Workflow-Management. Die vorhandenen Routing- und Booleschen Expressions-Matching-Lösungen weisen jedoch Einschränkungen in Bezug auf ihre Flexibilität, Leistung, Ausdruckskraft und Anwendbarkeit auf. Um diese Einschränkungen zu überwinden, schlagen wir neue effiziente Routing- und Matching-Algorithmen vor.

Obwohl der Filter-basierte Routing (FBR) Algorithmus ineffizient ist und unter Flexibilitätseinschränkungen leidet, wird er häufig in Content-basierten Publish/Subscribe-Systemen (Pub/Sub-Systemen) verwendet. Um diesen Einschränkungen zu begegnen, schlagen wir einen dynamischen, zielbasierten Routing-Algorithmus namens D-DBR vor. Dieser teilt Pub/Sub-Systeme in zwei unabhängige Vorgänge auf: Content-basierte Matching und zielbasiertes Multicasting. Der D-DBR Algorithmus weist niedrige Kosten für das Event-Matching und eine hohe Effizienz, Flexibilität und Robustheit für das Event-Routing in kleinen Overlay Netzwerken auf. Um die Skalierbarkeit zu erhöhen, ergänzen wir den D-DBR Algorithmus um einen neuen Routing-Algorithmus namens MERC. MERC teilt das Overlay in miteinander verbundene Cluster auf und wendet inhaltsbasierte und zielbasierte Mechanismen an, um Ereignisse innerhalb und zwischen (intra bzw. inter) Clustern zu routen. Experimentelle Ergebnisse zeigen, dass unsere Algorithmen den FBR Algorithmus hinsichtlich des Ereignisdiffusionsdurchsatzes um bis zu 700% übertrifft und die Ende-zu-Ende-Latenz um bis zu 55% reduziert.

Das konjunktiven Boolean Expression Matching ist eine wichtige Funktion für viele Anwendungen, unter anderem für Content-basierte Publish/Subscribe-Systeme. Die bestehenden Lösungen können jedoch nur eingeschränkt auf hochdimensionale Arbeitslasten und Umgebungen angewendet werden. Um diese Einschränkungen zu überwinden, entwickeln wir eine Datenstruktur namens PS-Tree, die Subskriptionen effizient in einer Dimension indizieren kann. Indem einzelne Prädikate in disjunkte Prädikatenräume aufgeteilt werden, erreicht PS-Tree eine hohe Matching-Leistung. Basierend auf PS-Tree, schlagen wir zuerst den konjunktiven Booleschen Ausdruck

Matching-Algorithmus PSTBloom vor. Durch das effiziente Filtern eines großen Teils von nicht übereinstimmenden Subskriptionen erzielt PSTBloom eine hohe Trefferquote, insbesondere in hochdimensionalen Anwendungen. Der PSTBloom Algorithmus und die verwendete Datenstruktur ermöglichen es außerdem den Index schnell aufzubauen. Darüber hinaus hat die Datenstruktur einen geringen Speicherbedarf. Im Vergleich zum aktuellen Stand der Technik zeigen umfangreiche Experimente, dass PSTBloom die Matching-Zeit, die Index-Bauzeit und den Speicherverbrauch um bis zu 84%, 78% bzw. 94% reduziert. Obwohl PSTBloom für viele Arbeitslastverteilungen effektiv ist, stellen sogenannte dichte Arbeitslasten neue Herausforderungen für PSTBloom und andere Algorithmen dar. Um diese effizient zu handhaben, schlagen wir außerdem den PSTHash-Algorithmus vor, der Subskriptionen in disjunkte mehrdimensionale Prädikaträume unterteilt. Umfassende Experimente an synthetischen und realen Datensätzen zeigen, dass PSTHash die Matching-Leistung für dichte Arbeitslasten um bis zu 92% verbessert.

Die effiziente Auswertung einer großen Anzahl von beliebigen Booleschen Ausdrücken spielt in vielen Anwendungen eine wichtige Rolle, z. B. für den Werbeaustausch, das automatische Targeting und die Veröffentlichung von Abonnentensystemen. Die meisten vorhandenen Lösungen unterstützen jedoch nur konjunktive Boolesche Ausdrücke. Die begrenzte Anzahl von Lösungen, die direkt mit beliebigen Booleschen Ausdrücken arbeiten können, weisen Anwendbarkeits- und Leistungseinschränkungen auf. Darüber hinaus ist es nicht immer effektiv, beliebige boolesche Ausdrücke in eine konjunktive Form zu normalisieren und dann vorhandene Verfahren zum Auswerten solcher Ausdrücke zu verwenden, da die Größe der auszuwertenden Ausdrücke exponentiell zunimmt. Aus diesen Gründen schlagen wir eine neuartige A-Baum-Datenstruktur vor (z. E. A-Tree), um beliebige Boolesche Ausdrücke effizient zu indizieren. Der A-Tree ist ein dynamischer Mehrwurzelbaum (z. E. “multi-rooted tree”), in dem Prädikate und Sub-Ausdrücke aus verschiedenen beliebigen Booleschen Ausdrücken aggregiert und geteilt werden. A-Tree verwendet dynamische Self-Adjustment-Richtlinien, um den Index anzupassen wenn sich die Arbeitslast ändert. Unsere umfassenden Experimente zeigen, dass A-Tree die vorhandenen Algorithmen für das Matching von Booleschen Ausdrücken bezüglich der Parameter: Matching-Leistung, der Indexaufbauzeit und des Speicherverbrauchs um bis zu 96%, 71% bzw. 65% übertrifft. Darüber hinaus erreicht A-Tree sogar bei konjunktiven Expression-Arbeitslasten eine bessere Matching-Rate als die meisten existierenden konjunktiven Booleschen Expression-Matching-Algorithmen, wie beispielsweise BETree und OpIndex.

Contents

Chapter 1

Introduction

1.1 Motivation

Due to its asynchronous nature and inherent decoupling properties, the distributed content-based publish/subscribe paradigm (`pub/sub`, for short) has been widely used in the design of many distributed applications, such as online news dissemination [42], workflow management [59], business process execution and monitoring [39], multi-player online gaming [11], network management and monitoring [48], and distributed system monitoring [61], to only name a few.

In the distributed `pub/sub` paradigm, a number of *brokers* are interconnected as an *overlay network* to provide the *event matching* and *event notification* service. The function of *event matching* is to retrieved the matching subscriptions for a incoming event, while the function of *event notification* is to route events from a *publisher* to interested *subscribers*.

The event routing and Boolean expression matching algorithms employed by a `pub/sub` system is crucial to managing performance, load distribution, and scalability. Moreover, the Boolean expression matching also plays an important role in a growing number of other applications, such as online advertising [43] and advertising exchanges [30]. However, the existing routing and matching solutions present limitations on flexibility, performance, expressiveness and applicability. To overcome these limitations, we propose new efficient content-based `pub/sub` routing and Boolean expression matching algorithms.

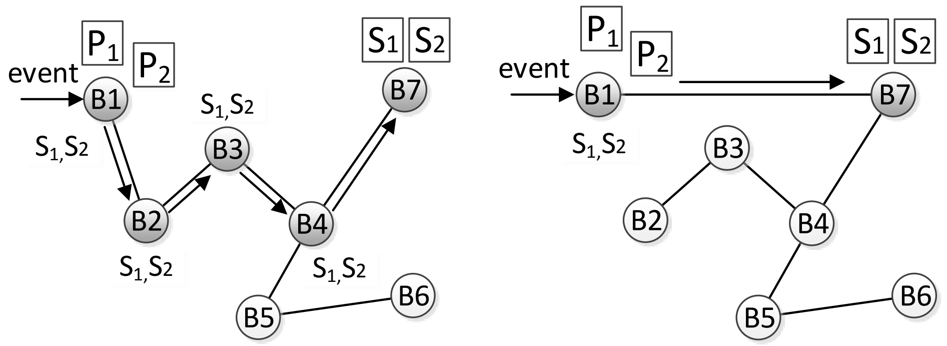
1.2 Problem Statement

It is a challenging undertaking to design efficient and scalable routing algorithms for `pub/sub` for two main reasons: First, a publisher does not know its interested subscribers in advance. Brokers need to identify interested subscribers for every issued event by matching the event against interest specifications in the form of subscriptions. Second, subscribers' interests are often highly diversified. The size of the potential destination set is usually very large and it is difficult to efficiently route each event to its destinations.

The design of efficient event routing protocols, especially in content-based `pub/sub` systems, is still an active area of research [38, 45, 65]. We observe that even the most widely used filter-based routing algorithm (FBR)[1] [14, 22, 36] suffers from the following four limitations: (1) Difficulty in supporting general overlay topologies, (2) subscription duplication, (3) redundant and repeated event matching, and (4) lack of flexibility in supporting overlay reconfiguration.

Difficulty in Supporting General Overlay Topologies: The original FBR algorithm was designed only for acyclic overlay networks [14, 45]. This design choice makes it difficult to support fault-tolerance and load balancing. Although, Li *et al.* extended FBR to support general overlays [38], their solution relies on two sources of additional complexity. First, advertisement broadcasting, which can generate a large number of redundant messages that need to be detected and eliminated. Second, a single subscription needs to be delivered to the same broker more than once, if the broker publishes more than one advertisement intersecting that subscription.

[1]In this thesis, we assume the advertisement mechanism is adopted in FBR, since it often improves the algorithm's scalability [47].



(a) Old overlay topology (b) New overlay topology

Figure 1.2.1: Limitations of the FBR algorithm

Subscription Duplication: In FBR, a subscription is stored at every broker along the routing paths from the source broker of that subscription to brokers with matching advertisements. For example, in Fig. 1.1(a), subscriptions S_1 and S_2 are redundantly stored at brokers B_1, B_2, B_3, B_4, and B_7. In many applications, the number of subscriptions is very large: In financial applications, there can be thousands of subscriptions [53] and security applications may have millions of subscriptions [19]. In these scenarios, important resources are used for storing the duplicated subscriptions.

Redundant and Repeated Event Matching: The FBR algorithm repeats the event matching operation at every intermediate broker along the paths from the publisher to interested subscribers. In the example in Fig. 1.1(a), an event issued at B_1 is repeatedly matched against all subscriptions stored at B_1, B_2, B_3, and B_4, before it is delivered to B_7. Since matching an event against a large number of subscriptions can be expensive, brokers may suffer from matching overhead and become overloaded.

Lack of Flexibility in Supporting Overlay Reconfiguration: As shown in Fig. 1.1(a), assuming the messaging load between brokers B_1 and B_7 is heavy, if the overlay were adjusted to that in Fig. 1.1(b), the routing efficiency would be improved. However, it is difficult to reconfigure the overlay in FBR because the routing state is stored at all intermediate brokers. Yoon *et al.* proposed three primitive operations to address this issue [65], yet, challenges such as how to generate the right sequence of operations have only recently been sketched and need further refinement [66].

Designing efficient conjunctive Boolean expression matching algorithms is also challenging for at least five main reasons. First, the algorithm must scale to a large number of Boolean expressions (i.e., a *subscription* in `pub/sub`) defined over a high-dimensional space. Second, the subscriptions may be unevenly distributed over the available dimensions and highly concentrated in some dimensions, which results in dense workloads; the algorithm should be able to efficiently handle dense workloads. Third, fast subscription-index construction and dynamic index updates need to be supported to accommodate changing interests. Fourth, the algorithm should be efficient at handling a high rate of arriving events on the premise of low Boolean expression matching latency. Last, the algorithm should support a rich subscription language to enable expressive modeling of user interests.

A *dimension* refers to the value domain underlying an attribute in a subscription. When all or part of the dimensions is associated with a large number of subscriptions, we consider the workload to be *dense*.

A large number of conjunctive Boolean expression matching algorithms exist [3, 27, 63, 56, 57, 58, 10, 68, 52, 51, 28, 44]. However, the existing solutions continue to suffer from limitations that affect performance and applicability. For example, `Propagation` [27] suffers from costs incurred while processing the predicate bit vector that tracks the matching predicates of a given event. `k-index` [63] does not support dynamic subscription updates. `OpIndex` [68] possesses high index construction costs when the arrival of subscriptions and events overlaps. Moreover, the existing solutions are not effective at handling dense workloads, which is an important property that is rarely considered by state-of-the-art algorithms.

The current solutions for matching Boolean expressions are primarily limited to the simple conjunctive Boolean expressions [68, 27, 63, 3]. Although this restriction is reasonable for many applications, some emerging applications require support for arbitrarily complex Boolean expressions, such as advertising exchange [30].

An advertising exchange is an electronic hub that connects online publishers to advertisers through intermediaries. An ad exchange can be represented as a directed graph, where the nodes are publishers, advertisers and intermediaries. An edge exists between node N_a and node N_b if N_a agrees to sell advertisement slots associated with user visits to N_b. Each edge is annotated with a Boolean expression that restricts the set of user visits that can be sold through that edge. When a user visits a publisher's web page, the user visit can be viewed as an attribute-value assignment. The goal is to rapidly find all the ad campaigns that can be satisfied by the user visit such that the best advertisement can then be selected to be shown to the user. In other words, the exchange has to rapidly evaluate arbitrary Boolean expressions to determine which expression satisfies the given assignment of attributes to values.

Designing efficient arbitrary Boolean expression matching algorithms is more challenging than conjunctive Boolean expressions matching for two reasons. First, in contrast to conjunctive Boolean expressions, arbitrary Boolean expressions contain not only *and* but also *or*, *not*, and other logical operators. Moreover, the same predicate can appear more than once in a single arbitrary Boolean expression. These features bring expressiveness and flexibility benefits, but they also introduce matching complexity. Second, different arbitrary Boolean expressions may contain a large number of common predicates and subexpressions. Efficiently identifying and sharing these predicates and subexpressions is necessary.

To the best of our knowledge, there are currently only four algorithms that have been proposed that can directly work on arbitrary Boolean expressions: `Dewey ID` [30], `Interval ID` [30], `BoP` [7] and `BDD` [10]. Moreover, these solutions exhibit limitations that affect their applicability and performance. The `Dewey ID` and `Interval ID` methods can only work in offline mode, which means that the Boolean expressions need to be known in advance, and the index cannot be dynamically changed after being built. The `BoP` method is an extension of the count-based conjunctive Boolean expression matching algorithms [68, 27, 63, 3]. However, `BoP` does not achieve comparable matching performance since filtering out nonmatching expressions based on the minimum number of matching predicates is inefficient for arbitrary Boolean expressions. Theoretically, `BDD` could support arbitrary Boolean expressions. However, this support was not implemented or verified in the published paper [10]. We first implemented `BDD` for arbitrary Boolean expression matching. `BDD` is not efficient since the tree could be very complex when there is a large number of distinct predicates in the system.

Many conjunctive Boolean expression matching algorithms have been proposed [3, 27, 63, 56, 57, 58, 10, 68]. A simple method is to translate each arbitrary Boolean expression into a set of conjunctive Boolean expressions and then utilize existing conjunctive Boolean expression matching algorithms. However, theoretically, the number of resulting conjunctive Boolean expressions can exponentially increase with the size of an arbitrary Boolean expression due to normalization [64, 7]. The translation introduces memory, matching and maintenance costs. Experiments on both synthetic and real-world workloads also show that this method is not effective.

1.3 Approach

We propose new efficient `pub/sub` routing, conjunctive Boolean expression matching and arbitrary Boolean expression matching algorithms to overcome the limitations of the existing solutions.

We observe that the aforementioned limitations of FBR result from the tight-coupling of event matching and event routing. We therefore consider decoupling them to address these limitations. Note that the concept of decoupling event matching and event routing in `pub/sub` is also explored in some related work such as link-matching [5], MEDYM [12] and DRP [13]. These approaches assume that a fully-connected topology or a statically configured, fixed maximum number of brokers are available. These assumptions may not always hold in practice, especially, when brokers are distributed across domains with restricted access and brokers may join or leave dynamically.

We continue to explore the idea of decoupling event matching and event routing to enable scalable `pub/sub` in a general and dynamic topology. We first propose a dynamic destination-based routing algorithm called D-DBR, which decouples the `pub/sub` system into two independent layers: Content-based matching and destination-based multicasting. The matching layer is responsible for subscription and event matching, whereas the multicasting layer is responsible for topology maintenance and message routing. When a message (i.e., advertisement, subscription, or event) is issued, it is first submitted to the matching layer to identify destination brokers. Then, the message is annotated with the addresses of those brokers and delivered to them via the multicasting layer. In D-DBR, subscriptions are not stored at any intermediate broker. An event only needs to be matched at its source and destination brokers. Changes of the overlay topology have no effect on the matching layer. Consequently, supporting a general overlay and dynamic overlay reconfiguration for fault-tolerance and performance optimization become straight forward, as we illustrate below.

D-DBR requires additional processing for the destination address list associated with each message. If the address list is long, more bandwidth is used. However, our extensive evaluations show that the resulting overhead is small, especially for small-sized overlays (with tens of brokers). For example, in an overlay with 70 brokers, when an event is delivered to 35 destination brokers, on average, each of the generated messages carries only 2.51 destination addresses (here, the average connection degree of the brokers is 3).

Although D-DBR is an effective solution, a factor limiting its scalability is that each broker needs to know all other brokers in the system, and thus, the topology maintenance cost can be expensive for large-scale overlays (with hundreds or more brokers). To mitigate this issue and to further boost scalability, we propose a new routing protocol called MERC — Match at Edge and Route intra–Cluster, — which is based on and complements D-DBR. MERC divides the overlay into interconnected clusters, where it applies content-based and destination-based mechanisms for inter- and intra-cluster event routing, respectively. In MERC, each broker only needs to be aware of brokers in the clusters it belongs to. As a result, the destination list overhead is mitigated, the topology maintenance cost is reduced, and the impact of change in one cluster can be isolated from brokers in other clusters.

We implemented both algorithms, D-DBR and MERC, in PADRES, an open-source content-based `pub/sub` system [40]. Our experimental results show that our algorithms outperform FBR in terms of improving the throughput by up to 700% and reducing the communication latency by up to 55% with an acceptable overhead.

To design efficient conjunctive Boolean expression matching algorithms, we first propose the *Predicate Space Tree* (`PS-Tree`), a data structure used to index subscriptions in one dimension. Then, we propose the `PSTBloom` and `PSTHash` matching algorithms for high-dimensional and dense workloads, respectively. In `PS-Tree`, a predicate of a subscription is regarded as a space, and an attribute-value pair of an event is regarded as a point. `PS-Tree` divides predicates into disjoint predicate spaces and maintains a many-to-many relationship between predicate spaces and subscriptions. Through `PS-Tree`, the problem of matching an attribute-value pair against a set of subscriptions is transformed into the problem of locating the predicate space to which the attribute-value pair belongs; `PS-Tree` efficiently solves this problem.

The example in Fig. 1.3.1 illustrates the relationship that `PS-Tree` maintains. In this example, there are two subscriptions: $S_1\{price, in, [0,4]\}$ and $S_2\{price, in, [2,4]\}$. The two predicates involved are divided into two disjoint predicate spaces: $[0,2)$ and $[2,4]$. The first predicate space $[0,2)$ is associated with $\{S_1\}$, while the second predicate space $[2,4]$ is associated with $\{S_1, S_2\}$. For an attribute-value pair, $\langle price, 3\rangle$, after determining the predicate space $[2,4]$ to which it belongs, the matching subscriptions $\{S_1, S_2\}$ can be directly retrieved.

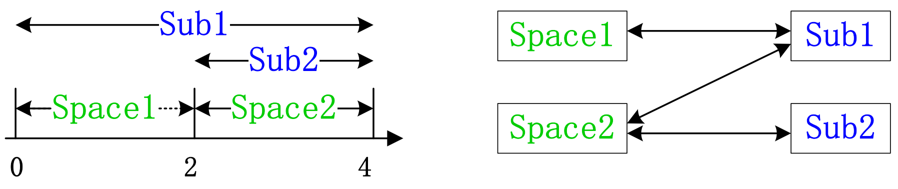


Figure 1.3.1: Relationship maintained by PS-Tree

Based on `PS-Tree`, we first propose the `PSTBloom` algorithm. For each subscription S in `PSTBloom`, one of its predicates is selected as the *access predicate*. Access predicates are evaluated before other predicates to retrieve candidate subscriptions, i.e., subscriptions that are likely to match the given input event. The access predicate is divided into several disjoint predicate spaces. Each predicate space corresponds to a leaf node in `PS-Tree`. S is associated with those leaf nodes that correspond to its access predicate. For an input event, each attribute-value pair of that event is matched against a corresponding `PS-Tree` to determine a set of partially matching subscriptions. These candidate subscriptions are then pruned by the Bloom filter signatures of the related leaf nodes. The signature of a leaf node is the Bloom filter created from subscription IDs. `PSTBloom` achieves good matching performance since a large proportion of unmatching subscriptions can be efficiently filtered out. Our comprehensive experiments show that `PSTBloom` outperforms state-of-the-art algorithms in terms of not only matching time but also index construction time and memory consumption.

`PSTBloom` is effective for many types of workloads, especially high-dimensional workloads, because `PSTBloom` can efficiently locate the small number of subscriptions whose access predicates are satisfied by an event. However, dense workloads present new challenges to `PSTBloom` and other algorithms: the number of partially matching candidate subscriptions could be large for an event. To overcome this limitation, we further design the `PSTHash` algorithm. `PSTHash` is also based on `PS-Tree`. However, in contrast to `PSTBloom`, `PSTHash` selects more than one predicate as *access predicates* for each subscription. These access predicates are divided into a set of disjoint multi-dimensional predicate spaces. `PSTHash` constructs a many-to-many relation between the multi-dimensional predicate spaces and the subscriptions. When an event is received in `PSTHash`, each attribute-value pair of that event is matched against a corresponding `PS-Tree` to determine the predicate space to which the attribute-value pair belongs. Then, a number of multi-dimensional predicate spaces is constructed. Through these multi-dimensional predicate spaces, the candidate subscriptions can be directly obtained for that event. `PSTHash` identifies fewer candidate subscriptions since `PSTHash` guarantees that all the selected access predicates in each candidate subscription have matching attribute-value pairs in the event. Under dense workloads, our experiments show that `PSTHash` achieves the best matching performance among the algorithms that we evaluated.

To design efficient arbitrary Boolean expression matching algorithms, we propose the aggregate tree (`A-Tree`) data structure to directly index arbitrary Boolean expressions. The basic idea of `A-Tree` is to represent an arbitrary Boolean expression as an *n-ary* tree and then combine these *n-ary* trees together into an aggregated tree with multiple roots. Common predicates and subexpressions are shared to reduce memory usage and improve matching performance. `A-Tree` supports dynamic self-adjustment based on the workloads. In contrast to `Dewey ID` [30] and `Interval ID` [30], `A-Tree` works in online mode, which means that the `A-Tree` index does not need to be built in advance and can be updated on the fly.

1.4 Contributions

In this section, we describe the contributions of this work with regard to each of the described problems. To provide efficient content-based `pub/sub` routing solutions, we make the following contributions:

- We propose the D-DBR algorithm, which overcomes the aforementioned limitations of the FBR algorithm and supports dynamic overlay reconfiguration.
- We propose the MERC routing protocol which routes events based on their content and destination information to enable a high degree of scalability.

To provide efficient conjunctive Boolean expression matching solutions, we make the following contributions:

- We propose the `PS-Tree` index, which achieves high predicate matching performance, exhibits high expressiveness and supports dynamic subscription updates.
- We propose `PSTBloom`, which presents advantages over existing solutions for reducing not only the Boolean expression matching latency but also the memory consumption and index construction latency.
- We propose `PSTHash`, which achieves the best matching performance given dense workloads.

To provide efficient arbitrary Boolean expression matching solutions, we make the following contributions:

- We develop the `A-Tree` data structure to efficiently index arbitrary Boolean expressions. `A-Tree` uniquely represents every shared predicate and subexpression by a single node to reduce memory cost and improve matching performance. Expression reorganization and index self-adjustment are supported to further optimize the index.

- Based on `A-Tree`, we propose an event matching algorithm that supports different logical operators. Moreover, the *zero suppression filter* and *propagation on demand* optimization methods are used to further reduce the matching cost.

Parts of the content and contributions of this work have been published or are submitted for review:

- Shuping Ji, C. Ye, J. Wei, and Hans-Arno Jacobsen. Merc: Match at edge and route intra–cluster for content-based publish/subscribe systems. In Proceedings of the 16th ACM/IFIP/USENIX Middleware Conference, pages 13–24. ACM, 2015.
- Shuping Ji, and Hans-Arno Jacobsen. PS-Tree-Based Efficient Boolean Expression Matching for High-Dimensional and Dense Workloads. PVLDB, 12(3): 251-264, 2018.
- Shuping Ji, and Hans-Arno Jacobsen. A-Tree: A Dynamic Data Structure to Efficiently Index Arbitrary Boolean Expressions. Technical Report, 07/2018 (submitted for review).

1.5 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we review the existing content-based `pub/sub` routing and Boolean expression matching algorithms. In Chapter 3, we present the expression language and matching semantics. The D-DBR algorithm is presented in Chapter 4. The MERC algorithm is presented in Section 5. In Chapter 6, we present the design of the `PS-Tree` index. In Chapter 7, we describe the design of the `PSTBloom` algorithm. In Chapter 8, we present the design of the `PSTHash` algorithm. The `A-Tree` data structure, index construction, event matching, the optimizations and property analysis are presented in Chapter 9. Chapter 10 reports extensively on our experimental results, and Chapter 11 concludes the thesis.

CHAPTER 2

Related Work

2.1 Publish/Subscribe Routing

In this section, we categorize existing event routing protocols for content-based `pub/sub` systems into three classes: Event flooding (EF) [21], multicast-based routing (MBR) [50, 54], and filter-based routing (FBR) [14, 22, 29, 36]. These solutions balance the tradeoff between routing accuracy, subscription duplication, redundant matching, overlay reconfigurability, and scalability differently.

Routing accuracy is an important criteria to evaluate a `pub/sub` routing algorithm's network efficiency. It is defined as the ratio between the number of brokers interested in an event over the number of brokers receiving that event when it is routed towards the interested brokers. We say a broker is *interested* in an event, if it serves a subscriber with subscriptions matching the event. For example, if an event has 10 interested brokers, but in total it is delivered to 20 brokers, — on its way toward the 10 interested brokers, — then, the routing accuracy for this event is 50%.

Event Flooding: In EF, each event originating at a publisher is flooded to all brokers and then matched against local subscriptions at every broker. This routing protocol is stateless and easily adapts to broker overlay changes. It also has low event matching overhead and saves memory resources by only storing local subscriptions at each broker. However, it suffers from the shortcoming that events may have to be propagated to a large number of uninterested brokers. In a general overlay with cycles, a large number of redundant messages may result. Therefore, this routing protocol suffers from low routing accuracy and is not scalable.

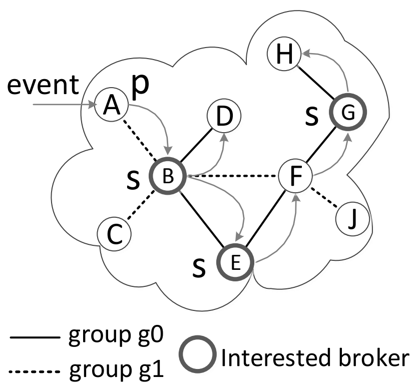


Figure 2.1.1: MBR algorithm

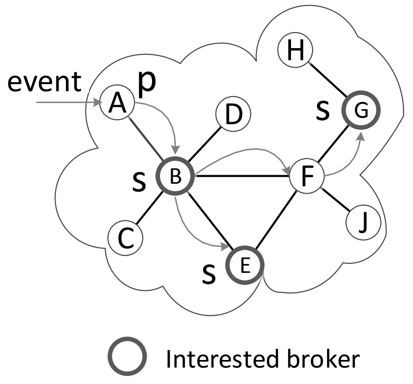


Figure 2.1.2: FBR algorithm

Gossip-style `pub/sub` [20] is in this category. It is an extension of the EF approach and uses gossiping to distribute events. This approach eases the problem of event duplication. However, events are not guaranteed to be delivered to all interested subscribers. Moreover, events need to be cached and additional memory resources are required.

Multicast-based Event Routing: In MBR, the event space is partitioned into a large number of disjoint multicast groups, for each of which a multicast tree is built. When an event is issued, it is first mapped to an appropriate group and then multicast on the corresponding spanning tree. For example, in Fig. 2.1.1, groups g_0 and g_1 are pre-computed. When an event is issued at Broker A, it may first be mapped to group g_0 and then it is multicast on g_0's spanning tree.

MBR improves the routing accuracy. In addition, existing multicast techniques, such as IP multicast and application-level multicast [25, 50] can be used to support event propagation. However, this solution may require a large number of groups, exponential in the number of brokers in the worst case.

To trade off the overhead, usually only a limited number of multicast groups are constructed [2]. In this case, subscribers with different interests may be clustered into the same group and events are forwarded to some uninterested brokers. For example, in Fig. 2.1.1, the issued event is forwarded to all brokers in the spanning tree of group g_0, among which, D, F, and H are uninterested brokers.

To improve the routing accuracy and reduce the bandwidth overhead, MBR can apply clustering algorithms [54, 60] to identify multicast groups. However, the effectiveness of clustering heavily depends on event and subscription locality in applications. As a result, this approach is not applicable to scenarios where subscriptions are highly diversified and workloads change over time.

Filter-based Routing: In FBR, advertisements are broadcast. Then, matching subscriptions are delivered to the advertisement source brokers in the reverse direction along the paths on which the advertisements were broadcast. Subsequently, routing trees rooted at source brokers are constructed. When an event is issued, routing decisions are made via successive content-based filtering at all brokers from the source to all destinations: Every broker along the way matches the event against its stored subscriptions and then forwards it only toward directions that lead towards matching subscriptions.

In FBR, every broker maintains a subscription routing table (SRT) and a publication routing table (PRT) for subscription and event delivery, respectively. An advertisement is stored in the SRT as a {*adv, lasthop*} tuple. The *lasthop* field indicates from which broker or client the advertisement came. A subscription is stored in the PRT as a {*sub, lasthop*} tuple. Again, *lasthop* also indicates via which broker or client the subscription came.

This approach has several advantages: First, it achieves better routing accuracy than MBR since events are only forwarded toward interested brokers. In the example presented in Fig. 2.1.2, the issued event is not forwarded to Broker D or H. Second, it scales well because each broker only needs to know its neighboring brokers. Finally, it is applicable to scenarios with highly diversified subscriptions. These advantages make FBR the most widely used routing protocol, despite the varied limitations discussed in Section 1.2.

A large body of work is dedicated to studying distributed content-based `pub/sub` systems. For example, designs include SIENA [14], Gryphon [5], JEDI [22], Herald [9], and PADRES [29, 40], to just name a few. Most of these systems adopt the idea of filter-based routing, which means the flow of events from publishers to subscribers is driven by the content of events at every routing step. An alternative is to deliver events between brokers directly based on their ultimate destination information. To the best of our knowledge, only few approaches explored this direction, e.g., the "*link matching*" algorithm [5], the MEDYM algorithm [12], and the DRP algorithm [13]. Since these approaches are closest in conception to our work, we focus in our below discussion on differentiating to them.

In the link matching algorithm, each broker has a copy of all present subscriptions, which are organized into a special data structure called parallel search tree (PST). Each broker calculates the subset of links to which an event should be forwarded. This algorithm achieves better matching performance than FBR but does not overcome the other limitations of FBR. In addition, the maintenance of PST introduces additional overhead, and it requires that subscriptions are not changed frequently, both are no concerns in our design.

MEDYM aims to provide ideal routing accuracy: Each event is only transferred to and through its interested brokers. To do so, every broker needs to build connections with all other brokers. As a result, a fully-connected overlay mesh is required. This requirement, however, is often hard to meet, especially, in large-scale networks. Different from MEDYM, our algorithms can be deployed on any kind of overlay network. Moreover, our approach can improve the routing efficiency by periodically adjusting the overlay topology based on the latest application workload and system load. Another shortcoming of MEDYM is that routing paths need to be dynamically computed for every event by every broker along the routing paths. This is a costly operation, especially, in sight of scenarios that process large numbers of events in an overlay with a large number of brokers. Our algorithms do not suffer from this limitation, since messages are always routed on the periodically updated dissemination paths.

Finally, DRP employs the idea of destination-based event routing. However, different from our algorithms, in DRP, destination addresses are stored in a fixed-size bit vector. This design makes the destination list overhead predictable for every message routed. However, in this way, DRP uses and potentially wastes more bandwidth than alternatives. Also, as a consequence, DRP limits the maximum number of brokers in the system. As a result, DRP exhibits scalability limitation, which, so far, have prevented it from reaching a wider adoption, as compared to FBR, for example.

2.2 Conjunctive Boolean Expression Matching

A large body of work is dedicated to studying Boolean expression matching in many contexts: trigger processing [15, 33], XPath/XML matching [26, 55, 37, 34], indexing in multi-dimensional space [41, 6, 31], and `pub/sub` matching [56, 68, 63, 27]. We concentrate on `pub/sub` matching since other contexts either use different languages or cannot scale to thousands of dimensions and millions of expressions. Existing solutions in `pub/sub` matching include `BE-Tree` [56, 57, 58], `OpIndex` [68], `Propagation` [27], `k-index` [63], `SIFT` [3], `Gryphon` [3], `H-Tree` [52], `TAMA` [69], `REIN` [51], `GEM` [28], and `SCAN` [3]. These solutions can be roughly classified into two classes: count-based methods and tree-based methods.

Count-based approaches [68, 27, 63, 3] usually design indexes to retrieve a set of candidate subscriptions. For these candidate subscriptions, count-based methods identify the matching subscriptions by counting the satisfied predicates of each subscription. Representative count-based approaches include `Propagation`, `k-index`, and `OpIndex`.

`Propagation` [27] creates indexes on predicates and maintains a *predicate bit vector*. Each predicate that occurs in one or more subscriptions is associated with a single entry in the predicate bit vector. When an event is received, `Propagation` first locates the satisfied predicates and sets related entries in the predicate bit vector to 1. Candidate subscriptions can be obtained through these predicates. Then, `Propagation` identifies a candidate subscription as matching if all of its predicates are satisfied. While `Propagation` is a novel algorithm, it suffers from the limitation that, to match each event, all bits in the predicate bit vector need to be reinitiated to 0. This operation is expensive if there are a large number of distinct predicates.

`k-index` [63] is an approach based on an inverted index. The basic idea behind `k-index` is to partition the subscriptions into subsets of predicates and to organize each predicate subset using the inverted list data structure. For each attribute-value pair in an incoming event, appropriate inverted list indexes are searched to identify predicates matching the attribute-value pair, and a counting method is used to determine the matching subscriptions for an event. `k-index` is effective in retrieving partially matching subscriptions; however, it suffers from the limitation that a range predicate must be rewritten to a disjunction of equality predicates, which increases the index's size due to the need for many inverted list entries for a single predicate.

`OpIndex` [68] is a state-of-the-art count-based method. This method adopts a two-level index structure and organizes subscriptions using inverted lists. In the first level, `OpIndex` selects an attribute for each subscription, and subscriptions with the same selected attribute are grouped together. In the second level, subscriptions are further partitioned based on their predicate operators. The predicates with the same operator are clustered together. For an incoming event, `OpIndex` retrieves all the satisfied predicates and then uses the counting method to locate the matching subscriptions. `OpIndex` has high matching performance. However, when the arrival of subscriptions and events overlaps, `OpIndex` suffers from high index construction costs since related indexes need to be reordered after each subscription is inserted. During the reordering process of an index, event matching is delayed, which results in high matching latency.

In contrast to count-based methods, tree-based methods [56, 3, 10] are designed to filter out unmatching subscriptions step by step. These methods usually recursively divide the search space by eliminating subscriptions upon encountering unsatisfied predicates. Compared to count-based methods, tree-based methods usually identify fewer candidate subscriptions, but the filtering process is more expensive.

`BE-Tree` [56], a representative tree-based method, uses a two-phase space cutting technique and organizes subscriptions in a tree index. The subscriptions are repeatedly partitioned by attribute, followed by value space partitioning. In `BE-Tree`, there are three classes of nodes: a partition node that maintains the space partitioning information (an attribute), a cluster node that maintains the space clustering information (a range of values), and a leaf node that stores the actual expressions. By configuring the maximum leaf node size, `BE-Tree` achieves a good trade-off between matching performance and memory consumption. However, `BE-Tree` indexes all subscriptions through a single tree, which constitutes a potential performance bottleneck.

`H-Tree` is also a tree-based method. `H-Tree` first selects a set of attributes to index. Then, the value domain of each attribute is divided into a number of overlapping cells. The cells of each attribute are connected level-by-level to form a tree. In this way, `H-Tree` divides the subscription space into a set of buckets. The number of buckets increases exponentially with the number of indexed attributes. As a result, `H-Tree` is not suitable for high-dimensional workloads.

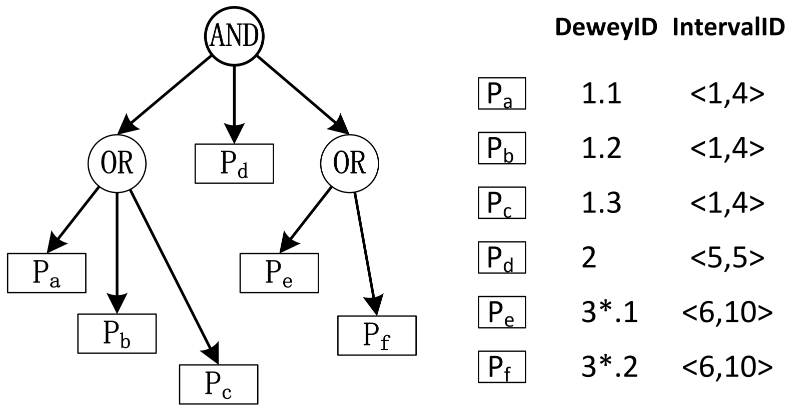
`GEM` [28] and `REIN` [51] are different from the count-based and tree-based methods presented above. The idea behind these two algorithms is to filter out the unmatching subscriptions one by one for each event. `GEM` and `REIN` are not suitable for scenarios in which the proportion of unmatching subscriptions is high. `TAMA` [69] is an approximate matching algorithm, which means that it can introduce false positives into the result set.

`PS-Tree` can be interpreted to store intervals and allows efficient querying of which stored intervals contain a given point. `Segment-Tree` [23] and `Interval-Tree` [18] are two index structures that provide similar capabilities. Thus, both approaches are related to `PS-Tree`. One limitation of `Segment-Tree` is that it is a static structure, meaning that `Segment-Tree` cannot be modified after it is built. A variant [4] of `Segment-Tree` supports dynamic updates at the cost of extra memory. `Interval-Tree` does not have this limitation but presents more expensive querying. Compared to `Segment-Tree` and `Interval-Tree`, `PS-Tree` achieves good querying performance, as shown in Section 10.2. Classical string and bit-vector tree indexing structures, such as a trie [24] and Radix-, Patrica- and Suffix-tree [46, 62], present a similar structure as that of `PS-Tree`. However, they are different because they do not provide the function to match attribute-value pairs against predicates.

2.3 Arbitrary Boolean Expression Matching

We categorize existing arbitrary Boolean expression matching algorithms into four classes: scan-based, count-based, tree-based and translation-based solutions.

Given an event, the naive matching method is to scan every arbitrary Boolean expression and match it against the event. This method is inefficient when there are a large number of expressions. The `Dewey ID` [30] and `Interval ID` [30] methods are variants of the scan-based method with two optimizations. First, an arbitrary Boolean expression is represented as a set of Dewey IDs or Interval IDs to reduce memory usage and improve evaluation performance. Second, for each event, the matching arbitrary Boolean expressions are retrieved through the matching Dewey IDs or Interval IDs. If matching Dewey IDs or Interval IDs do not exist, then the corresponding arbitrary Boolean expression will not be evaluated.

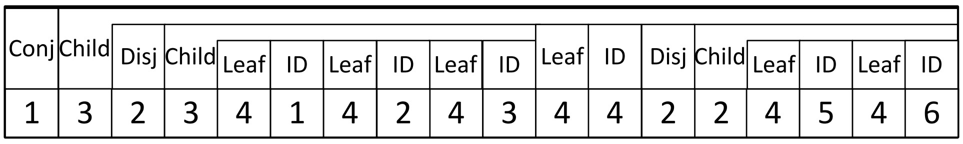


	DeweyID	IntervalID
P_a	1.1	<1,4>
P_b	1.2	<1,4>
P_c	1.3	<1,4>
P_d	2	<5,5>
P_e	3*.1	<6,10>
P_f	3*.2	<6,10>

Figure 2.3.1: Dewey ID and Interval ID Example

The `Dewey ID` and `Interval ID` methods work in two phases. The first phase is to annotate each arbitrary Boolean expression as a set of Dewey IDs or Interval IDs offline. Each ID corresponds to a leaf node. As shown in Fig. 2.3.1, the expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$ is annotated as the Dewey IDs $1.1, 1.2, 1.3, 2, 3*.1$, and $3*.2$ or Interval IDs <1,4>, <1,4>, <1, 4>, <5,5>, <6,10>, and <6,10>. These IDs correspond to the leaf nodes annotated by P_a, P_b, P_c, P_d, P_e and P_f, respectively. The second phase is the expression evaluation phase. Existing predicate matching solutions are used to retrieve the matching predicates. In this example, suppose that predicates P_a, P_d and P_e are matching. Then, the corresponding Dewey IDs $1.1, 2$, and $3 * .1$ or Interval IDs <1,4>, <5,5>, and <6,10> are used to compute whether the arbitrary Boolean expression is matching.

The `Dewey ID` and `Interval ID` methods are novel. However, these methods present two major limitations. First, all the expressions need to be known and annotated offline before the event matching starts, which limits its applicability. Second, different expressions are evaluated separately. The common subexpressions shared by different arbitrary Boolean expressions may be evaluated several times, which limits performance.



Conj	Child	Disj	Child	Leaf	ID	Leaf	ID	Leaf	ID	Leaf	ID	Disj	Child	Leaf	ID	Leaf	ID
1	3	2	3	4	1	4	2	4	3	4	4	2	2	4	5	4	6

Figure 2.3.2: BoP Encoding Example

The `BoP` [7] algorithm is a count-based method. Similar to the count-based conjunctive Boolean expression matching algorithms [27, 63, 68], the basic idea of `BoP` is to compute the minimum number of predicates that need to be satisfied for each arbitrary Boolean expression. Only when the number of satisfied predicates is greater than the minimal number is the expression further evaluated. Another optimization of the `BoP` algorithm is that an arbitrary Boolean expression is encoded into a compressed format to reduce the memory cost. For example, the expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$ is encoded into the format in Fig. 2.3.2.

The limitation of the `BoP` algorithm is that a nonmatching arbitrary Boolean expression may be incorrectly identified as a candidate when the minimum number of predicates is satisfied. Take the expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$ as an example. The minimum number of satisfied predicates is 3. When P_a, P_b, and P_c are matching is that expression identified as a candidate. However, that expression is not actually matching. Another limitation of `BoP` is that different candidate expressions are evaluated separately. The shared subexpressions may be evaluated several times, which further limits `BoP`'s matching performance.

A binary decision diagram (`BDD`) is a data structure that is used to represent a

Boolean function [8], and it has been successfully used in verification methods such as model checking [17]. Paper [10] uses shared BDDs to represent arbitrary Boolean expressions by assigning a Boolean variable to each unique predicate. Fig. 2.3.3 shows the BDD data structure of the expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$.

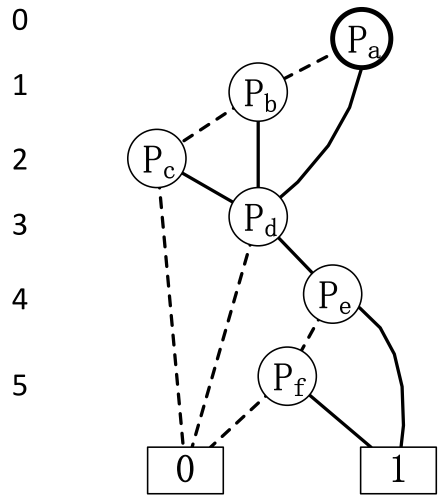


Figure 2.3.3: BDD Structure Example

BDD exploits the commonality (i.e., shared subexpression) between different arbitrary Boolean expressions. Consequently, the common subexpressions, corresponding to the BDD nodes, are evaluated only once for event matching. However, to represent arbitrary Boolean expressions with a BDD, each distinct predicate in the system needs a variable. When the number of variables becomes large, the BDD structure can become very complex since the size of a BDD may be exponential in the number of variables in the worst case. Moreover, the evaluation of events with BDDs involves the traversal of the entire BDD. Consequently, BDD has a high index construction cost, and the matching performance is limited.

Many conjunctive Boolean expression matching algorithms have been proposed, such as BE-Tree [56] and OpIndex [68]. The idea of translation-based methods is to first translate each arbitrary Boolean expression into a set of conjunctive Boolean expressions and then utilize the conjunctive Boolean expression matching algorithms. Take the expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$ as an example; it will be translated into six conjunctive Boolean expressions: $P_a \wedge P_d \wedge P_e$, $P_a \wedge P_d \wedge P_f$, $P_b \wedge P_d \wedge P_e$, $P_b \wedge P_d \wedge P_f$, $P_c \wedge P_d \wedge P_e$, and $P_c \wedge P_d \wedge P_f$. When any of these six expressions is matching, the arbitrary Boolean expression is matching. This is a simple method; however, it introduces translation and maintenance costs. Given the exponential increase in the size of expressions due to translation, the memory consumption increases and the matching performance deteriorates. Consequently, the translation-based method is not always effective for arbitrary Boolean expression matching.

Chapter 3

Matching Model

In this section, we first present the definitions of predicates, conjunctive Boolean expressions, a.k.a, subscriptions, arbitrary Boolean expressions, and events. Then, we specify the matching semantics. Finally, we review predicate selectivity, which is used to determine access predicates.

3.1 Expression Language

Predicate: A predicate is a triple consisting of an attribute, an operator and a set of values. A predicate is denoted as $P^{attr,op,vals}(x)$, or more concisely as $P(x)$. The attribute name P^{attr} uniquely represents a dimension. A predicate may contain more than one value. The number of values is determined by the operator. For example, if the operator is "$=$", the predicate contains a single value; if the operator is "$\in$", the predicate contains a set of values.

The expressiveness of the predicates is determined by the supported attribute value types and operators. To achieve high expressiveness, the predicates in our expression language support the standard relational operators $(<, \leq, =, \neq, >, \geq)$ and set operators $(\in, \notin)$ as well as SQL's BETWEEN operator (in) on numerical, enumeration, and string domains.

Subscription: A subscription is a conjunctive Boolean expression over predicates. Suppose that the total number of dimensions is n. Formally, a subscription S is defined over an n-dimensional space as follows:

$$S = \{P_1^{attr,op,vals}(x_1) \wedge ... \wedge P_k^{attr,op,vals}(x_k)\}, \quad k \leq n$$

Different predicates in the same subscription are required to belong to different dimensions: $P_i^{attr} \neq P_j^{attr},\ if\ i \neq j$. We refer to the size of a subscription S, denoted by $|S|$, as the number of predicates in S.

An *arbitrary Boolean expression* (`ABE`) is a Boolean function over a set of predicates. A Boolean function takes Boolean variables as inputs and produces a Boolean as output. Formally, an `ABE` is defined as follows:

$$ABE = f(P_1, ..., P_m)$$

The supported logical operators in the Boolean function f are *and*, *or*, *not*, *xor* and *xnor*. Note that it is allowed for the same predicate to appear in the `ABE` more than once, which is generally not supported by a conjunctive Boolean expression. This feature improves the expressiveness of arbitrary Boolean expressions. For example, in the arbitrary Boolean expression $(P_a \wedge P_b) \vee (\neg P_b \wedge P_c)$, the predicate P_b appears twice. We refer to the size of an arbitrary Boolean expression ABE, denoted as $|ABE|$, as the number of predicates in ABE.

Event: An event contains a set of attribute-value pairs. Formally, an event E is defined over an n-dimensional space as follows:

$$E = \{\langle attr_1, val_1\rangle, \dots , \langle attr_k, val_k\rangle\}, \quad k \leq n$$

For the same event, different attribute-value pairs are required to belong to different dimensions: $attr_i \neq attr_j, \;\; if\ i \neq j$. We refer to the size of an event E, denoted by $|E|$, as the number of attribute-value pairs in E.

3.2 Matching Semantics

A predicate accepts an input value x and outputs an evaluation result indicating whether that predicate constraint is satisfied. However, it is possible that an event only refers to a subset of attribute variables, which means that the attribute of a predicate does not appear in the event. To handle this case, we propose a new evaluation result called *undefined*.

$$P^{attr,op,vals}(x) \rightarrow \{true, false, undefined\}$$

A predicate $P^{attr,op,vals}(x)$ matches with an attribute-value pair $\langle attr, val\rangle$, denoted as $P^{attr,op,vals}(x) \simeq \langle attr, val\rangle$, if the following condition is satisfied:

$$\{P^{attr} = attr\} \wedge \{P^{attr,op,vals}(val) = true\}$$

If a predicate of a subscription S matches with an attribute-value pair $\langle attr, val\rangle$, we say that the subscription matches with that attribute-value pair, denoted by $S \simeq \langle attr, val\rangle$. Furthermore, a subscription S matches with an event E, denoted by $S \simeq E$, if the following condition is met:

$$\forall P(x) \in S \rightarrow \{\exists\langle attr, val\rangle \in E\} \wedge \{P(x) \simeq \langle attr, val\rangle\}$$

Given an event E and a set of subscriptions, retrieve all the subscriptions matched by E. We refer to this problem as the conjunctive Boolean expression matching problem.

An ABE matches with an event E, denoted as $ABE \simeq E$, if the following condition is satisfied:

$$ABE = f(P_1, ..., Pm) = true \mid E$$

Given an event E and a set of arbitrary Boolean expressions, retrieve all the expressions matched by E. We refer to this problem as the arbitrary Boolean expression matching problem.

3.3 Predicate Selectivity

Boolean expression matching can be seen as the process of filtering out unmatching subscriptions for a given event. We observe that different predicates have different pruning capacities, a.k.a., *predicate selectivity*. For a subscription S, we define the selectivity of its predicate $P(x)$ as the probability that S is identified as unmatching if $P(x)$ is used as the pruning predicate for an arbitrary event.

The selectivity of a predicate is determined by three factors: (1) the distribution of the event workload, (2) the operator of the predicate, and (3) the values of the predicate. Since the whole event workload is often unknown in advance, in our algorithms, we use statistics over historical events to calculate the selectivity of a predicate: a predicate with a higher number of matching events is considered to have lower selectivity. The advantage of our algorithms is that `PS-Tree` can be used to quickly obtain the number of matching historical events for a predicate. If no historical events exist, we heuristically rank the selectivity of predicates by their operators and values. The selectivity ranking of operators is $\{=\} > \{\in\} > \{in\} > \{<, \leq, >, \geq\} > \{\notin\} > \{\neq\}$. When the operators have the same selectivity, we consider predicates with wider value sets to have lower selectivity. For example, we consider the selectivity of $\{attr, in, [1, 10]\}$ to be lower than the selectivity of $\{attr, in, [1, 5]\}$.

Chapter 4

D-DBR Design

This section presents our D-DBR algorithm, which, as we demonstrate later, achieves better routing accuracy than EF and MBR, and overcomes the four limitations of FBR, presented above.

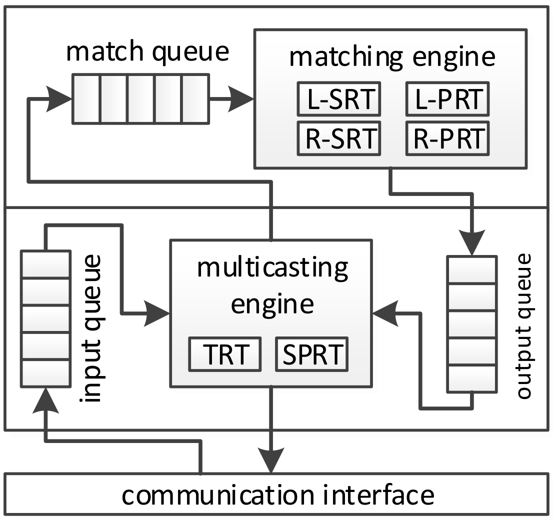


Figure 4.0.1: Layers of D-DBR

In addition, in D-DBR, the matching and the routing layers are decoupled, enabling the dynamic reconfiguration of the overlay. As a result, events can always be routed on periodically updated paths, and derived based on, for example, shortest-path and workload considerations. To enable optimized subscription propagation, D-DBR adopts the popular advertisement-based routing of FBR.

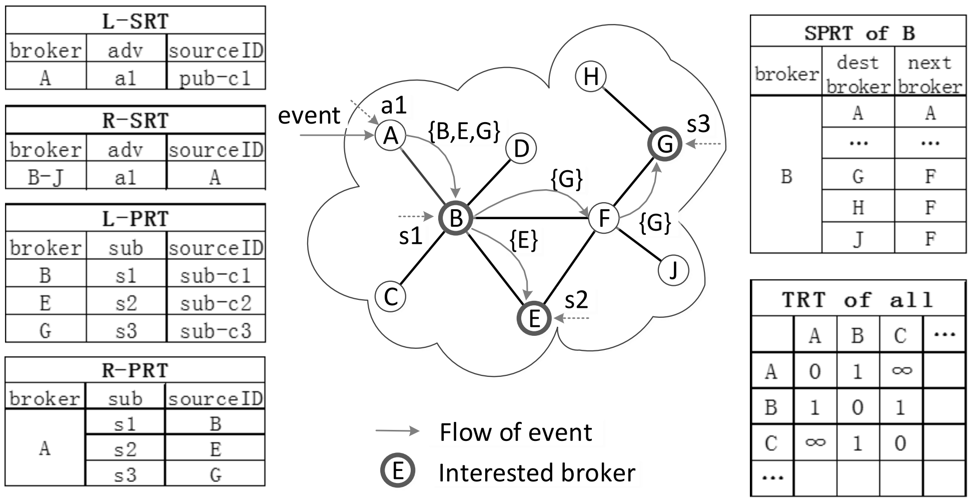
As shown in Fig. 4.0.1, in D-DBR, the `pub/sub` system is decoupled into two independent layers: Content-based matching and destination-based multicasting. The matching layer is responsible for event matching, whereas the multicasting layer is responsible for event routing. When a publisher issues an event at a broker, the event is matched against subscriptions managed by the broker's matching engine to obtain the addresses of brokers interested in the event (i.e., brokers hosting clients who are subscribed to the event.) The addresses are attached to the event. Then, the event is delivered to the interested brokers by the multicasting layer based on the event's destination addresses. Upon receiving an event, a destination broker matches the event against its local subscriptions and directly delivers it to the interested subscribers.

4.1 Content-based Matching Layer

Routing Tables: To reduce event matching cost, routing information (advertisements and subscriptions) from local clients and other brokers are stored separately. In total, each broker maintains four routing tables at the matching layer: *Local Subscription Routing Table* (**L-SRT**), *Remote Subscription Routing Table* (**R-SRT**), *Local Publication Routing Table* (**L-PRT**), and *Remote Publication Routing Table* (**R-PRT**). The tables' internal structure resembles the structure of the **SRT** and the **PRT** in FBR (see [29]), except that the last hop information is replaced by source hop information.

Let us use the example in Fig. 4.1.1 to illustrate these tables. In this example, a publisher, pub-c_1, issues an advertisement a_1 at Broker A and three subscribers, sub-c_1, sub-c_2, and sub-c_3, each, issue a subscription at brokers B, E, and G, respectively. Assume that all subscriptions match the advertisement. The advertisement a_1 is stored in the **L-SRT** of Broker A and in the **R-SRT** of all other brokers. The subscriptions s_1, s_2, and s_3 are stored in the **R-PRT** of Broker A, and in the **L-**

PRT of brokers *B*, *E* and *G*, respectively. In **L-SRT** and **L-PRT**, the `sourceID` indicates from which client the advertisement or subscription originated, whereas in **R-SRT** and **R-PRT**, the `sourceID` indicates from which broker the advertisement or subscription originated.



L-SRT		
broker	adv	sourceID
A	a1	pub-c1

R-SRT		
broker	adv	sourceID
B-J	a1	A

L-PRT		
broker	sub	sourceID
B	s1	sub-c1
E	s2	sub-c2
G	s3	sub-c3

R-PRT		
broker	sub	sourceID
A	s1	B
	s2	E
	s3	G

SPRT of B		
broker	dest broker	next broker
B	A	A
	...	...
	G	F
	H	F
	J	F

TRT of all				
	A	B	C	...
A	0	1	∞	
B	1	0	1	
C	∞	1	0	
...				

Figure 4.1.1: Event routing in D-DBR

Message Processing: In D-DBR, advertisements, subscriptions and events are delivered as messages. In each message, besides the `sourceID` field, the message header contains another field named `destIDList`, indicating to which clients and brokers the message needs to be delivered to. Alg. 1 specifies how the matching engine processes different messages.

The input to Alg. 1 is a message from the *match* message queue and the resulting output messages are added to the *output* queue, as shown in Fig. 4.0.1. Messages from local clients and other brokers are processed differently. Lines 1∼14 process messages from local clients as follows: (1) An advertisement is first inserted into the **L-SRT** and then attached to the *output* queue with the IDs of all other brokers as its destinations. (2) A subscription is first inserted into the **L-PRT** and then matched against advertisements in the **R-SRT** to obtain the IDs of interested brokers as its destinations. (3) An event is matched against subscriptions in both the **L-PRT** and the **R-PRT** to obtain the IDs of locally interested clients and remotely interested brokers as its destinations. In this way, advertisements can be broadcast to all other brokers, subscriptions can be delivered to all the brokers whose advertisements match the subscriptions, and events can be delivered to both locally interested clients and remotely interested brokers.

Algorithm 1 Message Processing in Matching Engine

```
Input: message m from queue match
Output: messages inserted into queue output
 1: if m is from a local client then
 2:     if m is an advertisement then
 3:         L-SRT.insert(m)
 4:         m.destIDList←all other brokers' IDs
 5:     else if m is a subscription then
 6:         L-PRT.insert(m)
 7:         advs←Match(m,R-SRT)
 8:         m.destIDList←advs.sourceID
 9:     else if m is an event then
10:         subs←Match(m,L-PRT) + Match(m,R-PRT)
11:         m.destIDList←subs.sourceID
12:     end if
13:     m.sourceID←localBrokerID
14:     output.enqueue(m)
15: else
16:     if m is an advertisement then
17:         R-SRT.insert(m)
18:         subs←Match(m,L-PRT)
19:         subs.destIDList←m.sourceID
20:         output.enqueue(subs)
21:     else if m is a subscription then
22:         R-PRT.insert(m)
23:     else if m is an event then
24:         subs←Match(m,L-PRT)
25:         m.destIDList←subs.sourceID
26:         output.enqueue(m)
27:     end if
28:     Forward(m)                                  ▷ for MERC algorithm
29: end if
```

Lines 16∼27 process messages from other brokers as follows: (1) An advertisement is first inserted into the **R-SRT** and then matched against subscriptions in the **L-PRT**. The matched subscriptions are inserted into the *output* queue with that advertisement's *sourceID* as their destination. (2) A subscription is simply inserted into the **R-PRT**. (3) An event is matched against subscriptions in the **L-PRT** only to obtain the IDs of locally interested clients as its destinations. In this way, when an advertisement is delivered to a broker, the broker's locally matched subscriptions are transferred to the source broker of that advertisement. When an event is delivered to a broker, that event is directly transferred to the broker's locally interested clients.

Separating the routing information of local clients from that of other brokers reduces the message matching cost for the following three reasons: (1) Subscriptions from local clients only need to be matched against advertisements from other brokers, (2) advertisements from other brokers only need to be matched against subscriptions from local clients, and (3) events from other brokers only need to be matched against subscriptions from local clients.

4.2 Destination-based Multicasting Layer

Routing Tables: To route a message to its destinations, the multicasting layer maintains two routing tables: The *Topology Routing Table* (**TRT**) and the *Shortest Path Routing Table* (**SPRT**). **TRT** is a matrix, where $\text{TRT}(i, j)$ represents the communication cost between brokers i and j. If there is a direct connection between brokers i and j, $\text{TRT}(i, j) = 1$, otherwise, ∞. **SPRT** includes a list of records (`destBrokerID, nextBrokerID`). Each record indicates the next hop on the path to a specific destination. All brokers share the same **TRT**, but each broker has its own **SPRT**. The **SPRT** is computed from the **TRT** by using the Dijkstra algorithm with cost $O(n^2)$, where n is the total number of brokers. Let us revisit the example in Fig. 4.1.1, where the **TRT** of all brokers as well as the **SPRT** of Broker B are presented. In this example, Broker B is connected to brokers H through F and G, and the distance-wise optimal next hop from Broker B to Broker H is Broker F.

Message Processing: The multicasting layer provides a simple and efficient destination-based one-to-many message delivery service. At this layer, advertisements, subscriptions, and events are routed in the same way, which simplifies the `pub/sub` system's design and implementation. Moreover, each message is delivered to its destinations along the shortest paths. And, once a message is received by a destination broker, the message's `destIDList` is reduced. This guarantees that each message is delivered to its destinations once and only once. As a result, general overlay topologies can be supported without redundant messages resulting from the broadcasting of advertisements.

Algorithm 2 Message Processing in Multicasting Engine

```
Input: messages in queue input and output
Output: messages inserted into queue match and messages sent to remote brokers
    and local clients
 1: for m←input.dequeue() do
 2:     if m is from a local client then
 3:         match.enqueue(m)
 4:     else
 5:         if localBrokerID∈m.destIDList then
 6:             match.enqueue(m)
 7:             m.destIDList.remove(localBrokerID)
 8:         end if
 9:         Multicast(m)
10:     end if
11: end for
12: for m←output.dequeue() do
13:     for destID∈m.destIDList do
14:         if destID is a client's ID then
15:             SendTo(m, destID)
16:             m.destIDList.remove(localBrokerID)
17:         end if
18:     end for
19:     Multicast(m)
20: end for
```

Alg. 2 processes messages from the network (stored in the *input* message queue) and messages generated by the matching engine (stored in the *output* message queue). In this algorithm, a message from a local client is simply inserted into the *match* queue for later processing by the matching engine. For a message from other brokers, if the broker is not one of its destinations, that message is simply forwarded. Otherwise, one copy of that message is inserted into the *match* queue. A message generated by the matching engine is first delivered to its destination clients and then forwarded to its destination brokers.

`SendTo` is a service provided by the underlying communication interface to deliver a message to a local client or a neighboring broker. `Multicast` is a function provided by the multicasting engine to forward a message to other brokers. Upon receiving a message with the destination list `destIDList`, the message is duplicated into several messages, and each one is attached with a new destination list $\texttt{destIDList}_i$:

$$< nextID_i, destIDList_i > \leftarrow ShortestPath(destIDList)$$

where $nextID_i$ is the distance-wise optimal next hop from the local broker to all destination brokers in $\texttt{destIDList}_i$. To implement the function `ShortestPath`, every destination in `destIDList` is used to retrieve the next hop from the **SPRT**. Then, destinations with the same next hop are merged into a single destination list. Finally, each duplicated message is sent to its distance-wise optimal next hop using the `SendTo` service.

4.3 Dynamic Overlay Reconfiguration

Since in our design, content-based matching and destination-based multicasting are decoupled, changes to the overlay do not impact the routing tables at the matching layer in the D-DBR algorithm. For example, in Fig. 4.1.1, if the connection between brokers B and F is lost, only the brokers' **TRT**s and **SPRT**s at the multicasting layer need to be updated. As a result, D-DBR can easily support dynamic overlay reconfiguration, enabling fault-tolerance and further performance optimizations. Below, we exemplify these two benefits to demonstrate the flexibility of our D-DBR algorithm.

Fault Tolerance: In D-DBR, brokers use a heart-beat mechanism to detect the status of their neighbors. A broker multicasts an overlay update message to other brokers when a neighbor's status changes. The brokers receiving the message update their **TRT**s and **SPRT**s accordingly. In this way, D-DBR can automatically recover from broker failures and disconnections. In Fig. 4.1.1, when the connection between brokers B and F is lost, the **TRT** and **SPRT** of Broker B would be updated. Then, events from Broker A to G would be transferred through Broker E. In networks with cycles, unless there are no connecting routing paths between the source and the destinations, messages can always be successfully delivered using another paths.

Performance Optimization: In `pub/sub`, the publication and subscription workload may vary over time. Capabilities to dynamically adapt the overlay to enable performance optimizations are important. Here, we sketch a simple distributed topology self-organizing algorithm for D-DBR. The basic idea is to set up connections between brokers which incur heavy communication load. In our algorithm, each broker periodically undergoes the following three steps:

(1) **Metric collection** – During a performance optimization cycle T[1], each broker collects its link-wise communication rates. The communication rate between brokers B_i and B_j, $rate_T(B_i, B_j)$, is defined as the number of messages transmitted from B_i to B_j or vice versa divided by T. In D-DBR, since messages are routed based on their destinations, the communication rate between any two brokers can be easily obtained.

(2) **Evaluation** – Based on the communication rates, broker B_i computes the *gain* of a candidate link to a non-neighbor broker B_r and the *loss* of an existing link to a neighbor broker B_n. Gain and loss are defined as follows:

$$gain(i, r) = rate_T(B_i, B_r) * (dist(B_i, B_r) - 1)$$

$$loss(i, n) = rate_T(B_i, B_n) * (dist'(B_i, B_n) - 1)$$

In these formula, $dist(B_i, B_r)$ represents the current distance between brokers B_i and B_r in terms of the number of hops on the distance-wise optimal path between B_i and B_r, whereas $dist'(B_i, B_n)$ represents the distance between B_i and B_n after the connection between them is lost.

Candidate links to non-neighboring brokers are ranked based on their *gains*, and existing links are ranked based on their *losses*. Based on the rank and a configuration

[1]T can be configured based on application requirements.

parameter *MaxDegree*[2], each broker communicates with its neighbors to decide which new links can be added and which old ones should be discarded: First, a broker, say B_i, computes its minimal loss $MinLoss(i)$ to free up a quota for a new link. $MinLoss(i)$ equals zero if its current link number is less than *MaxDegree*, otherwise, it equals the minimal *loss* of its existing links. Second, B_i selects the candidate link $l(i, j)$ with the largest *gain*. Third, if $gain(i, j) > MinLoss(i)$, then B_i communicates with B_j to obtain its current $MinLoss(j)$. Finally, if $gain(i, j) > MinLoss(i) + MinLoss(j)$, then the link $l(i, j)$ is added and other related links are discarded.

(3) **Execution** – The involved brokers update their connections, and a topology update message is issued by Broker B_i to notify all other brokers to update their **TRT**s and **SPRT**s accordingly.

In the algorithm, once an event is successfully transmitted to an intermediate broker, it becomes that broker's duty to deliver the message. So, no events will be lost even when topology reconfiguration happens.

4.4 Algorithm Analysis

Subscription Duplication: D-DBR does not redundantly duplicate subscriptions across all brokers on the routing paths. Instead, a subscription is only stored at its local broker and the brokers with matching advertisements. For example, in Fig. 4.1.1, the subscription s_3 is stored at brokers G and A, only, not at the intermediate brokers, such as B and F.

Matching Overhead: The matching overhead in D-DBR is reduced for three main reasons: First, the number of stored subscriptions is smaller and the matching cost at each source broker is thus smaller. Second, an event is not matched at any intermediate broker. Third, an event only needs to be matched against the local subscriptions at each destination broker. Since event matching is the most expensive operation in `pub/sub`, reducing matching cost significantly alleviates the overhead of

[2]MaxDegree indicates the total number of links each broker can maintain.

brokers and improves the pub/sub system's performance.

Flexibility and Robustness: D-DBR efficiently supports a general overlay with cycles. In addition, since content-based matching and destination-based multicasting are fully decoupled, D-DBR achieves good flexibility and robustness. It has good fault tolerance capabilities and supports dynamic overlay self-reconfiguration, as we exemplified, above.

Routing Accuracy: Since messages are routed on the shortest paths, D-DBR has a routing accuracy as good as that of the FBR algorithm. In some scenarios, where the load between brokers is not balanced, D-DBR's flexibility makes it possible to automatically reconfigure the overlay. As a result, in these scenarios, the lengths between the source and destination brokers can become shorter, and D-DBR can achieve better routing accuracy.

Destination List Overhead: An important factor that may limit the applicability of D-DBR is the destination list overhead. If the average destination list is long, more bandwidth is used. However, since a message's destination list can be reduced at every routing step by a factor of the fan-out of brokers in the message's delivery tree, the destination list overhead is in fact small. We demonstrate this point through real-world experiments and detailed simulations in Section **??**. On the other hand, events in content-based pub/sub systems often carry rich content. For example, in RSS applications, most feeds/events range from 1KB to 10KB, with a median of 5.8 KB [42]. In these scenarios, the destination list overhead consumes only a small fraction of the bandwidth.

Topology Maintenance Overhead: In D-DBR, a broker needs to know all other brokers in the overlay instead of only its own neighbor brokers. This introduces additional overhead for topology maintenance, which may become expensive when the overlay scales out. However, in small-sized networks, the overhead remains acceptable: heart-beat messages are only exchanged between neighboring brokers, a topology update message only needs to be delivered to a limited number of brokers, and the size of **TRT** and **SPRT** remains small. For example, in an overlay with 100 brokers, the **TRT** and **SPRT** of each broker only occupies about 11 kB of memory.

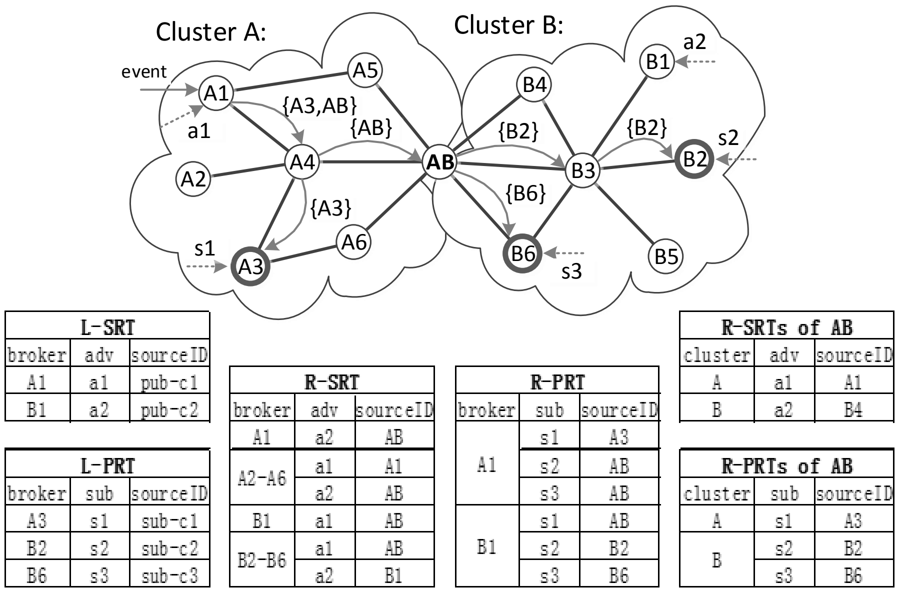
Chapter 5

MERC Design

For improved scalability in large-scale broker overlays, we propose another routing scheme called MERC—Match at Edge and Route intra–Cluster. MERC combines destination-based and content-based routing hierarchically. It has the advantages of D-DBR, i.e., low subscription duplication and low matching cost. It also overcomes the scalability limitation of D-DBR: In MERC, each broker needs to know a limited number of brokers, only, and the destination list is limited to brokers in the local cluster. Therefore, the topology maintenance overhead and the destination list overhead are both reduced.

In MERC, the broker overlay is divided into interconnected clusters of brokers. Some brokers, called *edge brokers*, are located at the edge of clusters and belong to more than one cluster, whereas the other brokers, called *internal brokers*, belong to only one cluster. Each broker only knows the addresses of brokers in clusters it belongs to. Content-based and destination-based mechanisms are adopted for inter- and intra-cluster event routing, respectively.

In MERC, when an event is issued, it is first matched against subscriptions at the local broker to identify interested brokers in the local cluster. Then, the event is delivered to these brokers along the distance-wise optimal paths, according to D-DBR. Once an event is received by an edge broker that also belongs to another cluster, the event is matched against subscriptions from that cluster at the edge broker to identify interested brokers in that cluster. It is then delivered to these brokers from the edge broker, again, according to D-DBR. This process is repeated until the event is delivered to all interested brokers in all clusters.



L-SRT		
broker	adv	sourceID
A1	a1	pub-c1
B1	a2	pub-c2

L-PRT		
broker	sub	sourceID
A3	s1	sub-c1
B2	s2	sub-c2
B6	s3	sub-c3

R-SRT		
broker	adv	sourceID
A1	a2	AB
A2-A6	a1	A1
	a2	AB
B1	a1	AB
B2-B6	a1	AB
	a2	B1

R-PRT		
broker	sub	sourceID
A1	s1	A3
	s2	AB
	s3	AB
B1	s1	AB
	s2	B2
	s3	B6

R-SRTs of AB		
cluster	adv	sourceID
A	a1	A1
B	a2	B4

R-PRTs of AB		
cluster	sub	sourceID
A	s1	A3
B	s2	B2
	s3	B6

Figure 5.0.1: Event routing in MERC

Fig. 5.0.1 shows an example of event routing in MERC. In this example, the broker overlay is divided into two clusters: Cluster A and Cluster B. Both are connected by an Edge Broker AB. There are two advertisements and three subscriptions: a_1, a_2, s_1, s_2, and s_3. In the example, we assume each advertisement matches all subscriptions. Now, when an event is issued at Broker A_1, it is first delivered to the interested brokers A_3 and AB in Cluster A. At the Edge Broker AB, it is matched against subscriptions from Cluster B and further transferred to the interested brokers B_2 and B_6 in Cluster B.

5.1 Routing Tables

In MERC, routing tables of internal brokers at both the matching layer and the multicasting layer are the same as those in D-DBR: Each broker maintains the same six routing tables: **L-SRT**, **R-SRT**, **L-PRT**, **R-PRT**, **TRT** and **SPRT**. However, the edge brokers have different routing tables. Besides an **L-SRT** and an **L-PRT**, an edge broker maintains a group of the other four routing tables for each cluster it belongs to. Note that in MERC, the `sourceID` of advertisements and subscriptions transferred by an edge broker from one cluster to another cluster is replaced by that edge broker's ID.

For example, Fig. 5.0.1 shows that the Edge Broker *AB* maintains two **R-SRT**s and two **R-PRT**s, each of which is identified by the cluster's name. Also, the `sourceID` of advertisements and subscriptions that are transferred by Broker *AB* from one cluster to another cluster is replaced by Broker *AB*'s ID.

5.2 Message Processing

In MERC, message processing for internal brokers is the same as that in D-DBR. For edge brokers, they override the function `Forward` in Alg. 1[1] to process messages at the matching layer with the new implementation shown in Alg. 3.

Algorithm 3 Implementation of function $Forward(m)$

```
1: if localBroker is an edge broker then
2:     m.sourceID←localBrokerID
3:     if m is an advertisement then
4:         m.destIDList←IDs of all brokers in other clusters
5:     else if m is a subscription then
6:         advs←Match(m,R-SRTs of other clusters)
7:         m.destIDList←advs.sourceID
8:     else if m is an event then
9:         subs←Match(m,R-SRTs of other clusters)
10:        m.destIDList←subs.sourceID
11:    end if
12:    output.enqueue(m)
13: end if
```

[1]Function Forward does nothing in Alg. 1.

Whenever a message from a specific cluster is delivered to an edge broker, the `sourceID` of that message is first replaced by that edge broker's ID. Then, that message is processed based on its type: An advertisement is forwarded to all brokers in other clusters, a subscription is forwarded to brokers in other clusters with matching advertisements, and an event is forwarded to brokers in other clusters with matching subscriptions.

For edge brokers, message processing at the multicasting layer is similar to that in D-DBR, except that several groups of routing tables (**TRT** and **SPRT**) may be used to route a message to its destinations.

5.3 Algorithm Analysis

Scalability: Compared with D-DBR, MERC achieves better scalability: First, a broker only needs to know brokers in the cluster or clusters it belongs to. Changes in one cluster do not affect brokers in other clusters. Topology maintenance overhead for each broker remains small, even when the overlay's size increases. Second, a message is not annotated with addresses of brokers in other clusters. The size of each message's destination list is thus limited by the number of brokers in the local cluster.

Performance: In MERC, subscriptions are stored at the local broker, remote interested brokers, and certain edge brokers. Events are matched at these brokers, only. Compared with FBR, MERC achieves higher performance, since each event does not have to be matched at all intermediate brokers. Compared with D-DBR, MERC's performance is lower, because MERC requires additional matching overhead at edge brokers. From a performance point of view, D-DBR fares best among these three algorithms. However, since destination list and topology maintenance overhead limit its scalability, D-DBR is not suitable for large-scale topologies (i.e., approximately 100 and more brokers). Therefore, it is better to opt for D-DBR and MERC in small-sized and large-sized overlays, respectively.

Other Considerations: The Internet resambles a collection of interconnected routing domains [16], which are groups of nodes under common administration sharing routing information. MERC follows this design: A single cluster represents an administrative domain and multiple clusters are connected in a hierarchical manner. In MERC, if an edge broker acts as a transit node, and an internal broker acts as a stub node, the `pub/sub` system follows the transit-stub model of the Internet [67]. So an appealing characteristic of MERC is that it provides a good starting point to construct large-scale `pub/sub` systems that mimic the structure of the Internet.

Chapter 6

PS-Tree Organization

`PS-Tree` is a novel tree index for subscriptions. The idea behind `PS-Tree` is to divide the set of values represented by all predicates into disjoint subsets, here referred to as *predicate spaces*. In other words, a predicate space is a value range in the predicate's dimension (i.e., value domain). Then, we construct a many-to-many relationship between predicate spaces and subscriptions. Thus, the problem of matching an attribute-value pair against a set of subscriptions is transformed into the problem of locating the predicate space to which an attribute-value pair belongs, which can be efficiently supported by `PS-Tree`.

6.1 PS-Tree Structure

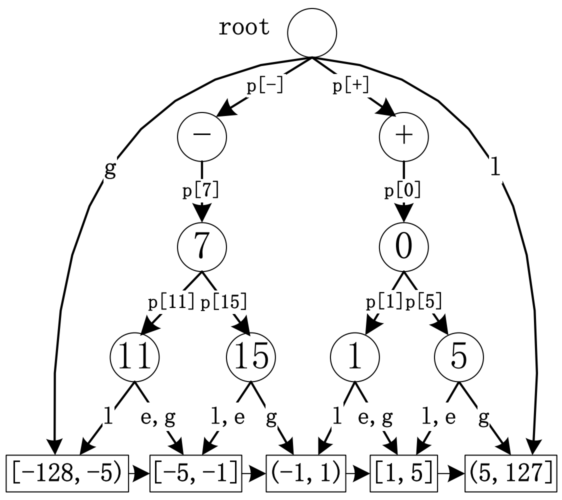
`PS-Tree` contains two types of nodes: leaf nodes and inner nodes. A leaf node represents a predicate space, while an inner node represents an "element" of the represented attribute values. Elements are specified differently for different value types. For example, we specify elements as digits for the integer type. The root node of `PS-Tree` is a special inner node that represents the starting element of the represented attribute values. The inner nodes on the path from the root node to a leaf node construct the represented attribute value and act as the boundary between two adjacent predicate spaces.

As shown in Fig. 6.1(a), an inner node contains three leaf node links (l, e, g) and an array of links to child inner nodes. The *length* of the inner node link array is set as the number of different elements. If an inner node corresponds to an attribute value, say, v, l of this inner node links to the leaf node whose predicate space is less than v, e links to the leaf node whose predicate space is equal to v, and g links to the leaf node whose predicate space is greater than v. As an optimization, e can link to the same leaf node as l and g. For the root node, a special consideration is that g links to the first leaf node, and l links to the last leaf node.

```
InnerNode {
   LeafNodeLink l;
   LeafNodeLink e;
   LeafNodeLink g;
   InnerNodeLink p[LENGTH];
}

LeafNode {
   LeafNodeLink next;
   Integer counter;
   list<Sub> subLinkedList;
   BloomFilter signature;
   Integer spaceId;
}
```

(a) Node design



(b) PS-Tree index example

Figure 6.1.1: PS-Tree index structure

In `PS-Tree`, a leaf node contains a *next link*, a *predicate counter*, an *event counter*, a *subscription linked list* and a *signature*. The next link points to another leaf node whose predicate space is adjacent and greater than that leaf node's predicate space. The predicate counter equals the number of predicates covering[1] the current leaf node's predicate space. The event counter equals the number of historical events that match with the current leaf node's predicate space. The subscription linked list contains links to subscriptions. The signature is the Bloom filter created from the

[1]In this thesis, we use the term *cover* to indicate that an interval contains another interval or a point.

subscription IDs. When a predicate of a subscription covers the predicate space of a leaf node, the ID of that subscription is inserted into the signature of that leaf node. The sub linked list and the signature are only required by `PSTBloom`. For `PSTHash`, a leaf node maintains a unique predicate space ID, which is not required by `PSTBloom`. As a result, we have two versions of `PS-Tree`, `PS-Tree`$_B$ and `PS-Tree`$_H$, which have different space complexities.

The elements are specified differently for different value types. In this thesis, we use the 8-bit byte type to illustrate the design of `PS-Tree`. Fig. 6.1(b) shows a `PS-Tree` instance with two subscriptions, $S_1\{attr, in, [-5, -1]\}$ and S_2 $\{attr, in, [1, 5]\}$, indexed. Each of these subscriptions contains one predicate in the dimension *attr*. In this example, the value type of this dimension is an 8-bit byte. For a byte, which is stored as a binary complement, we specify the most significant bit, the next 3 bits, and the last 4 bits as an element. Take the attribute value -5 as an example; its elements are "$-$", "7" and "11". In this `PS-Tree` instance, there are five leaf nodes. Correspondingly, the whole value domain $[-128, 127]$ of the dimension *attr* is divided into five predicate spaces: $[-128, -5)$, $[-5, -1]$, $(-1, 1)$, $[1, 5]$, and $(5, 127]$. These predicate spaces are mapped to the following five subscription sets, respectively: $\varnothing$, $\{S_1\}$, $\varnothing$, $\{S_2\}$, and $\varnothing$. Given an attribute value pair $\langle attr, 2\rangle$, the predicate space $[1, 5]$ to which it belongs can be quickly located through `PS-Tree`. Subsequently, the matching subscription set $\{S_2\}$ can be directly obtained.

6.2 Index Construction

Alg. 4 processes the inserted predicates and constructs the `PS-Tree` index. *InsertPredicate* takes two parameters, *pred* and *pstree*, as input. *pred* is the predicate to be inserted. The output is a list of leaf nodes whose predicate spaces are covered by the inserted predicate. *InsertPredicate* addresses predicates differently for different operators. Alg. 4 shows how the operator "$\geq$" is handled. Other operators are handled in a similar manner.

Algorithm 4 InsertPredicate($pred, pstree$)

```
1: if $pred.op$ $is \geq$ then
2:     startNode←Partition($pred.vals[0], pred.op, pstree$)
3:     endNode←pstree.root.l
4: end if
5: while $startNode \neq endNode.next$ do
6:     startNode.predCounter++
7:     leafNodes.add($startNode$)
8:     startNode←startNode.next
9: end while
10: return leafNodes
```

Algorithm 5 Partition($val, op, pstree$)

```
1: currNode←pstree.root
2: for $each\ elem \in val$ do
3:     path.push($currNode, elem$)
4:     if $currNode.p[elem] = null$ then
5:         currNode.p[elem]←CreateInnerNode()
6:     end if
7:     currNode←currNode.p[elem]
8: end for
9: if $currNode.e = null$ then
10:     iRNode←GetRNode($path$)
11:     iLNode←GetLNode($iRNode, pstree.root$)
12:     PartitionLeafNode($currNode, iLNode, op$)
13: else
14:     PartitionLeafNode($currNode, op$)
15: end if
16: return currNode.e
```

Partition is a function invoked by *InsertPredicate* to partition a predicate space in a `PS-Tree`. The input parameters include a value, an operator, and a `PS-Tree`. The output is a leaf node whose predicate space covers the input value. As shown in Alg. 5, if not all inner nodes corresponding to that value exist, new inner nodes are created. The function *GetRNode* is used to locate the inner node adjacent to and to the right of the current inner node. *GetLNode* is used to locate the minimal left inner node. *PartitionLeafNode* is used to partition the predicate space of a leaf node and create new leaf nodes. Except for the space ID and the next link, a newly created leaf node copies other content from the leaf node to be partitioned.

Algorithm 6 GetRNode(*path*)

```
1: while ⟨node, elem⟩←path.pop() do
2:     for pos←elm+1; pos ≤ LENGTH; pos++ do
3:         if node.p[pos] ≠ null then
4:             return node.p[pos]
5:         end if
6:     end for
7: end while
8: return node
```

Algorithm 7 GetLNode(*iRNode*, *root*)

```
1: if iRNode = root then
2:     return root
3: end if
4: iLNode = iRNode;
5: while iLNode.l = null do
6:     for pos←1; pos ≤ LENGTH; pos++ do
7:         if iLNode.p[pos] ≠ null then
8:             iLNode = iLNode.p[pos]
9:             break
10:        end if
11:    end for
12: end while
13: return iLNode
```

The function *GetRNode* is used to locate the inner node adjacent to and to the right of the current inner node. The input parameter is the path from the root node to the current node. As can be seen in Alg. 6, *GetRNode* reversely traverses the nodes on the path. The right side not *null* node pointer is returned. The time complexity of *GetRNode* is $O(LENGTH * N_e) = O(N_e)$, where N_e is the number of elements in an attribute value.

GetLNode is used to locate an inner node's minimal left inner node. The input parameters include an inner node link and the root node link. As can be seen in

Alg. 7, *GetLNode* contains a *while*-loop. The loop does not stop before locating an inner node whose l link is not null. The time complexity of *GetLNode* is also $O(N_e)$.

Alg. 8 and Alg. 9 both show how a leaf node is partitioned. In Alg. 8, the e link of $currNode$ points to the leaf node to be partitioned, while in Alg. 9, the l link of $iLNode$ points to the leaf node to be partitioned. How a leaf node is partitioned depends on the status of the `PS-Tree` and the operator. Alg. 8 and Alg. 9 show the detailed steps to partition a leaf node when the operator is $\geq$ and $=$.

In Alg. 8 and Alg. 9, the function *CopyLeafNode* creates a new leaf node by copying all the information from an existing leaf node. Similarly, the function *CreateInnerNode* creates a new inner node by setting every member variable to *null*. The time complexities of these two functions are $O(1)$.

Algorithm 8 PartitionLeafNode($currNode, op$)

```
 1: if $op$ $is \geq$ then
 2:     if $currNode.l = currNode.e$ then
 3:         leafNode = CopyLeafNode ($currNode.l$)
 4:         currNode.l.next = leafNode
 5:         currNode.e = leafNode
 6:     end if
 7: end if
 8: if $op$ $is =$ then
 9:     if $currNode.e = currNode.g$ then
10:         leafNode = CopyLeafNode ($currNode.e$)
11:         currNode.e.next = leafNode
12:         currNode.g = leafNode
13:     else if currNode.l = currNode.e then
14:         leafNode = CopyLeafNode ($currNode.l$)
15:         currNode.l.next = leafNode
16:         currNode.e = leafNode
17:     end if
18: end if
```

Algorithm 9 PartitionLeafNode($currNode, iLNode, op$)

```
1: if op is ≥ then
2:     leafNode = CopyLeafNode (iLNode.l)
3:     iLNode.l.next = leafNode
4:     currNode.l = iLNode.l
5:     currNode.e = leafNode
6:     currNode.g = leafNode
7:     iLNode.l = leafNode
8: end if
9: if op is = then
10:     leafNodeOne = CopyLeafNode (iLNode.l)
11:     leafNodeTwo = CopyLeafNode (iLNode.l)
12:     iLNode.l.next = leafNodeOne
13:     leafNodeOne.next = leafNodeTwo
14:     currNode.l = iLNode.l
15:     currNode.e = leafNodeOne
16:     currNode.g = leafNodeTwo
17:     iLNode.l = leafNodeTwo
18: end if
```

Algorithm 10 CreateInnerNode()

```
1: innerNode.l = null
2: innerNode.e = null
3: innerNode.g = null
4: for i = 1 to LENGTH do
5:     innerNode[i] = null
6: end for
7: return innerNode
```

6.3 Predicate Matching

Alg. 11 matches an attribute-value pair against a `PS-Tree` to locate the predicate space to which it belongs. Two situations can occur: (1) all the inner nodes corresponding to the value in that attribute-value pair exist, in which case the last inner node's link e is returned, or (2) not all corresponding inner nodes exist, in which case *GetRNode* and *GetLNode* are invoked to locate the inner node whose l links to the leaf node with the predicate space covering the value.

Algorithm 11 MatchPair($pair, pstree$)

```
1: currNode←pstree.root
2: for each elem ∈ pair.val do
3:     path.push($currNode, elem$)
4:     if $currNode.p[elem] \neq null$ then
5:         currNode←currNode.p[elem]
6:     else
7:         iRNode←GetRNode($path$)
8:         iLNode←GetLNode($currNode, pstree.root$)
9:         return iLNode.l
10:    end if
11: end for
12: return currNode.e
```

6.4 Dynamic Index Adjustment

PS-Tree supports dynamic index adjustment by providing the function *DeletePredicate*, which is used to delete the outdated predicates and nodes from a PS-Tree. As mentioned in Sec. 6.1, every leaf node contains a *predicate counter*, which is equal to the number of predicates covering the current leaf node's predicate space. When the predicate counter is zero, the corresponding leaf node is deleted to save memory.

In Alg. 12, *DeletePredicate* deletes a predicate from a PS-Tree instance. The main function of *DeletePredicate* is to determine all the leaf nodes whose predicate spaces are covered by that predicate. The predicate counters of these leaf nodes decrease by one. Similar to $InsertPredicate$, *DeletePredicate* executes differently for different operators. Lines 12∼20 exemplify the processing of "*in*" and "∈". As with $InsertPredicate$ of PS-Tree, the time complexity of $InsertPredicate$ is $O(N_e + N_p)$, where N_p represents the number of predicates indexed by current PS-Tree instance.

Algorithm 12 DeletePredicate($pred, pstree$)

```
1: if pred.operator is in then
2:     pairOne←⟨pred.attr, pred.vals[0]⟩
3:     pairTwo←⟨pred.attr, pred.vals[1]⟩
4:     startNode←MatchPair(pairOne, pstree)
5:     endNode←MatchPair(pairTwo, pstree)
6:     while $startNode \neq endNode.next$ do
7:         startNode.predCounter -= 1
8:         leafNodes.add(startNode)
9:         startNode←startNode.next
10:    end while
11: end if
12: if pred.operator is $\in$ then
13:    for $i = 1$ to pred.valueNum do
14:        pair←⟨pred.attr, pred.vals[i]⟩
15:        lNode←MatchPair(pair, pstree)
16:        lNode.predCounter -= 1
17:        leafNodes.add(lNode)
18:    end for
19: end if
20: return leafNodes
```

Here, we give an additional example that focuses on predicate space partitioning and dynamic `PS-Tree` index adjustment. Suppose, initially, that there are two subscriptions: $S1\{age, in, [20, 60]\}$ and $S2\{age, in, [30, 80]\}$. The value domain of the "age" dimension is $[1, 100]$. Therefore, at the beginning, the value domain is divided into 5 predicate spaces, $[1, 20), [20, 30), [30, 60]$, $(60, 80]$ and $(80, 100]$, with the counters 0, 1, 2, 1 and 0, respectively. When $S2$ is removed, the counters become 0, 1, 1, 0, 0, respectively. Then, the predicate space $[20, 30)$ is merged with $[30, 60]$, and $(60, 80]$ is merged with $(80, 100]$. As a result, there are three predicate spaces remaining: $[1, 20)$, $[20, 60]$, and $(60, 100]$.

6.5 Expressiveness

`PS-Tree` offers high expressiveness by supporting different value types and operators. To support a value type (integer, float, string, etc.), the only requirement is to specify the elements of values of that type. As shown in Fig. 6.1(b), we divide a byte type value into three elements. For other integer types, the elements are similarly specified as digits. For example, a 32-bit integer type value is divided into nine elements. We use characters as elements for the string type. For the float type, the most significant bit, the exponent bits and the mantissa bits are specified as elements.

`PS-Tree` supports an expressive set of operators; for numbers (e.g., integer, float and double), enumerations, and strings, the supported operations include relational operators ($<, \leq, =, \neq, >, \geq$), set operators ($\in, \notin$), and the SQL operator (in). The only requirement to support a specific operator in `PS-Tree` is that the predicate containing that operator is able to be divided into disjoint predicate spaces.

A further advantage of `PS-Tree` is that it isolates the specific value types and operators from the upper matching layer of `PSTBloom` and `PSTHash`. Thus, the upper layer works in the same way for different value types and operators.

6.6 Time and Space Analysis

Matching Time Complexity: In `PS-Tree`, matching an attribute-value pair against a set of subscriptions is achieved by locating the predicate space to which the attribute-value pair belongs. As shown in Alg. 11, the number of operations is linear in the number of elements of the represented attribute value. Thus, the matching complexity is $O(N_e)$, where N_e is the number of elements in an attribute value. For the integer type, N_e equals nine; therefore, the matching time complexity is only $O(1)$.

Predicate Insertion Time Complexity: In Alg. 4, two steps are needed to insert a predicate into a `PS-Tree`: (1) insert predicate values into the `PS-Tree` and (2) determine all leaf nodes covered by the predicate. The time complexity of the first step is also $O(N_e)$. For the second step, the number of operations needed equals the number of predicate spaces covered by the predicate. In the worst case, the number of predicate spaces in a `PS-Tree` is $2 * N_p + 1$, where N_p represents the number of predicates that have been inserted. Therefore, the predicate insertion time complexity is $O(N_e + N_p)$.

Space Complexity: `PS-Tree` requires space for inner nodes and leaf nodes. The number of leaf nodes is equal to the number of predicate spaces. In the worst case, the number of inner nodes is N_e times the number of leaf nodes. As analyzed above, the number of leaf nodes is $O(N_p)$. Therefore, the space complexity of $\texttt{PS-Tree}_H$, which is used by the `PSTHash` algorithm, is $O(N_e * N_p)$. For $\texttt{PS-Tree}_B$, which is used by the `PSTBloom` algorithm, additional memory is needed to store the subscription list. The space complexity of $\texttt{PS-Tree}_B$ is $O(N_p * (N_e + N_p))$. Note that this is the worst-case space complexity.

Chapter 7

PSTBloom Organization

Based on `PS-Tree`, we first design the `PSTBloom` algorithm. The idea behind `PSTBloom` is to select a predicate with high selectivity as the *access predicate* for each subscription. Then, the subscription is attached to those leaf nodes corresponding to its access predicate. The ID of the subscription is inserted into the Bloom filter signature of the leaf nodes corresponding to its other predicates. When an event is received, it is matched against a number of `PS-Trees` to locate its associated leaf nodes. Then, the set of subscriptions whose access predicates match with that event is directly located. Next, we further filter out unmatching subscriptions through the signatures of those leaf nodes. The correctness of `PSTBloom` is based on Lemma 1.

Lemma 1 Given an event E, the matching subscriptions for E are contained in the candidate subscription set $\{S(\langle attr_i, val_i\rangle) \mid \langle attr_i, val_i\rangle \in E\}$, where $S(\langle attr_i, val_i\rangle)$ represents the subscriptions whose access predicates match with the *i-th* attribute-value pair $\langle attr_i, val_i\rangle$ of E.

We sketch the proof for Lemma 1 as follows. If a subscription S matches with an event E, based on the matching semantics in Sec 3.2, we know that there exists a matching attribute-value pair in E for every predicate of S. Therefore, the access predicate of S has a matching attribute-value pair $\langle attr_i, val_i\rangle$ in E. Thus, S is contained in $\{S(\langle attr_i, val_i\rangle)\}$.

7.1 PSTBloom Structure

In PSTBloom, a PS-Tree is constructed for each dimension. As shown in Fig. 6.1(a), each leaf node of a PS-Tree contains a *subscription linked list* and a *signature*. Each link in the subscription linked list points to a subscription whose access predicate covers the current leaf node's predicate space. The signature is the Bloom filter over a set of subscription IDs. Each such subscription has one predicate (except the access predicate) covering the current leaf node's predicate space.

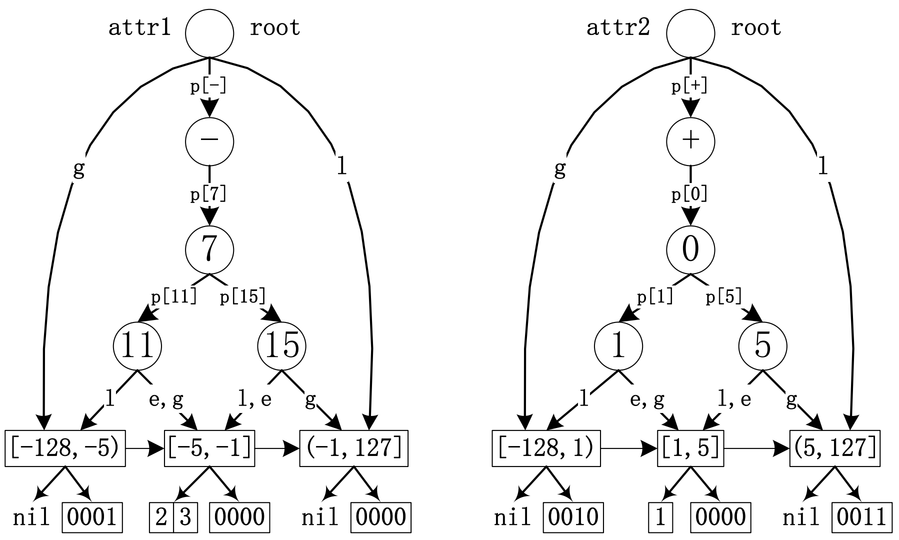


Figure 7.1.1: PSTBloom index structure

The example in Fig. 7.1.1 illustrates the PSTBloom index structure. In this example, the value type is byte. There are three subscriptions:

$$S_1 : \{attr_1, <, -5\}, \{attr_2, in, [1, 5]\}$$

$$S_2 : \{attr_1, in, [-5, -1]\}, \{attr_2, <, 1\}$$

$$S_3 : \{attr_1, in, [-5, -1]\}, \{attr_2, >, 5\}$$

PSTBloom constructs two PS-Trees, denoted as $attr_1$ and $attr_2$. For S_1, $\{attr_2, in, [1, 5]\}$ is selected as the access predicate, while for S_2 and S_3, $\{attr_1, in, [-5, -1]\}$ is selected as the access predicate. As shown in Fig. 7.1.1, a link to S_1 is inserted into the linked list associated with the leaf node with the predicate space $[1, 5]$ in the PS-Tree for $attr_2$. Links to S_2 and S_3 are inserted into the linked list associated with the leaf node with the predicate space $[-5, -1]$ of the PS-Tree for $attr_1$. The ID of S_1 is inserted into the signature of the leaf node with the predicate space $[-128, -5)$ in the PS-Tree for $attr_1$. The ID of S_2 and S_3 is inserted into the signature of the leaf nodes with the predicate space $[-128, 1)$ and $(5, 127]$ of the PS-Tree for $attr_2$, respectively.

7.2 Index Construction

Alg. 13 processes the inserted subscriptions. When a subscription is received, the predicate with the highest selectivity is selected as the access predicate. Then, every predicate is inserted into its corresponding PS-Tree. For each predicate, a set of leaf nodes is returned after invoking the operation *InsertPredicate* presented in Alg. 4. Based on whether that predicate is an access predicate, different operations are executed. If the predicate is an access predicate, a link to the subscription is inserted into the subscription linked list of each leaf node; otherwise, the subscription ID is inserted into the signature of each leaf node.

Algorithm 13 InsertSubscription($sub, pstb$)

```
1: accPred←**SelectAccPred**($sub, pstree$)
2: **for** *each pred* $\in$ *sub* **do**
3:     pstree←pstb.pstrees[pred.attr]
4:     leafNodes←**InsertPredicate**($pred, pstree$)
5:     **for** *each node* $\in$ $leafNodes$ **do**
6:         **if** $pred = accPred$ **then**
7:             node.subLinkedList.add($sub$)
8:         **else**
9:             node.signature.add($sub.id$)
10:        **end if**
11:    **end for**
12: **end for**
```

7.3 Event Matching

As shown in Alg. 14, `PSTBloom` takes three steps to match an event against subscriptions: (1) Match each attribute-value pair in the event against the `PS-Trees` to locate a set of leaf nodes. The subscriptions in the subscription linked lists attached to these leaf nodes are candidate subscriptions. (2) Prune a subscription if its ID is not contained in the signature of the related leaf nodes. (3) Match the event against the remaining subscriptions to further filter out false positives. In this step, the access predicate no longer needs to be checked.

Algorithm 14 MatchEvent($event, pstb$)

```
1: for each pair ∈ event do
2:     pstree←pstb.pstrees[pair.attr]
3:     leafNode←MatchPair(pair, pstree)
4:     leafNodes[pair.attr]←leafNode
5: end for
6: for each node ∈ leafNodes do
7:     for each sub ∈ node.subLinkedList do
8:         isCandidate←True
9:         for each pred ∈ sub do
10:            if pred = sub.accPred then
11:                Continue
12:            end if
13:            nodeSign←leafNodes[pred.attr].signature
14:            if ¬nodeSign.contain(sub.id) then
15:                isCandidate←False; break
16:            end if
17:        end for
18:        if isCandidate = True then
19:            if Match(event, sub) = True then
20:                matchingSubs.add(sub.id)
21:            end if
22:        end if
23:    end for
24: end for
25: return matchingSubs
```

7.4 Subscription Deletion

Alg. 15 shows how a subscription is deleted in `PSTBloom`: First, *DeletePredicate* is invoked for every predicate. Then, a set of leaf nodes is returned. For each leaf node, if its predicate counter is zero, that leaf node is removed from the `PS-Tree` it belongs to. Otherwise, based on whether the predicate is the access predicate, different steps are executed. If the predicate is the access predicate, the subscription is deleted from the subscription linked list; otherwise, the ID of the subscription is deleted from the signature of the leaf node. As with *InsertSubscription* of `PSTBloom`, the time complexity of this function is $O(|S| * (N_e + N_p))$.

Algorithm 15 DeleteSub($sub, pstb$)

```
1: for $each\ pred \in sub$ do
2:     pstree←pstb.pstrees[pred.atrr]
3:     leafNodes←DeletePredicate($pred, pstree$)
4:     for $each\ node \in leafNodes$ do
5:         if $node.predCounter = 0$ then
6:             RemoveNode($node, pstree$)
7:             Continue
8:         end if
9:         if $pred = sub.accPred$ then
10:            leafNode.subLinkedList.delete($sub$)
11:        else
12:            leafNode.signature.delete($sub.id$)
13:        end if
14:    end for
15:    MergePreds($leafNodes, pstree$)
16: end for
```

7.5 Time and Space Analysis

Matching Time Complexity: As analyzed in Sec. 6.6, `PS-Tree` needs $O(N_e)$ time to locate the predicate space to which an attribute-value pair belongs. Therefore, for `PSTBloom`, the time complexity to retrieve the candidate subscriptions is $O(|E| *$

N_e), where $|E|$ is the event size. The total time complexity for event matching is $O(|E| * N_e + N_c)$, where N_c is the number of candidate subscriptions.

Index Construction Time Complexity: To insert a subscription, each of its predicates is inserted into a corresponding `PS-Tree`. As analyzed in Sec. 6.6, the time complexity of this operation is $O(N_e + N_p)$. Thus, the index construction time complexity for `PSTBloom` is $O(N_s * |S| * (N_e + N_p))$, where N_s is the number of subscriptions and $|S|$ is the subscription size.

Space Complexity: `PSTBloom` maintains a `PS-Tree`, more specifically `PS-Tree`$_B$, for each dimension. As analyzed in Sec. 6.6, the space complexity of `PS-Tree`$_B$ is $O(N_p * (N_e + N_p)$. To store all `PS-Trees`, `PSTBloom` requires $O(N_d * N_p * (N_e + N_p))$ space, where N_d is the number of dimensions and N_p is the number of predicates in a dimension.

Chapter 8

PSTHash Organization

Although `PSTBloom` achieves good performance with respect to event matching, index construction, and memory use, it suffers from a limitation in addressing dense workloads because `PSTBloom` uses only one access predicate to select candidate subscriptions. Given dense workloads, the number of candidates could potentially be large.

To overcome this limitation, we design the `PSTHash` algorithm, which is also based on `PS-Tree`. The idea behind `PSTHash` is to select more than one, say, N, predicates with high selectivity as *access predicates* for each subscription. These access predicates are divided into a number of disjoint *N-dim* predicate spaces. Each *N-dim* predicate space contains N predicate spaces. For an event, a subscription is identified as a candidate only when all N access predicates are matched.

`PSTHash` differs from `PSTBloom` in its method of identifying candidate subscriptions. In `PSTHash`, a many-to-many hash table between *N-dim* predicate spaces and subscriptions is maintained. Given an event E, each attribute-value pair of E is matched against a corresponding `PS-Tree` to identify the predicate spaces to which the attribute-value pair belongs. In total, $|E|$ predicate spaces can be found. Then, $\binom{|E|}{N}$ *N-dim* predicate spaces are constructed. Through these *N-dim* predicate spaces, the candidate subscriptions can be directly retrieved by querying the hash table that relates the *N-dim* predicate spaces and the subscriptions. Given dense workloads, compared with `PSTBloom`, `PSTHash` identifies fewer candidate subscriptions and achieves better matching performance. Assuming that N is not greater than the size of all events and subscriptions[1], we formulate Lemma 2.

Lemma 2 Given an event E, the matching subscriptions for E are contained in the candidate subscription set $\{S(AV_{i_1}, ..., AV_{i_N}) \mid \{AV_{i_1}, ..., AV_{i_N}\} \in E, i_x \neq i_y, x \neq y\}$, where $S(AV_{i_1}, ..., AV_{i_N})$ represents the subscriptions whose access predicates match with the i_1*-th* to the i_N*-th* attribute-value pair of E.

We sketch a proof. If a subscription S matches with an event E, based on Sec. 3.2, there exists a matching attribute-value pair in E for every predicate of S. Therefore, any access predicate of S also has a matching attribute-value pair in E, which means that S is contained in a specific $S(AV_{i_1}, ..., AV_{i_N})$.

The number of access predicates N is an important configuration parameter in `PSTHash`. When N is large, the access predicates of a subscription are divided into more *N-dim* predicate spaces, `PSTHash` consumes more memory, and index construction becomes more expensive; however, the matching performance improves, thus producing a trade-off. The most suitable value of N is dependent on workload and beyond the scope of this thesis. For simplicity, in the following sections, we always set N equal to two.

8.1 PSTHash Structure

In addition to a set of `PS-Trees`, `PSTHash` utilizes two more data structures: a set of *2-dim* predicate space IDs and a hash table. A *2-dim* predicate space ID is constructed using 2 predicate space IDs. A predicate space ID is determined by a dimension ID and space ID pair. Thus, a *2-dim* predicate space ID is represented by 2 pairs of ⟨*dimension ID, space ID*⟩. The pairs are ordered in ascending order by their dimension ID values. The key of the hash table is a *2-dim* predicate space ID,

[1]We use the method of `PSTBloom` to address subscriptions and events whose size is less than N.

and the associated value is a subscription linked list.

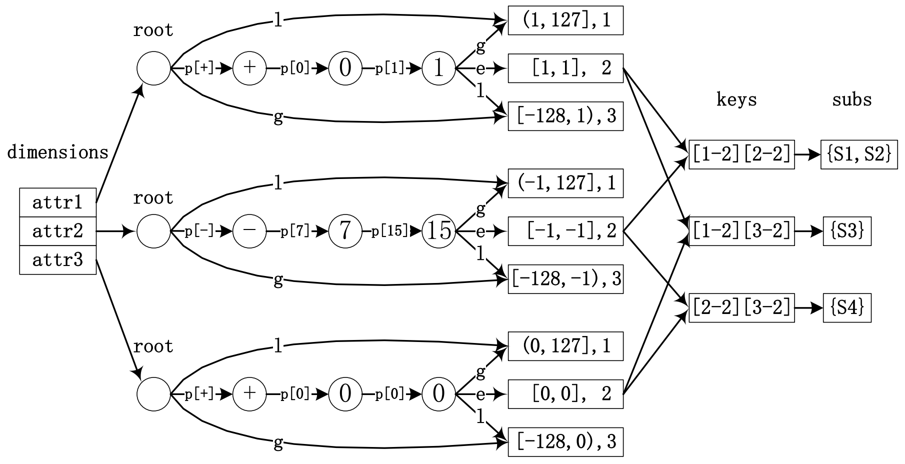


Figure 8.1.1: PSTHash index structure

Fig. 8.1.1 illustrates the index structure of `PSTHash`. Here, there are four subscriptions, all of which contain three predicates.

$$S_1 : \{attr_1, =, 1\}, \{attr_2, =, -1\}, \{attr_3, <, 0\}$$

$$S_2 : \{attr_1, =, 1\}, \{attr_2, =, -1\}, \{attr_3, >, 0\}$$

$$S_3 : \{attr_1, =, 1\}, \{attr_2, <, -1\}, \{attr_3, =, 0\}$$

$$S_4 : \{attr_1, <, 1\}, \{attr_2, =, -1\}, \{attr_3, =, 0\}$$

In this `PSTHash` index example, S_1 and S_2 are associated with the 2-dim predicate space whose ID is [1-2][2-2]. This ID indicates that the 2-dim predicate space is constructed using the second predicate space of $attr_1$ and the second predicate space of $attr_2$. Similarly, S_3 is associated with [1-2][3-2], and S_4 is associated with [2-2][3-2].

Assume that an event E $\{\langle attr_1, 1\rangle, \langle attr_2, -1\rangle, \langle attr_3, 2\rangle\}$ is received. The event is matched against these three `PS-Trees`. Three predicate spaces to which E belongs are identified: [1-2], [2-2] and [3-1]. These three predicate spaces are used to construct three 2-dim predicate spaces: [1-2][2-2], [1-2][3-1], and [2-2][3-1]. `PSTHash` uses these 2-dim predicate space IDs as keys to retrieve the candidate subscriptions S_1 and S_2.

These two candidate subscriptions are evaluated, and the matching subscription S_2 is found.

8.2 Index Construction

Alg. 16 processes subscriptions for insertion. When a subscription S is received, two predicates with high selectivity are selected as access predicates. Then, each access predicate is inserted into a corresponding `PS-Tree`. A set of leaf nodes is returned after invoking *InsertPredicate*. Using the dimension IDs and space IDs of those leaf nodes, a number of *2-dim* predicate space IDs are constructed. The subscription is inserted into the hash table with the *2-dim* predicate space ID as the key. When *InsertPredicate* is invoked, predicate spaces may need to be partitioned. In this situation, the records associated with the old space ID in the hash table are copied to new records using the new space ID as the key.

Algorithm 16 InsertSubscription($sub, psth$)

```
 1: accPreds←SelectAccPreds($sub, pstree, 2$)
 2: for each $pred \in accPreds$ do
 3:     pstree←psth.pstrees[pred.attr]
 4:     leafNodes←InsertPredicate($pred, pstree$)
 5:     for each $node \in leafNodes$ do
 6:         spaceIdSets[pred.attr].add($node.spaceId$)
 7:     end for
 8: end for
 9: for each $spaceId1 \in spaceIdSets[attr_1]$ do
10:     for each $spaceId2 \in spaceIdSets[attr_2]$ do
11:         key = [attr$_1$-spaceId1][attr$_2$-spaceId2]
12:         psth.hash[key].add($sub$)
13:     end for
14: end for
```

8.3 Event Matching

As shown in Alg. 17, `PSTHash` takes three steps to match an event against the subscriptions: (1) Match each attribute-value pair in the event against a corresponding `PS-Tree` to obtain $|E|$ predicate space IDs. (2) Construct $\binom{|E|}{2}$ *2-dim* predicate space IDs using those predicate space IDs and retrieve the candidate subscriptions. (3) Match the event against these subscriptions to find matched subscriptions. In this step, the two access predicates no longer need to be checked.

Algorithm 17 MatchEvent($event, psth$)

```
1: for each pair ∈ event do
2:     pstree←psth.pstrees[pair.attr]
3:     leafNode←MatchPair($pair, pstree$)
4:     predSpaces.add($pair.attr, leafNode.spaceId$)
5: end for
6: spaceIds←ConstructSpaces($predSpaces$)
7: for each spaceId ∈ spaceIds do
8:     subLinkedList←psth.hash[spaceId]
9:     for each sub ∈ subLinkedList do
10:        if Match($event, sub$) = $True$ then
11:            matchingSubs.add($sub.id$)
12:        end if
13:    end for
14: end for
15: return matchingSubs
```

8.4 Subscription Deletion

Alg. 18 shows how a subscription is deleted from `PSTHash`: *DeletePredicate* is invoked for every access predicate of the subscription to obtain a set of leaf nodes. Using the attribute names and space IDs of those leaf nodes, a number of 2-dimensional predicate space IDs are constructed. The subscription is deleted from the hash table with the 2-dimensional predicate space ID as the key. For each leaf node, if its predicate counter is zero, the leaf node is removed. As with *InsertSubscription* of `PSTHash`, the time complexity of this function is $O(N_e + (N_p)^2)$

Algorithm 18 DeleteSub($sub, psth$)

```
 1: for each $pred \in sub.accPreds$ do
 2:     pstree←psth.pstrees[pred.atrr]
 3:     leafNodes←DeletePredicate($pred, pstree$)
 4:     for each $node \in leafNodes$ do
 5:         spaceIdSets[pred.attr].add($node.spaceId$)
 6:         if $node.predCounter = 0$ then
 7:             RemoveNode($node, pstree$)
 8:         end if
 9:     end for
10:     MergePreds($leafNodes, pstree$)
11: end for
12: for each $spaceId1 \in spaceIdSets[attr_1]$ do
13:     for each $spaceId2 \in spaceIdSets[attr_2]$ do
14:         key = [attr$_1$-spaceId1][attr$_2$-spaceId2]
15:         psth.hash[key].delete($sub$)
16:     end for
17: end for
```

8.5 Time and Space Analysis

Matching Time Complexity: Similar to `PSTBloom`, the time complexity of locating the predicate spaces to which an event E belongs is $O(|E| * N_e)$. The number of *2-dim* predicate spaces is $\binom{|E|}{2}$. Therefore, the matching time complexity for `PSTHash` is $O(|E| * N_e + \binom{|E|}{2} + N_c)$, where N_c is the number of candidate subscriptions.

Index Construction Time Complexity: Given a subscription, `PSTHash` needs to insert its access predicates into `PS-Trees` and insert that subscription into a number of slots in the hash table. For these two operations, the time complexity is $O(N_e + N_p)$ and $O((N_p)^2)$, respectively. Therefore, the index construction complexity of `PSTHash` is $O(N_s * (N_e + (N_p)^2))$, where N_s is the number of subscriptions.

Space Complexity: `PSTHash` needs memory to store a `PS-Tree`, more specifically, `PS-Tree`$_H$, for each dimension. As analyzed in Sec. 6.6, the space complexity of `PS-Tree`$_H$ is $O(N_e + N_p)$. N_p is the number of access predicates in a dimension. `PSTHash` also needs memory for the hash table. In the worst case, the number of records in the hash table is $N_d * (N_p)^2$, where N_d is the number of dimensions. Thus, the space complexity of `PSTHash` is $O(N_d * N_e * N_p) + O(N_d * (N_p)^2) = O(N_d * N_p * (N_e + N_p))$.

Chapter 9

A-Tree Organization

In this section, we first discuss the `A-Tree` data structure. Then, we present how an `A-Tree` index is dynamically constructed with different optimization methods, such as expression reorganization and index self-adjustment. Finally, we analyze the space complexity and the index construction time complexity of `A-Tree`.

9.1 A-Tree Structure

In contrast to the binary decision diagram, `A-Tree` represents each arbitrary Boolean expression as an n-ary tree and aggregates a set of n-ary trees together. In `A-Tree`, we distinguish among three classes of nodes: a leaf node (*l-node*), which corresponds to a predicate; an inner node (*i-node*), which corresponds to a subexpression; and a root node (*r-node*), which corresponds to an arbitrary Boolean expression. `A-Tree` guarantees that the same predicate from different expressions corresponds to a unique *l-node* and the same subexpression from different expressions corresponds to a unique *i-node*. If two arbitrary Boolean expressions are the same, then they correspond to the same *r-node*. In Sec. 9.2.1, we will show how this property is efficiently guaranteed in the index construction process of `A-Tree`.

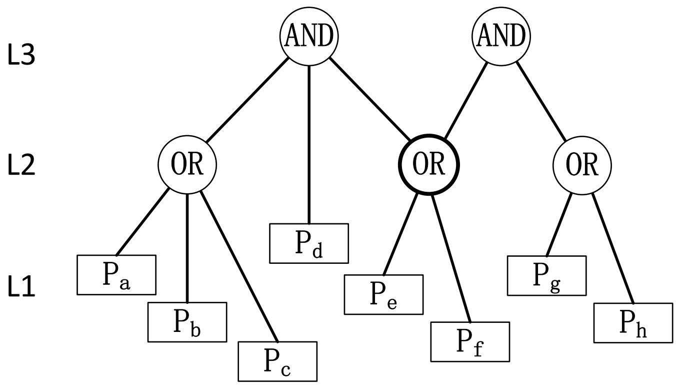


Figure 9.1.1: `A-Tree` Example

Suppose that we have two arbitrary Boolean expressions. The first one is $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$, and the second one is $(P_e \vee P_f) \wedge (P_g \vee P_h)$. Fig. 9.1.1 shows the `A-Tree` index constructed from these two expressions. As shown, the subexpression $(P_e \vee P_f)$ corresponds to an *i-node*, which is shared by the two n-ary trees of the two arbitrary Boolean expressions.

In `A-Tree`, each *l-node* and *i-node* can have any number of parent nodes, while each *i-node* and *r-node* can have any number of child nodes. An *i-node* and *r-node* store a logical operator, including *and*, *or*, *not*, *xor* and *xnor*. In the event matching process, each such node needs its child nodes' evaluation results to compute its own evaluation result based on the logical operator. The evaluation results flow upward in a bottom-up approach. To aid this matching process, each node of the `A-Tree` index keeps an integer called *level* to track that node's distance from the farthest *l-node* among all the nodes within its subtrees. The *level* of an *l-node* is set to be 1. An edge between two nodes increases this value by 1. For an *i-node* and *r-node* N, the *level* value is set according to the formula below:

$$level(N) = 1 + max\{level(C_i), \forall C_i | C_i \text{ is a child of } N\}$$

Before we discuss how the `A-Tree` index is dynamically constructed, we first define the *cost* of an `A-Tree` to construct a space-efficient and matching-efficient `A-Tree` structure. For each node, `A-Tree` consumes space to store that node. During event matching, it takes time to access that node. Thus, the cost of an `A-Tree` index has a positive correlation with the number of nodes. Meanwhile, when a node is matched, the parent nodes connected by its edges need to be evaluated. Therefore, in this thesis, we define the *cost* of an `A-Tree` index as the number of nodes plus the number of edges in the `A-Tree` index. Based on this definition, the *cost* of the `A-Tree` index shown in Fig. 9.1.1 is 25 since it contains 13 nodes and 12 edges. The goal of `A-Tree` index construction is to obtain an `A-Tree` index with the lowest possible cost given the incoming arbitrary Boolean expressions.

9.2 Index Construction

Index construction of `A-Tree` is important, which controls the degree of overlap achieved among differed expressions. A good index structure results in low memory consumption and high matching performance.

There are four challenges to overcome for building an `A-Tree` index with low cost. First, how to efficiently guarantee that every shared predicate, subexpression and expression are uniquely represented by a single node. Second, how to dynamically change the organization of an incoming arbitrary Boolean expression based on the current index structure to reuse a higher number of existing predicates and subexpressions. Third, how to dynamically adjust the existing index structure based on the newly incoming expression to ensure that the `A-Tree` index remains optimized regardless of the arrival order of the expressions. Fourth, how to efficiently remove expired expressions and dynamically adjust the index. In the following sections, we will discuss our solutions to these challenges one by one.

9.2.1 Node Uniqueness

To ensure that every shared predicate, subexpression and expression are uniquely presented by a single node in `A-Tree`, an *expression-to-node hash table* (H_{en}) is maintained. The key in this hash map is the id generated from the predicate, subexpression or expression; the value is the address of the node in the `A-Tree` index. For an incoming arbitrary Boolean expression, the first step is to identify the predicates and subexpressions. Take the arbitrary Boolean expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$ as an example; we identify the following six predicates and two subexpressions: P_a, P_b, P_c, P_d, P_e, P_f, $P_a \vee P_b \vee P_c$ and $P_e \vee P_f$. Note that the expressions $P_a \vee P_b$, $(P_a \vee P_b \vee P_c) \wedge P_d$, and $P_d \wedge (P_e \vee P_f)$ are not considered to be the subexpression of that arbitrary Boolean expression.

Algorithm 19 Insert($expr, H_{en}, atree$)

```
1: id ← generateID($expr$)
2: if $H_{en}[id] \neq null$ then
3:     return $H_{en}[id]$
4: else
5:     for $childExpr \in expr.childExprs$ do
6:         childNode ← Insert(childExpr, $H_{en}$, atree)
7:         childNodes.add(childNode)
8:     end for
9:     $H_{en}[id]$ ← createNewNode(expr, childNodes, atree)
10:    return $H_{en}[id]$
11: end if
```

Alg. 19 shows how the expression-to-node hash table, H_{en}, is used during the process of arbitrary Boolean expression insertion. If the expression already has a corresponding node in the `A-Tree` index, that node will be directly returned. Otherwise, nodes corresponding to its child expressions are first obtained, and then a new node is created in the `A-Tree` index. In Alg. 19, predicates, child expressions and the incoming arbitrary Boolean expression are all represented by the symbol *expr*.

9.2.2 Expression Reorganization

For the `A-Tree` index shown in Fig. 9.1.1, suppose that a new expression $P_a \vee P_b \vee P_c \vee P_d$ is received. Based on Alg. 19, a new node will be created with four child nodes corresponding to the predicates P_a, P_b, P_c and P_d, respectively. However, there is already an *inode* corresponding to $P_a \vee P_b \vee P_c$. If this node can be reused, then the cost of the new `A-Tree` index will be reduced.

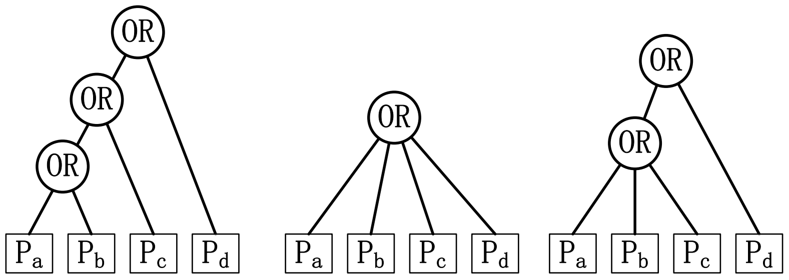


Figure 9.2.1: Different Presentations

Based on the associative law of the logical operator *or*, this incoming expression can be reorganized into different presentations, such as $((P_a \vee P_b) \vee P_c) \vee P_d$, $P_a \vee P_b \vee P_c \vee P_d$ and $(P_a \vee P_b \vee P_c) \vee P_d$. Fig. 9.2.1 shows the structures of these three presentations. The costs of these representations are 13, 9, and 11, respectively. Although the second presentation has the lowest cost, the third presentation is more suitable for the `A-Tree` index shown in Fig. 9.1.1 since the subexpression $P_a \vee P_b \vee P_c$ can be reused. Using the third presentation, the cost of the new `A-Tree` index only increases by 3, whereas using the second presentation, the cost of the new `A-Tree` index will increase by 5.

Given the current `A-Tree` index, by reorganizing an arbitrary Boolean expression, we can determine a representation that reuses more existing nodes in `A-Tree`. For different logical operators, different reorganization methods can be used. In this section, we mainly focus on the reorganization of the logical operators *and* and *or*. The reorganization of *not*, *xor* and *xnor* will be presented in Section **??**.

The reorganization of *and* and *or* is based on two properties. First, if several predicates or subexpressions are connected by the logical operator *and* or *or*, their orders do not change the evaluation result of the expression. For example, $P_a \vee P_b \vee P_c$ and $P_b \vee P_c \vee P_a$ are considered to be the same expression. Second, the same predicate or subexpression can be reused more than once. For example, if the expression $P_a \wedge P_b \wedge P_c \wedge P_d$ is inserted into an `A-Tree` index with two existing subexpressions $P_a \wedge P_b \wedge P_c$ and $P_c \wedge P_d$, then the inserted expression can be reorganized as $(P_a \wedge P_b \wedge P_c) \wedge (P_c \wedge P_d)$ such that those two existing subexpressions can be reused.

By considering an expression as the set of its child expressions, this expression reorganization problem can be translated into the set cover problem: given a set of child expressions (called the universe) and a collection S of m sets of expressions existing in `A-Tree`, we are attempting to identify the smallest subcollection of S whose union equals the universe. The set cover problem is *NP-hard*. We use the greedy algorithm shown in Alg. 20 to solve this problem as an approximate solution. The strategy is to choose the existing expression in `A-Tree` that contains the largest number of uncovered child expressions.

Algorithm 20 Reorganize($expr, atree$)

```
1: $U \leftarrow$ expr.childExprs
2: $C \leftarrow \varnothing$
3: while $U \neq \varnothing$ do
4:     select an $S \in$ atree that maximizes $S \cap U$
5:     $U \leftarrow U - S$
6:     $C \leftarrow C \cup \{S\}$
7: end while
8: return $C$
```

9.2.3 Index Self-adjustment

Unlike `Dewey ID` [30] and `Interval ID` [30], the `A-Tree` index is not built offline in advance, which means that the newly incoming arbitrary Boolean expressions are handled on the fly and the index structure is dynamically adapted.

In the above section, we proposed the method to dynamically change the organization of an incoming arbitrary Boolean expression based on the current `A-Tree` index structure. A different optimization direction is to dynamically adjust the `A-Tree` index structure based on the newly incoming arbitrary Boolean expression. For example, suppose that the current `A-Tree` index is as shown in Fig. 9.1.1. When a new expression $(P_a \vee P_b \vee P_c) \wedge P_d$ is received, the existing nodes corresponding to $P_a \vee P_b \vee P_c$ and P_d can be reused. Based on Alg. 19, a new node corresponding to the incoming expression $(P_a \vee P_b \vee P_c) \wedge P_d$ is created. In this situation, the new `A-Tree` index can be further optimized since the newly created node can be reused as the child of the existing node corresponding to the expression $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$. The motivation of `A-Tree` index self-adjustment is to ensure that the `A-Tree` index remains optimized regardless of the arrival order of the expressions.

Algorithm 21 SelfAdjust($newNode$)

```
1: for childNode ∈ newNode.childNodes do
2:     for parentNode ∈ childNode.parentNodes do
3:         if newNode.expr ⊂ parentNode.expr then
4:             update(childNode, parentNode, newNode)
5:         end if
6:     end for
7: end for
```

As shown in Alg. 21, the self-adjustment of `A-Tree` occurs after each new node is created. Through the new node's child nodes, the candidate nodes whose corresponding expression covers the new node's corresponding expression can be located. Then, the index is updated to reuse the new node as a child node of some existing nodes.

Algorithm 22 Insert($expr, H_{en}, atree$)

```
1: id ← generateID($expr$)
2: if $H_{en}[id] \neq null$ then
3:     $H_{en}[id]$.useCount += 1
4:     return $H_{en}[id]$
5: else
6:     for $childExpr \in expr.childExprs$ do
7:         childNode ← Insert(childExpr, $H_{en}$, atree)
8:         childNodes.add(childNode)
9:     end for
10:    Reorganize($expr, atree$)
11:    node ← createNewNode(expr, childNodes, atree)
12:    node.useCount = 1
13:    SelfAdjust($node$)
14:    $H_{en}[id]$ = node
15:    return node
16: end if
```

The complete arbitrary Boolean expression insertion process is shown in Alg. 22. In the node structure of `A-Tree`, an integer field called *useCount* is kept, which indicates the total number of predicates, subexpressions and expressions using this node. If an existing expression is inserted, then the *useCount* of the corresponding node is increased by 1. Otherwise, that expression is reorganized, a new node is created, and the new `A-Tree` index is self-adjusted. Before the expression is reorganized, its child expressions are first addressed in the same way to ensure that any child expression already has a corresponding node in the `A-Tree` index.

9.2.4 Expression Deletion

Deleting an expression is a straightforward and fast operation. When an expression is deleted, the *useCount* of its corresponding node N is decremented by 1. If the *useCount* becomes 0, the node N can be safely removed from the `A-Tree` index because it is already not required for any expression. However, it is still possible that N is used as the child node of another node P. In this case, the node P will be changed to consume the child nodes of N as its own child nodes. Then, the node N continues to be removed from the `A-Tree` index. Meanwhile, the corresponding record in the *expression-to-node* hash map is also removed. In this way, the `A-Tree` index achieves dynamic self-adjustment during expression deletion. As shown in Fig. 23, when an expression is processed, all of its child expressions are recursively processed in the same method.

Algorithm 23 Delete$(expr, H_{en}, atree)$

```
 1: id ← generateID($expr$)
 2: node ← $H_{en}[id]$
 3: node.useCount −= 1
 4: if node.useCount = 0 then
 5:     Remove($node, atree$)
 6:     $H_{en}[id]$ ← null
 7:     for $childExpr \in expr.childExprs$ do
 8:         Delete($childExpr, H_{en}, atree$)
 9:     end for
10: end if
```

9.3 Event Matching

An arbitrary Boolean expression is permitted to contain the following logical operators: *and*, *or*, *not*, *xor* and *xnor*. Compared to conjunctive Boolean expression matching, this flexibility increases the expressiveness for applications employing expression matching. However, it also makes arbitrary Boolean expression matching more complex compared to conjunctive expression matching. For example, for conjunctive Boolean expression matching, only the satisfied predicates need to be identified because only when all the predicates are satisfied is the conjunctive Boolean expression matching. However, for arbitrary Boolean expression matching, both the satisfied and unsatisfied predicates need to be identified since an arbitrary Boolean expression may be matching based on unsatisfied predicates. For example, the arbitrary Boolean expression $\neg(P_e \vee P_f)$ is matching when the predicates P_e and P_f are both unsatisfied.

`A-Tree`-based event matching proceeds in two phases: predicate matching and expression matching. Given an event, the predicate matching phase determines all the satisfied and unsatisfied predicates. As we discussed in Sec. **??**, an unsatisfied predicate means that the evaluation result of the predicate is *false*. Correspondingly, a predicate's evaluation result is *undefined* if the event does not contain the predicate's attribute. The expression matching phase locates all the satisfied arbitrary Boolean expressions. Several predicate matching algorithms have been proposed, such as `Segment-Tree` [23], `Interval-Tree` [18] and `Interval-Skip` [32]. `A-Tree` is compatible with all of these algorithms.

To run event matching, in addition to the `A-Tree` index, a set of queues are needed. In `A-Tree`, to verify whether an *i-node's* corresponding expression is satisfied, the evaluation results of its child nodes need to be known in advance. The evaluation sequence in `A-Tree` is from a low *level* node to a high *level* node. Thus, we keep a queue for each *level* to ensure that the nodes are verified in the correct order.

9.3.1 Matching Algorithm

As shown in Alg. 24, for each incoming event, after all the satisfied and unsatisfied predicates are identified, their corresponding *l-nodes* in `A-Tree` are located through the *expression-to-node hash table* (H_{en}). These *l-nodes* are queued into Q_1, which corresponds to the first level. After this initial step is finished, the matching algorithm first processes the lowest level unfinished queue. After finishing a queue Q_i, it starts to process the next higher level queue Q_{i+1} until the maximum queue Q_M is processed. For any queue Q_i, visiting a node N in Q_i involves the following two basic tasks:

Algorithm 24 Match($preds, H_{en}$)

```
1: for $pred \in preds$ do
2:     id ← generateID($pred$)
3:     l-node ← $H_{en}[id]$
4:     l-node.result ← pred.result
5:     $Q_1$.add(l-node)
6: end for
7: for $level = 1 \to M$ do
8:     while $Q_{level}$ is not empty do
9:         node ← $Q_{level}$.dequeue()
10:        result ← node.evaluate()
11:        node.clean()
12:        if $result = undefined$ then
13:            continue
14:        end if
15:        for all $parent \in node.parents$ do
16:            if $parent.operands.empty()$ then
17:                plevel ← parent.level
18:                $Q_{plevel}$.add(parent)
19:            end if
20:            parent.operands.add(result)
21:        end for
22:        if $result = true$ then
23:            matchingExprs.add($node.Exprs$)
24:        end if
25:    end while
26: end for
27: return matchingExprs
```

Evaluation of Result: Evaluation of a node N is performed according to the node type, operator and operands. If N is an *l-node*, the evaluation result of N is copied from the predicate matching phase. For an *i-node* and *r-node*, the evaluation result is determined by the operator and operands from its child nodes. When none of the operands is *undefined*, the result is evaluated normally. Otherwise, the result is determined by the following logics. (1) If an operand is *undefined* and the operator is *and*, then the evaluation result is determined by other operands. If any operand is *false*, then the evaluation result is *false*. Otherwise, the evaluation result is *undefined*. (2) If an operand is *undefined* and the operator is *or*, then the evaluation result is also determined by other operands. If any operand is *true*, then the evaluation result is *true*. Otherwise, the evaluation result is *undefined*. (3) If an operand is *undefined* and the operator is *not*, *xor* or *xnor*, then the evaluation result is *undefined*. The operands must be available immediately at the point of visiting N. This is ensured by the level-order bottom-up traversal of the `A-Tree`. The event matches with the arbitrary Boolean expression corresponding to N if the evaluation result of N is *true*. In this case, those arbitrary Boolean expressions are inserted into the matching expression list.

Propagation of Result: If the evaluation result of N is $undefined$, that result does not need to be propagated to the parent nodes of N since the default value of any operand is $undefined$. Otherwise, the result is propagated to its parent nodes. For a parent node P, we check whether its operands are empty. If yes, then P is inserted into the queue Q_i, where i is the level of P. The evaluation result of N is added into the operands of P.

9.3.2 Event Matching Example

In this section, we describe the execution steps of event matching using an example. In this example, we assume that there are six arbitrary Boolean expressions involving eight predicates P_a, P_b, P_c, P_d, P_e, P_f, P_g and P_h:

$$\begin{aligned}
S_1 &= (P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f) \\
S_2 &= (P_e \vee P_f) \wedge (P_g \vee P_h) \\
S_3 &= P_a \vee P_b \vee P_c \vee P_d
\end{aligned}$$

$$S_4 = (P_a \vee P_b \vee P_c) \wedge P_d$$
$$S_5 = (P_e \vee P_f) \wedge (P_g \vee P_h)$$
$$S_6 = \neg(P_g \vee P_h)$$

The `A-Tree` index built on these expressions is shown in Figure 9.3.1. In this `A-Tree` index, there are eight leaf nodes, four inner nodes and four root nodes. Each node is assigned an id. The leaf nodes 1, 2, 3, 4, 5, 6, 7, and 8 correspond to the predicates P_a, P_b, P_c, P_d, P_e, P_f, P_g and P_h, respectively. The inner nodes 9, 10, 11, and 13 correspond to the subexpressions $P_a \vee P_b \vee P_c$, $P_e \vee P_f$, $P_g \vee P_h$, and $(P_a \vee P_b \vee P_c) \wedge P_d$, respectively. The root nodes 12, 14, 15, and 16 correspond to the expressions $(P_a \vee P_b \vee P_c) \vee P_d$, $(P_e \vee P_f) \wedge (P_g \vee P_h)$, $\neg(P_g \vee P_h)$, and $(P_a \vee P_b \vee P_c) \wedge P_d \wedge (P_e \vee P_f)$, respectively. Among them, nodes 12, 13, 14, 15, and 16 have the linked list to the expression sets $\{S_3\}$, $\{S_4\}$, $\{S_2, S_5\}$, $\{S_6\}$, and $\{S_1\}$, respectively.

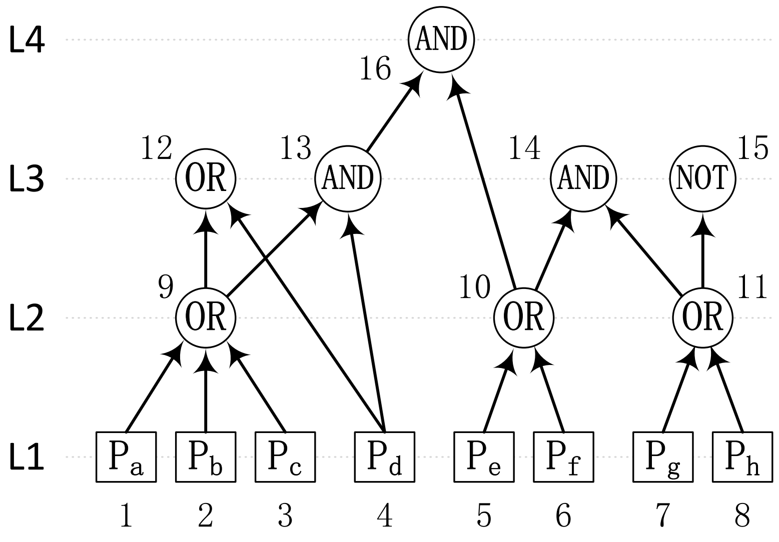


Figure 9.3.1: Matching Example

In this example, there are four queues: Q_1 to Q_4. Given an event E, before event matching starts, all these queues are empty. Assume that only the predicate P_a is satisfied, while the predicates P_e, P_g and P_h are unsatisfied. At the beginning, nodes 1, 5, 7, and 8 are queued into Q_1, with the evaluation results $true$, $false$, $false$ and $false$. Then, those queues are processed from level 1 to level 4.

Level 1: Nodes residing in Q_1 are processed one-by-one. First, node 1 is dequeued from Q_1. Since it only has one parent node 9, its result *true* is inserted into the operands of node 9. Node 9 is queued into Q_2. Second, node 5 is dequeued from Q_1. Its value *false* is inserted into the operands of node 10. Node 10 is also queued into the queue Q_2. Third, node 7 is dequeued from Q_1. Its value *false* is inserted into the operands of node 11, and node 11 is queued into Q_2. Fourth, node 8 is dequeued from Q_1. Its value *false* is also inserted into the operands of node 11.

Level 2: Q_1 is empty, and the nodes in Q_2 start to be processed. First, node 9 is dequeued from Q_2. Since its operator is *or* and one of its operands is *true*, its evaluation result is *true*. This evaluation result is inserted into the operands of nodes 12 and 13. Nodes 12 and 13 are queued into Q_3. Second, node 10 is dequeued from Q_2. Since its operator is *or*, one operand is *false*, and the other operand is *undefined*, its evaluation result is *undefined*. This evaluation result is not propagated. Third, node 11 is dequeued from Q_2. Since its operator is *or* and both operands are *false*, its evaluation result is *false*. This evaluation result is inserted into the operands of nodes 14 and 15. These two nodes are queued into Q_3.

Level 3: First, node 12 is dequeued from Q_3. Its operator is *or*, and one operand is *true*; thus, its evaluation result is *true*. The expression $S3$ associated with this node is identified as a matching expression. Second, node 13 is dequeued from Q_3. Since its operator is *and*, one operand is *true*, and the other operand is *undefined*, its evaluation result is *undefined*. Third, node 14 is dequeued from Q_3. Since its operator is *and*, one operand is *false*, and the other operand is *undefined*, its evaluation result is *false*. Fourth, node 15 is dequeued from Q_3. Since its operator is *not* and the only operand is *false*, its evaluation result is *true*. The expression $S6$ associated with this node is identified as a matching expression.

Level 4: Q_4 is empty. Consequently, there is no need to process any nodes. After the completion of the above steps, we have retrieved all the matching expressions S_3 and S_6.

9.3.3 Optimizations for Event Matching

The main cost of `A-Tree`-based event matching is the propagation of the matching result to a node's parent nodes and the later evaluation of the parent nodes. Two factors limit the event matching performance. First, when the evaluation result of a node is *false*, it still needs to be propagated to its parents. This process is expensive considering the potentially large number of unsatisfied predicates and subexpressions. Second, for a node N with the logical operator *and*, every child node's matching results are propagated to N. This is not necessary since if any child node is not satisfied, N will not be satisfied. In this section, we propose optimization solutions to overcome these two limitations.

Zero Suppression Filter: Because of the existence of the logical operators *not*, *xor* and *xnor* in arbitrary Boolean expressions, we need to distinguish between the *false* and *undefined* evaluation results. The *false* evaluation result needs to be propagated, which can be very expensive. To remove the cost to propagate *false* results, we propose the *zero suppression filter* optimization. Its basic idea is to remove the logical operators *not*, *xor* and *xnor* from an incoming arbitrary Boolean expression by applying the following laws:

$$\neg(E_1 \wedge E_2) = \neg E_1 \vee \neg E_2$$

$$\neg(E_1 \vee E_2) = \neg E_1 \wedge \neg E_2$$

$$E_1 \oplus E_2 = (E_1 \wedge \neg E_2) \vee (\neg E_1 \wedge E_2)$$

$$E_1 \otimes E_2 = (E_1 \wedge E_2) \vee (\neg E_1 \wedge \neg E_2)$$

In the negation removal procedure, by applying De Morgan's laws, all negations are pushed down level-by-level until directly before predicates. Then, negations are integrated into predicates. This procedure involves the inverse to a given relational operator: changing from greater than to less than, from equality to inequality, and so forth. Take the expression $\neg((age > 60) \vee (gender = female))$ as an example; it will be changed to $(age <= 60) \wedge (gender \neq female)$.

In this optimization, we remove the logical operators *not*, *xor* and *xnor* without increasing the number of expressions. When the evaluation result of any node is *false*, it is no longer propagated to any of its parents. Conversely, when computing the result of any node, if one of its operands is *undefined*, it is assumed to be *false*, and the result is computed accordingly. This optimization is promising because a *false* result is often obtained for a substantial number of leaf and inner nodes in the `A-Tree` while matching an event. Consequently, a large number of operations are saved, such as value update to parents, pushing nodes to queues, and so forth. Our experiments show that the event matching performance can be improved by up to 85% with this optimization.

Propagation On Demand: In `A-Tree`, for a node N with the logical operator *and*, only when all of its child nodes are satisfied is N satisfied. Based on this feature, we propose an optimization called *propagation on demand*. As the name suggests, we only propagate the matching result of $N's$ child nodes to N when it is needed. The basic idea of *propagation on demand* is as follows. If N is associated with the logical operator *and*, we randomly select one child node of N as the *access* child. In the `A-Tree` index, only that *access* child has a parent link to N. Meanwhile, N has links to its other child nodes. The concept of *access* child is similar to the concept of *access* predicate for conjunctive Boolean expression matching [27]. During the event matching process, only when the *access* child of N is satisfied is the matching result propagated to N and is N evaluated. During the evaluation process of N, we propagate the matching results of $N's$ other child nodes to N. However, if the *access* child of N is unsatisfied, no matching results will be propagated to N, and N will not be evaluated at all.

To support this optimization, we need to store the matching results of $N's$ child nodes because the matching results may be needed for the evaluation of N. Then, the question is when to clean up the stored matching results. Without cleaning, the matching of the next event will be affected. To solve this problem, we associate every matching result of a node to the signature of the current event. If the event signature of a matching result is not the same as the current processing event, then the matching result is considered to be *undefined*.

By this optimization, we can filter out a large number of unnecessary propagations: only when the *access* child is satisfied are the matching results propagated. This optimization is very effective for workloads with many *and* expressions. For example, on some conjunctive Boolean expression workloads, our experiments show this optimization improves the matching performance by up to 92%.

9.3.4 Optimized Event Matching Example

Here, we describe the execution steps of the optimized event matching using the same example as presented in Sec. 9.3.2. Since the *not* operator is removed for the *zero suppression filter* optimization, there are two more leaf nodes in the `A-Tree` index, as shown in Fig. 9.3.2. These two leaf nodes correspond to the predicates P_i and P_j. P_i is equal to $\neg P_g$, and P_j is equal to $\neg P_h$. Another difference in the `A-Tree` index is that there are not only *child-to-parent* links but also some *parent-to-child* links.

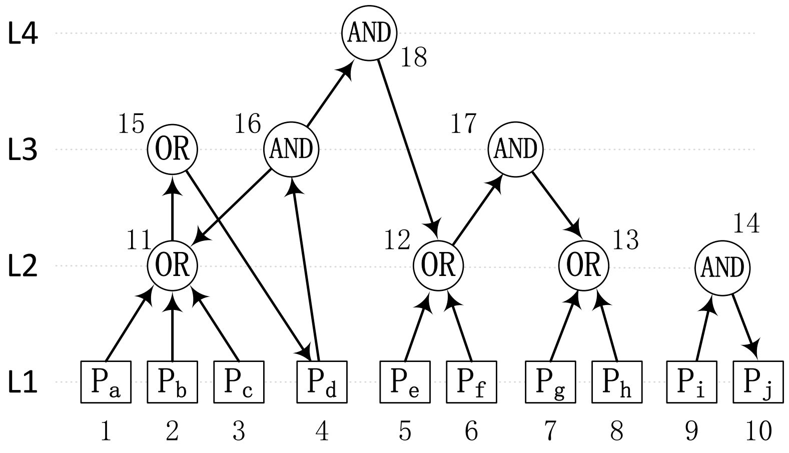


Figure 9.3.2: Optimized Matching Example

In this example, for the same incoming event E, only the satisfied predicates P_a, P_i and P_j need to be identified in the predicate matching phase. Thus, only nodes 1, 9, and 10 are queued into Q_1. Then, those queues are processed from level 1 to level 4.

Level 1: Nodes residing in Q_1 are processed one-by-one. First, node 1 is dequeued from Q_1, and its matching result *true* is only propagated to node 11. Node 11 is queued into Q_2. Second, node 9 is dequeued from Q_1, and its result is propagated to node 14. Node 14 is also queued into Q_2. Finally, node 10 is dequeued from Q_1, and its matching result *true* is stored in the node with the current event's signature.

Level 2: Q_1 is empty, and the nodes in Q_2 start to be processed. First, node 11 is dequeued from Q_2. Since its operator is *or* and one of its operands is *true*, its evaluation result is *true*. This evaluation result is only propagated to node 15, and node 15 is queued into Q_3. Second, node 14 is dequeued from Q_2. Since its operator is *and*, the stored matching result at node 10 is propagated on demand to node 14. Node 14 is identified as a satisfied node. Thus, the associated expression S_6 is identified as a matching expression.

Level 3: First, node 15 is dequeued from Q_3. Its operator is *or*, and one operand is *true*; thus, its evaluation result is *true*. The expression $S3$ associated with this node is identified as a matching expression. Q_4 is empty. Thus, there is no need to process to the next level. After completing the above steps, we have retrieved all the matching expressions S_3 and S_6.

9.4 Time and Space Analysis

Space Complexity: `A-Tree` needs spaces to store its nodes and links between child and parent nodes. In an `A-Tree` index, the number of leaf nodes is equal to the number of unique predicates, the number of inner nodes is equal to the number of unique subexpressions and expressions, and the number of links is equal to the number of *child-parent* relationships. For a single arbitrary Boolean expression with N_p predicates, the number of subexpressions and expressions is $O(N_p)$, and the number of *child-parent* relationships is also $O(N_p)$. Thus, the space needed for that single arbitrary Boolean expression is $O(N_p)$. For an `A-Tree` index constructed from N_{exp} arbitrary Boolean expressions, in the worst case, no predicates, subexpressions, and expressions are shared. In this situation, the number of nodes and *child-parent* links are all $O(N_{exp} * N_p)$. Thus, the space complexity of `A-Tree` is $O(N_{exp} * N_p)$. Note that many predicates and subexpressions are generally shared and the `A-Tree` index has a small memory footprint.

Index Construction Time Complexity: As shown in Alg. 22, if an incoming expression does not exist in the `A-Tree` index, then the cost of the insertion contains three parts: reorganization cost, new node creation cost and index self-adjustment cost. The time complexities of these three operations are $O(N_p^2)$, $O(1)$ and $O(N_p)$, respectively. N_p represents the number of predicates in the expression. Thus, the total `A-Tree` index construction time is $O(N_{exp} * N_p^2)$, where N_{exp} is the number of expressions.

Chapter 10

Experiments

10.1 Publish/Subscribe Routing

This section evaluates our D-DBR and MERC algorithms using experiments run on a computing facility and experiments based on simulations. We use the FBR algorithm[1] as a baseline. In our real-world experiments, we mainly evaluate system throughput, event delivery latency, overhead as measured due to subscription duplication, and brokers' CPU utilization. Through detailed simulations, we evaluate the destination list overhead, system robustness, routing accuracy, and network topology maintenance overhead.

10.1.1 Experiments on Computing Facility

We implemented the D-DBR and MERC algorithms in PADRES [40], a representative, open-source, content-based `pub/sub` system based on the FBR algorithm. The SciNet computing facility [49] was used as testbed, in which each node has 8 cores, a 2.66 GHz CPU, and 8 GB of memory. In our experiments, each broker is deployed on

[1]The original FBR algorithm was designed for acyclic overlay networks only [45]. For general overlay networks, we employ the extension to FBR proposed in [38].

a given node. The TCP/IP communication links between brokers represent the topology of the pub/sub system we deploy.

Since experimental results can be largely affected by the adopted overlay topology and workload, we use different topologies and workloads to study the performance of the algorithms. In our experiments, we consider two representative topologies, i.e., an acyclic linear topology and more general topologies generated by the Georgia Tech GT-ITM network topology generator [67].

One of the main challenges in evaluating a `pub/sub` system is the lack of a real-world application data trace. Previous work showed that in many `pub/sub` applications, subscriptions follow the Zipf or uniform distributions [42, 54, 35]. We experimented with both distributions. Event workloads draw from real-world stock quote trace datasets from Yahoo! Finance and subscriptions express interests in this stock data. The number of subscriptions ranges from tens to thousands.

Acyclic Linear Topology: To evaluate metrics like event delivery latency and to study the basic behaviour of D-DBR and MERC, we first experiment by using an acyclic linear topology, which is simple and intuitive and it allows us to control the number of intermediate brokers through which each event travels. In the experiments, varying numbers of brokers are interconnected in a straight line. One publisher and 100 subscribers are connected to the broker at the head and the tail of the line, respectively. For MERC, the linear topology is divided into two clusters with the same number of brokers: The first half of the brokers are located in the first cluster and the second half of the brokers are located in the second cluster. The middle broker serves as the edge broker for both clusters.

We measure the average event delivery latency and the brokers' CPU utilization with varying number of brokers and subscriptions. Event delivery latency is obtained by measuring the interval between the time when an event is issued by its publisher and the time when the event is received by a subscriber. To make sure the latency is accurately measured, in all experiments, publishers and subscribers are located on the same machine.

Fig. 10.1.1 to Fig. 10.1.3 show the results of three groups of experiments, in each of

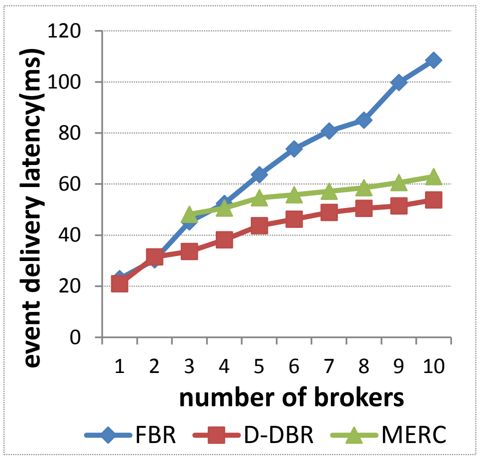


Figure 10.1.1: 100 subscriptions

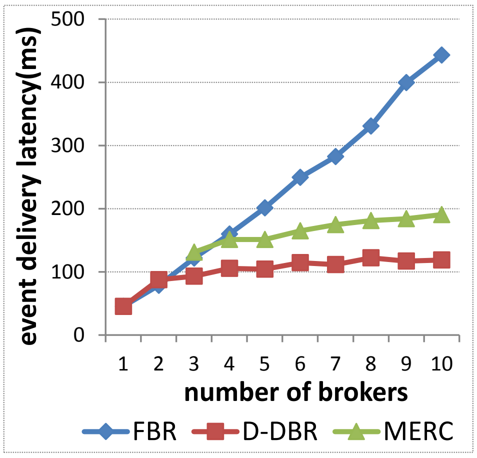


Figure 10.1.2: 800 subscriptions

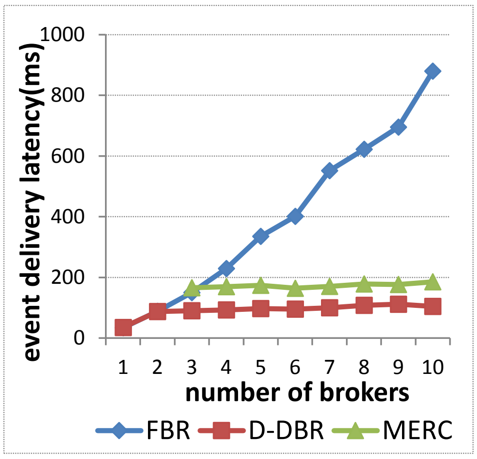


Figure 10.1.3: 2000 subscriptions

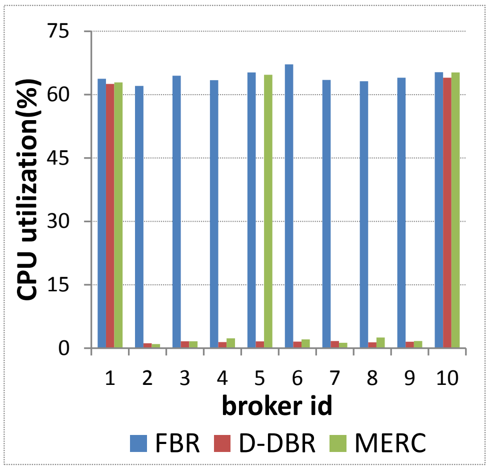


Figure 10.1.4: CPU utilization

which, the number of brokers increases from 1 to 10. We select 100, 800, and 2,000 subscriptions to represent the scenarios with a small, middle and large number of subscriptions, respectively. The event publishing rate is set to 3,000 messages/minute.

Fig. 10.1.1 shows that given 100 subscriptions, event delivery latencies of FBR, D-DBR and MERC all increase as the number of brokers increases. But the latencies of D-DBR and MERC are smaller than that of FBR. When there are 800 subscriptions, as shown in Fig. 10.1.2, the event delivery latency of FBR increases rapidly with increasing number of brokers, whereas the latencies of D-DBR and MERC only increase slightly. When the number of subscriptions increases to 2,000, the advantages of D-DBR and MERC are more significant. On the other hand, we can observe that the event delivery latency of MERC is roughly 1.5 times that of D-DBR. In these experiments, each event is matched three times in MERC and matched twice in D-DBR, which indicates the number of matches is an important factor contributing to the event delivery latency.

Fig. 10.1.4 presents ten brokers' CPU utilizations when there are 2,000 subscriptions in the system. For FBR, every broker's CPU utilization is about 63%; for D-DBR, the first and the last broker's CPU utilization is greater than 60%, and the remaining brokers' CPU utilizations are only about 2%; for MERC, only the first, the last and the middle broker's CPU utilization is greater than 60%. These experimental results show that, in an acyclic linear topology, D-DBR and MERC achieve better performance than FBR, especially, when there are a large number of subscriptions.

General Topology: To investigate the performance of our algorithms in general topologies, we did two groups of experiments. The first one is executed on a general overlay with 20 brokers, in which we mainly study the algorithms' performance when the subscriptions follow different distributions. The second one is executed on a general overlay with 100 brokers. In this group of experiments, we mainly evaluate the algorithms' maximum throughput and the corresponding event delivery latency, which are the two most important metrics for a `pub/sub` system. Some other metrics such as the destination list overhead, subscription duplication, and brokers' CPU and RAM utilization are also investigated.

In the overlay with 20 brokers, each broker has an average node degree equal to 3, while 20 publishers and 30 subscribers are randomly allocated to brokers in the system at the beginning of the experiment. Every subscriber issues a fixed number of subscriptions, ranging from 100 to 800. The event publishing rate of each publisher is also set to a fixed value. Subscriptions are synthesized according to the Zipf or uniform distributions.

Fig. 10.1.5 shows the event delivery latency of FBR and D-DBR when subscriptions follow the Zipf distribution (parameter is 0.5). In this experiment, the workload is the same for FBR and D-DBR. It can be seen that the event delivery latency of D-DBR is smaller than that of FBR. Moreover, D-DBR's event delivery latency is

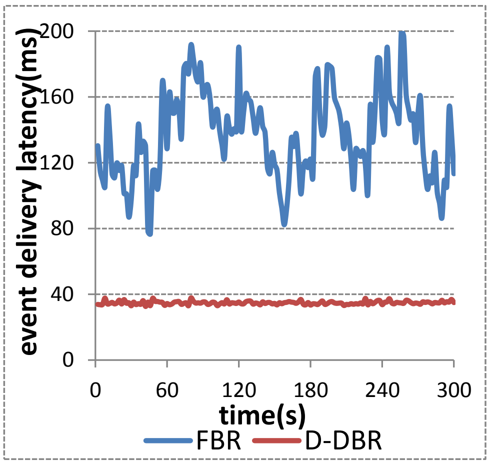


Figure 10.1.5: Latency for Zipf distribution

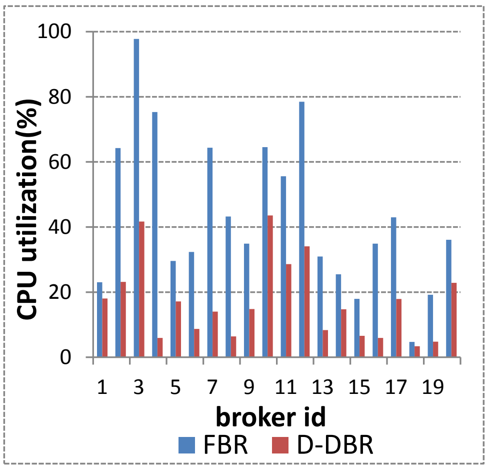


Figure 10.1.6: CPU utilization for Zipf distribution

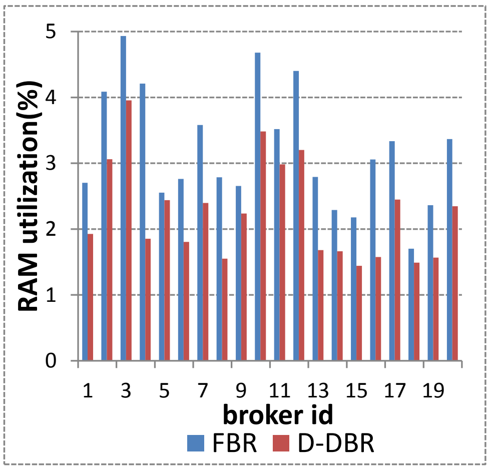


Figure 10.1.7: RAM utilization for Zipf distribution

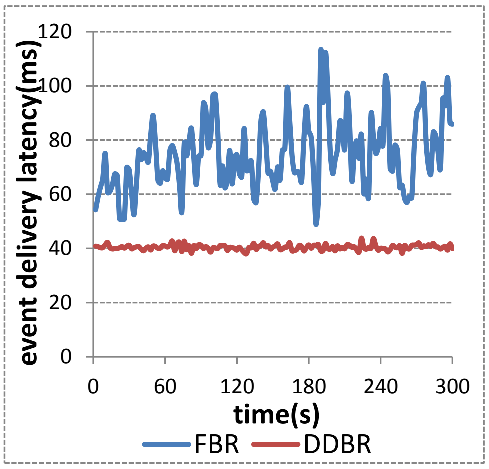


Figure 10.1.8: Latency for random distribution

more stable: During five minutes, D-DBR's event delivery latency is always about 35ms, while FBR's event delivery latency varies between 77ms and 200ms.

Fig. 10.1.6 and Fig. 10.1.7 show the 20 brokers' average CPU and RAM utilization, respectively. It can be seen on almost every broker that FBR consumes more computing resources than D-DBR. The difference on a specific broker is more obvious. For example, on Broker 4, FBR consumes about 75.4% of CPU, while D-DBR only consumes 6% of CPU. On almost every broker, FBR also consumes more memory resources than D-DBR, though, the difference is not as pronounced.

Fig. 10.1.8 shows the event delivery latency of FBR and D-DBR when subscriptions follow the uniform distribution. Compared with Fig. 10.1.5, we know that the event delivery latency of FBR is reduced, however, the event delivery latency of D-DBR slightly increases. The uniform distribution leads to a more balanced workload on every broker. FBR benefits from a balanced workload since brokers are less likely to become overloaded. D-DBR suffers from a balanced workload since its dynamic overlay reconfiguration mechanism becomes less effective. But, as evident from Fig. 10.1.8, under this kind of balanced workload, D-DBR still performs much better than FBR.

In the second group of experiments, we constructed an overlay of 100 brokers for FBR and D-DBR, in which each node has an average node degree of 6. For MERC, we constructed an overlay of 100 brokers following the transit-stub model [67]. There are 10 transit nodes acting as edge brokers and 90 stub nodes acting as internal brokers. In total, there are 10 clusters. Here, the average node degree is also 6. Unlike internal brokers, each edge broker is deployed on a node with 8 cores. For FBR, D-DBR, and MERC, every broker has an attached publisher and subscriber, which issues 1 advertisement and 30 subscriptions, respectively. The subscriptions follow the Zipf distribution.

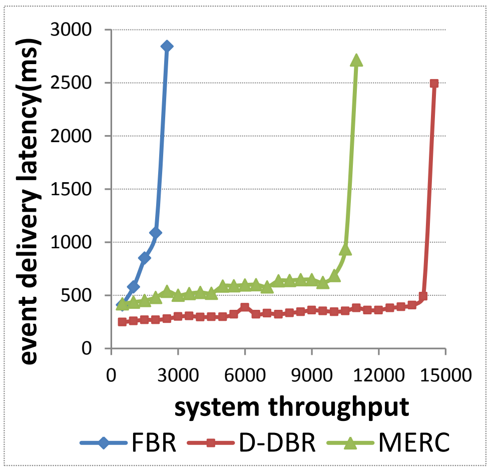


Figure 10.1.9: Throughput and latency

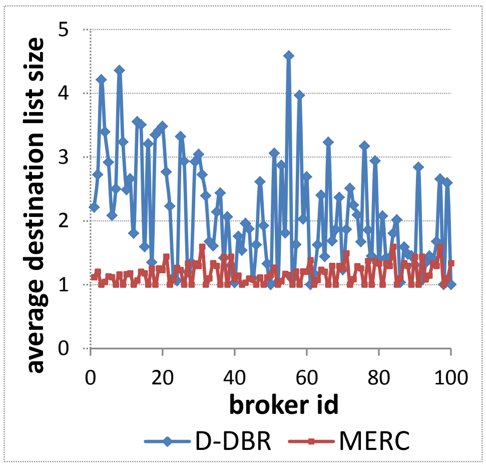


Figure 10.1.10: Destination list size distribution

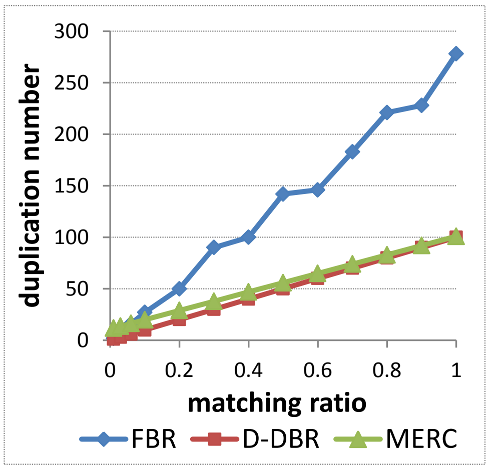


Figure 10.1.11: Subscription duplication

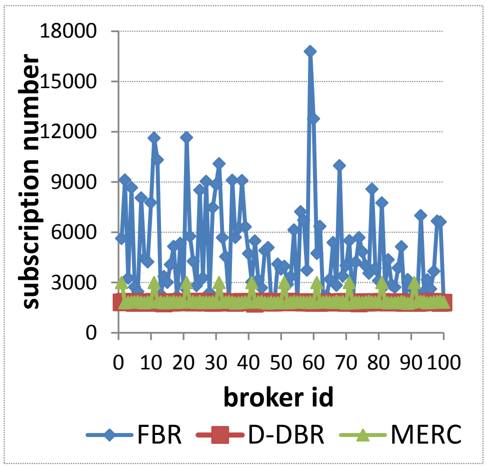


Figure 10.1.12: Subscription number distribution

Fig. 10.1.9 compares the performance of FBR, D-DBR and MERC. The results show that D-DBR exhibits the best performance and MERC lies between D-DBR and FBR. When the event publishing rate (messages/minute) increases from 2,000 to 2,500, the event delivery latency of FBR increases from 1,087 ms to 2,843 ms. However, the event delivery latency of D-DBR is only 491 ms when the event publishing rate is 14,000. Thus, as compared with FBR, D-DBR improves throughput by up to 700% and reduces the communication latency by up to 55% at the same time. The experiments also show that the latency of MERC is stable and slightly higher than D-DBR when the event publishing rate increases to 11,000. This suggests that when powerful edge brokers are used, MERC offers better performance than FBR.

In this group of experiments, we recorded the messages received and sent by each broker and computed the average destination list overhead on all brokers. Fig. 10.1.10 shows that the average destination list size at each broker ranges from 1 to 5 for D-DBR and 1 to 1.6 for MERC. For the whole system, on average, MERC exhibits a smaller destination list size than D-DBR(1.18 vs. 2.12).

In FBR, advertisements need to be broadcast. In this group of experiments, even though only 100 advertisement messages are issued, more than 38,000 duplicated advertisement messages are detected. In D-DBR and MERC, advertisements are routed in the same way as events. No duplicated advertisement messages are generated.

Fig. 10.1.11 shows duplication numbers for each issued subscription when the ratio of brokers with matching advertisements varies. D-DBR exhibits a good subscription duplication behaviour, which means a subscription is only duplicated at brokers with matching advertisements. MERC also presents a good subscription duplication behaviour. However, for FBR, subscriptions are heavily duplicated. For example, when 80% of the brokers have matching advertisements, each subscription is duplicated 221 times, on average[2].

Fig. 10.1.12 shows the number of subscriptions maintained by each broker when 60% of the brokers have matching advertisements. Overall, the 100 subscribers issue 3,000 different subscriptions, each broker stores 5,005, 1,800, and 1,945 subscriptions for FBR, D-DBR, and MERC, respectively.

10.1.2 Experiments Based on Simulations

Through real-world experiments, we evaluated the impact of our algorithms on timing-based metrics, such as event delivery latency. For a more comprehensive study evaluating deterministic properties of our algorithms, we resort to simulations, which simplify experimentation and suffice to study effects determined by message counts, data structure sizes, and routing table entries. For example, the destination list overhead can be affected by the size of the topology, the average broker degree (i.e., the number of neighbours a broker is connected to), and the ratio of interested brokers.

Destination List Overhead: This section investigates the destination list overhead of the algorithms as the number of brokers, the average broker degree, and the ratio of interested brokers for each event varies. In our experiments, different topologies are generated with the Georgia Tech GT-ITM network topology generator. For a topology, which represents a configuration of experimental factors, we evaluated

[2]To support FBR in general overlays, a broker may need to store many copies of the same subscription [38].

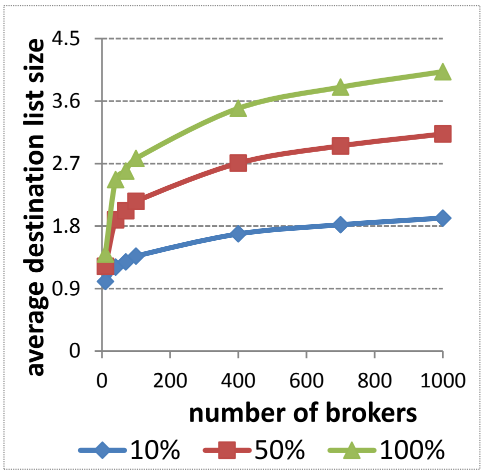


Figure 10.1.13: Average destination list size (broker degree is 6)

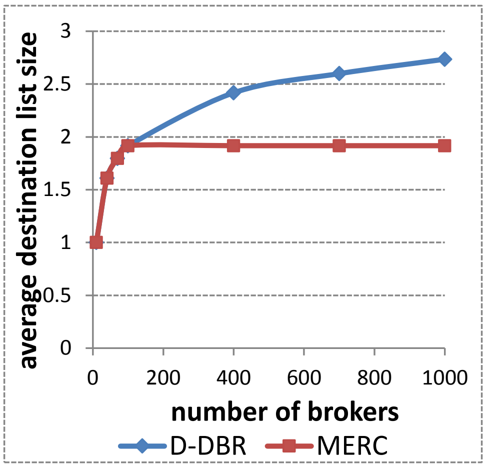


Figure 10.1.14: Average destination list size (broker degree is 9)

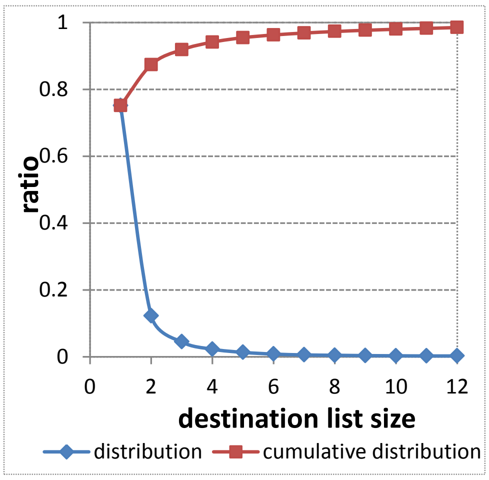


Figure 10.1.15: Destination list size distribution

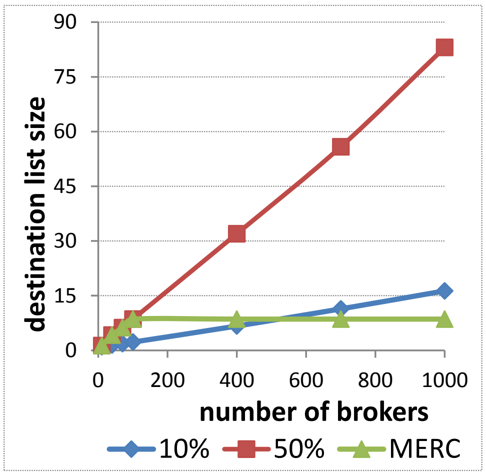


Figure 10.1.16: Average destination list size at source brokers

D-DBR and MERC by routing 10,000 events. For each event, the source broker and destination brokers are randomly selected using different random seeds.

Fig. 10.1.13 shows the average destination list size for D-DBR when the broker degree is fixed to 6. It can be seen that the average destination list size grows when the number of brokers increases. In addition, the more brokers are interested in an event, the longer the average destination list becomes. However, this experiment demonstrates that the average destination list size is quite small, even in a relatively large overlay. For example, in an overlay with 700 brokers, even when an event is issued to all brokers, on average, each generated message carries the IDs of only 3.8 brokers.

Fig. 10.1.14 shows the average destination list size for D-DBR and MERC when the broker degree is fixed to 9. In this experiment, each message is delivered to 50% of the brokers. Comparing Fig. 10.1.14 with Fig. 10.1.13, we see the destination list size can be reduced by increasing the brokers' degree. In the aforementioned example, the average destination list size is reduced from 3.8 to 3.2, when the broker degree increases from 6 to 9. As compared with D-DBR, MERC has a smaller average destination list size. When the cluster size is set to 100, as shown in Fig. 10.1.14, the average destination list size for MERC is only 1.92, when the number of brokers increases to 1,000.

Fig. 10.1.15 shows the distribution of destination list size and its cumulative distribution in a topology with 700 brokers for D-DBR. The average broker degree is set to 6 and an event is delivered to 10% of the brokers. The results show that more than 95% of the destination list sizes are smaller than 5 and more than 99% of them are smaller than 12.

When an event is close to its source broker, the corresponding event messages may carry long destination lists. We study this effect by computing the average destination list size at source brokers for all events. As shown in Fig. 10.1.16, for D-DBR, the destination list size increases linearly as the network scales up. Moreover, the larger the ratio of interested brokers, the longer the destination list becomes. So, for D-DBR, in a large-scale network, if a broker issues lots of events to a large number of brokers, its destination list overhead can be significant. However, this problem does not exist for MERC, since messages are only annotated with addresses of the brokers in the local cluster. In Fig. 10.1.16, for MERC, the average destination list size at source brokers is only 8.6, even when the overlay network has up to 1,000 brokers.

Routing Accuracy: In D-DBR, the overlay topology can be dynamically adjusted for performance optimization purposes. This can improve its routing accuracy, as we show in this experiment.

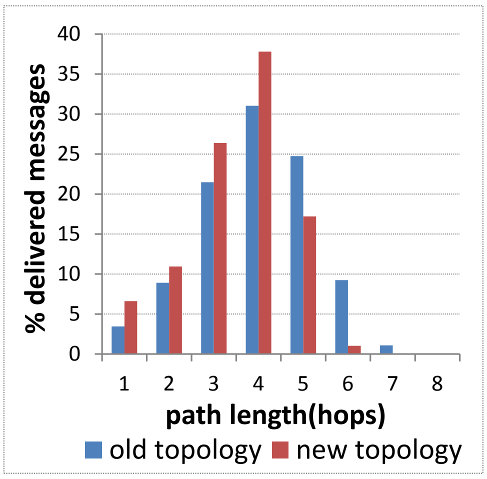


Figure 10.1.17: Routing accuracy for old and new topology

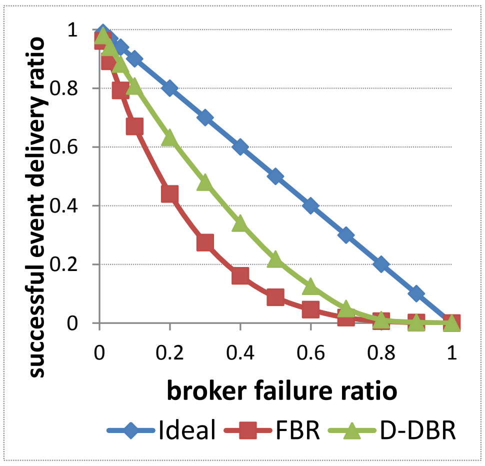


Figure 10.1.18: Robustness under broker failures

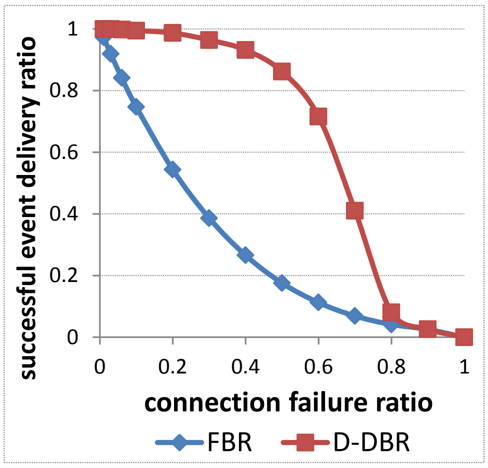


Figure 10.1.19: Robustness under connection failures

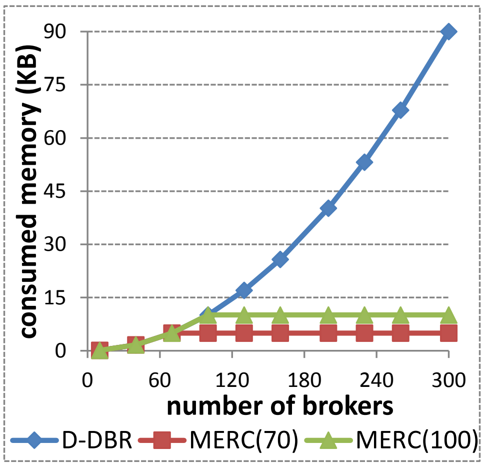


Figure 10.1.20: Consumed Memory

In an overlay with 70 brokers, the workload between every two brokers is randomly selected within the range of 1 to 100 messages/minute. We computed the number of hops each message travels in this overlay. Then, we dynamically reconfigured the overlay, using the solution proposed in Section 4.3, and computed the number of hops each message travels in the new overlay.

Fig. 10.1.17 shows that after the overlay reconfiguration, the number of hops each message travels is reduced. Overall, the average number of hops for all messages is reduced from 3.97 to 3.51, which means the routing accuracy is improved from 0.252 to 0.285.

System Robustness: To study the system robustness under FBR and D-DBR, we compare the successful event delivery ratio when brokers crash or their connections are lost. In our experiments, the overlay has 100 brokers and the average broker degree is 6. Each broker is attached with one publisher and one subscriber. A subscriber issues one subscription and a publisher publishes one event, and each event is delivered to all the subscribers.

Fig. 10.1.18 shows that D-DBR is more robust than FBR under broker failures. For example, when 50% of the brokers fail, 21.8% of events are successfully delivered using D-DBR, whereas only 8.7% of events are successfully delivered using FBR. Fig. 10.1.19 shows that D-DBR is also more robust than FBR for connection failures. For example, when 50% connections are lost, 86.2% of the events are successfully delivered using D-DBR, whereas only 17.5% of the events are successfully delivered using FBR.

MERC applies the D-DBR algorithm for intra-cluster event routing and thus inherits D-DBR's flexibility and robustness. However, the failure of an edge broker may cause a whole cluster to become disconnected from the system in the worst case. Therefore, MERC may suffer from single points of failure caused by edge brokers. We plan to investigate this problem in future work.

Topology Maintenance Overhead: The topology maintenance overhead for D-DBR and MERC consists of two parts: Memory overhead to store routing tables and network communication overhead for overlay reconfiguration. Fig. 10.1.20 shows the consumed memory to store routing tables for D-DBR and MERC. In this figure, the size of each cluster is set to 70 or to 100 for MERC. It can be seen that in D-DBR, the memory overhead grows rapidly with the number of brokers, while in MERC, the memory overhead is limited by the number of brokers in each cluster. If the cluster size is set to 70, a broker consumes at most 5KB of memory to store the routing tables. If set to 100, a broker consumes at most 11KB of memory. The network communication overhead is also different for D-DBR and MERC: The number of messages for an overlay reconfiguration is equal to the number of brokers in the system for D-DBR and is equal to the number of brokers in each cluster for MERC.

10.2 Predicate Matching

PS-Tree, more precisely, PS-Tree$_B$, can be interpreted to store intervals and allows querying which of the stored intervals contain a given point. Interval-Tree [18] and Segment-Tree [23] are two index structures that provide similar capabilities. Thus, both represent approaches related to PS-Tree. Here, we conduct the following comparative evaluations.

Table 10.2.1: PSTree Querying Performance

Index	Querying	Construct	Memory
SCAN	23.11 s	-	-
Interval-Tree	2.52 s	16.02 ms	1.61 MB
Segment-Tree	6.27 ms	29.71 ms	6.86 MB
PS-Tree	0.71 ms	91.88 ms	54.13 MB

In this group of experiments, we compare the query time, index construction time and memory consumption of PS-Tree, Segment-Tree and Interval-Tree. SCAN, which represents the naive scanning method, is used as a baseline. Table 10.2.1 shows the experimental results when there are 100K intervals and 100K query points. As shown in this table, although PS-Tree exhibits higher index construction time and memory use, its query performance is the best. Compared to Interval-Tree and Segment-Tree, PS-Tree reduces the matching time by 99.97% and 89%, respectively, which suggests that PS-Tree is more suitable for Boolean expression matching, where the number of queries is much higher than the number of intervals. Moreover, PS-Tree supports more operators, such as "$\in$" and "$>$", which are not supported by Segment-Tree and Interval-Tree. Another advantage of PS-Tree over Segment-Tree and Interval-Tree is that the PSTHash algorithm can only be supported by PS-Tree.

10.3 Conjunctive Boolean Expression Matching

This section evaluates `PSTBloom` and `PSTHash` using both synthetic and real-world datasets. `BE-Tree`, `OpIndex`, `Propagation`, `k-index`, and `SCAN` (a sequential scan of the subscriptions) are selected as baselines. With the exception of `SCAN`, these algorithms have been shown to exhibit good performance in the literature[3]. All algorithms are implemented in C[4] and compiled with gcc 4.8.4 using the optimization level O3 on a Ubuntu 14.04 system. All experiments were run on an Intel 2.66 GHz machine with 512 GB of memory.

In the experiments, we consider a variety of controlled experimental conditions: workload size, workload distribution, dimension number, dimension cardinality, subscription size, event size, matching probability, and predicate selectivity.

10.3.1 Workloads

We first used the `BE-Gen` workload generator [56] to generate synthetic workloads. Table 10.3.1 summarizes the parameters and settings, with the default values highlighted in bold. To evaluate scalability, we vary the number of subscriptions from 300K to 100M. The attributes of the predicates were drawn from the distribution $P(r) = \frac{C}{r^{\alpha}}, r \neq 0$. When α is 0, the distribution is Uniform; otherwise, the distribution is Zipf. The number of dimensions varies from 100 to 30K. The default number of dimensions is set to 100 and 30K to represent low and high dimensionality, respectively. We vary the dimension cardinality from 3 to 1K. We vary the average subscription size from 5 to 30 and the event size from 30 to 130. The matching probability varies from 0.1% to 50%. The equality operator ratio varies from 0% to 100% to represent different predicate selectivities. Compared with the related approaches,

[3]We also compared `PSTBloom` and `PSTHash` with `SIFT`, `Gryphon`, `REIN`, and `GEM`. These algorithms do not exhibit comparable performance; thus, we omitted the experimental results to focus on better-performing algorithms.

[4]The authors of some related approaches kindly provided the source code of their implementations [56, 68, 51]. For consistency, we reimplemented `OpIndex` and `REIN` in C because the original versions were written in C++.

our workloads are comprehensive, thereby exploring a richer parameter space. For example, the workloads used by `OpIndex` all follow the Uniform distribution, and the number of subscriptions increases to only 1M in `BE-Tree`.

Table 10.3.1: Parameters of the Synthetic Datasets

Subscription Number	300K, **1M**, 3M, 10M, 30M, 100M
The α in Zipf	**0**, 1, 2, 3, 4, 5
Dimension Number	**100**, 300, 1K, 3K, 10K, **30K**
Dimension Cardinality	3, 10, 30, **100**, 300, 1K
Avg. Subscription Size	**5**, 10, 15, 20, 25, 30
Avg. Event size	**30**, 42, 54, 66, 78, 90
Matching Probability	**0.001**, 0.005, 0.01, 0.05, 0.1, 0.5
Equal. Operator Ratio	0, **0.2**, 0.4, 0.6, 0.8, 1.0

The second synthetic dataset uses the query logs from the SIGMOD 2013 contest to represent keyword-based subscriptions [1]. We transform a query into a Boolean expression whereby each keyword is treated as an equality predicate. If a keyword has more than six characters, we transform it into a predicate using the first three characters as the attribute name and the next three characters as the attribute value. For example, “boolean" is transformed into $\{boo, =, \text{“}lea\text{”}\}$. Otherwise, a keyword is transformed into a predicate using the first half of its characters as the attribute name and the remaining characters as the attribute value. For example, “vldb" is transformed into $\{vl, =, \text{“}db\text{”}\}$. This transformation results in Boolean expressions in a space of 17,577 dimensions. The document dataset is transformed into events using a similar method.

In addition to these synthetic datasets, we also derived a real-world workload based on a display ads dataset of an online shopping site for subscriptions and events. When a user visits the site, product advertisements are shown to the user. In the backend advertisement inventory, an advertisement specifies conditions to promote products to users. The conditions include channel (e.g., mobile, PC, or tablet), region (e.g., CA, DE, or CN), ad position, etc. By translating conditions into predicates, we model advertisements as subscriptions. When a user interacts with the website (e.g., surfs or logs in), the user's session is bound to a set of attributes such as the login channel, the login region and the user's profile. By translating the profile and attributes into attribute-value pairs, we model each session as an event. For example, if a user is male, the resulting event contains an attribute-value pair $\langle gender, male \rangle$.

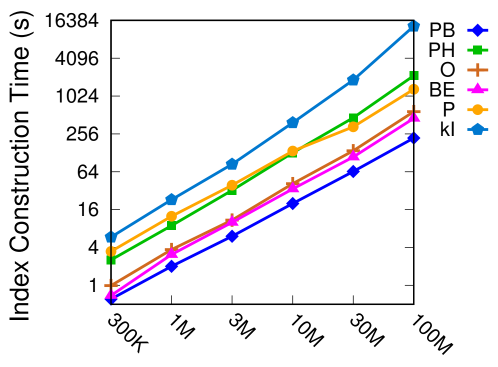
10.3.2 Experiments on Synthetic Workloads

The first set of experiments was conducted on the synthetic datasets. We first report on the index construction time. Then, we evaluate the matching performance with respect to workload size, distribution, number of dimensions, etc. Finally, the memory use of each index is reported.

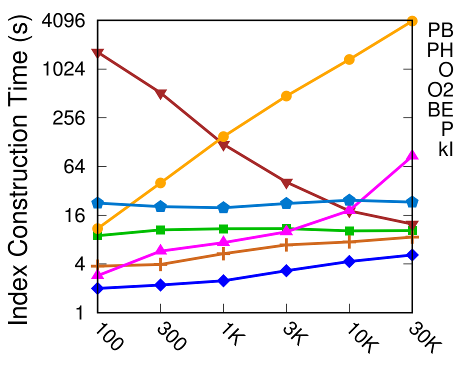
Index Construction Time

Our experiments show that not only workload size but also the number of dimensions, subscription size, and equality operator ratio affect the index construction time. As shown in Fig. 10.1(a), all the algorithms' index construction times increase with the number of subscriptions. Among the algorithms shown, `PSTBloom` exhibits the lowest index construction time: when there are up to 100M subscriptions, compared with the next-best algorithm `BE-Tree`, `PSTBloom` reduces the index construction time by 78%.

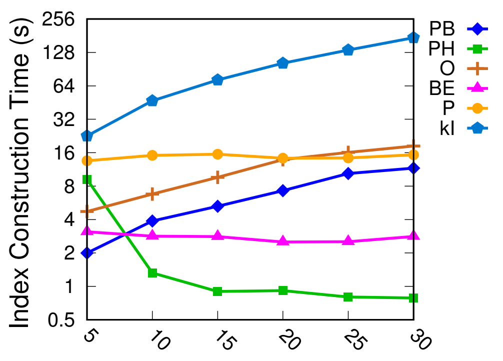
Fig. 10.1(b) shows how the number of dimensions affects the index construction time. As shown, `BE-Tree` and `Propagation` increase quickly with the number of dimensions, whereas `PSTBloom`, `PSTHash` and `k-index` are not sensitive to the number of dimensions. For `OpIndex`, after each subscription is inserted, its index needs to be reordered before event matching can resume. An optimization adopted by `OpIndex` is to sort its index after all subscriptions are inserted. The light brown line shows the index construction time of `OpIndex` when this optimization is utilized. However, when the arrivals of subscriptions and events overlaps, `OpIndex` cannot operate in this manner. The brown line shows the index construction time of `OpIndex` when the index is kept up to date after each subscription is inserted. As shown, for dense workloads (when the number of dimensions is small), the index construction time of `OpIndex` is three orders of magnitude larger than that of `PSTBloom`, `BE-Tree`, and `PSTHash`.



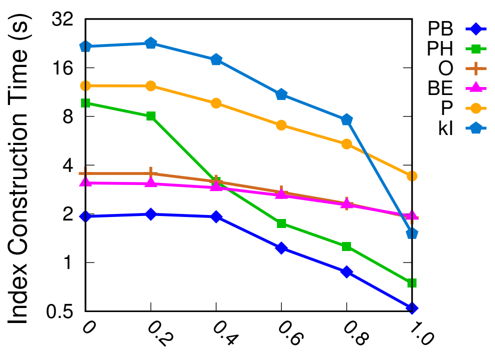
(a) Workload Size



(b) Dimension number



(c) Subscription Size



(d) Equal Operator Ratio

Figure 10.3.1: Index Construction Time

Fig. 10.1(c) shows how the average subscription size affects the index construction time. The index construction time of `PSTBloom`, `OpIndex` and `k-index` increases with the subscription size. `BE-Tree` and `Propagation` are not sensitive to subscription size, whereas the index construction time of `PSTHash` decreases. `PSTHash` possesses this advantage because it builds indexes only on access predicates. When there are more predicates in each subscription, access predicates with high selectivity are more likely to be selected.

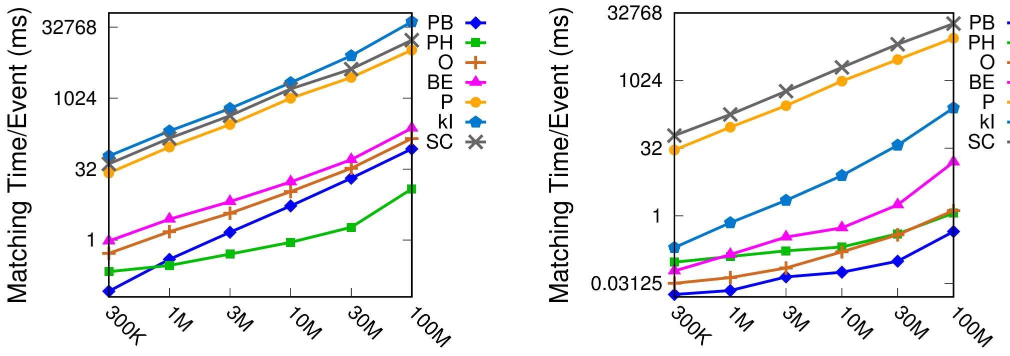
The effect of the equality operator ratio on the index construction time is shown in Fig. 10.1(d). All algorithms show a decrease in the index construction time when the equality operator ratio increases. `PSTHash` decreases most quickly because a higher equality operator ratio results in fewer predicate spaces being covered by the inserted predicate. `PSTBloom` achieves the lowest index construction time. For a similar reason, the advantage of `PSTBloom` is more obvious when the equality operator ratio is high.

Matching Time

The matching time is among the most important metrics for Boolean expression matching algorithms. In this section, we present extensive experiments under a variety of controlled conditions. In particular, the number of dimensions is a distinguishing factor among matching algorithms. For each controlled condition, we ran two experiments: (1) with 100 dimensions and (2) with 30K dimensions. Since the default number of subscriptions is 1M, these settings represent dense and sparse workloads, respectively.

Workload Size: We consider the matching time as we increase the number of subscriptions processed. As illustrated by Fig. 10.3.2, all algorithms scale linearly with respect to the number of subscriptions. Among them, `PSTHash` increases slowest, especially for dense workloads. In Fig. 10.2(a), when there are 300K subscriptions, `PSTBloom` performs best. Compared with `OpIndex`, `PSTBloom` reduces the matching time by 84%. When the number of subscriptions is greater than 1M, `PSTHash` performs best. When there are up to 100M subscriptions, `PSTHash` reduces the matching time by 92% compared to `OpIndex`. In Fig. 10.2(b), `PSTHash` performs as well as `OpIndex` when the number of subscriptions increases to 30M, and `PSTBloom` always performs best.

Workload Distribution: Workload distribution is another distinguishing factor among matching algorithms. In Fig. 10.3.3, the workload distribution is $P(r) = \frac{C}{r^\alpha}$. When α is 0, the distribution is Uniform; when α is greater than 0, the distribution is Zipf. Given a Zipf distribution, a few popular dimensions are associated with a large number of subscriptions, resulting in dense workloads. As shown, under the Zipf distribution, `PSTHash` performs best regardless of the number of dimensions evaluated (here, 100 or 30K). An interesting finding is that `OpIndex` outperforms `BE-Tree` under the Uniform distribution; however, `BE-Tree` outperforms `OpIndex` under the Zipf distribution because the index retrieval time of `OpIndex` increases when there are a large number of subscriptions associated with a few popular dimensions. For a similar reason, under the Zipf distribution, `k-index` performs even worse than `SCAN`.



(a) 100 Dims (b) 30K Dims

Figure 10.3.2: Varying Workload Size

Number of Dimensions: As shown in Fig. 10.4(a), when the number of subscriptions is fixed and the number of dimensions increases, the matching times of all algorithms decrease, except for those of `Propagation` and `SCAN`. Compared to `PSTHash` and `BE-Tree`, the matching times of `PSTBloom`, `OpIndex` and `k-index` decrease quickly. Intuitively, these three algorithms are more suitable for high-dimensional workloads. However, the prerequisite is that the number of subscriptions does not increase simultaneously. In Fig. 10.4(b), we increase the number of

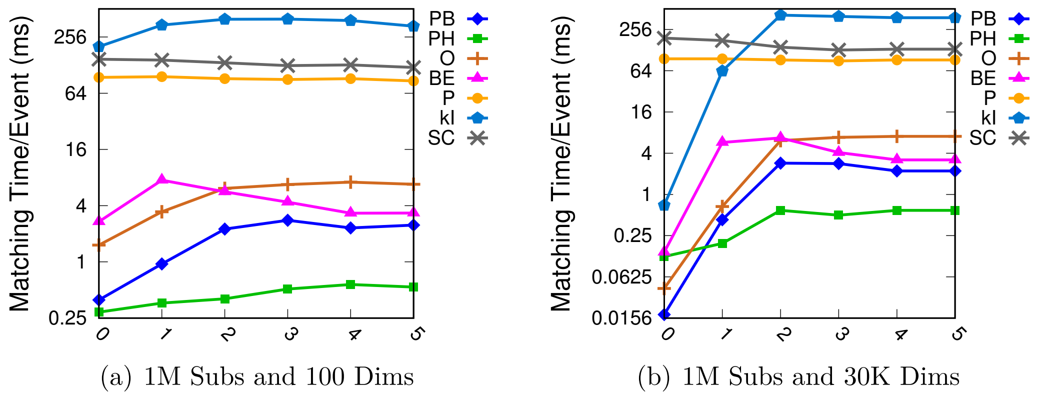


Figure 10.3.3: Varying Workload Distribution

dimensions while keeping the number of subscriptions per dimension fixed. As shown, PSTBloom, OpIndex and k-index are no longer sensitive to the number of dimensions. By combining the findings of these two experiments, we observe that PSTBloom, OpIndex and k-index are more suitable for sparse workloads. In this experiment, when the number of dimensions is greater than 300, PSTBloom always achieves the best performance.

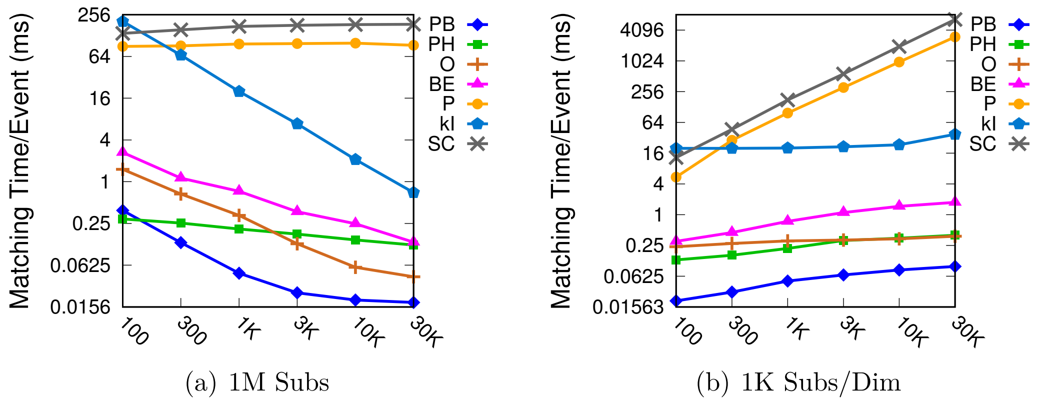
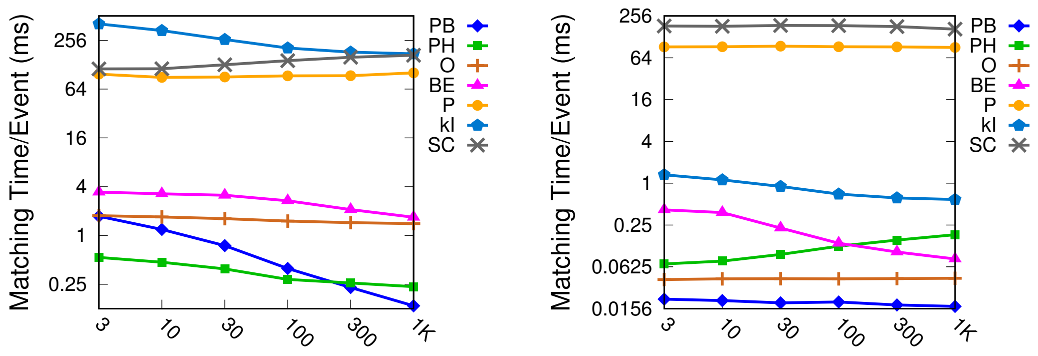


Figure 10.3.4: Varying Number of Dimensions

Dimension Cardinality: Fig. 10.3.5 shows how the dimension cardinality affects the matching time. As shown in Fig. 10.5(a), PSTHash performs best when the dimension cardinality is less than 300, while PSTBloom performs best for higher-dimension cardinalities. For example, when the dimension cardinality is 1K, compared to that of OpIndex, the matching time of PSTBloom is reduced by up to 91%.



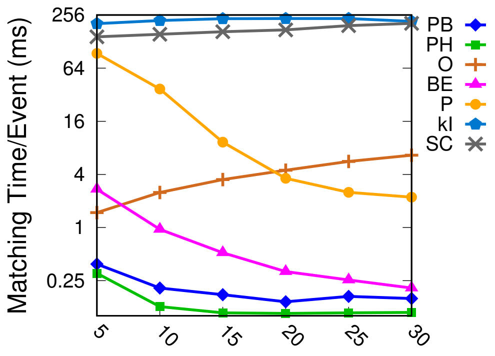
(a) 1M Subs and 100 Dims (b) 1M Subs and 30K Dims

Figure 10.3.5: Varying Dimension Cardinality

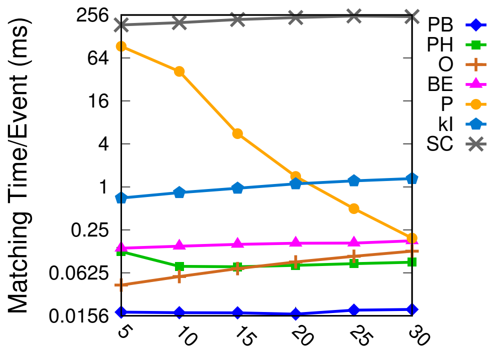
Subscription Size: Another important workload characteristic is the subscription size. As shown in Fig. 10.6(a), given dense workloads, `PSTBloom`, `PSTHash`, `BE-Tree`, and `Propagation` all present lower event matching times as the average subscription size increases because these algorithms select predicates with high selectivity to prune subscriptions. When there are more predicates in a subscription, predicates with high selectivity are more likely to be found. Under sparse workloads, the effect of subscription size is not obvious, except for `Propagation`. In these experiments, only `OpIndex` performs worse as the average subscription size increases. When the average subscription size increases from 5 to 30, the matching time of `OpIndex` increases by 350% and 200% for dense and sparse workloads, respectively, because the index size of `OpIndex` increases linearly with the total number of predicates. Larger subscription sizes result in larger index scanning costs.

Event Size: When the average number of attribute-value pairs of an event increases, there are more candidate subscriptions. As shown in Fig. 10.3.7, under both dense and sparse workloads, all algorithms, except for `Propagation` and `SCAN` show higher matching times as the event size increases. Given dense workloads, when the average event size increases to 90, `PSTHash` performs best. Compared to `BE-Tree`, `PSTHash` reduces the matching time by 85%. Given sparse workloads, when the average event size increases to 90, `PSTBloom` performs best. Compared to `OpIndex`, `PSTBloom` reduces the matching time by 80%.

Matching Probability: We refer to the *matching probability* as the expected ratio

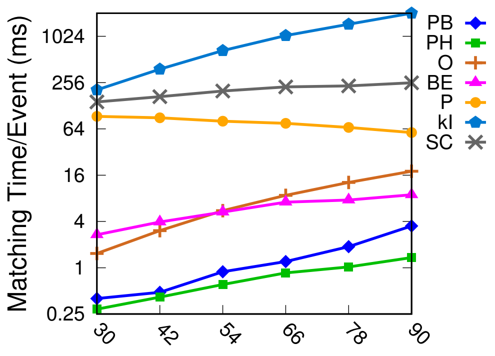


(a) 1M Subs and 100 Dims

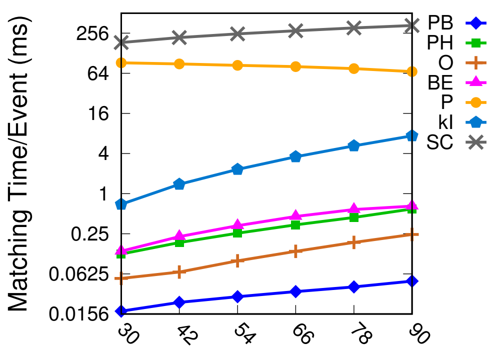


(b) 1M Subs and 30K Dims

Figure 10.3.6: Varying Subscription Size

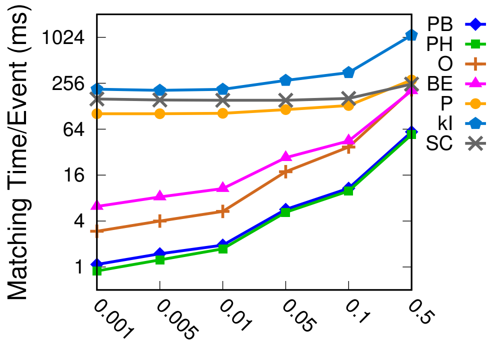


(a) 1M Subs and 100 Dims

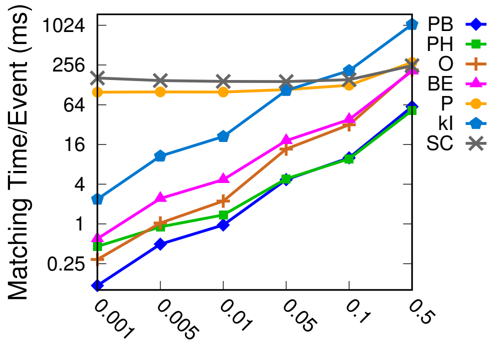


(b) 1M Subs and 30K Dims

Figure 10.3.7: Varying Event Size

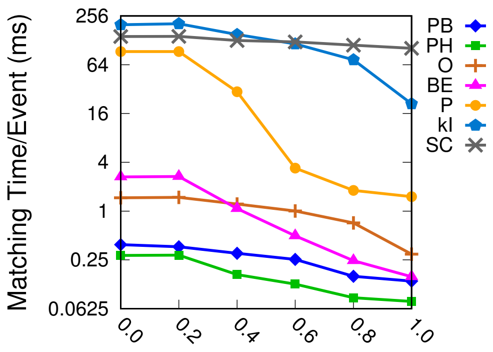


(a) 1M Subs and 100 Dims

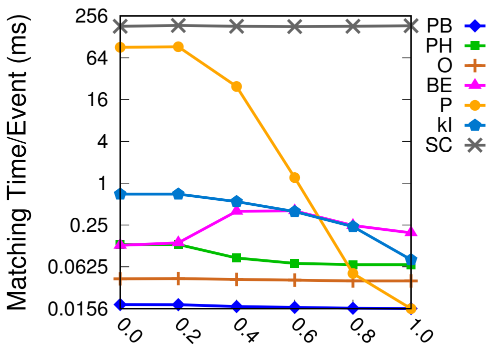


(b) 1M Subs and 30K Dims

Figure 10.3.8: Varying Matching Probability



(a) 1M Subs and 100 Dims



(b) 1M Subs and 30K Dims

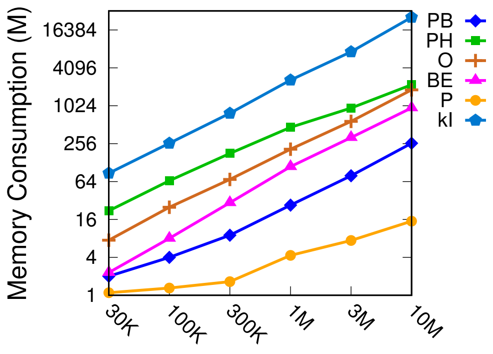
Figure 10.3.9: Varying Equal Operator Ratio

of subscriptions that match for a given event. A higher matching probability means that more subscriptions match. In Fig. 10.3.8, with the exception of `Propagation` and `SCAN`, the algorithms' matching times increase with the matching probability under both dense and sparse workloads. When the matching probability increases to 50%, `BE-Tree`, `OpIndex` and `Propagation` show similar matching times as that of `SCAN`. In contrast, both `PSTBloom` and `PSTHash` need only approximately 25% of this matching time for both dense and sparse workloads.

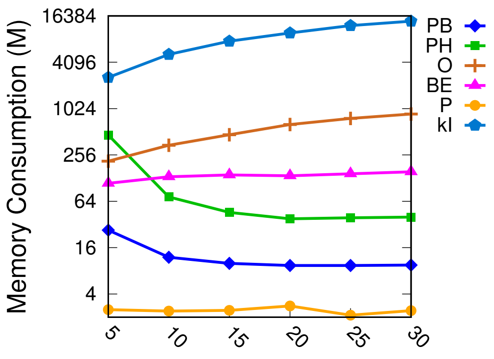
Equality Operator Ratio: In this experiment, we study the effect of the ratio of equality vs. nonequality predicates per subscription. Fig. 10.3.9 shows that the general trend is that the matching time of all algorithms decreases as the percentage of equality predicates increases. Most notably, when subscriptions consist of only equality predicates, `Propagation` achieves a substantial performance gain and is as good as `PSTBloom` under sparse workloads. Given dense workloads, the matching performance of `PSTHash` always ranks first. Moreover, when the equality operator ratio is higher, the advantage of `PSTHash` becomes more obvious.

Memory Consumption

All indexes considered in this thesis are memory resident. Here, we evaluate memory use. To accurately report the memory consumption of each index, we calculate



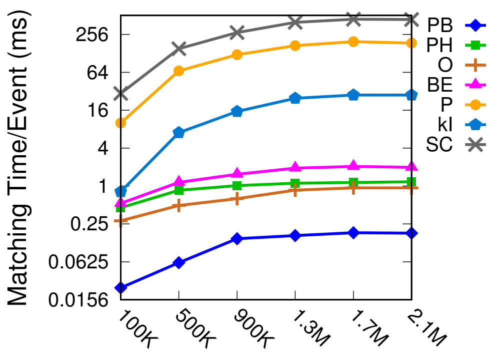
(a) Varying Workload Size



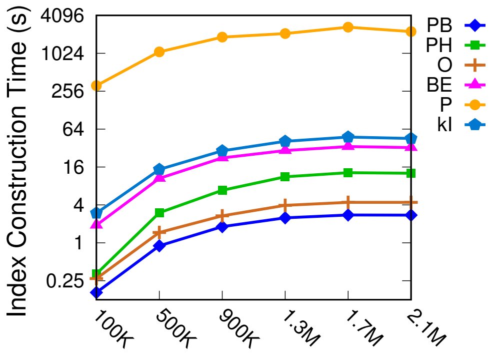
(b) Varying Sub Size

Figure 10.3.10: Memory Consumption

the memory use of the runtime processes before and after all subscriptions are inserted into each index. Fig. 10.10(a) shows the memory use as the number of subscriptions increases. Unsurprisingly, all algorithms require more memory when there are more subscriptions. However, the memory use of `PSTBloom` and `PSTHash` increases slower than that of `OpIndex` and `BE-Tree` because the number of predicate spaces maintained in `PS-Trees` increases slower than the number of subscriptions. Fig. 10.10(b) shows the memory use as the average subscription size increases: `OpIndex` and `k-index` require more memory, `BE-Tree` and `Propagation` remain stable, and `PSTBloom` and `PSTHash` need less memory. In these two experiments, `Propagation` needs the least amount of memory. Compared with `BE-Tree` and `OpIndex`, `PSTBloom` reduces memory use by up to 94% and 99%, respectively.



(a) Matching Time



(b) Construction Time

Figure 10.3.11: Varying Workload Size

10.3.3 Experiments on Query Logs

In this synthetic dataset, which stems from query logs, there are 2.1M subscriptions in a space of 17,577 dimensions. On average, each subscription contains 2.78 predicates, and each event contains 93.07 attribute-value pairs. As shown in Fig. 10.11(a), under this high-dimensional workload, the matching performance ranking is in the following order: `PSTBloom`, `OpIndex`, `PSTHash`, `BE-Tree`, `k-index`, `Propagation`, and `SCAN`. `OpIndex`, `PSTHash`, and `BE-Tree` have similar matching latencies, and `PSTBloom` performs considerably better than these three algorithms. Fig. 10.11(b) shows the index construction time. The ranking of the index construction time is the same as that of the matching time. The difference is that `PSTBloom` and `BE-Tree` have similar index construction times as `OpIndex` and `k-index`, respectively.

10.3.4 Experiments on the Ads Dataset

By transforming advertisements into subscriptions, in this workload, we obtain 3M subscriptions. The number of predicates in a subscription ranges from 1 to 56. On average, each subscription contains 8 predicates, and each event contains 20 attribute-value pairs. The number of dimensions is 122, which means that this workload is a dense workload.

As shown in Table 10.4.2, under this workload, `PSTHash` achieves the best matching performance, followed by `PSTBloom`. The matching performance of `BE-Tree` is a little better than that of `OpIndex`. Compared with `BE-Tree`, `PSTHash` reduces the matching time by 89%. `PSTBloom` achieves the shortest index construction time. Moreover, compared with `BE-Tree` and `OpIndex`, `PSTBloom` needs less memory. This experimental result is roughly consistent with the experimental results on synthetic workloads.

Table 10.3.2: Experiments on the Ads Dataset

Index	Matching	Construct	Memory
SCAN	435.27 ms	-	-
k-index	616.42 ms	140.76 s	7.32 GB
Propagation	168.73 ms	62.43 s	14.41 MB
OpIndex	3.21 ms	24.92 s	574.38 MB
BE-Tree	2.26 ms	13.70 s	393.21 MB
PSTBloom	0.82 ms	7.81 s	55.93 MB
PSTHash	0.24 ms	8.22 s	759.72 MB

10.4 Arbitrary Boolean Expression Matching

This section evaluates `A-Tree`-based arbitrary Boolean expression matching using both synthetic and real-world datasets. We compare against the following alternative arbitrary Boolean expression matching algorithms: `BoP` [7], `Dewey ID` [30], `Interval ID` [30], `BDD` [10], `Translation`, and `Scan` (a sequential scan of the expressions). Since a conjunctive Boolean expression is a special case of an arbitrary Boolean expression, we also compare these algorithms to the `BE-Tree` and `OpIndex` conjunctive Boolean expression matching algorithms. `BE-Tree` and `OpIndex` are selected because they perform better than other existing conjunctive Boolean expression matching algorithms, such as `Propagation` [27], `k-index` [63], `Gryphon` [3], `SIFT` [3], `TAMA` [69], `REIN` [51] and `GEM` [28]. We implemented all the arbitrary Boolean expression matching algorithms in C by ourselves [5] and compiled with gcc 4.4.8 using optimization level O3 on an Ubuntu 16.04 system. All experiments were run on an Intel 2.66 GHz machine with 128 GB of memory.

In the experiments, we consider a variety of controlled experimental conditions: workload size, arbitrary expression tree depth, the number of child nodes, expression duplicated number, dimension number, dimension cardinality, dimension distribution, and event size.

[5]The authors of `BE-Tree` and `OpIndex` kindly provided the source code of their implementations. The authors of the remaining approaches did not provide the source codes.

10.4.1 Workloads

To compare with the existing conjunctive Boolean expression matching algorithms, we first use the `BE-Gen` workload generator [56] to generate a set of conjunctive workloads. Since we mainly focus on arbitrary expression workloads, we select some fixed typical parameters: the number of dimensions is 1000, the average expression size is 6, the average event size is 20, the predicate distribution is uniform, and the number of expressions varies from 30K to 10M.

To synthesize arbitrary Boolean expression workloads, we created a new workload generator called `ABE-Gen`. In `ABE-Gen`, the generation of an arbitrary Boolean expression, i.e., an n-ary tree, starts from the root node and recursively moves to the child nodes. For each node, we first decide the logical operator, which is selected from *and*, *or*, *not*, *xor* and *xnor*. If the logical operator is selected as *and* or *or*, we then decide the number of child nodes. In addition to the logical operator and child number, we also use another parameter called *tree depth* to control the expression generation. When a node's depth is equal to the maximum tree depth, that node will be directly identified as a leaf node. For each leaf node, we generate a predicate. The generation of predicates is controlled by a set of parameters, such as dimension number, dimension cardinality and dimension distribution.

To generate a wide distribution on the n-ary trees, i.e., arbitrary Boolean expressions, we use a wide range of parameters and settings, as shown in Table 10.4.1, with the default values highlighted in bold. To evaluate scalability, we vary the number of expressions from 30K to 10M. The default logical operator distribution is 40% *and*, 40% *or*, 10% *not*, 5% *xor* and 5% *xnor*. The tree depth varies from 1 to 6. The child number varies from 2 to 12. In real world workloads, the same arbitrary Boolean expression often appears for more than once, so, we also conduct experiments to vary the expression duplicated number from 0 to 5. The predicates' attributes are drawn following the distribution $P(r) = \frac{C}{r^\alpha}, r \neq 0$. When α is 0, the distribution is uniform; otherwise, the distribution is Zipf. The number of dimensions varies from 100 to 30K. The default number of dimensions is set to 1000. We vary the dimension cardinality from 3 to 1K. We vary the average event size from 5 to 30.

Table 10.4.1: ABE-Gen Parameters on Synthetic Datasets

Expression Number	30K, 100K, 300K, **1M**, 3M, 10M
Operator Distribution	40%, 40%, 10%, 5% 5%
Tree Depth	1, 2, **3**, 4, 5, 6
Child Number	2, **4**, 6, 8, 10, 12
Expression Repeat Times	**0**, 1, 2, 3, 4, 5
The α in Zipf	**0**, 1, 2, 3, 4, 5
Dimension Number	100, 300, **1K**, 3K, 10K, 30K
Dimension Cardinality	3, 10, 30, **100**, 300, 1K
Event Size	5, 10, 15, **20**, 25, 30

In addition to these synthetic datasets, we also designed a real-world workload based on an internet company's Ads dataset to generate arbitrary Boolean expressions and events. When a user surfs on the company's webiste, product advertisements are shown to the user. In the backend advertisement inventory, an advertisement specifies conditions to promote products to users. The conditions include channel (e.g., mobile, PC, or tablet), region (e.g., US, DE, or CN), ad position, and so forth. By translating conditions into predicates, we model advertisements as arbitrary Boolean expressions. When a user interacts with the website (e.g., surfs or logs in), the user's session is bound to a set of attributes, such as the login channel, the login region and the user's profile. By translating the profile and attributes into attribute-value pairs, we model each session as an event. For example, if a user is male, then the resulting event contains an attribute-value pair $\langle gender, male\rangle$.

10.4.2 Experiments on Conjunctive Workload

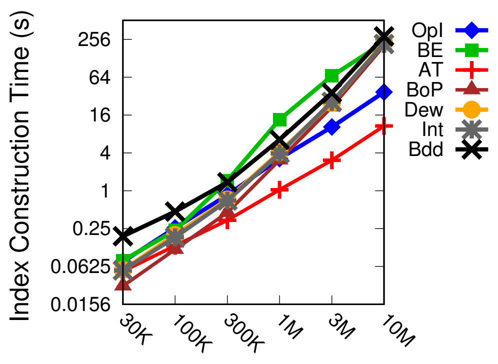
The first set of experiments was conducted on the synthetic conjunctive expression datasets. We mainly investigate the scalability of different algorithms with respect to the workload size. The index construction time, matching performance and memory consumption are reported. We also evaluate the matching performance of different algorithms with respect to the expression duplicated number.

Index Construction Time: As shown in Fig. 10.1(a), all the algorithms' index construction times increase with the number of expressions. Among the algorithms shown, `BoP` exhibits the lowest index construction time when there are 30K expressions. However, when there are up to 10M expressions, `A-Tree` achieves the fastest index construction; compared with the next-best algorithm `OpIndex`, `A-Tree` reduces the index construction time by 71.1%. `A-Tree` presents fast index construction because common shared predicates and subexpressions can always be reused.

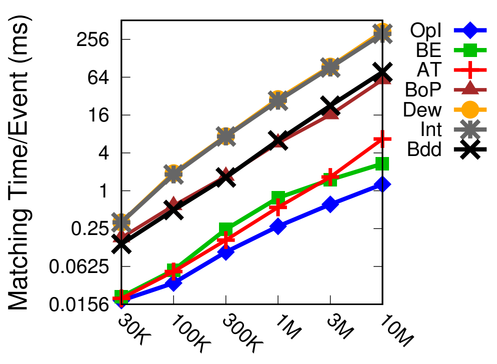
Matching Time: The matching time is the most important metric for Boolean expression matching algorithms. As illustrated in Fig. 10.1(b), all algorithms scale linearly with respect to the number of expressions. Under different workloads, `A-Tree` achieves matching performance comparable with `BE-Tree`. `OpIndex` achieves the best matching performance under this group of workloads. `BoP` and `Interval ID` present matching performances similar to those of `BDD` and `Dewey ID`, respectively. The matching performances of `A-Tree` are much better than those of `BoP` and `BDD` which are further better than those of `Interval ID` and `Dewey ID`.

For many applications, the same expression may appear more than once. Thus, we also investigate the matching performances of these algorithms under duplicated expression workloads. Fig. 10.1(c) shows the matching time when each unique expression is duplicated 0, 1, 2, 3, 4, and 5 times. Zero means that the 1M expressions are not duplicated at all. As shown, only the matching times of `A-Tree` and `BE-Tree` do not increase with the expression duplicated number. This result is because the expressions will be indexed together by `A-Tree` and `BE-Tree` if the expressions are the same. However, duplicated expressions are not identified by the other algorithms and thus are proccesed independently.

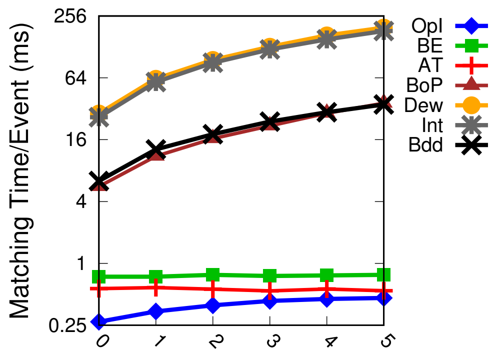
Memory Consumption: All indices considered in this thesis reside in memory. Here, we evaluate memory usage. To accurately report the memory consumption of each index, we calculate the memory use of the runtime processes before and after all expressions are inserted into each index. Fig. 10.1(d) shows the memory use as the number of expressions increases. Unsurprisingly, all algorithms require more memory when there are more expressions. However, the memory use of `A-Tree` increases slower: when there are 30K expressions, the memory use of `A-Tree` is greater than that of `BE-Tree`. However, when there are 10M expressions, the memory use of `A-Tree` becomes smaller than that of `BE-Tree`. `BE-Tree` always consumes



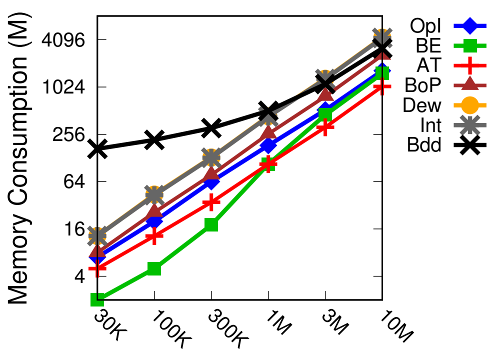
(a) Varying Number of Expressions



(b) Varying Number of Expressions



(c) Varying Repeated Times



(d) Varying Number of Expressions

Figure 10.4.1: Conjunctive Workloads

less memory than `OpIndex`.

10.4.3 Experiments on Arbitrary Workload

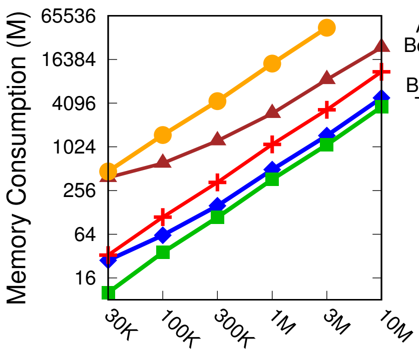
Under the workloads presented in the above section, `OpIndex` achives better matching performance than `BE-Tree`. Though our experiments show `BE-Tree` presents better performance than `OpIndex` under some other workloads, we only adopt `OpIndex` as the underling algorithm for the `Translation`-based arbitrary Boolean expression matching solution. `Dewey ID` and `Interval ID` are proposed in the same paper [30]. The basic idea of these two algorithm are similar and `Interval ID` always performs a little better than `Dewey ID`. So, in this group of experiments, we only use `Interval ID` for comparison. This group of experiments were conducted on the synthetic arbitrary expression datasets. Under some workloads, the memory needed by `Translation` and `BDD` exceeds the available memory of 128GB. So, we first report on the memory consumption of each index. Then, we evaluate the matching performance with respect to workload size, expression tree depth, dimension number, etc. Finally, index construction is reported.

Memory Consumption

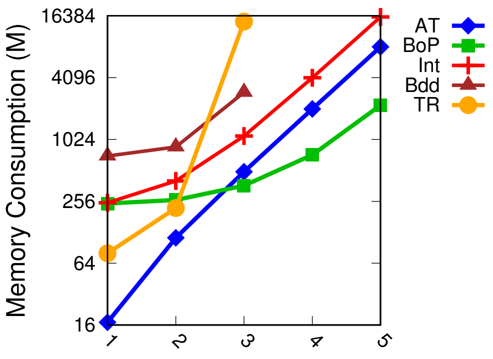
For arbitrary Boolean expression workloads, our experiments show the memory consumption of different algorithms are affected by the workload size, expression tree depth, the number of child nodes and expression duplicated number.

Fig. 10.2(a) shows the memory use as the number of expressions increases. Unsurprisingly, all algorithms require more memory when there are more expressions. However, the memory use of `A-Tree` and `BDD` increases slower than that of `BoP`, `Interval ID` and `Translation`. This is because the common predicates and subexpressions can be shared in the `A-Tree` and `BDD` index. `BoP` requires the least memory because it compresses the expressions and only requires a count for each expression in the index. `Translation` needs the most memory since a single arbitrary expression can be translated into up to $4^4 = 256$ conjunctive expressions. When there are

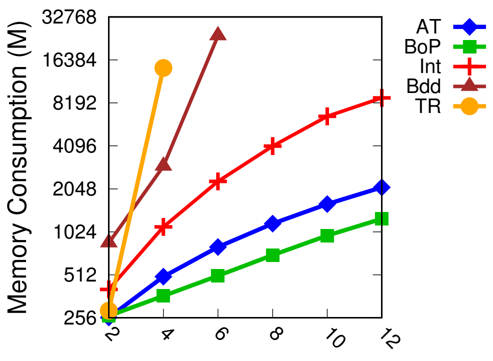
10M arbitrary expressions, The available 128GB memory is not enough for the `Translation` solution.



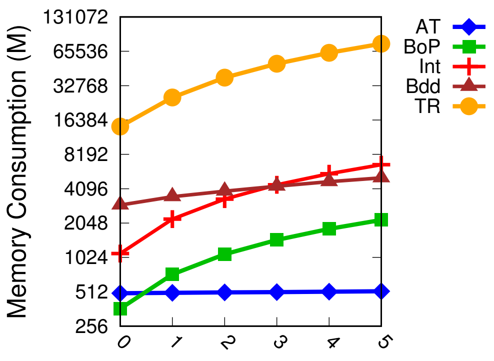
(a) Varying Number of Expressions



(b) Varying Tree Depth



(c) Varying Number of Children



(d) Varying Repeated Times

Figure 10.4.2: Memory Consumption

Fig. 10.2(b) shows the memory use as the average arbitrary expression tree depth increases. It can been seen both `Translation` and `BDD` runs out of memory when the tree depth increases to 4. For `Translation`, the reason is that a single arbitrary expression can be translated into a large number of conjunctive expressions. For `BDD`, the reason is that the complexity of the `BDD` index increases expotentially with the number of predicates in an expression. `A-Tree` needs the least memory when the tree depth is 1. While `BoP` needs the least memory when the tree depth increases to 3.

Fig. 10.2(c) shows the memory use as the average number of child nodes increases. Similar to the tree depth experiments, `Translation` runs out of memory when the child number increases to 6, while `BDD` runs out of memory when the child number increases to 8. The ranking of the left algorithms is `BoP`, `A-Tree` and `Interval ID`. Fig. 10.2(d) shows the memory use as the expression duplicated number increases. It can been seen, only the memory use of `A-Tree` and `BDD` do not obviously increase with the expression repeated times. When the expression duplicated number increases to 2, `A-Tree` consumes the least memory. When the expression repeated times is 5, `A-Tree` reduces the memory use by 76%.

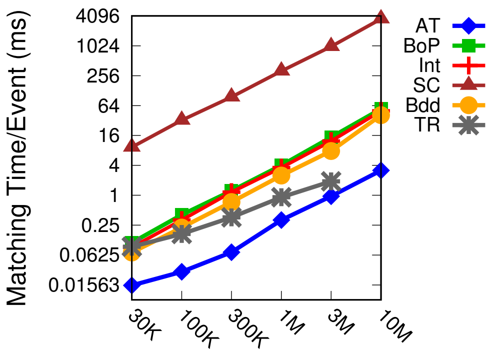
Matching Time

The matching time is among the most important metrics for Boolean expression matching algorithms. In this section, we present extensive experiments under a variety of controlled conditions.

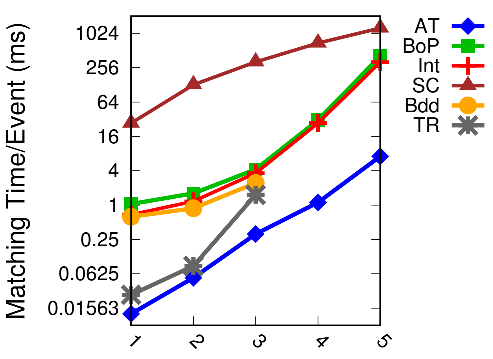
Workload Size: We consider the matching time as we increase the number of subscriptions processed. As illustrated by Fig. 10.3(a), all algorithms scale linearly with respect to the number of expressions. Among them, `A-Tree` always performs the best when the number of expressions increases from 30K to 10M. When there are 10M expressions, `A-Tree` reduces the matching time by up to 92.3% compared to the next-based algorithm `BDD`. The matching performances of `BoP`, `Interval ID` and `BoP` are very similar. `BDD` is slightly better than `BoP` and `Interval ID`. `A-Tree` achieves the best matching performance because the common predicates and expressions are evaluated at most once. Moreover, the unnecessary propagation of evaluation results are cut off by the event matching optimizations.

Expression Tree Depth: As shown in Fig. 10.3(b), when the expression tree depth increases, the matching time of all algorithms increases. However, the matching performance of `Translation` increases faster. Compared with `Translation`, `A-Tree` reduces the matching time by 53% and 81% when the tree depth is 1 and 3, respectively. Moreover, when the tree depth increases to 4, the `Translation` and `BDD` runs out memory. It means the `Translation` and `BDD` method is not suitable for complex arbitrary Boolean expression workloads.

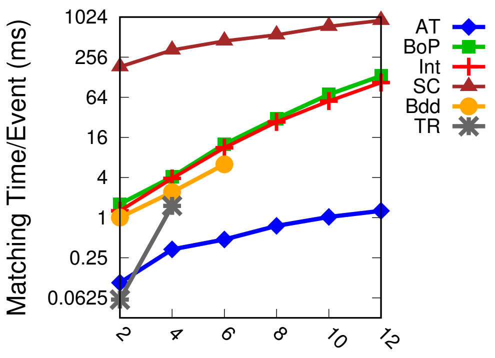
Child Number: The effect of child number is similar to the effect of expression



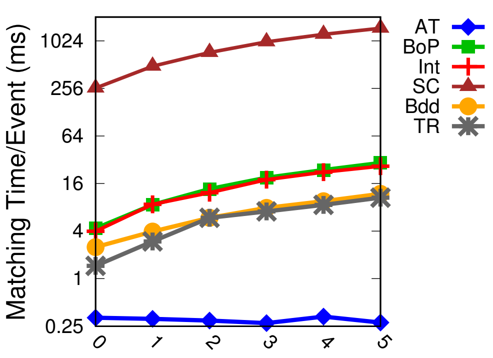
(a) Varying Number of Expressions



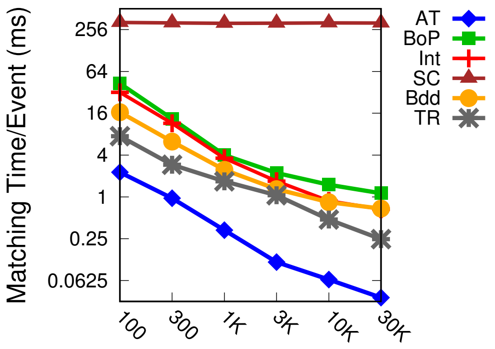
(b) Varying Tree Depth



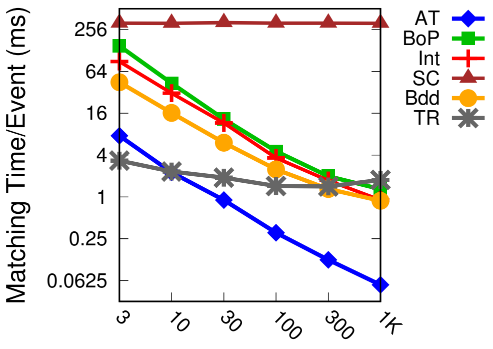
(c) Varying Number of Children



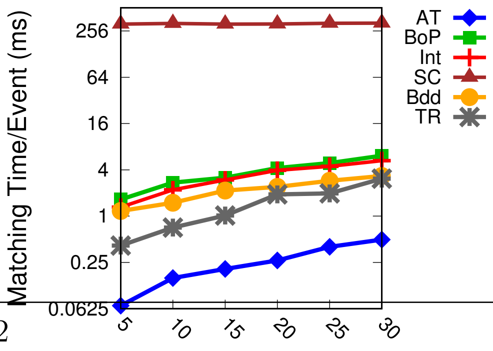
(d) Varying Repeated Times



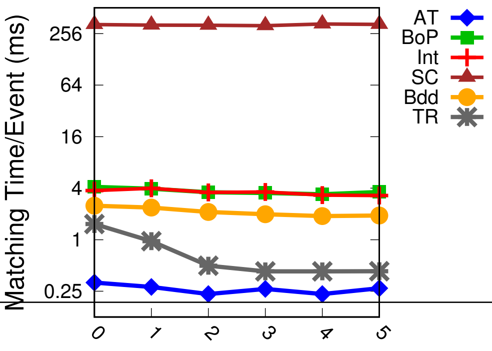
(e) Varying Number of Dimensions



(f) Varying Dimension Cardinality



(g) Varying Event Size



(h) Varying Zipf Distribution

Figure 10.4.3: Event Matching Time

tree depth. As can be seen in Fig. 10.3(c), `Translation` and `BDD` can not scale to expressions with large number of child nodes. The ranking of the left algorithms is `A-Tree`, `Interval ID`, `BoP` and `Scan`.

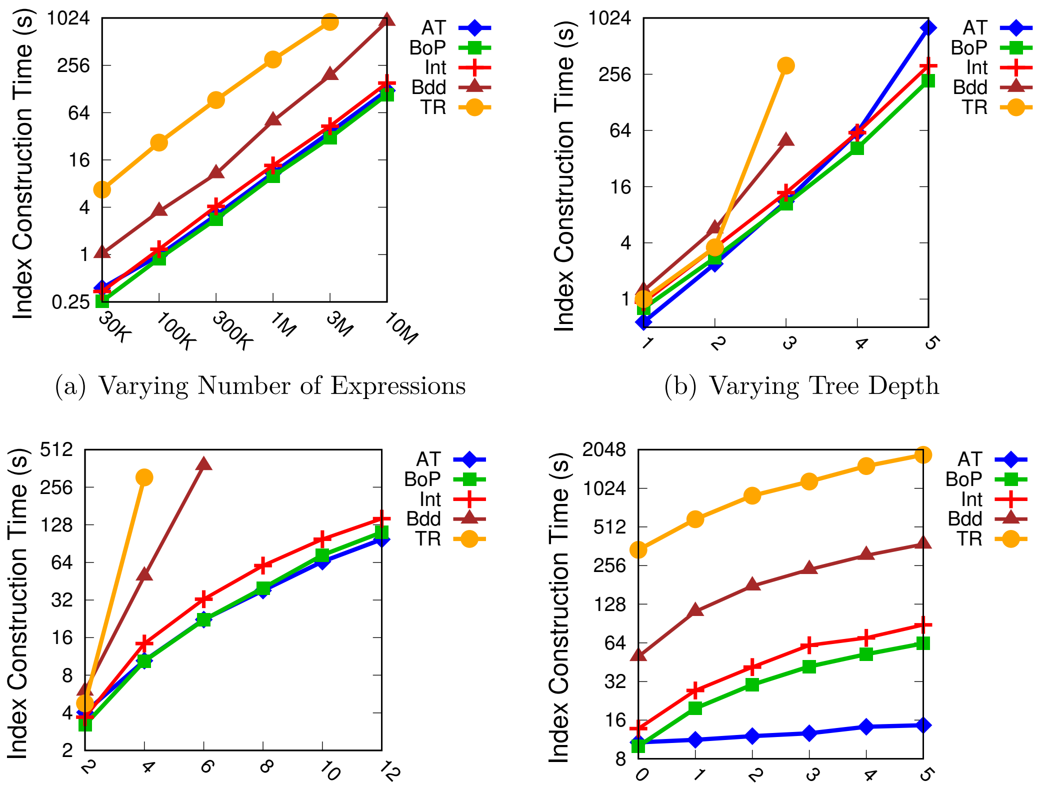
Expression Repeated Times: In this experiment, we study the effect of the expression repeated times. Fig. 10.3(d) shows that the general trend is that the matching time for all algorithms except `A-Tree` increases as the expression duplicated number increases. The matching time of `BDD` also increases. The reason is that the evaluation of events with `BDD` involves the traversal of the entire `BDD` index. We adopted an optimization to the `BDD`-based matching, which was not proposed in the original paper: traverse an expression only one there exists a matching predicate. However, the same expression still need to be evaluated for more than once if it is duplicated. The reason is that `BDD`-based matching is from top-to-bottom, which is different from `A-Tree`-based matching.

Number of Dimensions: As shown in Fig. 10.3(e), when the number of expressions is fixed and the number of dimensions increases, the matching times of all algorithms decrease, except for `Scan`. Compared to `BoP`, `Interval ID`, `BDD`, and `Translation`, the matching time of `A-Tree` decrease quickly. The ranking of all algorithms do not change as the the number of dimensions increases.

Dimension Cardinality: Fig. 10.3(f) shows how the dimension cardinality affects the matching time. It can been seen the matching times of `A-Tree`, `BoP`, `Interval ID` and `BDD` decrease as the number of dimension cardinality increases. The reason is that when the dimension cardinality increases, the number of maching predicates and expressions decreases. When the dimension cardinality increases to 1K, `BoP`, `Interval ID` and `BDD` present very similar matching performance.

Dimension Distribution: In Fig. 10.3(h), the X-axis represents the value of α in $P(r) = \frac{C}{r^{\alpha}}$. It can be seen the matching performance of `A-Tree`, `BoP`, `Interval ID` and `BDD` are not obviously affected by the distribution. The reason is that even though the predicates follow a Zipf distribution, the whole arbitrary Boolean expression will not follow the Zipf distribution. The predicate matching phase can be affected, but the major expression matching phase are not affacted.

Event Size: When the average number of attribute-value pairs of an event increases, there are more matching predicates and expressions. As shown in Fig. 10.3(g), all algorithms except `Scan` present higher matching times as the event size increases. The ranking of these algorithms is `A-Tree`, `Translation`, `BDD`, `Interval ID` and `Scan`. The ranking does not change as the event size increases.



(a) Varying Number of Expressions

(b) Varying Tree Depth

(c) Varying Number of Children

(d) Varying Repeated Times

Figure 10.4.4: Index Construction Time

Index Construction Time

The index construction times of different algorithms are affected by workload size, tree depth, child number and expression duplicated number. As can been seen in Fig. 10.4(a), the index construction times of all the algorithms increase with the expression number. Among them, the index construction times of `A-Tree`, `BoP` and `Interval ID` are very similar. The index construction times of `Translation` and `BDD` are higher. The same phenonmenon is observed in Fig. 10.4(b) and Fig. 10.4(c). In these three groups of experiments, the expressions are all distinct. When we increase the expression duplicated number, as we can see in Fig. 10.4(d), the index construction of `A-Tree` becomes faster than the other algorithms. When the expression repeated times increases to 5, compared to `BoP`, `A-Tree` reduces the index construction time by 77%.

10.4.4 Experiments on the Ads Dataset

By transforming advertisements into arbitrary Boolean expressions, in this workload, we obtain 1,392,196 expressions. The number of predicates in an expression ranges from 1 to 56. Each event contains 20 attribute-value pairs. The number of dimensions is 122.

Table 10.4.2: Experiments on the Ads Dataset

Index	Matching	Construct	Memory
SCAN	994.1 ms	-	-
Translation	7.1 ms	588.2 s	25,822 MB
BDD	7.8 ms	238.3 s	4,310 MB
Interval ID	17.9 ms	61.2 s	4,432 MB
Dewey ID	19.5 ms	68.8 s	4,640 MB
BoP	18.9 ms	41.9 s	1,456 MB
ATree	0.27 ms	12.3 s	509 MB

As shown in Table 10.4.2, under this workload, `A-Tree` achieves the best matching performance, followed by `Translation` and `BDD`. `A-Tree` also achives the best index construction performance and memory foot print. The reason is that in this workload, lots of predicates, subexpressions and expressions are duplicated for many times. The matching performance of `BoP` is similar with `Dewey ID` and `Interval ID`. But the index construction performance and memory use of `BoP` is better than `Dewey ID` and `Interval ID`. `Interval ID` performes slightly better than `Dewey ID`. Compare with the next-best algorithms, `A-Tree` reduces the matching time, index constrution time and memory use by 96%, 71%, and 65%, respectively. This experimental result is roughly consistent with the experimental results on synthetic workloads.

Chapter 11

Conclusions

The existing content-based publish/subscribe routing and Boolean expression matching solutions present limitations on flexibility, performance, expressiveness and applicability. To overcome these limitations, we propose new efficient routing and Boolean expression matching algorithms. Our proposed new routing algorithms include D-DBR and MERC. For conjunctive Boolean expression matching, we proposed `PSTBloom` and `PSTHash` algorithm. While for arbitrary Boolean expression matching, we propose the `A-Tree`-based matching.

D-DBR exhibits low processing overhead by reducing the number of event matching computations and exhibits higher flexibility by decoupling event matching from event routing at the cost of requiring global topology knowledge and destination list information in each message. D-DBR is well-suited for small-scale networks, such as one may find at the enterprise- and data center-level. For larger scale networks, such as in inter-data center and wide area application scenarios, MERC is more suitable, because each broker only needs the knowledge of a small part of the overall topology, and, consequently, the destination list overhead is kept in check. Moreover, brokers in MERC are organized as a structured and hierarchical network of clusters offering better system management and maintenance opportunities. Our experimental results show that D-DBR and MERC perform well across a range of evaluation scenarios with significant performance improvements, especially, when there are a large number of subscriptions. A limitation of MERC is that edge brokers may constitute single points of failure, thus, may require a fail-over mechanism, and may benefit from running on more powerful machines.

`PS-Tree` is a novel data structure we proposed to index predicates. `PS-Tree` can efficiently constructs a many-to-many relation between predicate spaces and subscriptions. Through `PS-Tree`, the problem of predicate matching is transformed into the problem of locating the predicate space to which an attribute-value pair belongs. `PS-Tree` offers excellent query performance and good expressiveness.

Based on `PS-Tree`, we first propose the `PSTBloom` algorithm. `PSTBloom` selects a predicate with the high selectivity as the access predicate for each subscription. Then, the subscription is associated with its corresponding leaf nodes. Through `PS-Tree`, `PSTBloom` can efficiently filter out all the subscriptions whose access predicates do not match with a received event. Then, Bloom filter signatures are used to further filter out most unmatching subscriptions. `PSTBloom` is efficient at handling many workload distributions, especially high-dimensional workloads. `PSTBloom` achieves fast index construction and requires a small amount of memory. However, `PSTBloom` and other algorithms do not meet the challenge presented by dense workloads. To overcome this limitation, we further propose the `PSTHash` algorithm. `PSTHash` selects more than one access predicate for each subscription and constructs a many-to-many relation between multi-dimensional predicate spaces and subscriptions. Only when an event matches with all access predicates of a subscription is the subscription identified as a candidate subscription. Compared with `PSTBloom` and other existing algorithms, `PSTHash` achieves the best matching performance for dense workloads.

To efficiently index arbitrary Boolean expressions, we proposed a novel multiroot tree data structure called `A-Tree`. In `A-Tree`, the same predicate is uniquely represented by a leaf node, and the same subexpression is uniquely represented by an inner node. The same predicates and subexpressions of different arbitrary Boolean expressions are shared. In this way, `A-Tree` achieves a good memory footprint. Moreover, during event matching, it is guaranteed that the same predicate and subexpression are evaluated once for each event.

`A-Tree` not only supports dynamic expression reorganization based on the current index structure but also supports dynamic index self-adjustment based on the incoming expressions. In this way, the `A-Tree` index remains optimized regardless of the incoming order of different expressions. Moreover, the `A-Tree` index supports the removal of expired expressions and can be self-adjusted after the removal. In contrast to the existing `Dewey ID` [30] and `Interval ID` [30] methods, `A-Tree` works in a online mode, which means that the `A-Tree` index does not need to be built in advance.

Based on the `A-Tree` index, we propose algorithms to match events against arbitrary Boolean expressions from bottom to top. Different logical operators are supported. The matching process consists of *evaluation of result* and *propagation of result*. To remove the cost of propagating the *false* evaluation result, we proposed the *zero suppression filter* optimization. To avoid the unnecessary propagation of evaluation results to a node with the *and* logical operator, we propose the *propagation on demand* optimization.

We conducted extensive experiments using both synthetic and real-world datasets. The results show that our algorithms outperform state-of-the-art approaches for routing and matching performance under different types of worloads.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] http://sigmod.kaust.edu.sa, Jan. 2013.

[2] M. Adler, Z. Ge, J. F. Kurose, D. Towsley, and S. Zabele. Channelization problem in large scale data dissemination. In *ICNP'01*, pages 100–109. IEEE, 2001.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *ACM PODC*, pages 53–61, 1999.

[4] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *IEEE FOCS*, pages 560–569, 1996.

[5] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS'99*, pages 262–272. IEEE, 1999.

[6] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Springer ICDT*, pages 217–235, 1999.

[7] S. Bittner and A. Hinze. The arbitrary boolean publish/subscribe model: making the case. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 226–237. ACM, 2007.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[9] L.-F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *Hot Topics in Operating Systems Workshop*, pages 87–92. IEEE, 2001.

[10] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *IEEE ICSE*, pages 443–452, 2001.

[11] C. Canas, K. Zhang, B. Kemme, J. Kienzle, and H.-A. Jacobsen. Publish/Subscribe network designs for multiplayer games. In *ACM Middleware*, 2014.

[12] F. Cao and J. P. Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Middleware*, pages 292–313. Springer-Verlag New York, Inc., 2005.

[13] A. Carzaniga, C. Hall, and A. L. Wolf. Practical high-throughput content-based routing using unicast state and probabilistic encodings. *Faculty of Informatics, University of Lugano, Tech. Rep*, 6, 2009.

[14] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3):332–383, 2001.

[15] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. In *PVLDB*, pages 254–262, 2000.

[16] D. D. Clark. Policy routing in internet protocols. *Policy*, 1989.

[17] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT press, 1999.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, 2009.

[19] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worm epidemics. *ACM TOCS*, 26(4):9, 2008.

[20] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In *DEBS workshop*, pages 1–8. ACM, 2003.

[21] P. Costa and G. P. Picco. Semi-probabilistic content-based publish-subscribe. In *ICDCS'05*, pages 575–585. IEEE, 2005.

[22] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE TSE*, 27(9):827–850, 2001.

[23] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[24] R. De La Briandais. File searching using variable length keys. In *Western joint computer conference*, pages 295–298. ACM, 1959.

[25] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM TOCS*, 8(2):85–110, 1990.

[26] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM TODS*, pages 467–516, 2003.

[27] 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 *ACM SIGMOD*, pages 115–126, 2001.

[28] W. Fan, Y. Liu, and B. Tang. Gem: An analytic geometrical approach to fast event matching for multi-dimensional content-based publish/subscribe services. In *IEEE INFOCOM*, pages 1–9, 2016.

[29] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovskii. The PADRES distributed publish/subscribe system. pages 12–30, July 2005.

[30] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *ACM SIGMOD*, pages 3–14, 2010.

[31] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, 1984.

[32] E. N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures*, pages 153–164. Springer, 1991.

[33] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *IEEE ICDE*, pages 266–275, 1999.

[34] S. Hou and H.-A. Jacobsen. Predicate-based filtering of xpath expressions. In *IEEE ICDE*, page 53, 2006.

[35] H. Jafarpour, S. Mehrotra, and N. Venkatasubramanian. Dynamic load balancing for cluster-based publish/subscribe system. In *SAINT'09*, pages 57–63. IEEE, 2009.

[36] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS'05*, pages 447–457. IEEE, 2005.

[37] G. Li, S. Hou, and H.-A. Jacobsen. Routing of xml and xpath queries in data dissemination networks. In *IEEE ICDCS*, pages 627–638, 2008.

[38] G. Li, V. Muthusamy, and H.-A. Jacobsen. Adaptive content-based routing in general overlay topologies. In *Middleware*, pages 1–21. Springer-Verlag New York, Inc., 2008.

[39] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB*, 4(1):2, 2010.

[40] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB*, page 2, 2010.

[41] 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, 1994.

[42] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *IMC'05*, pages 3–3. USENIX Association, 2005.

[43] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *PVLDB*, 1(1):451–462, 2008.

[44] A. Margara and G. Cugola. High-performance publish-subscribe matching using parallel hardware. *IEEE TPDS*, pages 126–135, 2014.

[45] J. Martins and S. Duarte. Routing algorithms for content-based publish/subscribe systems. *IEEE CST*, 12(1):39–58, 2010.

[46] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, pages 514–534, 1968.

[47] G. Muhl, L. Fiege, F. C. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *MASCOTS'02*, pages 167–176. IEEE, 2002.

[48] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *Network, IEEE*, 8(3):26–41, 1994.

[49] U. of Toronto SciNet Consortium. http://www.scinet.utoronto.ca.

[50] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In *Middleware*, pages 185–207. Springer-Verlag New York, Inc., 2000.

[51] S. Qian, J. Cao, Y. Zhu, and M. Li. Rein: A fast event matching approach for content-based publish/subscribe systems. In *IEEE INFOCOM*, pages 2058–2066, 2014.

[52] S. Qian, J. Cao, Y. Zhu, M. Li, and J. Wang. H-tree: An efficient index structurefor event matching in content-basedpublish/subscribe systems. *IEEE TPDS*, pages 1622–1632, 2015.

[53] C. Raiciu, D. S. Rosenblum, and M. Handley. Revisiting content-based publish/subscribe. In *ICDCS Workshops 2006*, pages 19–19. IEEE, 2006.

[54] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *ICDCS'02*, pages 133–142. IEEE, 2002.

[55] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. Gpx-matcher: a generic boolean predicate-based xpath expression matcher. In *ACM EDBT*, pages 45–56, 2011.

[56] M. Sadoghi and H.-A. Jacobsen. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *ACM SIGMOD*, pages 637–648, 2011.

[57] M. Sadoghi and H.-A. Jacobsen. Analysis and optimization for boolean expression indexing. *ACM TODS*, page 8, 2013.

[58] M. Sadoghi and H.-A. Jacobsen. Adaptive parallel compressed event matching. In *IEEE ICDE*, pages 364–375, 2014.

[59] M. Sadoghi, M. Jergler, H.-A. Jacobsen, R. Vaculin, and R. Hull. Safe distribution and parallel execution of data-centric workflows over the publish/subscribe paradigm. *IEEE TKDE*, 2015.

[60] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS'05*, pages 320–326, 2005.

[61] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 21(2):164–206, 2003.

[62] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, IEEE Annual Symposium on*, pages 1–11, 1973.

[63] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. Indexing boolean expressions. *PVLDB*, 2(1):37–48, 2009.

[64] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Transactions on Database Systems (TODS)*, 19(2):332–364, 1994.

[65] Y. Yoon, V. Muthusamy, and H.-A. Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *ICDCS'11*, pages 800–811. IEEE, 2011.

[66] Y. Yoon, N. Robinson, V. Muthusamy, S. McIlraith, and H.-A. Jacobsen. Towards planning the transformation of distributed messaging middleware. In *IEEE ICDCS*, June 2015.

[67] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM'96*, volume 2, pages 594–602. IEEE, 1996.

[68] D. Zhang, C.-Y. Chan, and K.-L. Tan. An efficient publish/subscribe index for e-commerce databases. *PVLDB*, 7(8):613–624, 2014.

[69] Y. Zhao and J. Wu. Towards approximate event processing in a large-scale content-based network. In *IEEE ICDCS*, pages 790–799, 2011.