QpIndex is an event matching algorithm which is designed to deal with the situation where there are too many event types. So, the total number of attributes appeared in all event types is very large, maybe in the scale of tens or hundreds of thousand. In other words, the data space composed of all attributes is really high-dimensional. One question concerning QpIndex is that, for different event types , does the matching performance of QpIndex behave stably?

An Efficient Publish/Subscribe Index for E-Commerce Databases

The features of a subscription:
1. the number of predicates
2. the matching probability
3. the operators of predicates
4. the pivot attribute
5. the set of attributes

Dongxiang Zhang   Chee-Yong Chan   Kian-Lee Tan

Department of Computer Science
School of Computing, National University of Singapore

{zhangdo,chancy,tankl}@comp.nus.edu.sg

verb:
prompt, show propose,
provide, conduct, retrieve,

ABSTRACT

1.
2
3
4.
5.
6
7.

Many of today's publish/subscribe (pub/sub) systems have been designed to cope with a large *volume* of subscriptions and high event arrival rate (*velocity*). However, in many novel applications (such as e-commerce), there is an increasing *variety* of items, each with different attributes. This leads to a very high-dimensional and sparse database that existing pub/sub systems can no longer support effectively. In this paper, we propose an efficient in-memory index that is scalable to the volume and update of subscriptions, the arrival rate of events and the variety of subscribable attributes. The index is also extensible to support complex scenarios such as prefix/suffix filtering and regular expression matching. We conduct extensive experiments on synthetic datasets and two real datasets (AOL query log and Ebay products). The results demonstrate the superiority of our index over state-of-the-art methods: our index incurs orders of magnitude less index construction time, consumes a small amount of memory and performs event matching efficiently.

1. INTRODUCTION

P/S
1.
2
3
4.

Publish/subscribe, or pub/sub for brevity, has been well-studied in the last two decades [3, 6, 9, 16, 20, 22, 26], with deployment in a variety of applications including online advertising [16], stock market [6] and social media monitoring [9]. A pub/sub system contains two types of roles, information provider and information consumer. The information provider publishes information in the form of *events*. The information consumer subscribes interesting events in the form of *boolean expression*. These two roles can be interconnected either via a simple client/server model [12,20,22,26] or over a network of brokers routing events in a distributed paradigm [3, 7, 14]. The system has to ensure a timely delivery of matching events to the subscribers.

Existing pub/sub systems, however, are designed with two factors in mind: a large volume of subscriptions and a high event arrival rate. However, pub/sub systems are increasingly being adopted in e-commerce applications with a wide variety of items, each with different attributes. The database can be modelled as a sparse and high dimensional table, and an event is a tuple in this high dimen-

sional table. To explain this, we use the Amazon product database as a working scenario.

EXAMPLE 1. *We can model the Amazon product database as an information provider and customers as information consumers. Since Amazon has launched a Wish List to collect customer intention in product purchasing, we can extend this function for a customer to specify the conditions under which (s)he will purchase the item. The event would be either the launch of a new product or a discounted product on sale. An example of a subscription would be in the form of a boolean expression. e.g., (model=iphone5s ∧ color=silver ∧ price≤580). A product is represented by a list of attribute-value pairs. e.g., (model=iphone5s ∧ color=silver ∧ storage capacity=16GB ∧ price=550 ∧ contract=no). The customer will be notified whenever there is a product in the database satisfying all the specified constraints. However, there are more than 200 million[1] items hosted in the Amazon product database. Moreover, there is a wide variety of products and they may have very different attributes. The product database can be modelled as a very wide and sparse table. The pub/sub system has to be scalable to the number of columns as new products are continually being inserted.* □

In the following, we summarize several applications with high dimensions of attributes for which a pub/sub system may add value:

- **Electronic Commerce.** Online electronic commerce companies like Amazon, Ebay and Taobao[2] have large number of products in many different categories. Information extraction techniques [13] can be adopted to extract attribute-value pairs from the unstructured web page to support faceted search [8] and pub/sub. For example, Taobao, the largest online shopping website in China with more than 800 million products[3], has integrated faceted search in the system to facilitate customers filtering from a great number of search results. Similarly, these systems can allow customers to subscribe to products they are interested in and receive a timely notification when a match occurs. Such a pub/sub model may emerge as a new business intelligence model to improve online shopping experience.

- **Groupon and Deal Websites.** Groupon and other deal websites have the pub/sub gene in nature. Instead of going through every deal sent to the registered email address, it would be more convenient for users to only subscribe the deals they

---

[1]This number is acquired by submitting an arbitrary keyword query like "-asdsddafd" to the Amazon product search engine.
[2]http://www.taobao.com
[3]http://www.alexa.com/siteinfo/taobao.com

The difference between search and match:
1. definition
   1) Given a condition, search is the process of find all condition-satisfying records from a database.
   2) Match is the duality of search. Given a record, match aims at obtaining all conditions that are evaluated true for the given record.

2. indexed objects
   1) Search focuses on proposing new data structures that are used to index records.
   2) Match is performed based on the data structures indexed over conditions.

3. time
   1) Given a condition and a set of records, search is carried out currently, not extending to future.
   2) Based on a set of pre-defined conditions, match is towards future, waiting for coming records to identify conditions that are evaluated true.

are interested in using boolean expressions. Similar to the product database, the deals also show great variety in terms of the subscribable attributes.

- **Google Base**[4]. Google Base, which later becomes Google Merchant Center, allows users to upload any structured or unstructured product feeds in various file format. A real-time pub/sub system on top of Google Base would be of utmost importance to business dealers, e.g., to monitor the potential competitors within an area.

- **Web Tables and Semantic RDF Database**. In recent years, harvesting knowledge from the web [11, 24, 25, 28] has attracted more and more attention. For example, Google's Freebase [1] has collected and published more than 39 million real world entities, with more than $140,000$ attributes. These structured or semi-structured harvested results are invaluable. Agents can subscribe to such information for decision making, just analogous to brokers subscribing to stock price.

The Knowledge Graph API is a replacement to the Freebase API that is planned to shut down in early 2016.

To understand how existing systems cope with boolean expression matching when events come from a sparse and high dimensional table, we conducted an experimental study using two recently proposed pub/sub indexes [5]: k-index [26] and BE-Tree [20, 22]. k-index partitions subscriptions into inverted lists while BE-Tree uses hierarchical clustering to organize the data. Although in [20,22], k-index was reported to be inferior to BE-Tree in datasets with hundreds of attributes, we have new findings when we further increase the dimension space. Figure 1 shows the index construction time and event matching time for a uniformly distributed dataset when the number of attributes grows from 20K to 60K. The results shed interesting insights that were not previously reported: *The inverted index solution not only significantly outperforms the BE-Tree in terms of index construction time, but also demonstrates better scalability in terms of event matching time!* However, due to its ineffective partitioning mechanism, k-index consumes more memory than BE-Tree and supports only a subset of the operators that BE-Tree can handle.
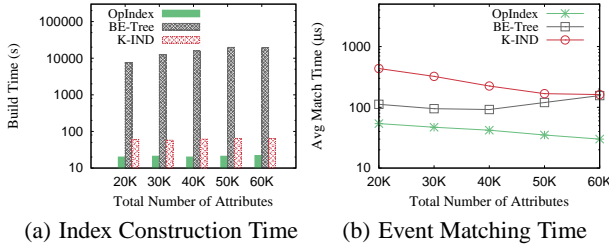

Figure 1: Performance w.r.t. increasing number of attributes

Our findings prompted us to design a more efficient, expressive and compact index, which we name OpIndex, to support pub/sub for e-commerce data that exhibits a large number of dimensions. OpIndex adopts a two-level index structure and organize the subscriptions using inverted lists. In the first level, we select a pivot attribute for each subscription, and subscriptions with the same pivot attribute are grouped together. In the second level, subscriptions are further partitioned based on their predicate operators. In

[4] http://base.google.com
[5] The implementation of the two indexes was kindly provided by the authors of BE-Tree.

this manner, the predicates with the same operator are clustered so that we can design specific index to support various operators and to enhance the subscription expressiveness. The effectiveness is demonstrated in Figure 1: OpIndex achieves better event matching performance with much smaller construction cost.

In summary, the contributions of this paper include

1. We show that pub/sub applications in e-commerce are becoming increasingly important. Furthermore, we identify a gap in existing pub/sub systems - they cannot cope effectively for applications with very high dimensional table.

2. We propose a novel index structure, OpIndex, which is scalable with respect to the volume, velocity and variety of the data. In particular, OpIndex is more efficient, has low memory requirement and maintenance cost, and can be easily extended to support more expressive subscriptions (i.e., can support prefix/suffix and regular expression matches).

3. We provide a comprehensive complexity analysis of our OpIndex in terms of the memory overhead, and data insertion and query processing cost.

4. We conduct extensive experiments on synthetic and real datasets (AOL query log and Ebay Products). The results show that OpIndex is superior in terms of index construction time, memory cost and query processing time.

The remaining of the paper is organized as follows. We present our boolean expression model and problem definition in Section 2. In Section 3, we review existing pub/sub works. In Section 4, we propose OpIndex and analyze the memory consumption and insertion cost. Event matching algorithm as well as query processing complexity analysis are presented in Section 5. We discuss extensions of our index to support complex operators in Section 6. Extensive experiment results are reported in Section 7. Section 8 concludes the paper.

2. BOOLEAN EXPRESSION MODEL

In pub/sub systems, a subscription is represented as a boolean expression which provides flexibility for users to specify their interests. In this section, we present the boolean expression model as well as the matching semantics.

2.1 Predicate

The most basic unit in a boolean expression model is a predicate. A predicate is determined by three elements: an attribute A, an operator f_{op} and an operand $\bar{o}$. A predicate accepts an input value x and outputs a boolean value indicating whether or not the operator constraint is satisfied:

$$P^{(A, f_{op}, \bar{o})}(x) \rightarrow \{0, 1\}$$

In this paper, we adopt a data model that is more general and expressive than that used in the state-of-the-art index methods [20,22, 26]. Besides supporting numerical, categorical, and string attribute domains with the standard relational operators ($<, \leq, =, \neq, >, \geq$), set operators ($\in_s, \notin_s$), and interval operators ($\in_i, \notin_i$) [6], our model can also support complex operators such as the prefix, suffix, and regular expression matching operators for the string domain; we discuss how complex operators are supported by our approach in Section 6.2.

[6] We make a distinction between the operators $\in_s$ (representing SQL's IN operator) and $\in_i$ (representing SQL's BETWEEN operator).

2.2 Boolean Expression

A boolean expression is a combination of predicates in either Conjunctive Normal Form (CNF) or Disjunctive Normal Form (DNF). To simplify the presentation, we assume that a boolean expression is represented in DNF with a single clause (i.e., simply a conjunction of predicates). We will discuss how to handle more general forms of boolean expressions in Section 6. Thereafter, a subscription S is defined over n predicates as follows:

$$S : P_1^{A,f_{op},\bar{o}}(x) \wedge P_2^{A,f_{op},\bar{o}}(x) \wedge \ldots \wedge P_n^{A,f_{op},\bar{o}}(x)$$

We refer to the size of a subscription S, denoted by $|S|$, as the number of predicates in S.

Table 1 shows a small collection of six subscriptions that we will be using as our running example in the rest of this paper.

Table 1: Example Subscriptions

S_1	$A = 2 \wedge B \in_s \{3,6,9\}$
S_2	$A \leq 8 \wedge C \geq 2$
S_3	$C = 6 \wedge B \leq 4 \wedge E \in_i [3,12]$
S_4	$A = 2$
S_5	$D \geq 12 \wedge E \leq 9$
S_6	$B \in_s \{3,6\} \wedge C \leq 4 \wedge D \geq 10 \wedge E \leq 7$

2.3 Event

An information publisher publishes an event in the form of a collection of attribute-value pairs. We model an event as a conjunction of equality predicates.

$$E : (A_{i_1} = \bar{o}_1) \wedge (A_{i_2} = \bar{o}_2) \wedge \ldots \wedge (A_{i_m} = \bar{o}_m)$$

> number of

We refer to the size of an event, denoted by m or $|E|$, as the number predicates in the event E. For example, an event about iPhone may look like the following:

$$(model = iphone5 \wedge color = white \wedge price = 800 \wedge size = 16GB)$$

2.4 Boolean Expression Match

Given a subscription S and an event E, S matches E if it satisfies two requirements, namely, attribute match and value match.

DEFINITION 1. *Attribute Match*
There is an attribute match between a subscription S and an event E if for any attribute occurring in S, it also appears in E.

We use $S \sim_A E$ to denote an attribute match. For example, $(A \leq 3 \wedge B = 2)$ is not an attribute match to $A = 2$.

DEFINITION 2. *Value Match*
There is a value match between a subscription S and an event E if for any attribute A occurring in S and E, we have $P^{A,f_{op},\bar{o}}(\bar{o}_i) = 1$, where $P^{A,f_{op},\bar{o}} \in S$ and $(A = \bar{o}_i) \in E$.

We use $S \sim_V E$ to denote a value match. Now we can define the boolean expression match.

> [1, 4] -> [1, 5]

DEFINITION 3. *Boolean Expression Match*
A subscription S is said to match an event E, denoted by $S \sim E$, if $S \sim_A E$ and $S \sim_V E$.

Given a subscription collection $\mathbb{S}$ and a published event E, our goal is to find all the subscriptions $S \in \mathbb{S}$ such that $S \sim E$.

> 1. Attribute Match is only applied to the case that a boolean expression is simply a conjunction of predicates (represented in DNF with one clause). How to relax the definition of Attribute Match to adapt to the general DNF and CNF?
>
> 2. According to the definitions of Attribute Match (AM) and Value Match (VM), VM covers AM. In other words, if there is a VM between S and E, which necessarily deduces that there is a AM between S and E.

3. RELATED WORK

Pub/sub systems have been extensively studied for over a decade; and there has been a lot of focus on indexing support to efficiently identify matching subscriptions (e.g., [12,26,29]). The basic idea is to partition the subscription database into subsets of predicates using some hashing scheme and organize each predicate subset using the inverted list data structure. For each predicate p in an incoming event, appropriate inverted list indexes are searched to identify subscription predicates that match p, and a counting algorithm is used to determine matching subscriptions for an event.

The k-index [26] is the state-of-the-art approach based on inverted-list index. The subscription predicates are partitioned into subsets using a three-level partitioning scheme: the subcriptions are first partitioned based on their size, and the predicates in a subscription are further partitioned based on the predicate's attribute and value. For example, for a predicate A=1 in a subscription of size 3, the predicate will be partitioned into the subset associated with the partition key $(3, A, 1)$. By using the subscription size as the primary partitioning key, the k-index is able to prune away inverted-list searches for subscriptions with size larger than that of the event.

A drawback of k-index is that a range predicate in a subscription needs to be rewritten into a disjunction of equality predicates, which increases the size of the k-index with many inverted-list entries for a single subscription predicate. As an example, Figure 2 illustrates the k-index entries for our running example subscription database in Table 1. Note that for the predicate $A \leq 8$ in subscription S_2, assuming the domain of A is $\{1, 2, \ldots\}$, the predicate is rewritten as $(A = 1) \vee \ldots \vee (A = 8)$ which requires eight entries $(2, A, i)$, $i \in [1, 8]$, to be created in the k-index.

n	(A,v)	List	n	(A,v)	List	n	(A,v)	List
1	(A,2)	S_4		(D,…)	S_5		(A,1)	S_2
2	(A,1)	S_2	2	(E,1)	S_5		(B,3)	S_6
	(A,2)	S_1, S_2		(E,…)	S_5		(B,6)	S_6
	(A,…)	S_2		(E,9)	S_5		(C,1)	S_6
	(A,8)	S_2		(B,1)	S_3	4	(C,…)	S_6
	(B,3)	S_1		(B,…)	S_3		(C,4)	S_6
	(B,6)	S_1		(B,4)	S_3		(D,10)	S_6
	(B,9)	S_1	3	(C,6)	S_3		(D,…)	S_6
	(C,2)	S_2		(E,3)	S_3		(E,1)	S_6
	(C,…)	S_2		(E,…)	S_3		(E,…)	S_6
	(D,12)	S_5		(E,12)	S_3		(E,7)	S_6

Figure 2: k-Index for subscriptions in Table 1 (n = subscription size, A = predicate attribute, v = predicate value)

More recently, a new index method, the BE-Tree, was shown to outperform the k-index [20, 22]. Unlike the k-index, the BE-Tree uses a two-phase space-cutting technique and organizes the subscriptions in a hierarchical index. The subscriptions are repeatedly partitioned by attribute followed by a value space partitioning. Figure 3 shows an example of BE-Tree indexing the subscriptions in Table 1. The p-directory stores the attributes selected for partitioning. In this example, the p-directory contains two attributes A and B, associated with two different p-nodes. If an event does not contain attribute A, all the subscriptions in the subtree of p-node A can be pruned. Then, the subscriptions are partitioned by the associated attribute value. The value space is organized in a hierarchy of intervals with different length. Each subscription is attached to the smallest interval that can cover the predicate. For example, S_3 contains a predicate B≤4 and is inserted into p-node=B with value interval $[1,4]$. Given an event $B = 5$, all the subscriptions attached to intervals that are not stabbed by $B = 5$ can be pruned. S_5 is inserted into another branch because it does not contain attribute A or B. In the leaf nodes, inverted lists of bitmaps are maintained

for efficient evaluation of a predicate. The key of the list is the attribute-value pair, the same as that in *k*-index. As the number of attributes increases, BE-Tree generates more *p-nodes* which incurs higher construction, optimization and access cost. Moreover, both the *k*-index and BE-Tree support only the standard basic predicate operators but not more advanced matching operators such as prefix/suffix and regular expression matching operators. In contrast, our approach can support such complex matching operators (to be elaborated in Section 6.2).
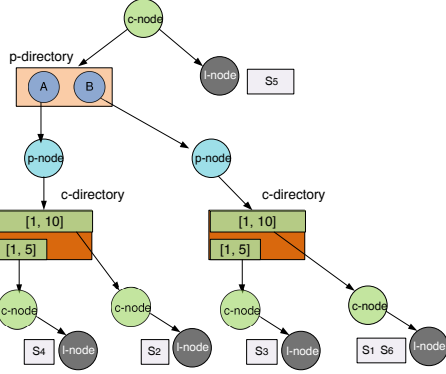

Figure 3: BE-Tree for subscriptions in Table 1

Index methods to support ranked pub/sub matching, where only the top-k matching subscriptions are returned, have also been proposed including score-optimal R-tree [16], *k*-index [26], and a modified variant of BE-Tree [21]. Our index focuses on efficient filtering and we plan to support top-*k* pub/sub matching in future. Other directions in pub/sub subscription matching include support for XPath-based subscriptions (e.g., [10,19,27]), stateful event matching (e.g., [5,6,9,15,16]) where subscriptions may span multiple events and efficient routing solutions in a content network [5,6,14,18,23]. Since we are interested in the problem of efficient event matching without considering network communication, the pub/sub in a content network is beyond the scope of this paper. For more information, readers can refer to the survey in [2].

4. INDEX STRUCTURE

In this section, we present our new index method, named *OpIndex*, to efficiently retrieve matching subscriptions for a given input event. OpIndex uses a novel, two-level partitioning scheme to organize the subscription predicates into disjoint subsets each of which is independently and efficiently indexed to minimize the number of candidate subscriptions accessed for event matching. In this way, our index design provides a highly efficient and extensible approach for subscription matching which can support complex predicate matching operators beyond the standard operators supported in current state-of-the-art methods [20,22,26].

In OpIndex, each subscription S in the database is associated with a judiciously selected attribute termed its *pivot attribute*, denoted by δ_S, which is one of the attributes contained in S. The subscriptions are partitioned using a two-level partitioning scheme as follows: first, the subscriptions in the database are partitioned based on their pivot attributes into subscription lists, and the predicates in each subscription list are then further partitioned based on the predicate operator into predicate lists. Each predicate list is then independently indexed using an efficient method that is appropriate for the predicate operator. Given an input event, appropriate predicate lists are accessed via their corresponding indexes; and a

counting-based approach is used to identify the matching subscriptions.

For convenience, Table 2 summarizes the key notations used in this paper.

Table 2: Notation Table

$P^{A_i, f_{op}, \bar{o}}$	A predicate defined over attribute A with operator f_{op} and operand $\bar{o}$
S	A subscription
E	An event
δ_S	The pivot attribute of subscription S
d	The total number of distinct attributes (or dimensions)
N	The number of subscriptions in the subscription database
Γ	The number of predicates in a subscription
m	The number of attributes in an event
σ	The cardinality of an attribute domain
w	The number of bits in a segment signature
b	The number of segment in an inverted list

4.1 Level 1: Subscription Partitioning

In the first level of partitioning, the subscriptions in the database $\mathbb{S}$ are partitioned into disjoint subscription lists based on the pivot attribute of each subscription as follows:

$$\mathbb{S} = L_{\langle A_1 \rangle} \cup L_{\langle A_2 \rangle} \cup \ldots \cup L_{\langle A_d \rangle}$$
$$L_{\langle A_i \rangle} = \{S | S \in \mathbb{S} \wedge \delta_S = A_i\}$$

Here, each $L_{\langle A_i \rangle}$ denote the subscription list associated with the pivot attribute A_i.

From the definition of attribute match, we know that if a subscription S matches an event E, then all the attributes in S have to appear in E. Clearly, if S contains an attribute A_i that does not occur in E, then S will definitely not match E. Thus, given E, we only need to consider the subscriptions whose pivot attribute occurs in E as stated in the following result.

LEMMA 1. *Given an event E, the candidate matching subscriptions for E are contained in the subscription lists* $\{L_{\langle A_i \rangle} | A_i \in E\}$.

To minimize the number of candidate matching subscriptions to be accessed for an input stream of events $\mathbb{E}$, the problem of selecting the pivot attribute for a subscription S is modeled as a visibility minimization problem [17]. Let $\Delta(A)$ denote the frequency of an attribute A in an event stream $\mathbb{E}$. We choose attribute A to be the pivot attribute for a subscription S if A appears the least frequently in $\mathbb{E}$ among all the attributes in S; i.e.,

$$\delta_S = \arg_{A \in S} \min \Delta(A) \tag{1}$$

We can compute $\Delta(.)$ based on an event log or using the subscription database $\mathbb{S}$ to approximate the attribute frequency distribution in $\mathbb{E}$.

The following result establishes a desirable property of our pivot attribute selection criteria.

LEMMA 2. *Given a stream of published events* $\mathbb{E}$, *using* $\delta_S = \arg_{A \in S} \min \Delta(A)$ *to select pivot attributes for partitioning subscriptions minimizes the number of candidate matching subscriptions accessed to match the events in* $\mathbb{E}$.

PROOF. *By Lemma 1, we know that the candidate matching subscriptions are contained in* $\{L_{\langle A_i \rangle} | A_i \in E\}$. *Let* $f(A_i)$ *represents the frequency of attribute* A_i *in* $\mathbb{E}$. *Given a subscription* S, *if the pivot attribute for* S *is* A_i, *then the subscription* S *will be accessed* $f(A_i)$ *times in order to match all the events in* $\mathbb{E}$. *Since we want* $f(A_i)$ *to be as small as possible, we define the pivot attribute to be the attribute with the minimum visibility to events* $\mathbb{E}$.

From the perspective of events, the number of candidate matching subscriptions increases with the number of matching subscriptions, limited by the number of all subscriptions, the upper bound denoted by B. It is optimal for a matching algorithm that the number of candidate matching subscriptions accessed to match the events equals the number of matching subscriptions.

Suppose there are 3 events {e1, e2, e3} and 5 subscriptions {s1, s2, s3, s4, s5}. The number of matching subscriptions for e1, e2 and e3 is 2, 1 and 3, respectively. Then the range of the number of candidate matching subscriptions is [2, 5], [1, 5] and [3, 5] for e1, e2 and e3, respectively.

To simplify discussion, let e1, e2 and e3 be the same event type T, and s1, ..., s5 be all interested in T. According to Lemma 2, each subscription is accessed 3 times. So, the total number of candidate matching subscriptions is 15. Suppose there is an efficient matching algorithm that is capable of pruning most of unmatching subscriptions, substantially decreasing the upper bound B and making it far smaller than the number of all subscriptions. For example, if the number of candidate matching subscriptions of e1, e2 and e3 is reduced to 3, 3 and 4, then the total number is 10, much smaller than that of OpIndex.

EXAMPLE 2. *Figure 4 depicts the first-level partitioning of the subscriptions $\mathbb{S}$ in Table 1 into three lists of subscriptions. In this example, $\Delta(.)$ is derived based on the attribute frequency in $\mathbb{S}$ which results in the three pivot attributes A, C, and D being selected. Thus, given an event $E : (A = 2) \wedge (B = 6)$, the subscriptions in L_C and L_D are guaranteed not to match E; therefore, the subscriptions in these two lists need not be accessed for matching event E.*

Sub	Pivot Attribute
S_1	A
S_2	A
S_3	C
S_4	A
S_5	D
S_6	D

L_A — S_1 — S_2 — S_4
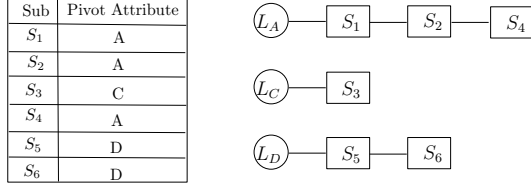
L_C — S_3

L_D — S_5 — S_6

Figure 4: The first level partitioning of subscriptions in Table 1

4.2 Level 2: Predicate Partitioning

In the second level of partitioning, the predicates in each subscription list $L_{\langle \delta_S \rangle}$ are further partitioned based on the predicate operator into predicate lists; i.e.,

$$L_{\langle \delta_S \rangle} = L_{\langle \delta_S, f_{op_1} \rangle} \cup L_{\langle \delta_S, f_{op_2} \rangle} \cup \ldots \cup L_{\langle \delta_S, f_{op_k} \rangle}$$
$$L_{\langle \delta_S, f_{op_i} \rangle} = \{P | f_{op_i} \in P \wedge P \in S \wedge S \in L_{\langle \delta_S \rangle}\}$$

Each predicate list $L_{\langle \delta_S, f_{op_i} \rangle}$ is then independently indexed using an efficient method appropriate for the predicate operator.

Our approach supports both the standard predicate operators (i.e., $<, \leq, =, \neq, >, \geq, \in_s, \notin_s, \in_i, \notin_i$) as well as more complex matching operators (to be discussed Section 6.2). In the following discussion, we shall explicitly consider only the three most common relational operators ($=, \leq, \geq$) to simplify the presentation. Other relational operators ($\neq, <, >$) are treated similarly and are omitted here. Predicates with set or interval comparison operators are rewritten using the common relational operators. For example, $B \in_s \{3, 6, 9\}$ is rewritten as $(B = 3 \vee B = 6 \vee B = 9)$, and $E \in_i [3, 12]$ is rewritten as $(E \geq 3 \wedge E \leq 12)$[7].

In the rest of this section, we discuss how a predicate list $L_{\langle \delta_S, f_{op} \rangle}$, where $f_{op} \in \{=, \leq, \geq\}$, is organized as an inverted-list structure to efficiently process an event $E : (A = \bar{o})$. Given a global ordering of attributes, we use the pair $(A_i, \bar{o})$ as the sorting key and the predicates $P^{A_i, f_{op}, \bar{o}}$ in each predicate list $L_{\langle \delta_S, f_{op} \rangle}$ are sorted in non-descending order of $(A_i, \bar{o})$. In other words, the predicates are first sorted by the attribute and ties are broken by the comparison of operand. In this way, the matching of an event $E : (A = \bar{o})$ against a predicate list $L_{\langle \delta_S, f_{op} \rangle}$, where $f_{op} \in \{=, \leq, \geq\}$, is performed efficiently using a range scan on $L_{\langle \delta_S, f_{op} \rangle}$. Specifically, if f_{op} is '=', we perform an equality search with $(A, \bar{o})$; if f_{op} is '$\leq$', we perform a range scan with $[(A, \bar{o}), (A, +\infty)]$; and if f_{op} is '$\geq$', we perform a range scan with $[(A, -\infty), (A, \bar{o})]$. Here, $-\infty$ and $+\infty$, denote, respectively, the minimum and maximum values of attribute A.

In our implementation of the inverted list structures, we use two optimizations to speed up range scans on predicate lists. The first optimization splits the attribute space into b segments and uses a directory with b entries to index each predicate list $L_{\langle \delta_S, f_{op} \rangle}$. Each entry corresponds to a contiguous segment of predicates in the list.

(margin annotation: "What are ties? how to understand?")

(margin annotation: "The first optimization uses the divide-and-conquer policy to reduce the search space.")

---

[7]In contrast to the k-index approach, our approach does not rewrite an interval-operator predicate into a disjunction of equality predicates and therefore avoids the problem of generating many index entries for an interval-operator predicate.

The predicates having the same attribute will belong to the same segment in $L_{\langle \delta_S, f_{op} \rangle}$. In this way, given an event $E : (A = \bar{o})$, we only need to access the segment containing attribute A. The second optimization introduces a w-bit signature for each segment: for each predicate $P^{A_i, f_{op}, \bar{o}}$ in a segment, we apply a hash function h on $(A_i, f_{op}, \bar{o})$ to select a bit position in w; the selected bit in that segment's signature is then set to 1. The hash function h is defined as follows: if f_{op} is '=', then h is a function of both A_i and $\bar{o}$; otherwise, h is a function of only A_i. The intuition is that a predicate matching on equality operator requires both the attribute and operand to be identical. However, operators '$\leq$' and '$\geq$' are less restrictive and we cannot take advantage of the operand for pruning in the hash function.

EXAMPLE 3. *Consider the matching of an event $E : (A = \bar{o})$ against a predicate list $L_{\langle \delta_S, = \rangle}$. We apply the first optimization by using attribute A to search the directory on $L_{\langle \delta_S, = \rangle}$ to determine the segment in $L_{\langle \delta_S, = \rangle}$ that possibly contain predicates for attribute A. Next, we apply the second optimization by computing the hash value $h(A, =, \bar{o})$ to determine a bit position and check if the selected bit is turned on in the selected segment's signature. If the bit is off, then we conclude that there are no matching predicates for the event in $L_{\langle \delta_S, = \rangle}$; otherwise, we perform a range scan on the selected segment in $L_{\langle \delta_S, = \rangle}$ to search for matching predicates.* □

4.3 Index Construction

Our OpIndex for a subscription database consists of two components. The first component is a collection of predicate lists $\{L_{\langle A_1, = \rangle}, L_{\langle A_1, \leq \rangle}, L_{\langle A_1, \geq \rangle}, \ldots, L_{\langle A_d, = \rangle}, L_{\langle A_d, \leq \rangle}, L_{\langle A_d, \geq \rangle}\}$ derived from the two-level partitioning scheme that we have described. The predicate lists are used to search for matching subscription predicates during event processing. The second component is a collection of counter arrays $\{V_{A_1}, \ldots, V_{A_d}\}$, corresponding to the collection of subscription lists $\{L_{\langle A_1 \rangle}, \ldots, L_{\langle A_d \rangle}\}$. The counter arrays are used by a counting-based algorithm to detect matching subscriptions for an event. For each subscription S_j in $L_{\langle A_i \rangle}$, the counter value $V_{A_i}[j]$ represents the number of predicates in S_j that have not been matched during the processing of an event. These counter values are initialized to the number of predicates in the respective subscriptions before the start of an event matching, and the counter value for a subscription S_j is decremented by one for each predicate in S_j that matches the event being processed. Thus, a subscription S_j in $L_{\langle A_i \rangle}$ matches an event iff $V_{A_i}[j]$ is reduced to zero. To facilitate the efficient updating of these counter values, for each predicate p in a predicate list, we also store a pointer to the counter array entry corresponding to the subscription that contains p.

Algorithm 1 shows the algorithm to insert a new subscription S into an OpIndex. If S contains any set/interval predicate operator, we first rewrite S in terms of the standard relational operators as described in Section 4.2. Next, we determine the subscription's pivot attribute δ_S and append a new entry e in the counter array V_{δ_S} for S. For each predicate $P^{A_i, f_{op}, \bar{o}} \in S$, we insert the predicate along with a pointer to e into the predicate list $L_{\langle \delta_S, f_{op} \rangle}$. The directory on $L_{\langle \delta_S, f_{op} \rangle}$ and the appropriate segment signature are updated as follows. If $P^{A_i, f_{op}, \bar{o}}$ becomes the first predicate in its inserted segment in $L_{\langle \delta_S, f_{op} \rangle}$, we update the directory on $L_{\langle \delta_S, f_{op} \rangle}$ to reflect this. In addition, we compute the hash value $h(A_i, f_{op}, \bar{o})$ to select a bit in the segment's signature and set this bit to 1.

EXAMPLE 4. *Figure 5 shows the OpIndex for the subscription database in Table 1. The subscriptions are first partitioned into three subscription lists $L_{\langle A \rangle}$, $L_{\langle C \rangle}$, and $L_{\langle D \rangle}$; and each subscription list is further partitioned into three predicate lists corresponding to*

617

Algorithm 1: Insert (Subscription S)

1. Determine the pivot attribute δ_S
2. Append a new entry e in V_{δ_S}
3. **for** each predicate $P^{A_i, f_{op}, \bar{o}} \in S$ **do**
4.   Insert $(P^{A_i, f_{op}, \bar{o}}, ptr)$ into $L_{\langle \delta_S, f_{op} \rangle}$, where ptr is a pointer to e
5.   Update the directory for $L_{\langle \delta_S, f_{op} \rangle}$ & the appropriate segment's signature for $P^{A_i, f_{op}, \bar{o}}$
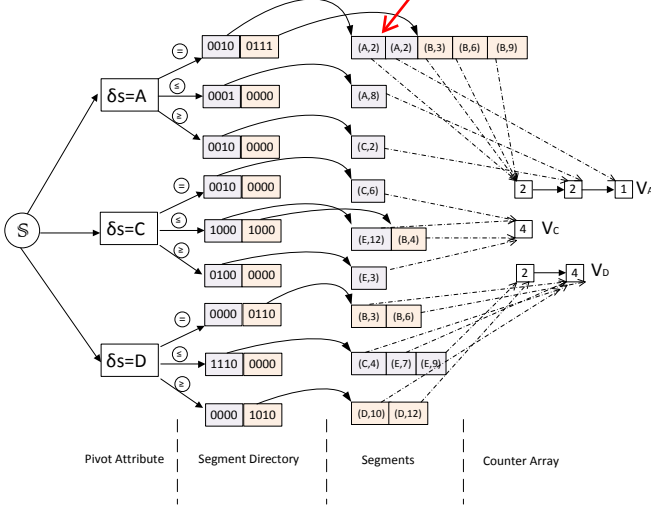

Figure 5: Index Structure

the predicate operators '=', '$\leq$', and '$\geq$'. Each list is split into two segments stored contiguously: one with attributes $\{A, C, E\}$ and the other with $\{B, D\}$. Each segment is associated with a 4-bit signature and the predicates in it are sorted by $(A_i, \bar{o})$. There are three counter arrays V_A, V_C, and V_D, corresponding to the three pivot attribute partitions, and each entry in the segment stores a pointer to its subscription's counter array entry. □

4.4 Space and Construction Complexity

We now analyze the space and construction complexity of OpIndex. To facilitate the analysis, we make the following assumptions:

- The number of predicates Γ in a subscription follows a uniform distribution in $[1, \Gamma_{max}]$, where Γ_{max} is the maximum subscription size. Thus, the average subscription size is $\Gamma_{avg} = \frac{\Gamma_{max}}{2}$.

- Each attribute A_i occurs at most once in a subscription and the probability of A_i occurring in a predicate follows a uniform distribution.

- All the attributes are associated with domain $[1, \sigma]$. The probability of an operand $\bar{o} \in [1, \sigma]$ occurring in a predicate follows a uniform distribution.

- There are three possible predicate operators '=','$\leq$' and '$\geq$', each of which is equally likely to appear in a predicate.

- The size of a segment signature is w bits and each predicate list is organized into b segments, where b is a small number.

LEMMA 3. *The number of predicates in a segment is $\eta = \frac{N\Gamma_{avg}}{3bd}$.*

PROOF. *The average number of predicates in a partition $L_{\langle A_i \rangle}$ is $\frac{N}{d} \cdot \Gamma_{avg}$. Since there are three operators with the same frequency, after the partition by operator, the number of predicates in $L_{\langle A_i, f_{op} \rangle}$ is $\frac{N\Gamma_{avg}}{3d}$. Since each predicate is equally likely to be inserted into any of the b segments of $L_{\langle A_i, f_{op} \rangle}$, the size of each segment is $\eta = \frac{N\Gamma_{avg}}{3bd}$.*

The following result establishes the linear space complexity of OpIndex.

LEMMA 4. *The space complexity of OpIndex is $O(N\Gamma_{avg})$.*

PROOF. *OpIndex consists of four data structures: predicate lists, counter arrays, segment directories and segment signatures. Since there is one counter entry for each subscription, the size of the counter arrays is $O(N)$ given N subscriptions. Since each predicate is inserted into a unique predicate list, the space requirement of the predicate lists is $O(N\Gamma_{avg})$. The space requirement for the segment pointers and signatures is $O(bd)$ and can be ignored compared to that of predicate lists. Therefore, the final space complexity of OpIndex is $O(N\Gamma_{avg})$.*

The insertion procedure consists of three steps: find the pivot attribute, append an entry in the counter array and insert each subscription predicate into the apprpriate predicate list. Its overall time complexity to insert a subscription is given by the following result.

LEMMA 5. *The time complexity of inserting a subscription is $O(\Gamma_{avg} \log \eta)$*

PROOF. *The cost of the pivot attribute selection is $O(\Gamma_{avg})$ to find the attribute with the ~~maximum~~ frequency in the event collection. The append cost in the second step is $O(1)$ since the counter array is not required to be sorted and we can simply append the entry to the end of the array. Finally, for each predicate, it takes $O(1)$ to find the corresponding segment to insert the predicate, $O(\log \eta)$ to insert the predicate in order and $O(1)$ to update the segment signature. Therefore, to insert a subscription with $O(\Gamma_{avg})$ predicates, the time complexity is $O(\Gamma_{avg} \log \eta)$.* □

5. QUERY PROCESSING

Algorithm 2 provides an overview of how OpIndex retrieves matching subscriptions for an input event E. Before the start of the matching, we initialize the set of matching subscriptions R to be empty, and each counter array value to its respective subscription count.

To search for matching subscription predicates, we enumerate the candidate pivot attributes A_j from the set of distinct attributes appearing in the event E (step 3). If A_j is indeed a pivot attribute, we enumerate each attribute-value pair $(A_i, \bar{o})$ in E to search the predicate lists $L_{\langle A_j, f_{op} \rangle}$, $f_{op} \in \{=, \leq, \geq\}$, for predicates that match $A_i = \bar{o}$. To speed up the range-scan searches on $L_{\langle A_j, f_{op} \rangle}$, the two optimizations described in Section 4.2 are applied (steps 7 and 8). For each matching predicate P returned by the scan, we decrement the appropriate counter array value $V_{A_j}[ptr]$ using the subscription pointer ptr associated with P. If the counter value reduces to zero, we have a matching subscription for E which is added to the result set R.

EXAMPLE 5. *Consider the processing of the event $(B = 6 \land C = 3 \land E = 9)$ using OpIndex in Figure 5. Among the three attributes in the event, only attribute C is used as a pivot attribute. Therefore, only the three predicate lists $L_{\langle C, = \rangle}$, $L_{\langle C, \leq \rangle}$, and $L_{\langle C, \geq \rangle}$ are searched for matching predicates. This example demonstrates*

Algorithm 2: Match (Event E)

1. Initialize $R \leftarrow \{\}$
2. Initialize the counter array values
3. **for** each distinct attribute A_j appearing in E **do**
4.   **if** A_j is a pivot attribute **then**
5.   **for** each $(A_i = \bar{o}_i) \in E$ **do**
6.   **for** each operator $f_{op} \in \{=, \leq, \geq\}$ **do**
7.   Determine the segment seg in $L_{\langle A_j, f_{op}\rangle}$ corrp. to A_i
8.   **if** the $h(A_i, f_{op}, \bar{o}_i)^{th}$ bit of seg's signature is set **then**
9.   **for** each matching entry (P, ptr) in the scan of seg **do**
10.   Decrement $V_{A_j}[ptr]$ by one
11.   **if** $V_{A_j}[ptr] = 0$ **then**
12.   Add the subscription corrp. to $V_{A_j}[ptr]$ into R
13. **return** R

the effectiveness of partitioning subscriptions using pivot attributes to minimize the number of accessed subscriptions: although subscriptions S_2 and S_5 partially match the event, they are not accessed at all because they are stored in subscription lists whose pivot attributes do not appear in the event. In contrast, the k-index approach would have accessed all the three partially matching subscriptions. □

5.1 Query Processing Complexity

In the following, we analyze the query processing complexity based on the same assumptions in Section 4.4. First, we estimate the matching probability between a predicate $P^{A_i, f_{op}, \bar{o}}$ and an event predicate $A_i = \bar{o}_i$.

LEMMA 6. *The probability of a predicate $P^{A_i, f_{op}, \bar{o}}$ matching $A_i = \bar{o}_i$ is $\kappa = \frac{1}{3}(1 + \frac{2}{\sigma})$.*

PROOF. *There are three cases to consider depending on the predicate operator. If f_{op} is '=', then there is a match if $\bar{o}_i = \bar{o}$; if f_{op} is '$\leq$', then there is a match if $\bar{o} \in [\bar{o}_i, \sigma]$; and if f_{op} is '$\geq$', then there is a match if $\bar{o} \in [1, \bar{o}_i]$. Since the domain of each A_i is $[1, \sigma]$, the predicate operand $\bar{o}$ and operator f_{op} are each uniformly distributed, the probability for a predicate to match an event is given by κ as*

$$\kappa = \frac{1}{3}\left(\frac{1}{\sigma} + \sum_{\bar{o} = \bar{o}_i}^{\sigma} \frac{1}{\sigma} + \sum_{\bar{o} = 1}^{\bar{o}_i} \frac{1}{\sigma}\right) = \frac{1}{3}\left(1 + \frac{2}{\sigma}\right)$$

□

Next, we estimate the number of predicates matching a query event E.

LEMMA 7. *Given an event E, the expected number of matching predicates for E is $\psi = O(\frac{mN\kappa}{d}(1 + \frac{(m-1)(\Gamma_{avg}-1)}{d}))$.*

PROOF. *Given a query event E of size m, we need to access m subscription lists $\{L_{\langle A_1 \rangle}, \ldots, L_{\langle A_m \rangle}\}$, where $\{A_1, \ldots, A_m\}$ are the attributes in E that are also pivot attributes. Clearly, each subscription in $L_{\langle A_i \rangle}$ must contain a predicate with attribute A_i. Since each subscription list has $\frac{N}{d}$ subscriptions, there are $\frac{N}{d}$ predicates in $L_{\langle A_i \rangle}$ that contain attribute A_i. Since the average subscription size is Γ_{avg}, there are $(\Gamma_{avg} - 1)\frac{N}{d}$ predicates in $L_{\langle A_i \rangle}$ that* do *contain attribute A_i, and each of these predicates has a probability of $\frac{m-1}{d}$ to contain an attribute in E. Therefore, the expected number of predicates in $L_{\langle A_i \rangle}$ that contain attributes in E is $O(\frac{N}{d}(1 + \frac{(m-1)(\Gamma_{avg}-1)}{d}))$.*

do or do not?

By Lemma 6, the number of predicates matching an event is given by $O(\frac{mN\kappa}{d}(1 + \frac{(m-1)(\Gamma_{avg}-1)}{d}))$. □

LEMMA 8. *Given an event E of size m, the query processing cost is $O(m^2 \cdot \log\eta(1 - (1 - \frac{1}{w})^{\eta}) + \psi)$.*

PROOF. *Based on Algorithm 2, the number of predicate lists that needs to be searched for processing E is $O(m^2)$. In each predicate list, a segment has η predicates (by Lemma 3), and the size of each segment signature is w bits. The probability that a bit in a segment signature is not set is given by $(1 - \frac{1}{w})^{\eta}$. Therefore, the probability that a segment needs to be searched for an event predicate $A_i = \bar{o}$ is $1 - (1 - \frac{1}{w})^{\eta}$. If the search cannot be pruned by the signature, the time complexity to search for the first matching predicate in a segment is $O(\log\eta)$ using binary search. For each matching predicate found, we incur a constant cost to update its corresponding subscription counter. By Lemma 7, the total cost incurred to update the subscription counters of matching predicates is given by ψ. Therefore, the overall time complexity to process an event E is given by $O(m^2 \cdot \log\eta(1 - (1 - \frac{1}{b})^{\eta}) + \psi)$.* □

1/b -> 1/w

6. DISCUSSIONS

In this section, we discuss how OpIndex can be extended to handle general CNF/DNF subscriptions as well as support more complex predicate operators.

6.1 Handling General CNF/DNF Subscriptions

Our discussion so far has considered only simple boolean-expression subscriptions consisting of a conjunction of predicates. We now discuss how our approach can be extended to handle more general boolean expressions in DNF or CNF:

DNF: $(P_{11} \wedge P_{12} \wedge \ldots \wedge P_{1n_1}) \vee \ldots \vee (P_{m1} \wedge P_{m2} \wedge \ldots \wedge P_{mn_m})$

CNF: $(P_{11} \vee P_{12} \vee \ldots \vee P_{1n_1}) \wedge \ldots \wedge (P_{m1} \vee P_{m2} \vee \ldots \vee P_{mn_m})$

For subscriptions in DNF, we can consider each conjunctive clause in such a subscription as a simple subscription; i.e., $S = S_1 \vee S_2 \vee \ldots \vee S_n$ with each $S_i = P_{i1} \wedge \ldots \wedge P_{in_i}$. Therefore S is a matching subscription so long as any S_i is a matching subscription. Thus, a set of DNF subscriptions is simply decomposed into a collection of simple subscriptions which can be handled by OpIndex. This straightforward approach to handle DNF subscriptions works for both k-index and BE-Tree as well.

Our approach can also be generalized with two extensions to handle subscriptions in CNF. The first extension deals with pivot attribute selection and subscription partitioning. To correctly detect matching CNF subscriptions, each subscription S is now associated with a set of pivot attributes (instead of a single pivot attribute) since it is not necessarily the case that there exists a specific attribute in S that must occur in every event that matches S. To minimize the number of pivot attributes associated with a subscription $S = S_1 \wedge \ldots \wedge S_m$, we choose the disjunctive clause S_i in S with the least number of predicates[8], and all the attributes in S_i form the set of pivot attributes of S. Thus, a subscription with a set of ℓ pivot attributes will appear in ℓ subscription lists.

The second extension for subscriptions in CNF generalizes the counting-based approach to detect matching subscriptions: we maintain a m-bit bitmap (instead of an integer counter value) for each subscription, where m is the maximum number of disjunctive clauses in a subscription. For a subscription S with k disjunctive clauses,

---

[8]To break ties, we pick the disjunctive clause that minimizes the sum of its attribute frequency.

For a CNF subscription, each clause is a disjunction of multiple predicates. One problem with OpIndex is that the bit corresponding with a clause may be updated multiple times where there are multiple predicates satisfied. It is inefficient to update a bit multiple times. To improve the matching efficiency of CNF subscriptions, some shortcut optimizations should be utilized.

$k \leq m$, its bitmap is initialized and updated as follows. The first k bits in the bitmap of S, which are used to represent whether the k disjunctive clauses in S have been matched by an event, are initialized to ones and the remaining bits are initialized to zeros. Whenever any predicate in the ith disjunctive clause of S is matched, the bitmap is updated by setting its ith bit to zero. Therefore, S is a matching subscription iff its bitmap value is 0. Note that this bitmap scheme is also applicable for the k-index approach to handle CNF subscriptions. For the BE-Tree approach, which can handle only DNF subscriptions, a CNF subscription would need to be rewritten to DNF which would result in a more complex subscription with an increased matching overhead.

6.2 Supporting Complex Predicate Operators

One key advantage of OpIndex's two-level partitioning approach is that each predicate list $L_{\langle \delta_s, f_{op} \rangle}$ can be indexed independently with an efficient method that is appropriate for the predicate operator f_{op}. In Section 4.2, we have presented an inverted-list structure organization to efficiently support $f_{op} \in \{=, \leq, \geq\}$. In this section, we illustrate OpIndex's extensiblity feature by considering how to support the prefix-match operator for string values.

The prefix-match operator is a useful string matching operator, which is also supported in SQL in the form A LIKE 'xyz%' to retrieve records where the value of attribute A begins with 'xyz'. An efficient approach to index string values for the prefix-match operation is the well-known trie index. We can apply the trie index to index subscription predicates involving the prefix-match operator as follows. Given a prefix-match predicate with attribute A and prefix string $\bar{o}$, we map this predicate into a string of the form "A#$\bar{o}$", where '#' denote a special delimiter that does not appear in the attribute name and the attribute's domain values. The collection of transformed strings are then indexed using a trie index.

denote -> denotes

Figure 6(a) shows a hypothetical implementation interface of a trie index. Here, Item defines the structure of an index entry, insert is a function to insert a new entry into the index, and match is a function to retrieve all index entries that satisfies an input prefix-match query (represented by the structure Query). Figure 6(b) shows the modifications to the index's interface for the index to be integrated into OpIndex's framework. To index the transformed strings for OpIndex, the new structure NewItem not only contains the transformed predicate string (represented by Item) but also the identifier of the subscription that contains the indexed predicate (represented by sid) and a pointer to the subscription's counter array (represented by eid). In addition, there is also a new function matchSub which calls the original match function to retrieve matching subscription predicates and update their corresponding counter values; matching subscriptions are added to the result variable.

Similarly, we can apply the above ideas to support other complex predicate operators such as the regular expression matching (RE-match) operator. Specifically, given a predicate list for the RE-match operator, we can apply index methods such as the RE-Tree [4] to index the collection of predicates in the list. In our experiments, we shall evaluate the performance of OpIndex for prefix-match predicates using the trie index.

7. EXPERIMENTS

This section presents results of an extensive performance study of our proposed OpIndex in comparison with k-index and BE-Tree. The implementation of k-index and BE-Tree was kindly provided by the authors of BE-Tree and in the form of binary executable. We also compare with the unoptimized version of OpIndex, denoted as OpIndex-BS, which does not use bucket and signature to

```
struct Item{           class Index{
    ...                    void insert(NewItem item);   NewItem -> Item
}                          vector<Item> match(Query query);
                       }
            (a)
------------------------------------------------------- Item -> NewItem
struct NewItem{        class Index{
    int eid;               void insert(NewItem item);
    int sid;               vector<Item> match(Query query);
    Item item;             vector<int> matchSub(Query query){
}                              for(NewItem item : match(query))
                                   if(--counter[item.eid]==0)
                                       result.add(item.sid);
                               return result;
                           }
                       }
            (b)
```
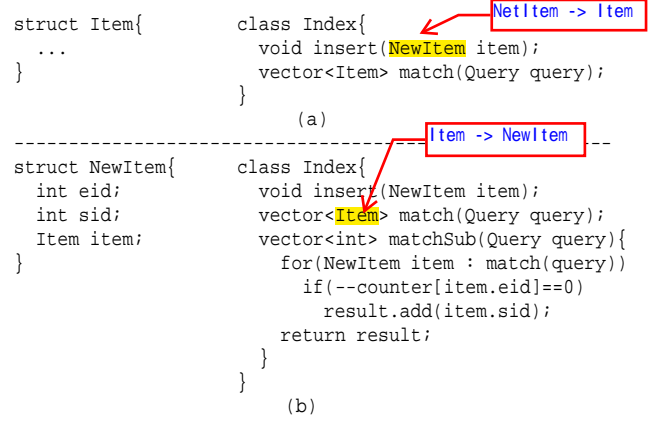
Figure 6: Example to illustrate the extensibility of OpIndex

improve performance. All the indexes are memory resident and implemented in C++. We conduct the experiments on a server with 128GB memory, 64KB L1 cache and 512KB L2 cache, running Centos 5.6.

7.1 Data Generator

To generate synthetic datasets, we implemented our own data generator instead of using BE-Gen [20]. This provides us with better flexibility to customize the generator for our specific requirements such as generating datasets with prefix operator. For uniformly distributed datasets, the generator follows the assumptions in our complexity model in Section 4. All the attributes and operands in a subscription are randomly selected. Three operators '=', '$\leq$' and '$\geq$' are supported. An input parameter θ_1 controls the percentage of '=' operators with the remaining percentages distributed equally between the '$\leq$' and '$\geq$' operators. The performance with respect to the set operator '$\in_s$' and interval operator '$\in_i$' will be evaluated on the real datasets. The generator also generates datasets in which both the attribute and operand follow the Zipf distribution.

Table 3 summarizes the parameters and their settings, with the default values highlighted in bold in our synthetic datasets. We vary the subscription number from 1 million to 40 million to test the scalability. The subscription size tends to be smaller than event size. Moreover, we vary Γ_{max} from 4 to 20 and m from 20 to 120. The default number of attributes in the synthetic datasets is set to 20,000. In our implementation, we set the number of segments in a directory to be 32 and the number of bits in a machine word is 64.

Table 3: Parameters and Settings on Synthetic Datasets

Number of subscriptions N	1M, 10M, 20M, 30M, 40M
Number of dimensions d	20K, 30K, 40K, 50K, 60K
Maximum subscription size Γ_{max}	4, 8, 12, 16, 20
Maximum event size m	20, 40, 60, 80, 100, 120
Percentage of equal operator θ_1	20%, 40%, 60%, 80%, 100%
Value space σ	50, 200, 800, 3200, 12800
Zipf	0, 0.2, 0.4, 0.6, 0.8, 1.0

Besides the synthetic datasets, we also design two data generators from real datasets. The first generator uses the AOL query log [9] to simulate keyword subscriptions. A keyword query is transformed into a boolean expression. Each keyword is treated as an attribute. Its operator is '=' and the operand is set to 1. For example, the query "vldb hangzhou" will be converted to (vldb=1

---

[9] http://www.gregsadetsky.com/aol-data/

∧ hangzhou=1). In this way, the model serves as a filtering condition of AND semantics used in keyword search. Moreover, we can extend the model to consider the term frequency as a filtering condition. For example, (vldb∈$_i$[5,20] ∧ hangzhou∈$_i$[2,8]) is a more precise filtering condition. At the publisher side, we use two datasets, Twitter and Wikipedia, as the event sources. We randomly select 10,000 documents from each dataset to publish. The average event length (in terms of the number of keywords) is 5.4 in Twitter and 123 in Wikipedia. In our implementation, we first extract the 50,000 most frequent keywords. The reason is that BE-Tree crashes when the dimension is too high and we use 50,000 as an upper bound. For both datasets, we generated two types of subscriptions. One uses operator '=' and the other uses interval operator '∈$_i$'. The combination of subscription operators and event sources results in four different datasets: Twitter$_=$, Twitter$_∈$, Wiki$_=$ and Wiki$_{∈_i}$.

The second data generator uses Ebay product information to generate subscriptions and events. In each web page of product description, there is a section named Item specifics which contains structured information of the product. It lists the important attributes and values about the product. We crawled 296,846 products from Ebay and extracted 10,204 unique attributes. To generate a subscription, we follow the assumption that the more common an attribute is, the more likely it will be used as a filtering condition. However, the attribute distribution in Ebay is rather skew. For example, 31 percent of products are associated with attribute brand and 17 percent with attribute country of manufacture. Hence, we count the frequency $f(A)$ for each attribute, take the $log(f(A))$, which is similar to handling $tf\text{-}idf$ and normalize it to form a probability distribution. The attributes in the generated subscriptions will follow this distribution. At the publisher side, we assume that the information provider publishes new products to subscribers. Therefore, we randomly pick 10,000 products with different number of attributes to publish.

7.2 Performance Trade-off in BE-Tree

In [20], BE-Tree was reported to be not highly sensitive to the node capacity parameter (the maximum number of entries stored in a leaf node). However, we observed that, when the number of dimensions grows to very large, this parameter plays an important role in the trade-off between index construction cost and event matching performance. In Figure 7, we vary the node capacity from 5 to 250 and report the build time and average matching time in a uniformly-distributed dataset. When the node capacity grows from 5 to 150, the index construction becomes 15 times faster but the performance of query processing degrades 15 times as well. In [20], it is suggested that the parameter should be set based on the matching rate (the number of matching subscriptions in terms of the total number of subscriptions). Since in very high dimensional space, the matching rate is smaller than 1%, we set the node capacity to 5 in the following experiments. We note that this essentially biased the experimental comparison in favor of BE-Tree.

7.3 Experiments on Synthetic Datasets

The first set of experiments was conducted on the synthetic dataset. We first report the memory usage and index construction time. Then, we evaluate the matching performance with respect to parameters $\mathbb{S}$, Γ_{max}, m, θ_1 and σ, followed by an experiment using the Zipf distribution.

7.3.1 Memory Consumption

Recall that all the indexes are memory-resident. Our first task is to examine the memory consumption. However, since we only

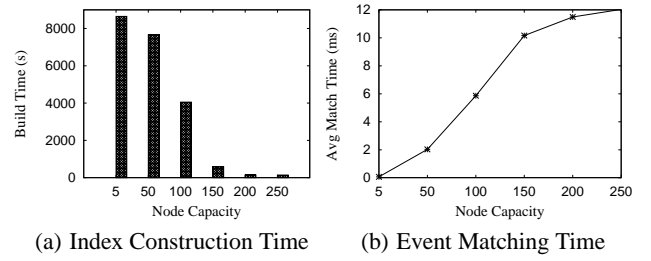(a) Index Construction Time (b) Event Matching Time

Figure 7: Increasing node capacity of BE-Tree

have the binary executable files for our comparison methods, we cannot report the exact index size. As an approximation, we run the algorithms and report the memory usage in the event matching stage for BE-Tree and k-index. For our index, we do not deallocate the memory occupied by the subscriptions after reading them from input file, although our matching algorithm does not need to access them any longer. Thus, we report our memory usage in the worst case which is in favor of the two comparison indexes. For this experiment, we are interested in examining two parameters: N and σ. The results are shown in Figure 8(a) and Figure 8(b).

When the number of subscriptions N increases from 1 million to 40 million, BE-Tree and our index demonstrate similar patterns in memory usage. Their memory cost slowly grows and the consumption by BE-Tree is around 2 times more than our index. However, the performance of k-index degrades dramatically, taking up 7 times more memory. If the operator in a predicate is not '=', k-index has to transform it into multiple predicates of the form $A =\bar{o}$. This replication causes the index to quickly run out of memory.

The value space σ also plays an important role. As shown in Figure 8(b), when we increase σ from 50 to 12,800, k-index runs out of memory and the usage when $\sigma = 3200$ and $\sigma = 12,800$ cannot be reported. The performance of BE-Tree also degrades a lot. Its memory usage grows from 1GB to 10GB. This is because both k-index and BE-Tree maintain attribute-value inverted lists and more inverted lists are built when σ increases. Our OpIndex partitions the subscriptions into predicate lists whose key is the pivot attribute and operator. Its memory consumption is not affected by σ (always 0.2 GB in Figure 8(b)).

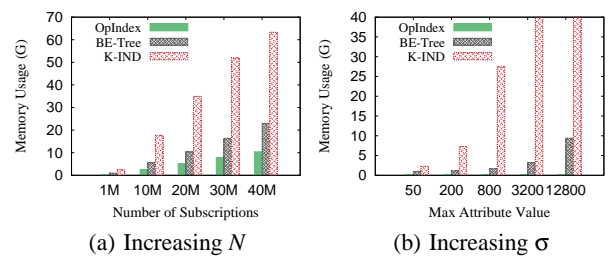(a) Increasing N (b) Increasing σ

Figure 8: Memory Consumption

7.3.2 Index Insertion Time

In this experiment, we use the index insertion time to approximately represent the update cost. The reason is that the binary executable files do not provide the command to support update operations. If we consider an update as a deletion followed by an insertion, the update cost will be around two times of the insertion

cost. <u>We report the performance with respect to N, and Γ_{max} in Figure 9(a) and Figure 9(b).</u>
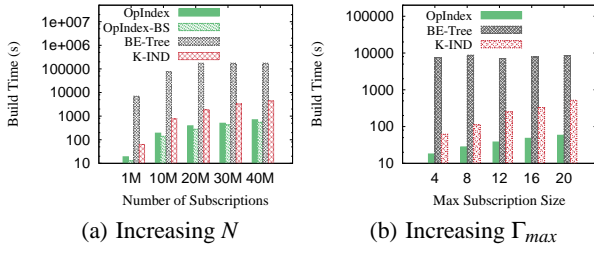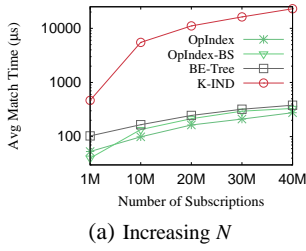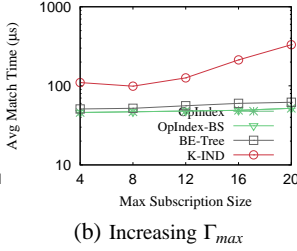

(a) Increasing N (b) Increasing Γ_{max}

Figure 9: Index Insertion Time

The index insertion time of OpIndex is three orders of magnitude better than BE-Tree and one order of magnitude better than k-index. This is because in our index, the three operators are treated in a uniform manner. The partition scheme is effective and the data structure is scalable. The optimized version takes slightly longer construction time than OpIndex-BS as it needs to build buckets and maintains additional fields. k-index generates multiple predicates when the operator is '$\leq$' or '$\geq$', which incurs much higher insertion overhead. When the number of dimensions is very high, BE-Tree incurs long processing time in attribute selection and other optimization techniques to guarantee a good matching performance.

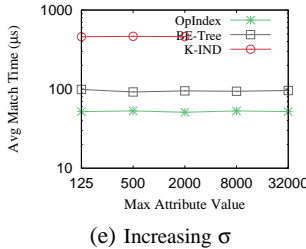### 7.3.3 Matching time with increasing N

(a) Increasing N (b) Increasing Γ_{max}
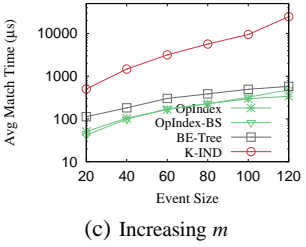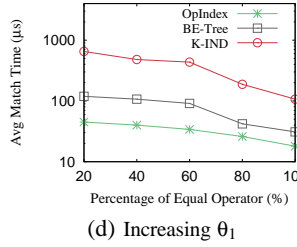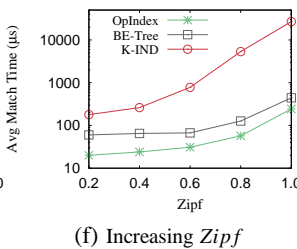
(c) Increasing m (d) Increasing θ_1

(e) Increasing σ (f) Increasing $Zipf$

Figure 10: Matching time on synthetic datasets

The performance of pub/sub matching with increasing subscription number is reported in Figure 10(a). Our index achieves the

best event matching time, which is more than 10X better than k-index. The optimized version scales better than OpIndex-BS because when data size increases, the inverted list becomes longer and the cost of binary search is more expensive. It becomes more effective to reduce the number of binary searches. <u>k-index loses badly for <mark>three reasons.</mark></u> First, partitioning by subscription size is not as effective as partitioning by pivot attribute. <u>Second,</u> its number of inverted lists is much larger than OpIndex, leading to higher lookup cost. <u>Third,</u> its update of counter array is more expensive as it requires random access on the whole array, whose size is the number of subscriptions. In comparison, our counter arrays are much smaller and can be fit in the cache. BE-Tree scales well because of the hierarchical clustering and the optimization mechanisms.

7.3.4 Matching time with increasing Γ_{max}

The running time of increasing Γ_{max} on the three indexes are shown in Figure 10(b). Our OpIndex demonstrates the best scalability due to its data structures and optimized matching algorithm. The running time of k-index increases linearly with Γ_{max}. For BE-Tree, its performance slightly improves at the beginning but later <mark>degrades dramatically</mark> when Γ_{max} increases to 20.

[margin note: This phenomenon can not be observed in the figure. BE-Tree -> k-index?]

7.3.5 Matching time with increasing m

As shown in Figure 10(c), <u>all the indexes are sensitive to m.</u> When m increases, the running time of OpIndex scales similarly to BE-Tree. The OpIndex-BS does not scale as well and its performance degrades to become close to BE-Tree when m is large. k-index performs the worst and does not scale well with m.

7.3.6 Matching time with increasing θ_1

Figure 10(d) shows the matching time when the percentage of '$=$' operator increases. The performance of all the indexes becomes better because '$=$' has high pruning power when σ is large, resulting in a small matching result set. Furthermore, k-index and BE-Tree are more sensitive to this parameter than OpIndex, demonstrating a dramatic performance improvement when θ_1 becomes large. The reason is that they both need to maintain inverted lists whose key is a pair of attribute name and value which naturally supports operator "$=$" and requires operator transformation for other operators as discussed in Section 3.

7.3.7 Matching time with increasing σ

As shown in Figure 10(e), the event matching time stays stable in all the three indexes for increasing value space. BE-Tree and k-index guarantee the filtering performance at the expense of more memory resource and index construction cost. For our index, the number of matching predicates barely changes when σ increases from 50 to $12,800$. This can be verified by our complexity analysis in which the matching probability is estimated as $\kappa = \frac{1}{3} + \frac{2}{3\sigma}$ and decreases from 0.3466 to 0.3334.

7.3.8 Matching time with increasing $Zipf$

We also test the performance when the attribute and value of subscriptions and events follow the Zipf distribution. The result in Figure 10(f) shows that when we gradually increase the skewness of datasets, OpIndex always achieves the best performance and scales better than k-index.

7.4 Experiments on AOL Search Log

The subscriptions derived from AOL query log support two types of operators: equal operator '$=$' and interval operator '$\in_i$'. We vary the number of subscriptions from 1 million to 5 million and report the index construction time in Figure 11(a) and Figure 11(b). When

only operator '=' appears in the subscriptions, the build time of k-index and our index is close. However, when interval operator $\in_i$ is involved, index construction is longer for k-index. BE-Tree does not scale well in the very high dimensional space. It requires two orders of magnitude more insertion time than our index in the real datasets.

The running time of matching tweets and Wikipedia articles using different operators is shown in Figures 11(c)-11(f). In Twitter dataset, the event is small in length. Our index achieves very good matching performance: the running time of OpIndex is 4-9 times faster than BE-Tree and two orders of magnitude better than k-index. When the event length grows to more than 100, as shown in the results of Wikipedia datasets, our index still shows the best performance. The results show that our index works well when the attribute distribution is skew. The pivot attribute is effective in pruning.
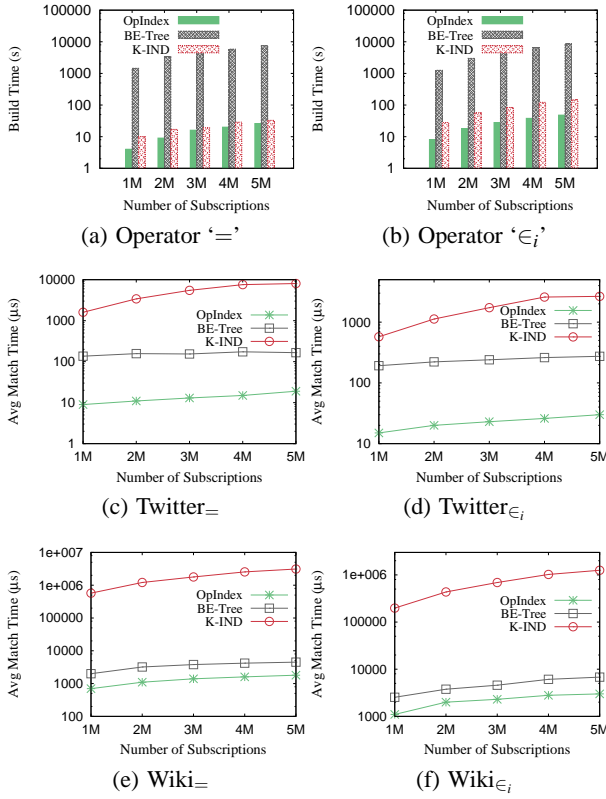

Figure 11: Matching time with increasing N in AOL

7.5 Experiments on Ebay Dataset

The experiment results, including index construction cost and event matching time, on Ebay dataset are reported in Figure 12. Again, k-index spends similar construction time to our index and orders of magnitude better than BE-Tree. When N and Γ_{max} increase, OpIndex always demonstrates the best event matching performance. We also note that as Γ increases, the performance advantage over BE-Tree is more significant. This is because our index first partition the subscriptions based on the pivot attribute. When Γ_{max} increases, it is more likely to find a pivot attribute with small frequency in the event sources to improve the pruning power. In Figure 12(f), the performance of BE-Tree shows a clearly degrad-

partition -> partitions

ing pattern. The set operator $\in_s$ is less powerful than operator $=$ in pruning. It takes more time to prune a longer subscription.
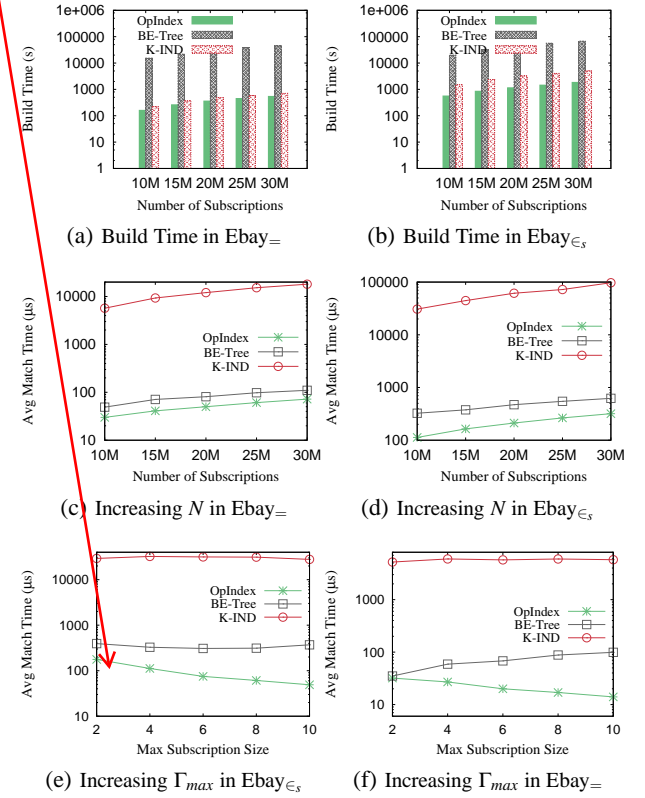

Figure 12: Experiment Results on Ebay

7.6 Experiments on CNF and DNF Matches

As discussed in Section 6, OpIndex can be extended to support CNF and DNF matching. Since the implementation of BE-Tree and k-index does not support general CNF and DNF matches, we only report the matching performance of OpIndex in Figure 13. The default maximum number of clauses is set to 5. When we vary the number of subscriptions from 1 million to 20 million, the matching time of DNF scales better than CNF because a CNF may be inserted into multiple pivot attribute partitions, incurring more scanning cost. When the number of clauses increases, the matching time of CNF scales better because it has a higher probability to find a clause with only one predicate. In that case, there is no replicate insertion.
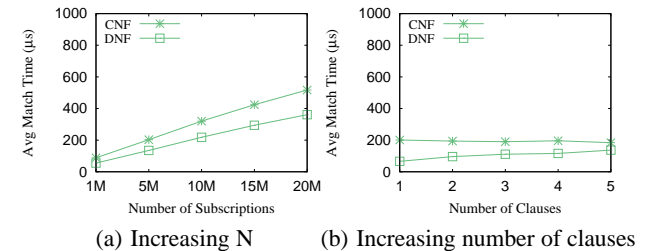

Figure 13: Performance of CNF and DNF matches

7.7 Experiments on Prefix Operator

In the last experiment, we examine the performance of our index on the prefix operator. We use the words in Wikipedia for dataset generation. The dataset contains 1 million subscriptions with 10, 100 attributes, among which 10,000 attributes are numeric and the remaining 100 are string. The operators include only '=' and 'prefix'. Since BE-Tree and k-index cannot support the prefix operator, we only report the running time of our index with respect to increasing percentage of prefix operator and increasing prefix length in Figure 14. We can see that the running time increases as more prefix operators appear in the subscription. This is because 'prefix' is a more expensive operator than '='. However, it still takes less than 0.4ms, which is considered acceptable, to match an event when all the subscriptions are based on prefix operator. When the prefix length increases, the performance is stable, even slightly improved due to fewer matching subscriptions.

(a) Increasing ratio   (b) Increasing prefix length
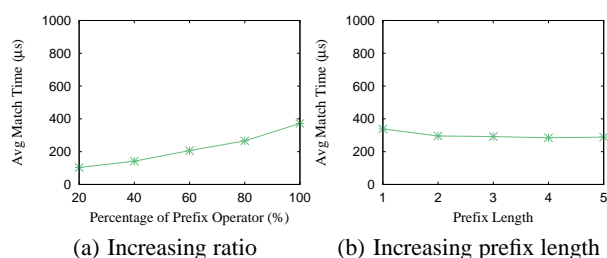
Figure 14: Performance of prefix operator

8. CONCLUSION

In this paper, we tackled the problem of efficient pub/sub match in E-commerce databases where the volume, velocity and especially variety are tamed together. Existing solutions cannot cope effectively for applications with very high dimensional tables. Thus, we proposed an efficient, scalable and extensible index, which adopts a two-level partitioning scheme and can be extended to support complex scenarios such as prefix/suffix and regular expression matches. Extensive experiments were conducted in synthetic and real datasets. The results showed that our index achieved the best performance in terms of memory consumption, index construction and query processing cost.

9. ACKNOWLEDGEMENT

10. REFERENCES

[1] Freebase Data Dumps. https://developers.google.com/freebase/data.

[2] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical Report 15-05, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienzia", Rome, Italy, 2005.

[3] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.

[4] C. Y. Chan, M. N. Garofalakis, and R. Rastogi. Re-tree: An efficient index structure for regular expressions. In *VLDB*, pages 263–274, 2002.

[5] B. Chandramouli, J. Phillips, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems. In *VLDB*, pages 878–889, 2007.

[6] B. Chandramouli and J. Yang. End-to-end support for joins in large-scale publish/subscribe systems. *PVLDB*, 1(1):434–450, 2008.

[7] A. K. Y. Cheung and H.-A. Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Trans. Comput. Syst.*, 28(4):9, 2010.

[8] W. Dakka and P. G. Ipeirotis. Automatic extraction of useful facet hierarchies from text databases. In *ICDE*, pages 466–475, 2008.

[9] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[10] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[11] H. Elmeleegy, J. Madhavan, and A. Y. Halevy. Harvesting relational tables from lists on the web. *PVLDB*, 2(1):1078–1089, 2009.

[12] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.

[13] R. Ghani, K. Probst, Y. Liu, M. Krema, and A. E. Fano. Text mining for product attribute extraction. *SIGKDD Explorations*, 8(1):41–48, 2006.

[14] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Middleware*, pages 254–273, 2004.

[15] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD Conference*, pages 761–772, 2007.

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

[17] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *ICDE*, pages 356–365, 2008.

[18] T. Milo, T. Zur, and E. Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *SIGMOD Conference*, pages 749–760, 2007.

[19] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over xml streams. In *SIGMOD Conference*, pages 253–264, 2012.

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

[21] M. Sadoghi and H.-A. Jacobsen. Relevance matters: Capitalizing on less (top-k matching in publish/subscribe). In *ICDE*, pages 786–797, 2012.

[22] M. Sadoghi and H.-A. Jacobsen. Analysis and optimization for boolean expression indexing. *ACM Trans. Database Syst.*, 38(2):8, 2013.

[23] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.

[24] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 4(9):528–538, 2011.

[25] G. Weikum and M. Theobald. From information to knowledge: harvesting entities and relationships from web sources. In *PODS*, pages 65–76, 2010.

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

[27] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418, 2006.

[28] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD Conference*, pages 481–492, 2012.

[29] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, 1994.

Generally, a matching algorithm is based on an index structure which clusters subscriptions or predicates according to a grouping rule.
1. For the case of grouping subscriptions, clusters containing matching subscriptions are located and checked when matching events. Usually, these clusters contains some other unmatching subscriptions besides the matching ones. The more the unmatching subscriptions, the lower the index efficiency of the matching algorithm. Let k be the number of matching subscriptions and n be the number of subscriptions to be checked. Index efficiency can be denoted by k/n.

2. For the case of grouping predicates, the subsets containing matching predicates are located and checked, in addition to using counting methods to determine the matching subscriptions. Let m be the number of predicates in subscriptions and p be the number of predicates to be checked. Index efficiency can be denoted by (k*m)/p.