A-Tree: A Dynamic Data Structure for Efficiently Indexing Arbitrary Boolean Expressions

SIGMOD ’21

Background & Introduction

- 问题
  - Given event *e* and a set of attribute-value pairs, retrieve all the expressions matched by *e*, called **arbitrary Boolean expressions(ABEs) matching**.
  - Effort over the past 25 years: focused on conjunctive Boolean expressions.
- 挑战
  - 规模大、响应快、复杂(not only “and”, but also “or, not”⋯)
- 目标
  - Minimize the matching time
  - Maintain small index construction time
  - Minimal memory use

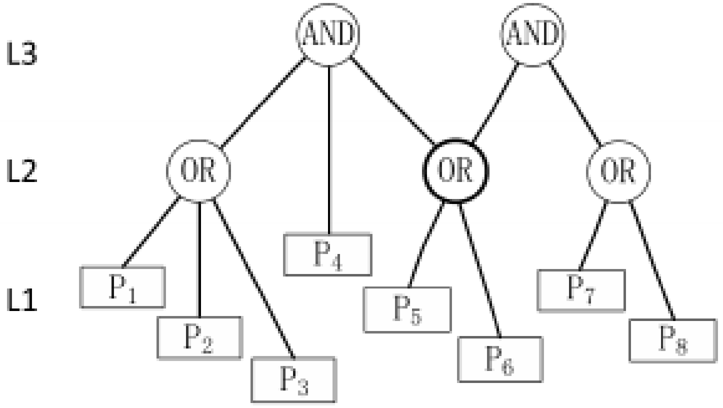
Matching Semantics (sec.3)

- Predicate
  - *<attr, op, vals>*
  - e.g. P1<age,>=,18>
- Expression
  - *f (P1, …,Pm)*
  - e.g. (age>=18)&&(gender=‘male’)
- Event
  - $E = \{<attr1,val1>, \ldots , <attrn,valn>\}$
  - a set of attribute-value pairs
- Matching
  - $E \vdash f \Rightarrow V$
  - V is true if all i : attri= attr $\wedge \langle vali,op,vals \rangle$ = true

Background & Introduction

- 应用
  - 在线广告：广告分发
  - 在线新闻传播：匹配用户感兴趣的新闻
  - Complex event processing(CEP)：match event stream against query
  - Content-based pub/sub system：match event against subscriptions
- 场景
  - Ad exchanges: Directed graph
    - Nodes: publishers, advertisers and intermediaries
    - Edges: sourcing relationships. Boolean expression
    - 来自Efficiently Evaluating Complex Boolean Expressions的例子：广告商A，广告分发者B，目标: (Age ∈ {4} ∧ Interest ∈ {NFL}) ∨ (Age ∈ {5} ∧ Interest ∈ {NBA}), 当一个用户访问publisher Web page,就要进行匹配
  - System monitoring: 错误或关键日志信息
    - e.g. source="mobile" and (type="error" or level="critical").

Basic Idea



- A-TREE
  - aggregate tree
  - n-ary tree
- Subexpression Sharing
  - Different expressions often contain many common predicates and subexpressions.
  - Sharing of predicates, subexpressions and expressions.
  - 减少内存消耗和匹配时间，享元模式?
- Dynamic self-adjustment
  - No need to build in advance
  - Support insert and delete

Content

- Related work (sec.2)
- A-Tree data structure (sec.3-4)
- A-Tree-based event matching algorithms (sec.5)
  - together with zero suppression filter and propagation on demand optimizations that reduce matching time.
- Experimental analysis (sec.6)
  - Compared to existing ABE matching algorithms

Current algorithms (sec.2)

- Dewey ID
- Interval ID
  - The Dewey ID and Interval ID
  - Although these approaches are efficient at evaluating a single expression, they are not efficient at concurrently evaluating a large number of expressions.
- BoP
  - does not achieve a comparable matching time since filtering non-matching expressions based on the minimum number of matching predicates is inefficient for ABEs.
- BDD
  - Support ABEs? neither discussed nor verified

Related Word (sec.2)

- Scan-based
  - 对于一个event，扫描每一个ABE获得匹配
  - 在有多个表达式的情况下性能差
- Dewey ID and Interval ID
  - 缺点

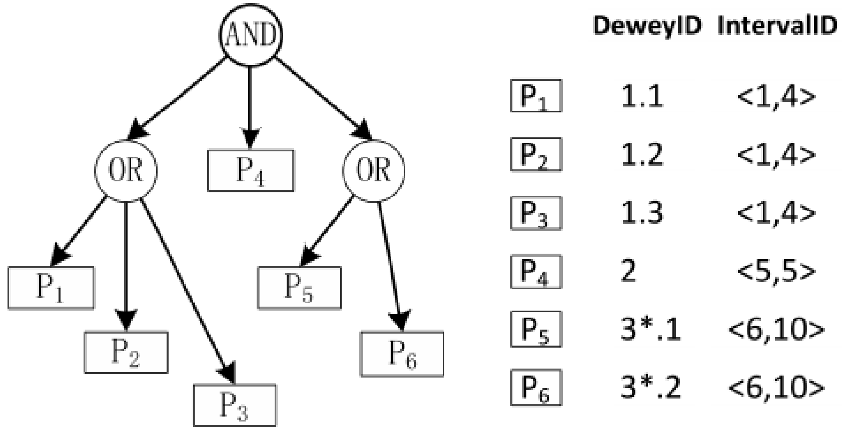


Figure 1: Dewey ID and Interval ID Example

1. Shared subexpressions are evaluated multiple times for a single event, which limits performance.
2. Moreover, both methods are not efficient at pruning nonmatching expressions to narrow down the matching candidates.

Related Word (sec.2)

- Count-based
  - 计算最小可满足的predicate数
  - 只有满足才进一步计算
- BoP
  - 数据压缩

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: BoP Encoding Example

  - 对于不匹配的表达式会有开销
    - $(P1 \vee P2 \vee P3) \wedge P4 \wedge (P5 \vee P6)$
    - Min=3, only P1 P2 P3 =TRUE
  - 相同的子表达式需要多次计算

Related Word (sec.2)

- Tree-based
  - 有点类似决策树
  - (P1∨ P2∨ P3) ∧ P4∧ (P5∨ P6).

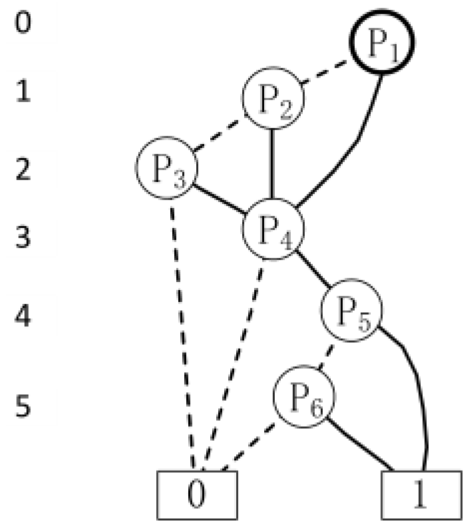


Figure 3: BDD Structure Example

- BDD
  - high memory use
    - 每一个predicate建立一个“node”
    - 变量增加，结构会变得非常复杂
  - high matching time
    - 需要自顶向下的进行遍历

Matching Semantics (sec.3)

- Predicate
  - *<attr, op, vals>*
  - e.g. P1<age,>=,18>
- Expression
  - *f (P1, …,Pm)*
  - e.g. (age>=18)&&(gender='male')
- Event
  - *E = {<attr1,val1>, … , <attrn,valn>}*
  - a set of attribute-value pairs
- Matching
  - $E \vdash f \Rightarrow V$
  - V is true if all i : attri= attr $\wedge$ $\langle$vali,op,vals$\rangle$ = true

A-Tree Organization (sec.4)

- Content
  1. A-Tree data structure
  2. How A-Tree index is dynamically constructed
  3. Space use and insertion cost
- 3 included in 1&2

A-Tree Structure (sec.4.1)

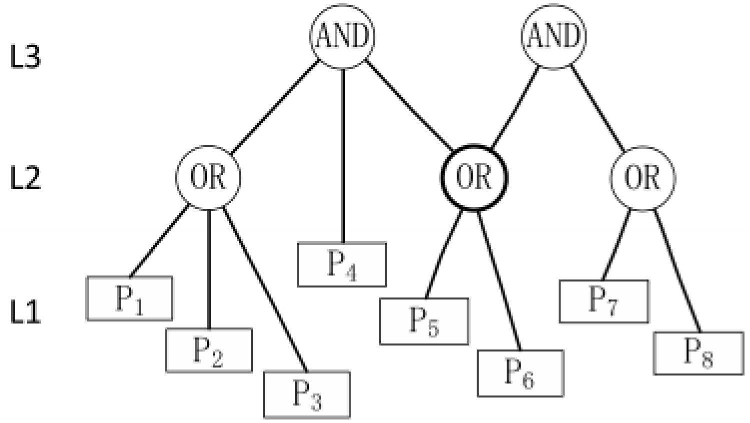


Figure 4: A-Tree Example

- Leaf node (l-node) -> predicate
- Inner node (i-node) -> subexpression
- Root node (r-node) -> ABE
- Constraint:
  - Same predicate, subexpression, and expression correspond to a unique l-node, i-node, and r-node
  - Each l-node andi-node can have any number of parent nodes
  - Each i-node and r-node can have any number of child nodes.

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

A-Tree Space Cost (sec.4.1)

- For single ABE, Np predicates
  - Space: O(Np)
- For Nexp expressions
  - Worst: O(Nexp*Np)
  - Avg: O((Nexp/Nse)*(Np/Nsp))

  Nse and Nsp are the average number of times an expression and a predicate are shared

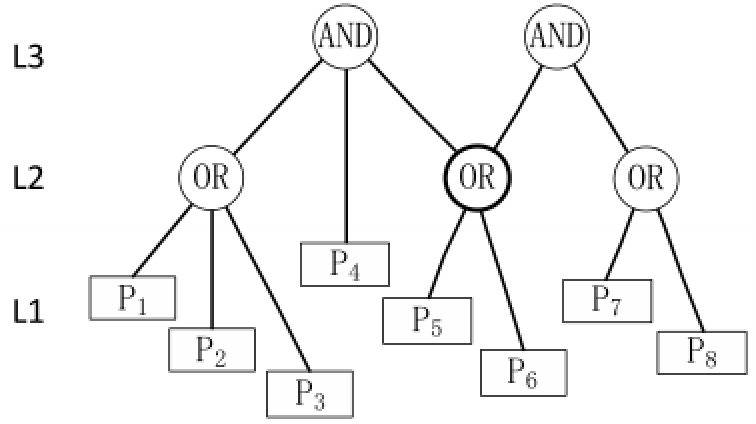


Figure 4: A-Tree Example

Index Construction (sec.4.2.1)

- 想法
  1. 共享的子表达式被唯一表示为A-Tree里的一个node
  2. 引入的ABE结构会根据现有的A-Tree改变
  3. 现有的A-Tree结构会根据新引入的表达式改变
  4. 过期的表达式会被删除

Index Construction (sec.4.2.1)

- “共享的子表达式被唯一表示为A-Tree里的一个node”
- Generated ID
  - Unique ID, address of a node
  - Generate based on IDs of its predicates

    why? 避免相同语义的表达式有不同的表示方法
- How?
  - Translate and, or, not, xor and xnor into the arithmetic operators add and multiply
  - The bitwise operators not, xor, and xnor
- e.g.
  - Expr1: $(P1 \vee P2 \vee P3) \wedge P4 \wedge (P5 \vee P6)$
  - Expr2: $P4 \wedge (P3 \vee P2 \vee P1) \wedge (P6 \vee P5)$
  - Both are **(Id1*Id2*Id3)+Id4+(Id5*Id6)**

Expression Reorganization(sec.4.2.2)

- “引入的ABE结构会根据现有的A-Tree改变”
- 目的
  - 为了识别是否有可以share的subexpression
  - 把不同的表达形式识别成一样的

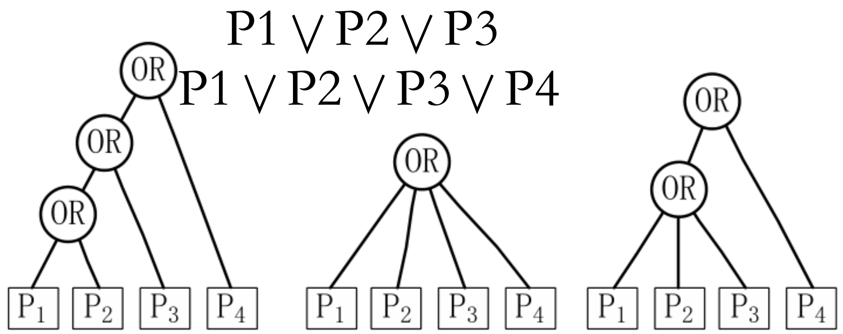


Figure 5: Different Expression Structures

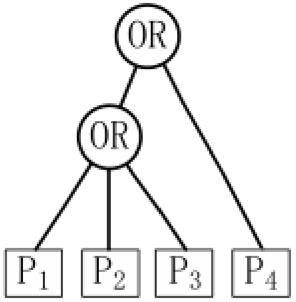
Algorithm 2 Reorganize(*expr*, *atree*)

$U \leftarrow$ expr.childExprs
$C \leftarrow \varnothing$
while $U \neq \varnothing$ **do**
    select an $S \in$ atree that maximizes $|\, S \cap U \,|$
    if $S = \varnothing$ **then**
        break
    $U \leftarrow U - S$
    $C \leftarrow C \cup \{S\}$
expr.childExprs $\leftarrow C \cup U$

O(Np^2)

Index Self-adjustment(sec.4.2.3)

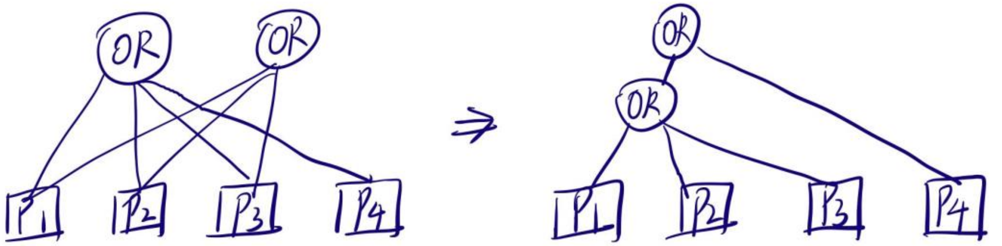
- “现有的A-Tree结构会根据新引入的表达式改变”
- A-Tree的优势
  - 在线构建，不需要离线预构建
- Motivation
  - 不管表达式的到达顺序如何，都能保持优化
  - e.g. 先 $P1 \vee P2 \vee P3$ 后 $P1 \vee P2 \vee P3 \vee P4$ 可以被优化，但是反过来可以吗?



Algorithm 3 SelfAdjust(*newNode*)

```
for childNode ∈ newNode.childNodes do
    for parentNode ∈ childNode.parentNodes do
        if newNode.expr ⊂ parentNode.expr then
            update(childNode, parentNode, newNode)
```

O(Np)



Insert Algorithm(sec.4.2.3)

Algorithm 4 Insert($expr, H_{en}, atree$)

ID ← generateID($expr$)
if $H_{en}[id] \neq null$ **then**
    $H_{en}[id]$.useCount += 1   O(1)
    return $H_{en}[id]$
else
    Reorganize($expr, atree$)   O(Np^2)
    for $childExpr \in expr.childExprs$ **do**
        childNode ← Insert($childExpr, H_{en}, atree$)
        childNodes.add($childNode$)
    node ← createNewNode($expr, childNodes, atree$)   O(1)
    node.useCount ← 1
    SelfAdjust($node$)   O(Np)
    $H_{en}[id]$ ← node
    return node

$O((1/N_{se}) * (N_p/N_{sp})^2).$

Expression Deletion(sec.4.2.4)

- “过期的表达式会被删除”
- useCount -=1
  - If decrement to 0, safely delete.
- 非常简单、直接

Algorithm 5 Delete($expr, H_{en}, atree$)

ID ← generateID($expr$)
node ← $H_{en}[id]$
node.useCount −= 1
if node.useCount = 0 **then**
    Remove($node, atree$)
    $H_{en}[id] \leftarrow null$
    for $childExpr \in expr.childExprs$ **do**
        Delete($childExpr, H_{en}, atree$)

$$O((1/N_{se}) * (N_p/N_{sp})).$$

Event Matching(sec.5)

- Complex
  - Not only conjunctive(“and”)
  - ¬(P5 ∨ P6) P5 and P6 are true, still unsatisfied
- 自底向上
  - evaluate()
    - 如果是lnode，就是它的predicate；
    - 如果是inode或者rnode，根据operator和operands
  - undefined
    - event does not contain the predicate’s attribute
    - 计算规则如表

Algorithm 6 Match($preds, H_{en}$)

```
for pred ∈ preds do
    ID ← generateID(pred)
    l-node ← H_en[id]
    l-node.result ← pred.result
    Q_1.add(l-node)
for level = 1 → M do
    while Q_level is not empty do
        node ← Q_level.dequeue()
        result ← node.evaluate()
        node.clean()
        if result = undefined then
            continue
        for all parent ∈ node.parents do
            if parent.operands.empty() then
                plevel ← parent.level
                Q_plevel.add(parent)
            parent.operands.add(result)
        if result = true then
            matchingExprs.add(node.exprs)
return matchingExprs
```

Table 2: Evaluation on the Operand *Undefined*

Operator	Operand1	Operand2	Result
and	*undefined*	*true*	*undefined*
and	*undefined*	*false*	*false*
or	*undefined*	*true*	*true*
or	*undefined*	*false*	*undefined*
not, xor, xnor	*undefined*	*any*	*undefined*

Optimization(sec.5.2)

- Main cost
  - result propagation
  - subsequent evaluation
- 举例
  - 所有的false都会被传递给上层，实际上并不是所有的false都有这种必要
  - 上层的操作为and的时候，只要下层有一个为false，其他的下层predicate其实都不重要了

Optimization Method(sec.5.2)

- **Zero Suppression Filter**
  - 消除not、xor、xnor
  - false无需向上传递
  - 如果一个operand是undefined，就假定为false?

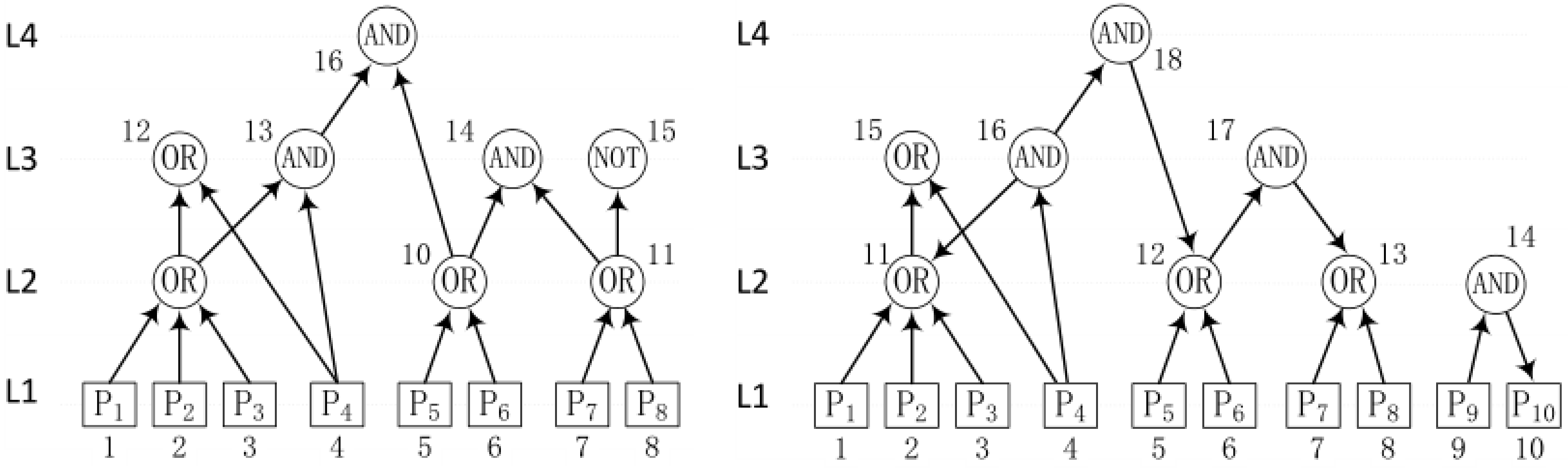
by applying the following laws: (1) $\neg(E_1 \wedge E_2) = \neg E_1 \vee \neg E_2$, (2) $\neg(E_1 \vee E_2) = \neg E_1 \wedge \neg E_2$, (3) $E_1 \oplus E_2 = (E_1 \wedge \neg E_2) \vee (\neg E_1 \wedge E_2)$, and (4) $E_1 \otimes E_2 = (E_1 \wedge E_2) \vee (\neg E_1 \wedge \neg E_2)$

- **Propagation On Demand**
  - 针对上层是and的情况
  - 随机挑选一个子节点为access child
  - 上层节点持有到其他child nodes的link
  - 只有access child为true，上层节点才计算其他子节点
  - child nodes需要保存result，需要通过一个event版本号来保存状态，避免影响其他event （这样不会影响并发性吗？）

Optimization Example(sec.5.3)

$$S_1 = (P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$$
$$S_2 = (P_5 \vee P_6) \wedge (P_7 \vee P_8)$$
$$S_3 = P_1 \vee P_2 \vee P_3 \vee P_4$$
$$S_4 = (P_1 \vee P_2 \vee P_3) \wedge P_4$$
$$S_5 = (P_5 \vee P_6) \wedge (P_7 \vee P_8)$$
$$S_6 = \neg(P_7 \vee P_8)$$

(a) Expression Samples

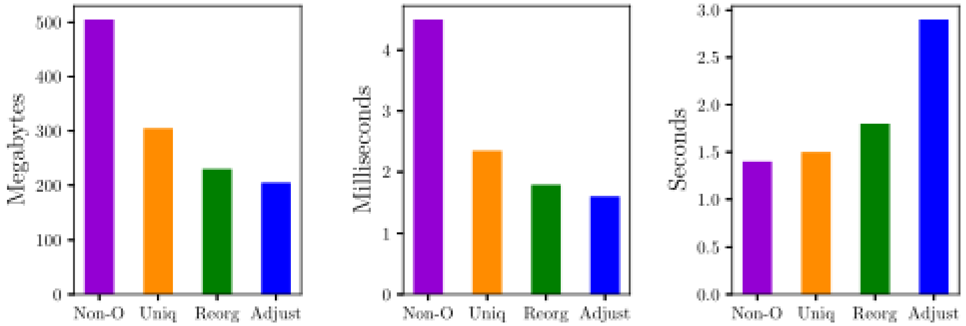


(b) Unoptimized Matching (c) Optimized Matching

Figure 6: Event Matching Example

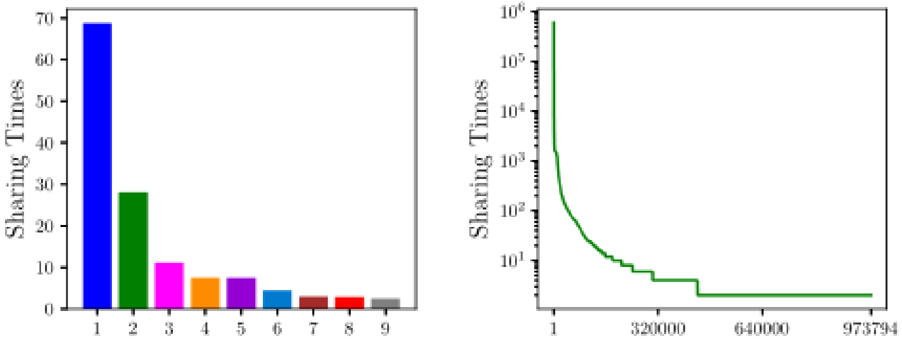
Experiment(sec.6)

- Sharing analysis
- Optimization analysis



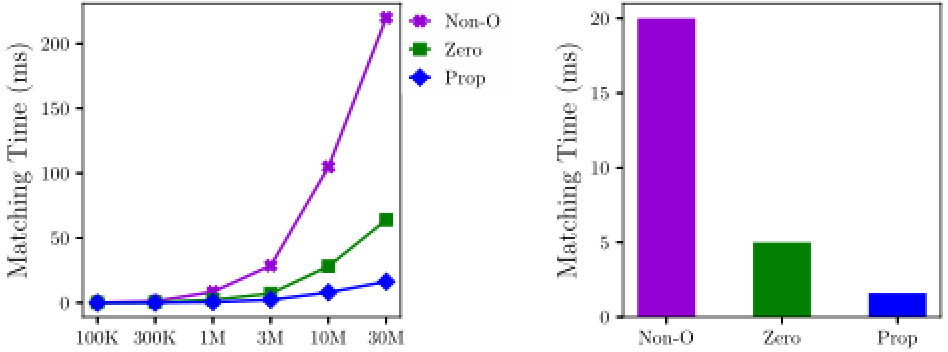
(a) Memory Consumption (b) Matching Time (c) Construction Time

Figure 9: Uniqueness and Dynamicity



(a) Varying Layer (b) Predicate Id

Figure 7: Subexpression Sharing Analysis



(a) Varying Expression Number (b) Ads Dataset

Figure 10: Matching Optimizations

Experiment(sec.6)

- Comparation

Table 4: Experiments on the Ads Dataset

Index	Matching	Construct	Memory
ATree	1.6 ms	2.9 s	205 MB
BDD	11.5 ms	25.6 s	823 MB
Translation	13.7 ms	11.1 + 14.6 s	320 + 4,592 MB
BoP	44.1 ms	8.6 s	684 MB
Interval ID	53.6 ms	12.2 s	1775 MB
Dewey ID	62.7 ms	14.8 s	1887 MB
SCAN	529.1 ms	-	-

Conclusion

- A-Tree
- Shared subexpression
- 值得学习的地方
  - 对于and的处理
- 可能可以优化的地方
  - Propagation On Demand中，child nodes的状态如何保存
  - 对于or，是否可以“熔断”?