Reporter: 温晴

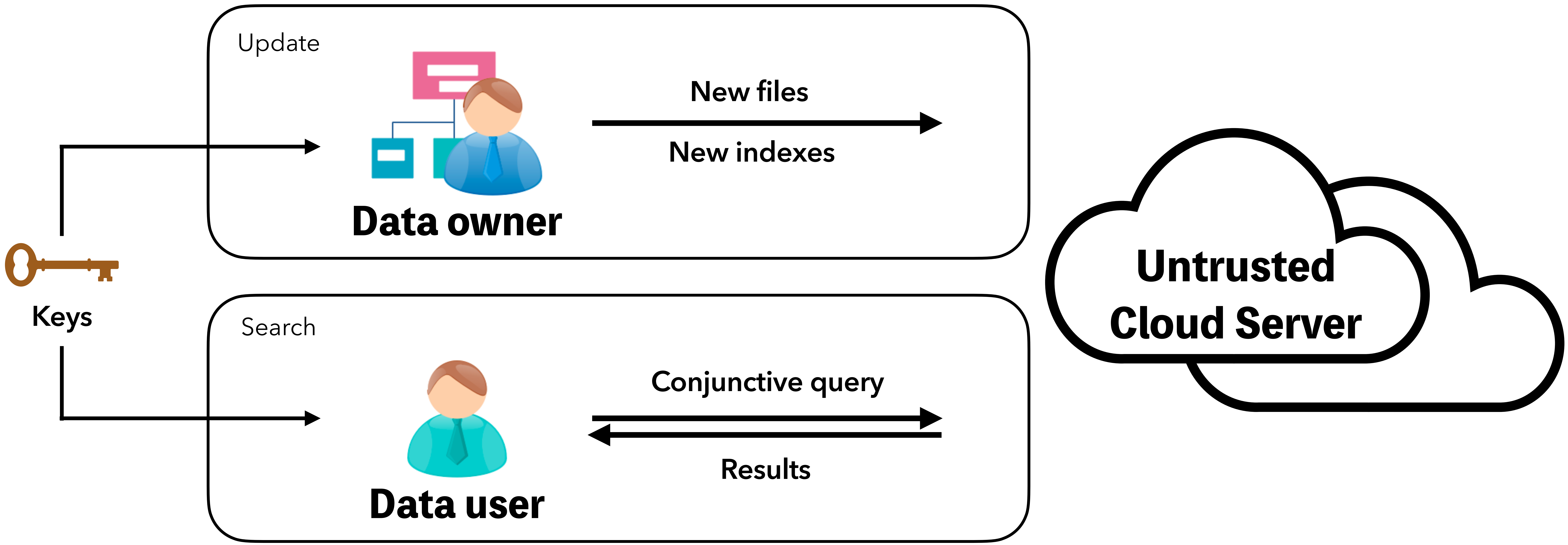
VBTree: forward secure conjunctive queries over encrypted data for cloud computing

Zhiqiang Wu, Kenli Li

The VLDB Journal (2019) 28:25-46

Content

Induction - Model



Induction - Problem

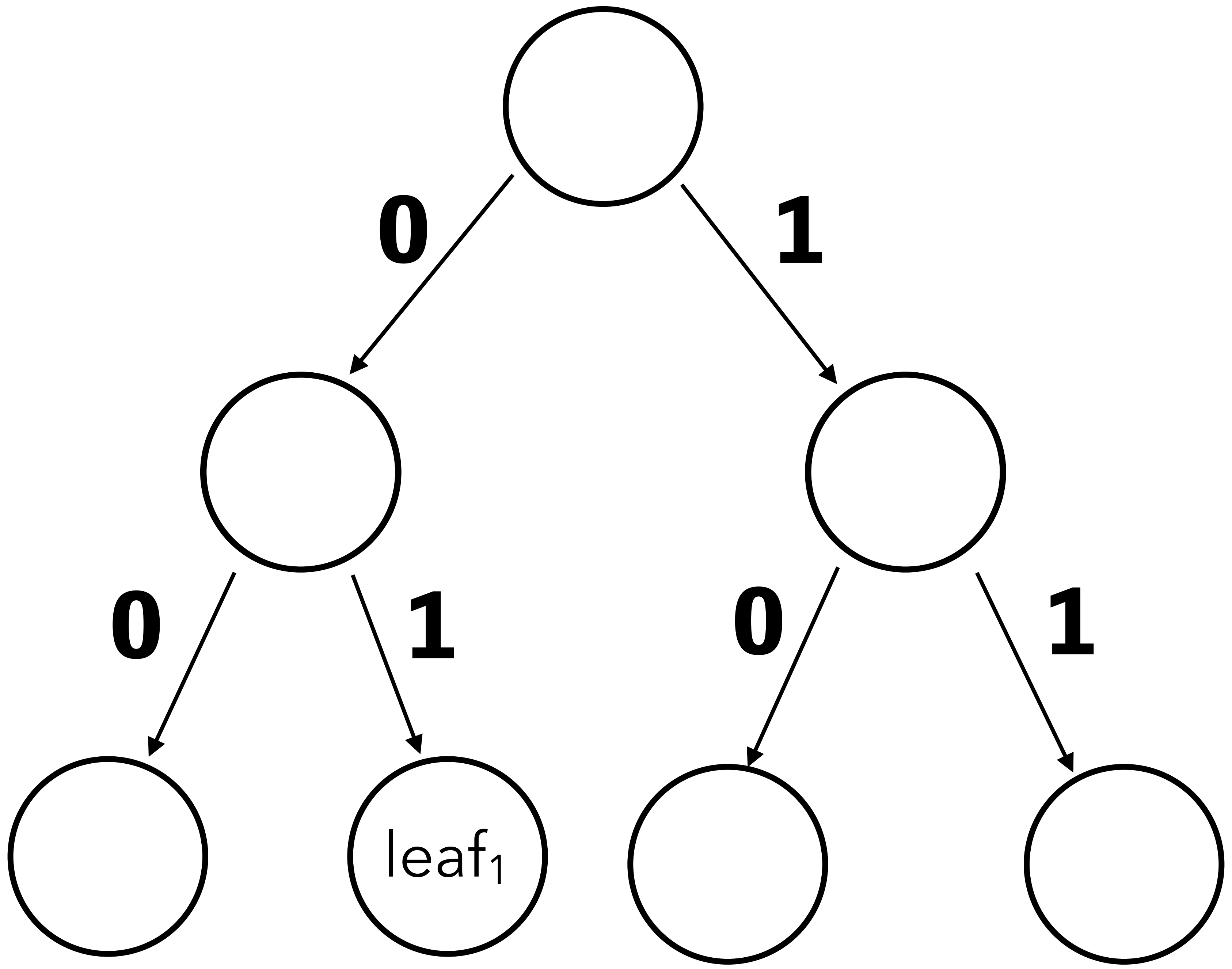
- Few SSE works achieve forward security and conjunctive queries simultaneously.
- Conjunctive queries schemes
  - The tree-based SSE schemes are branch-leaked.
  - The Bloom-filter-based ones are lacking scalability in dynamically modifying.
- Forward secure schemes
  - Single-keyword schemes.
  - Data user's storage is large.

Induction - Solution

- Virtual binary tree (VBTree)
  - The tree only exists in a logical view, and all of the elements are actually stored in a hash table.
  - VBTree avoids leaking branch privacy.
- Version control repository (VCR)
  - VCR is used to record and control versions of keywords and queries.
  - VCR makes the update procedures leak nothing about user's historical queries.

Introduction - Notation

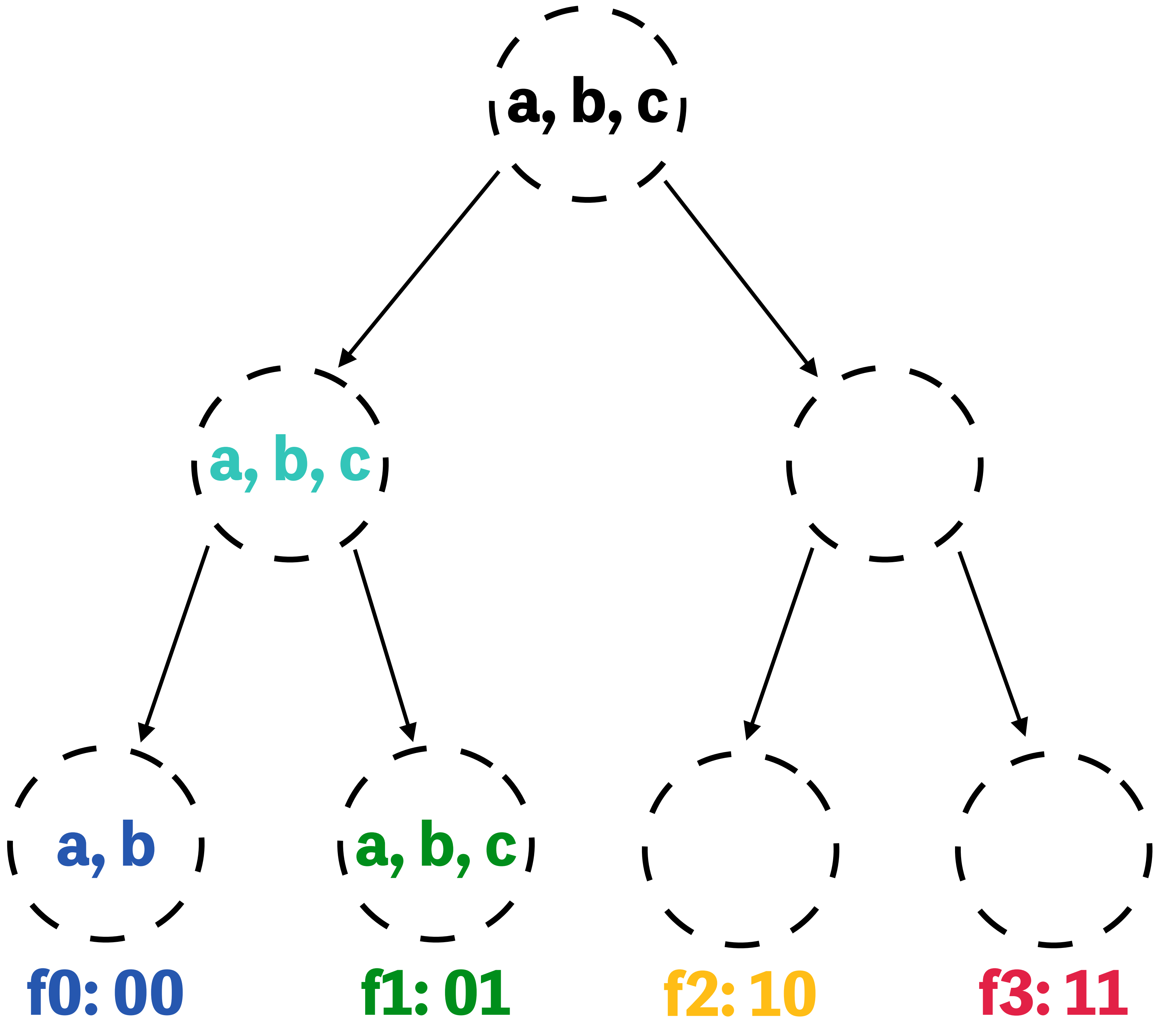
Complete binary tree



Notation	Meaning
L	Height
Path(v)	A string concatenated with all tree branches from root to v.
Node(i)	A set of all traversed nodes from the root to the i^{th} leaf

Eg: $L = 3$ Path(leaf$_1$) = '01' Node(1) = {root, root->left, leaf$_1$}

Static Scheme - Concept



Virtual binary tree

- A virtual binary tree (VBTree) is an encrypted full binary tree stored in a random oracle.
- The value of WD(w, i) in the hash table are a set of key-value pairs:

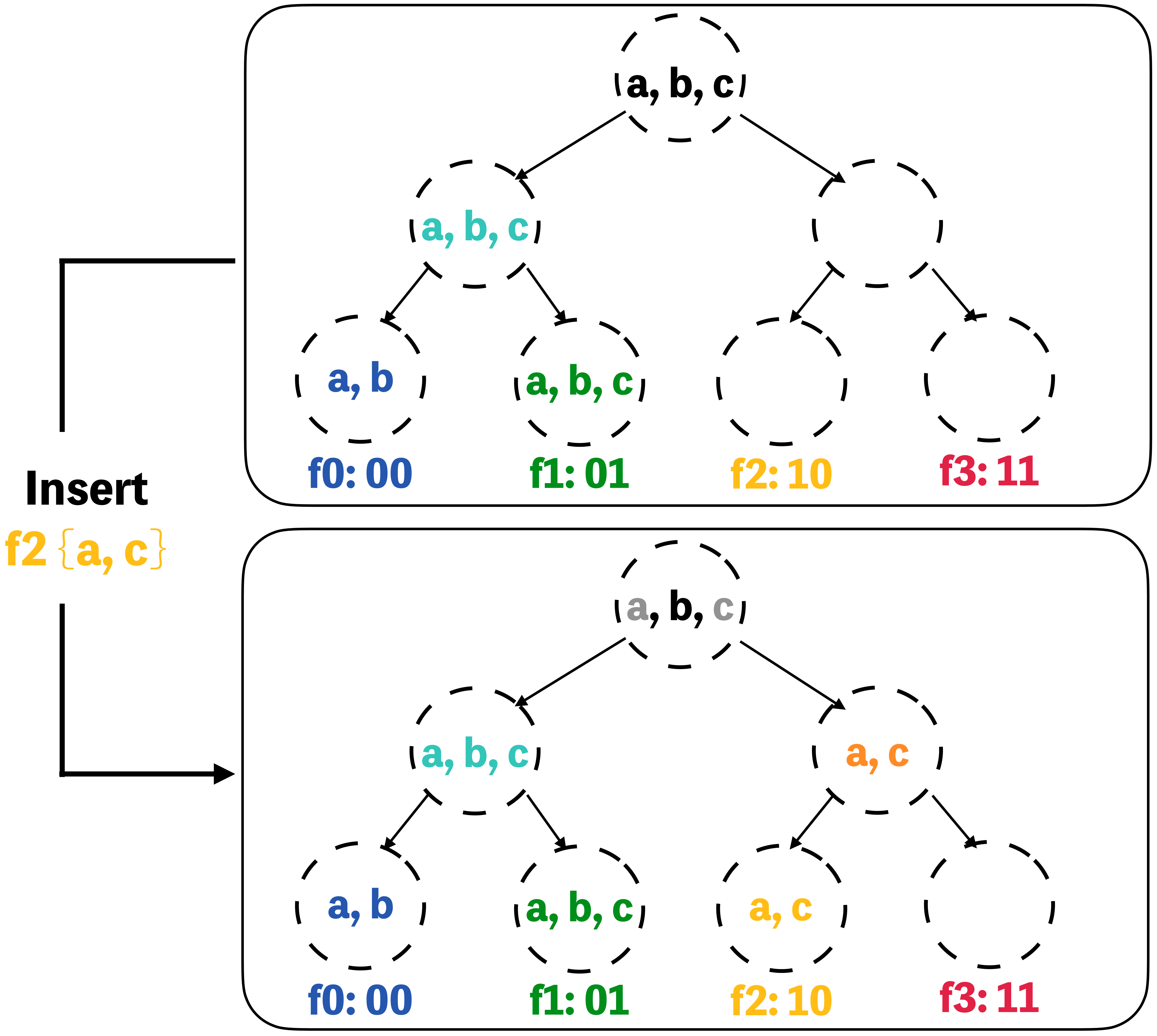
$$\mathbf{WD(w, i)} = \left\{ \boxed{H_1(\,\mathrm{Path}(v) \| F_K(w)\,)} \boxed{1} \right\}_{v \in \mathbf{Node(i)}}$$

Eg: $WD(‘c’, 1) = \{\, (H_1(‘’\|F_K(‘c’)), 1),$

$(H_1(‘0’\|F_K(‘c’)), 1),$

$(H_1(‘01’\|F_K(‘c’)), 1)\,\}$

Static Scheme - Construction



The static construction of a VBTree:

1. For every (w, i), the data owner <u>inserts L items</u> into the VBTree from the root to the $leaf_i$.
2. The data owner <u>inserts some random values</u> into the VBTree and <u>sends the tree</u> to the cloud as an index.
3. The data owner <u>tells the cloud n</u>.

Static Scheme - Search (Top-down)

- Given a query $q = w_1 \wedge w_2 \wedge \ldots \wedge w_u$, the trapdoor is $T(q) = \{F_k(w_1), F_k(w_2), \ldots, F_k(w_u)\}$.

Algorithm 1 $\mathcal{VB}$Tree.Search $\mathcal{A}$lgorithm ($\mathcal{VSA}$-1)

// Cloud:
$Search(\mathcal{T}; \textbf{p}\text{ath}, \{F_K(w_1), F_K(w_2), \cdots, F_K(w_u)\})$

1: **for** i=1 to u **do**
2: Let $b_i \leftarrow \mathcal{T}$.ContainsKey($H_1(\textbf{\textit{path}}||F_K(w_i))$)
3: If b_i=false, then return 'not found'
4: **end for**
5: If the length of $\textbf{p}$ath is $L-1$, then convert it to a number as a file identifier and return one result.
6: Invoke, Search($\mathcal{T}$, $\textbf{\textit{path}}$||'0', $\{F_K(w_1), \cdots, F_K(w_u)\}$)
7: Invoke, Search($\mathcal{T}$, $\textbf{\textit{path}}$||'1', $\{F_K(w_1), \cdots, F_K(w_u)\}$)

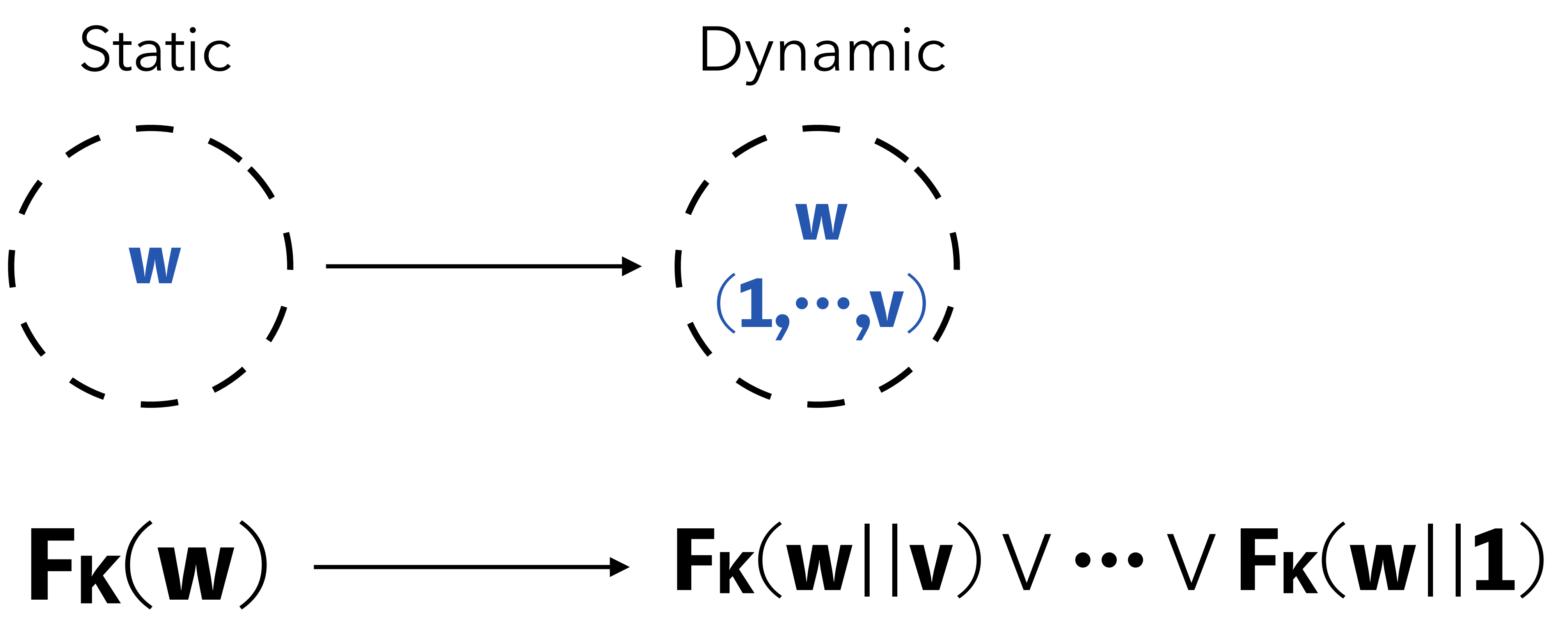
- *VSA-1* can search from any tree node by setting the path to its corresponding value.
- The cloud can search from the sub-root, whose path value is a string of $L - \lceil \log_2 n \rceil - 1$ zeros
- The query time is

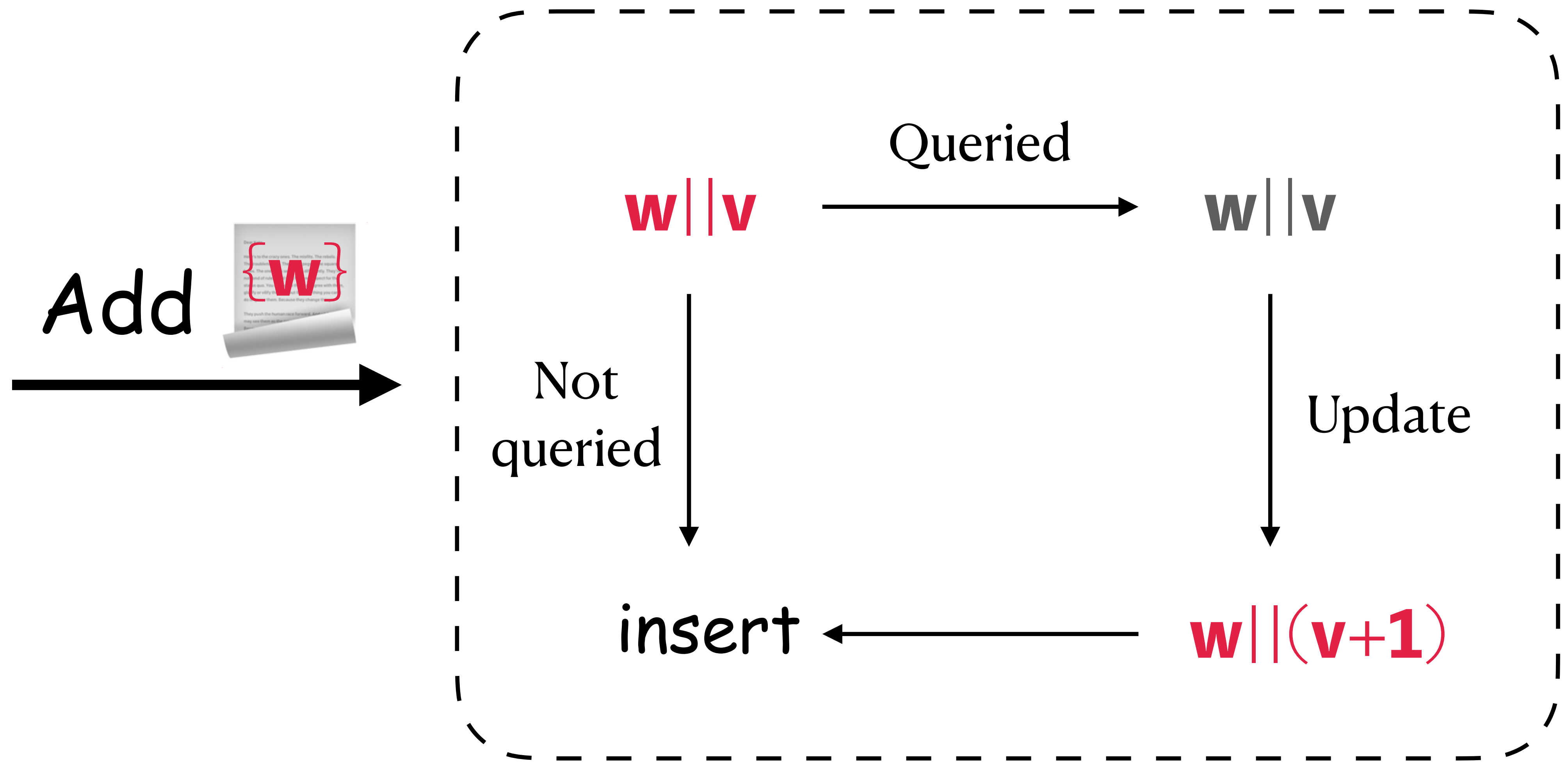
$$O(|q| \min_{w \in q}\{|DB(w)|\} \log_2 n).$$

Dynamic Scheme

- Label keywords with versions.
  - Given a keyword w, its vth version is denoted as w||v, and its vth version trapdoor is denoted by $F_K(w||v)$.

- Historical search queries and updates are managed by a storage mechanism, which is called version control repository (VCR).





Dynamic Scheme - VCR

Version control repository

Local repository (LR)

w	V_l	b	n_l

Cloud repository (CR)

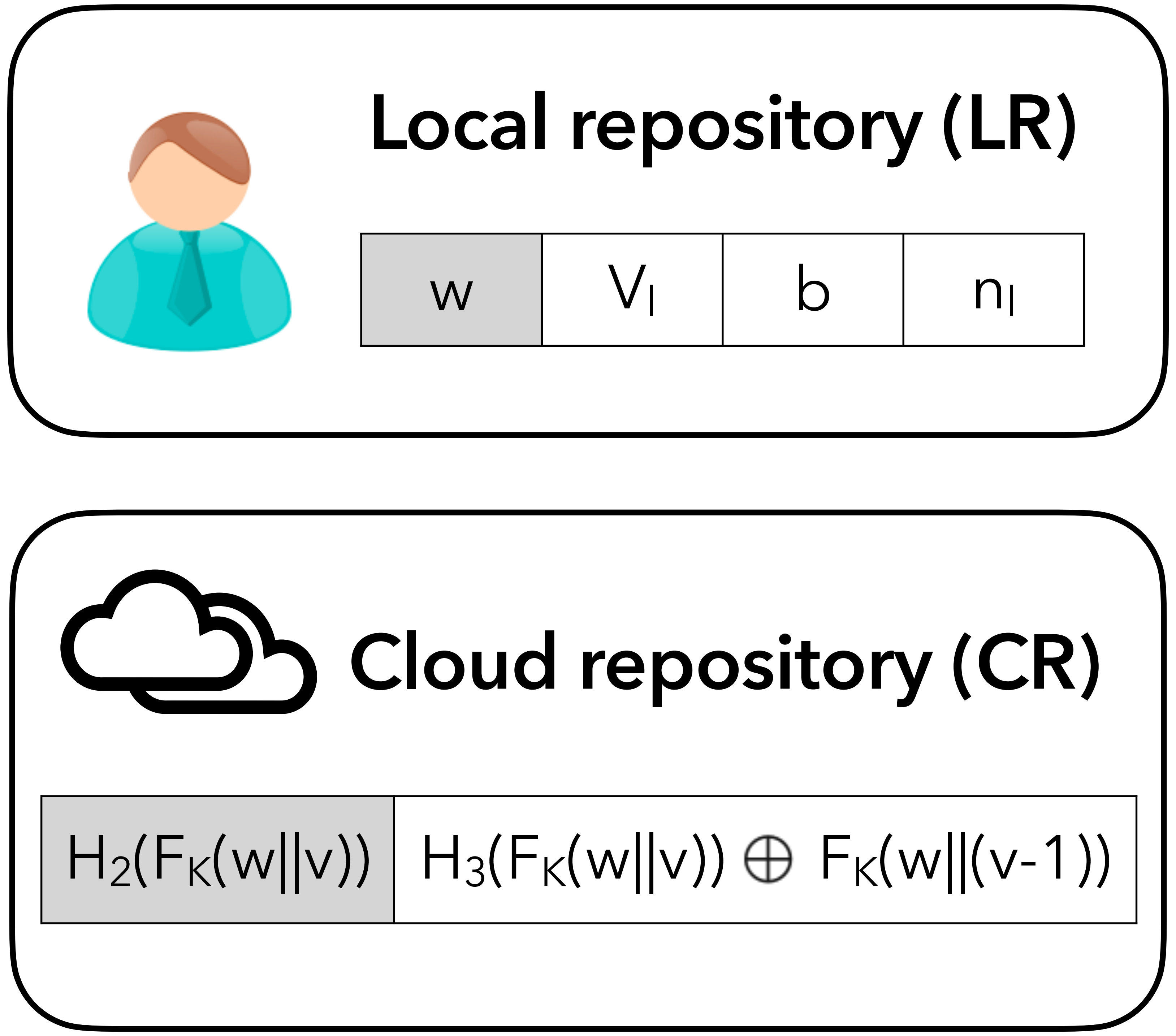
$H_2(F_K(w\|\|v))$	$H_3(F_K(w\|\|v)) \oplus F_K(w\|\|(v-1))$

LR is a client-side hash table. LR(w) denotes the usage information of a keyword w:

- The number $LR(w).V_l$ denotes the latest version of the keyword w.
- The bit LR(w).b denotes that whether the latest version of keyword w has been queried by the data users or not.
- The number $LR(w).n_l$ denotes the last file identifier that matches the keyword w.

Dynamic Scheme - VCR

Version control repository



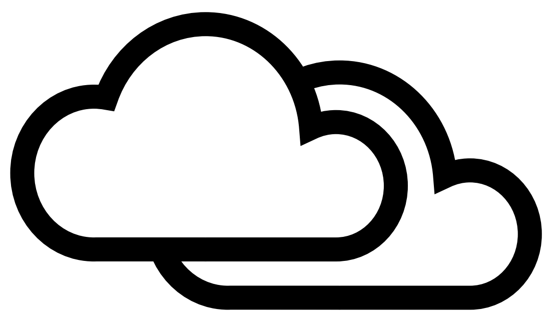
CR is a cloud-side hash table. From CR, given a new version trapdoor, the cloud can get all the corresponding historical trapdoors.

The cloud can get $F_K(w||(v-1)) \leftarrow CR[H_2(F_K(w||(v)))] \oplus H_3(F_K(w||(v)))$ and other subversions by $F_K(w||v)$.

Dynamic Scheme - Add

Add a keyword w of a file with identifier n

⑥ (T_{up_1}, T_{up_2})



① According LR(w).b, update the version or not.

② Create $T_{up1} \leftarrow WD(w,n) - WD(w, LR(w).n_l)$.

③ If LR(w).b is true, create $T_{up_2} \leftarrow \{(H_2(F_K(w||v)), H_3(F_K(w||v)) \oplus F_K(w||(v-1)))\}$ and set LR(w).b to false.

④ Set LR(w).n_l to n.

⑤ Pad the size of T_{up1} to L with random values.

⑦ $\mathcal{T} \leftarrow \mathcal{T} \bigcup T_{up_1}$ and $CR \leftarrow CR \bigcup T_{up_2}$

Dynamic Scheme - Search(1)

Algorithm 3 $\mathcal{VB}$Tree.Search $\mathcal{A}$lgorithm ($\mathcal{VSA}$-3)

// Cloud:

$Search(\mathcal{T};$ **p**ath, $\{\triangle_1, \triangle_2, \cdots, \triangle_u\})$

```
1: for i=1 to u do
2:     for j=1 to | △_i | do
3:         Let b_i ← T.ContainsKey(H_1(path||x_ij))
4:         if b_i=false then
5:             Remove x_ij from △_i, i.e., △_i ← △_i − {x_ij}
6:         else
7:             break
8:         end if
9:     end for
10:    If | △_i | = 0, return 'not found.'
11: end for
12: If the length of path is L − 1, then convert it to a number as a file
    identifier and return one result.
13: Invoke: Search(T, path||'0', {△_1, △_2, ··· , △_u})
14: Invoke: Search(T, path||'1', {△_1, △_2, ··· , △_u})
```

用户构造查询陷门: $T(q) \leftarrow \bigcup_{w\in q} F_K(w||LR(w).V_l)$

云服务器查询CR，获含有历史版本查询陷门的合取范式CNF：$\triangle_1 \wedge \triangle_2 \wedge \cdots \wedge \triangle_u$

$$\triangle_i = F_K(w_i||v) \vee F_K(w_i||(v-1)) \vee \cdots \vee F_K(w_i||(v-e_{w_i}))$$

$$\triangle_i = \{x_{i1}, x_{i2}, \ldots, x_{ip(i)}\} \quad |\triangle_i| = p(i)$$

conjunctive query time is $O((\sum_{w\in q}(e_w+1))\min_{w\in q}\{|DB(w)|\}\log_2 n)$.

Dynamic Scheme - Search(2)

create a new thread to run following codes.
for $i = 1$ to u **do**
  for all $y \in \triangle_i$ **do**
    ignore the lastest trapdoor, i.e., if $y = x_i$ continue;
    remove a historical trapdoor from CR by a key, i.e., $CR \leftarrow CR - \{H_2(y)\}$.
    get a result set from the tree by the subquery y, i.e., $\mathcal{IDs} \leftarrow DB(y)$.
    for all $id \in \mathcal{IDs}$ **do**
      remove old version items, $\mathcal{T} \leftarrow \mathcal{T} - \{(H_1(Path(v)||y), 1)\}_{v \in Nodes(id)}$
      insert the latest items, $\mathcal{T} \leftarrow \mathcal{T} \bigcup \{(H_1(Path(v)||x_i), 1)\}_{v \in Nodes(id)}$
    end for
  end for
end for

If the cloud learns that CR contains many versions of w, the cloud creates a new thread to reorganize the tree and CR after a search query.

对于此次查询中涉及到的每一个关键词w的非最新版本陷门(y)进行如下操作：

1. 清除它在CR中对应记录。

2. 对于y能查到的所有文件，将VBTree中相关的WD(w,id)中的版本号y替换为最新版本x_i

Dynamic Scheme - Delete

tree$_A$	$H_1($ A:Path(v)||F_K(w))	1	added keywords
tree$_B$	$H_1($ B:Path(v)||F_K(w))	1	deleted keywords

- stored in the same hash table
- indistinguishable operations

The search result is the difference of the two result sets.

不彻底的删除

Optimizations - Traversal width optimization

Data File Partition Problem (DFPP)

Given a set of files F of even size, Partition F into a file set F_1 and a file set F_2, such that $|F_1|=|F_2|$ and $|W(F_1) \cap W(F_2)|$ is the minimized value.

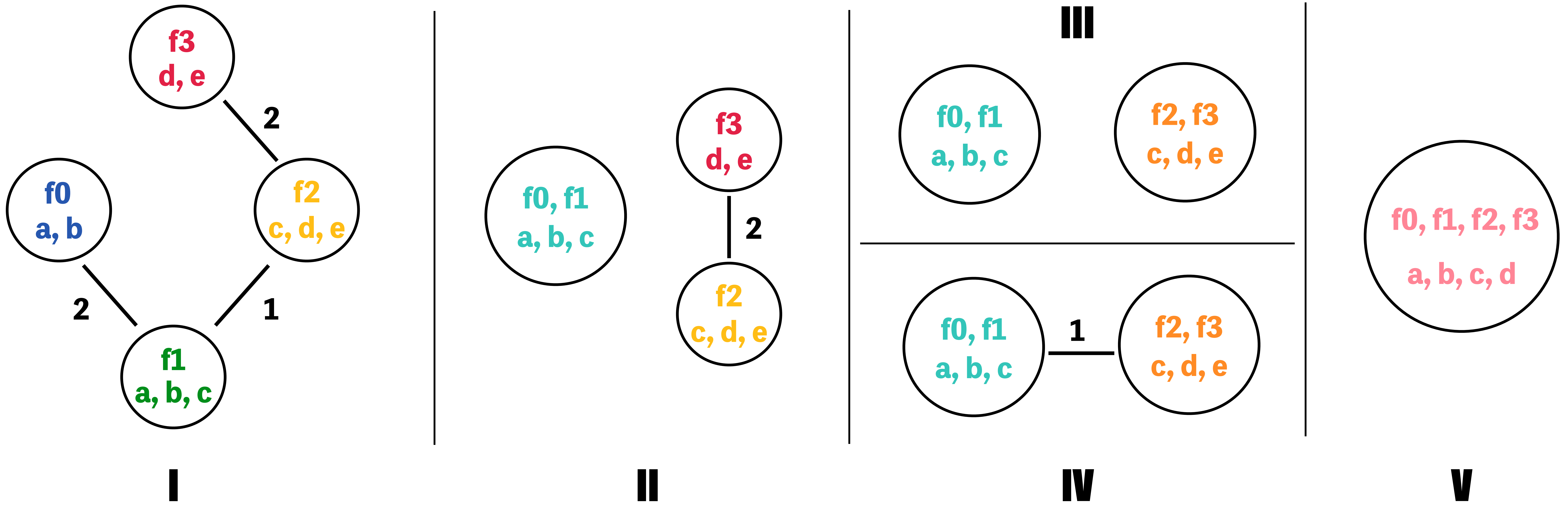
Solution:

- Graph-based algorithm
- TF-IDF-based algorithm

Optimizations - Traversal width optimization

Graph-based algorithm

The file list of the final vertex contains optimized order of file insertions.



Optimizations - Traversal width optimization

TF-IDF-based algorithm

Term Frequency $$TF(w, d_i) = \frac{1}{m_f}$$

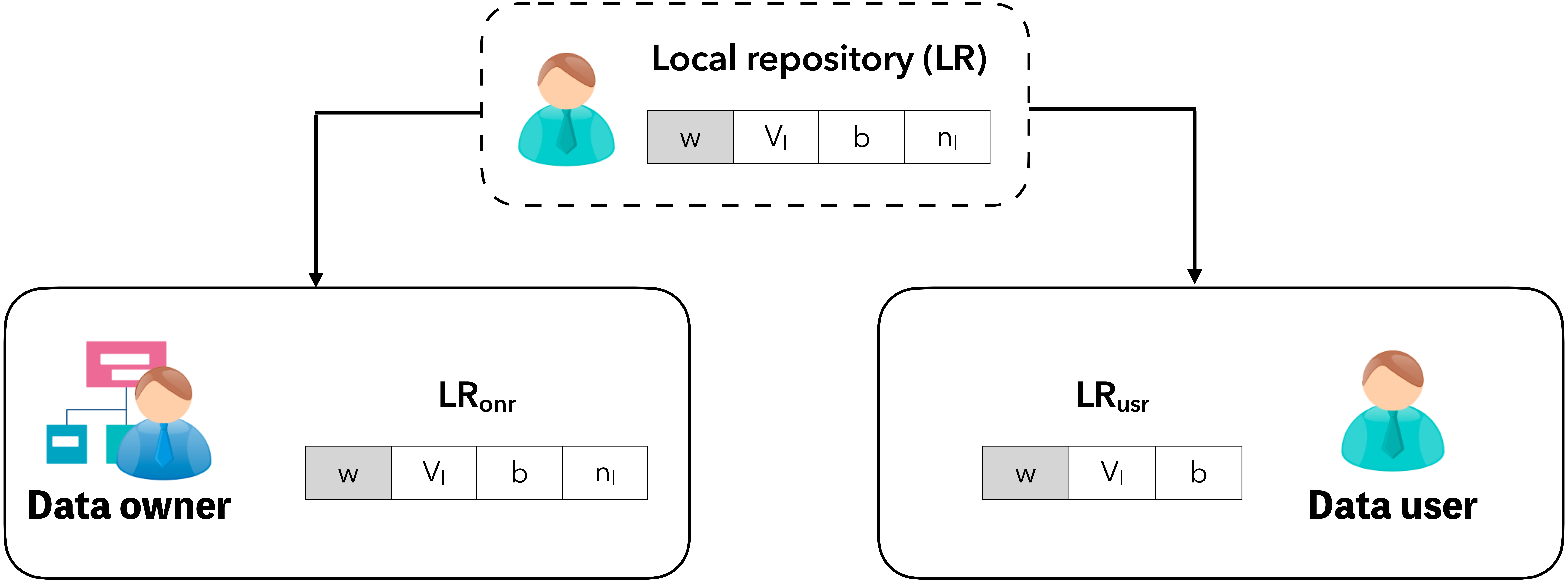
Inverse Document Frequency $$IDF(w, \mathcal{F}) = \log_2 \frac{n}{|DB(w)|}$$

$$score(d_i) = \sum_{w \in d_i} TF(w, d_i) IDF(w, \mathcal{F})$$

① Use the TF-IDF-based algorithm to sort the files and to split the sorted files into $\lceil \frac{n}{P} \rceil$ parts.

② Use the graph-based algorithm to optimize each of the subset files.

Optimizations - Reducing client-side storage



Optimizations - Removing user storage

Algorithm 5 $\mathcal{V}$CR.$\mathcal{U}$pdate $\mathcal{A}$lgorithm ($\mathcal{VUA}$-1)

// Data owner:

$Trapdoor(K, LR_{onr})$

1: Get the maximal version from LR_{onr}, i.e., $v_u \leftarrow max_{w \in W}\{LR_{onr}(w).V_l\}$.

2: Gather LR_{usr} of version v_u of all users.

3: For each keyword w in W, skip all keywords of version v_u

1. Let $v_0 \leftarrow LR(w).V_l$
2. Prepare a key-value token o for update, $o \leftarrow (H_2(F_K(w||v_u)), H_3(F_K(w||v_u)) \oplus F_K(w||v_0))$
3. Generate a trapdoor, and add all o, $T_{up_2} \leftarrow T_{up_2} \bigcup\{o\}$ Set to the new version, i.e., $LR_{onr}(w).V_l \leftarrow v_u$.
4. Mark w with non-leaked, i.e., $LR_{onr}(w).b \leftarrow false$.

4: Send T_{up_2} to the cloud.

5: Send v_u to the users.

// Cloud:

$Update(T_{up_2}, CR)$

1: Update CR, i.e., $CR \leftarrow CR \bigcup T_{up_2}$.

// Data users:

$Refresh(LR_{usr})$

1: Remove the storage, i.e., $LR_{usr} \leftarrow \phi$.

- 将所有关键字更新至同一个版本，版本号为所有关键字的历史最大版本号v_u
- 用户可以在零存储的条件下使用$F_K(w||v_u)$对VBTree进行搜索

Experiment - comparison

Dynamic VBTree 是唯一同时实现了连接查询和前向安全的SSE机制

Table 1 Comparison of current tree-based SSE schemes and typical forward-private schemes

Scheme	Security	Query type	Forward private	Index size	Search time	Update time	Owner storage	User storage
KRB [6]	IND-CKA2	Conj.	x	$O(mn)$	$O(x \log n)$	$O(m \log n)$	$O(m \log n)$	$O(1)$
PBTree [11,13]	IND-CKA	Range	x	$O(n \log n)$	$O(r \log n)$	$O(\log n)$	$O(n \log n)$	$O(1)$
KBB [14]	COA	Multi-k.	x	$O(mn)$	$O(m^2 + xm \log n)$	$O(m^2 \log n)$	$O(mn)$	$O(m^2)$
IBTree [9]	IND-CKA2	Conj.	(Static)	$O(cn \log n)$	$O(x \log n)$	–	–	$O(1)$
Sophos [7]	IND-CKA2	Single-k.	√	$O(N)$	$O(t_a r_u)$	$O(t_a)$	$O(m \log n)$	$O(m \log n)$
FFSSE [8]	IND-CKA2	Single-k.	√	$O(N)$	$O(r_u)$	$O(1)$	$O(m \log n)$	$O(m \log n)$
VBTree (static)	IND-CKA2	Conj.	(Static)	$O(N \log n)$	$O(x \log n)$	–	–	$O(1)$
VBTree (dynamic)	IND-CKA2	Conj.	√	$O(N \log n)$	$O(r_q x \log n)$	$O(L)$	$O(m \log n)$	$O(1) \sim O(m_u)$

$x = \min_{w \in q} |DB(w)|$, r_q denotes the number of update times of the keywords of q after the last search of these keywords, m_u denotes the number of newly updated keywords, n denotes the number of files, with m the number of keywords that can be queried, with N the number of keyword/document pairs,

Experiment - dataset

Enron e-mail dataset (unstructured)

Contents and metadata of e-mails are all considered as keywords with prefix attribute strings.

① sort the files by the TF-IDF algorithm.

② split the files into multiple parts with each group size P = 8192.

③ use the graph-based algorithm to optimize each file group.

Gowalla dataset (structured)

A five-dimensional data table. A record can be considered as a file, and the number of (w, id) pairs N is five times the number of records n.

sort the data table by attributes in turn (first sort the table by this attribute with more high-frequency words).

Experiment - storage

The optimization algorithm can significantly reduce the index size of the Enron dataset.

	n	m	N	M	Owner (MB)	EDB (GB)
(1) EN	10E4	97E4	41E6	17E7	55	4.0
(2) EN	50E4	26E5	16E7	67E7	157	15.5
(3) EO	50E4	26E5	16E7	48E7	157	11.1
(4) GN	64E5	14E5	32E6	20E7	65	4.6
(5) GO	64E5	14E5	32E6	17E7	65	4.0

'E' means the Enron dataset. 'G' means the Gowalla dataset. 'O' means an optimized index, and N means non-optimized. 'M' is the number of items in the hash table of the VBTree. 'Owner' denotes the storage size of the data owner

Experiment - construction time

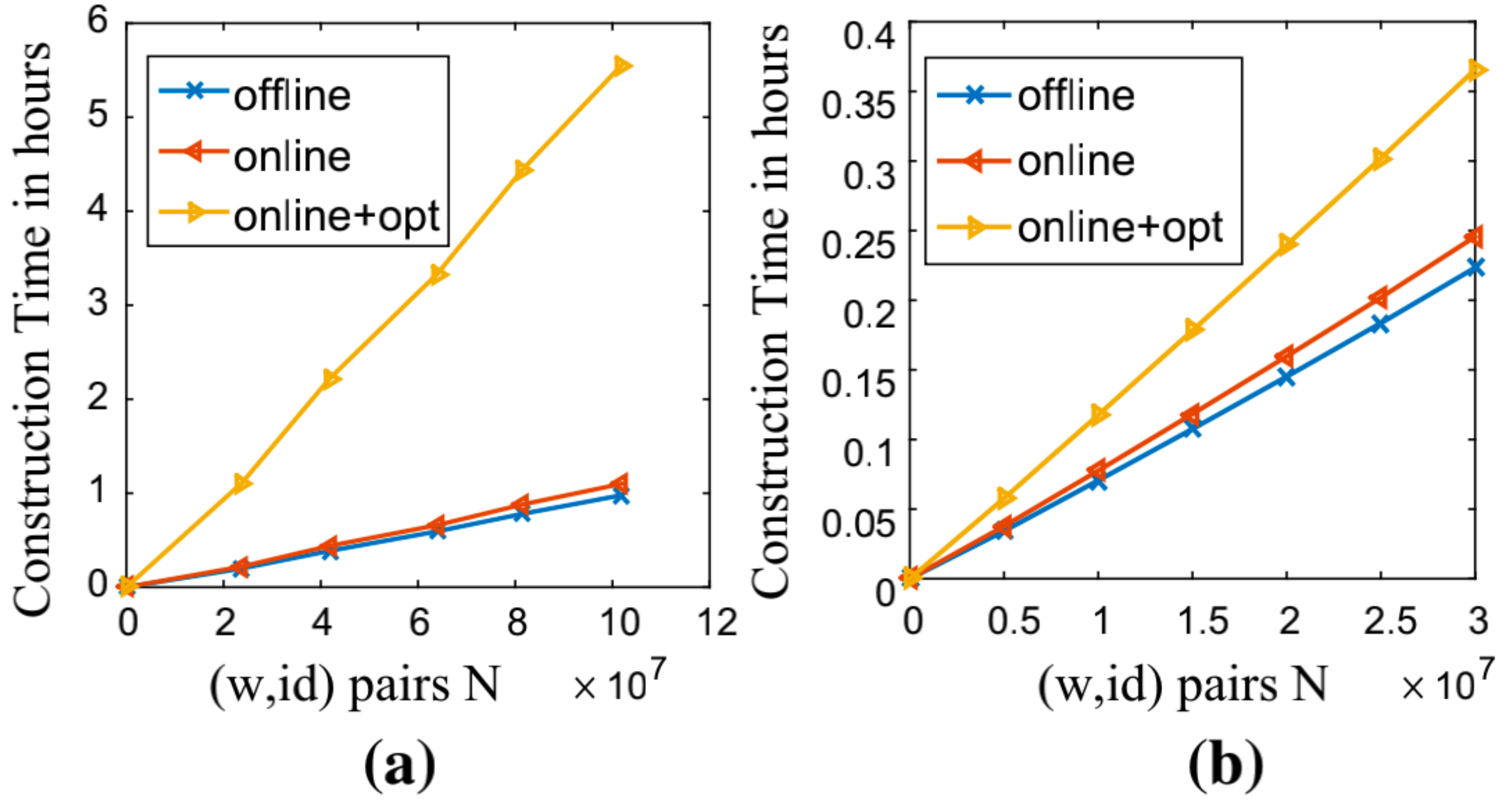


Fig. 4 Off-line and online constructions. **a** Enron dataset, **b** Gowalla dataset

Although the graph-based indexing algorithm is time-consuming, the time is paid for later faster conjunctive queries.

Experiment - conjunctive queries

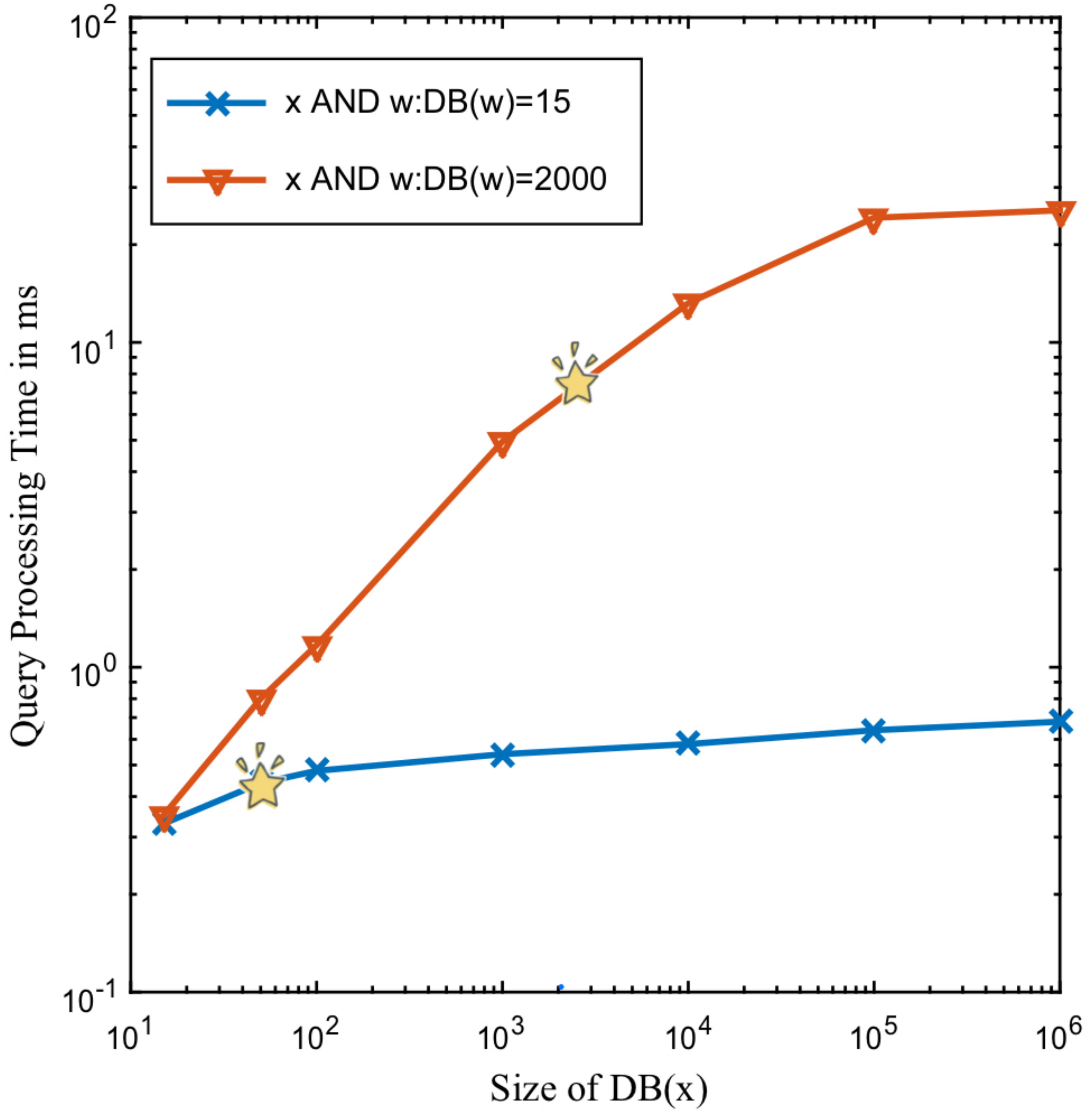


Fig. 5 Two-dimensional conjunctive queries

The two-dimensional conjunctive query time is mainly related to the minimum size of subqueries.

在 $\min_{w\in q}\{|DB(w)|\}$ 值固定之后，查询时间的曲线趋于平缓。

Experiment - optimized queries

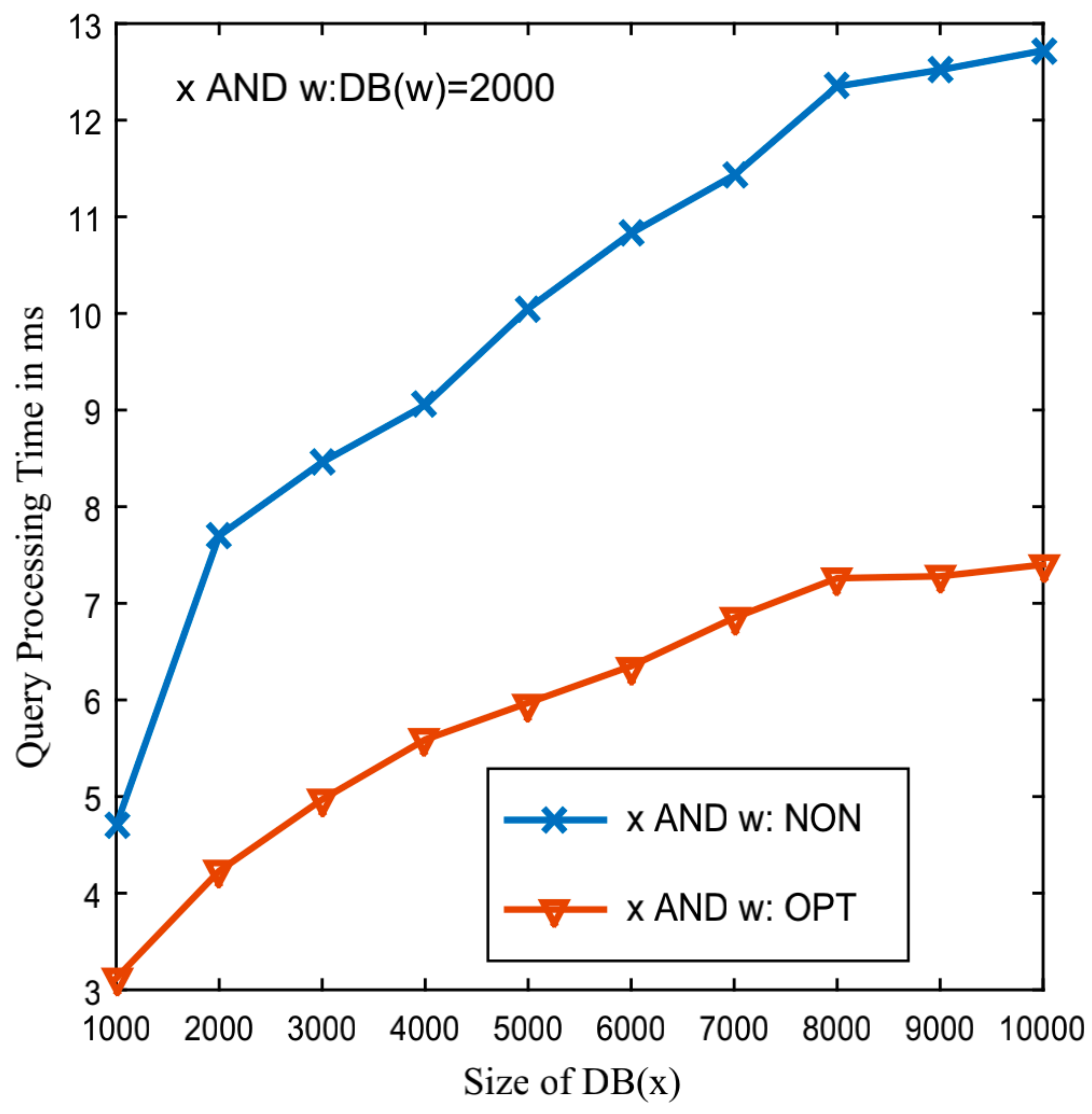


Fig. 6 Optimized two-dimensional queries, Enron dataset

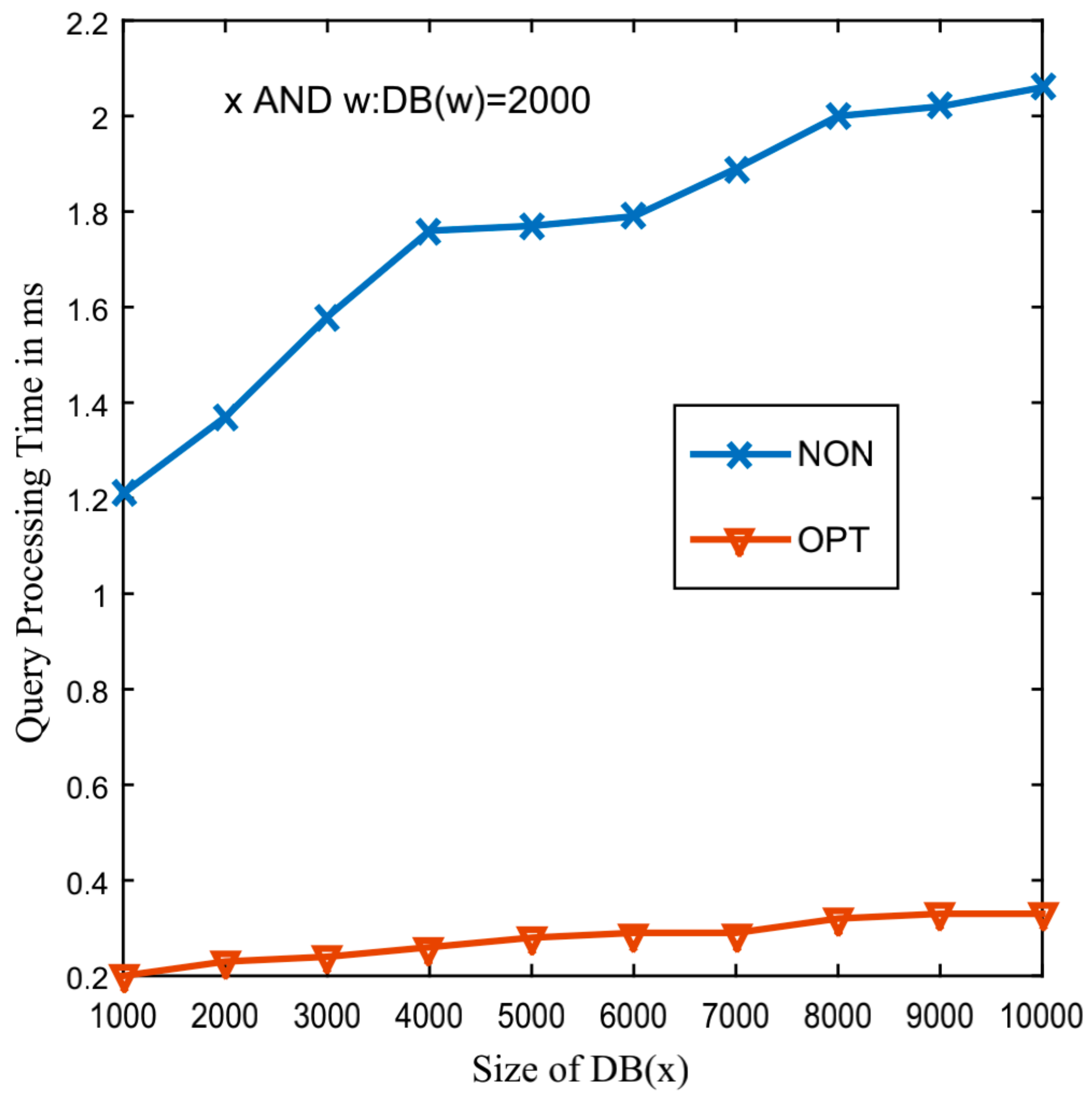


Fig. 7 Optimized two-dimensional queries, Gowalla dataset

The index optimization algorithm can significantly reduce the query time.

Experiment - queries

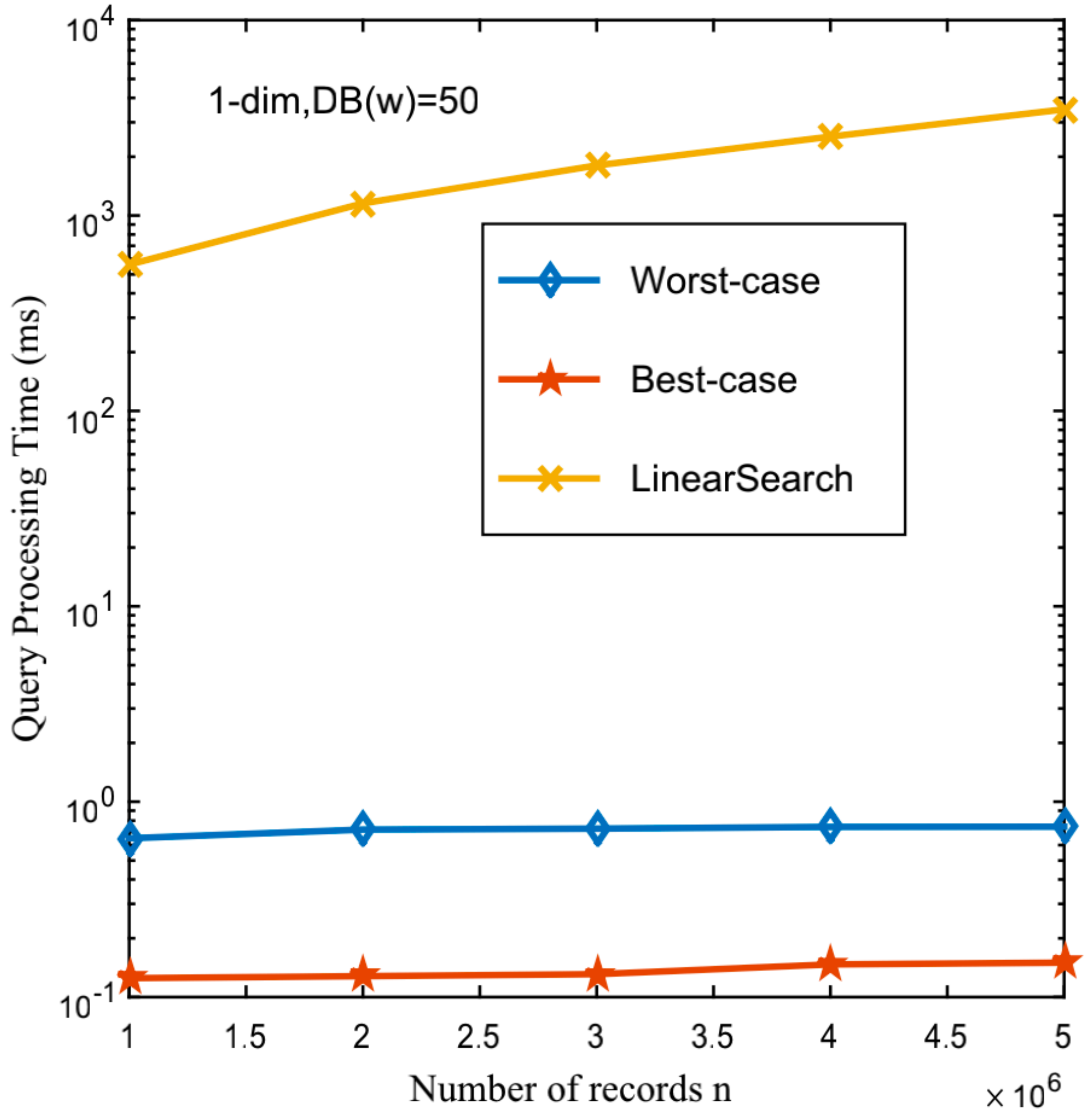


Fig. 8 DB(w) = 50, one-dimensional queries

The linear search is costly compared to the top-down search algorithm.

Experiment - queries

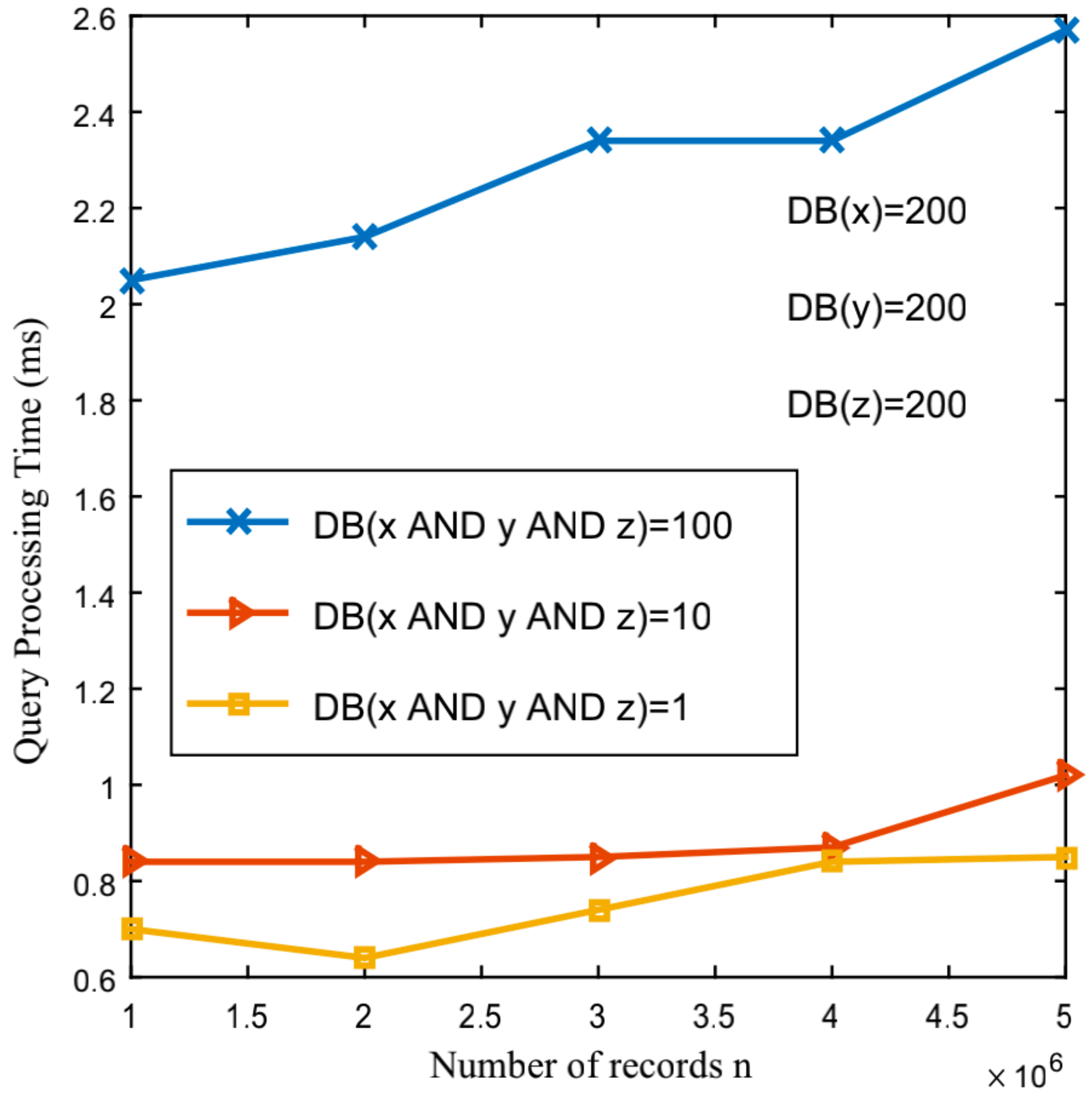


Fig. 9 Three-dimensional queries

A smaller final result size will make the conjunctive query more efficient.

Experiment - update

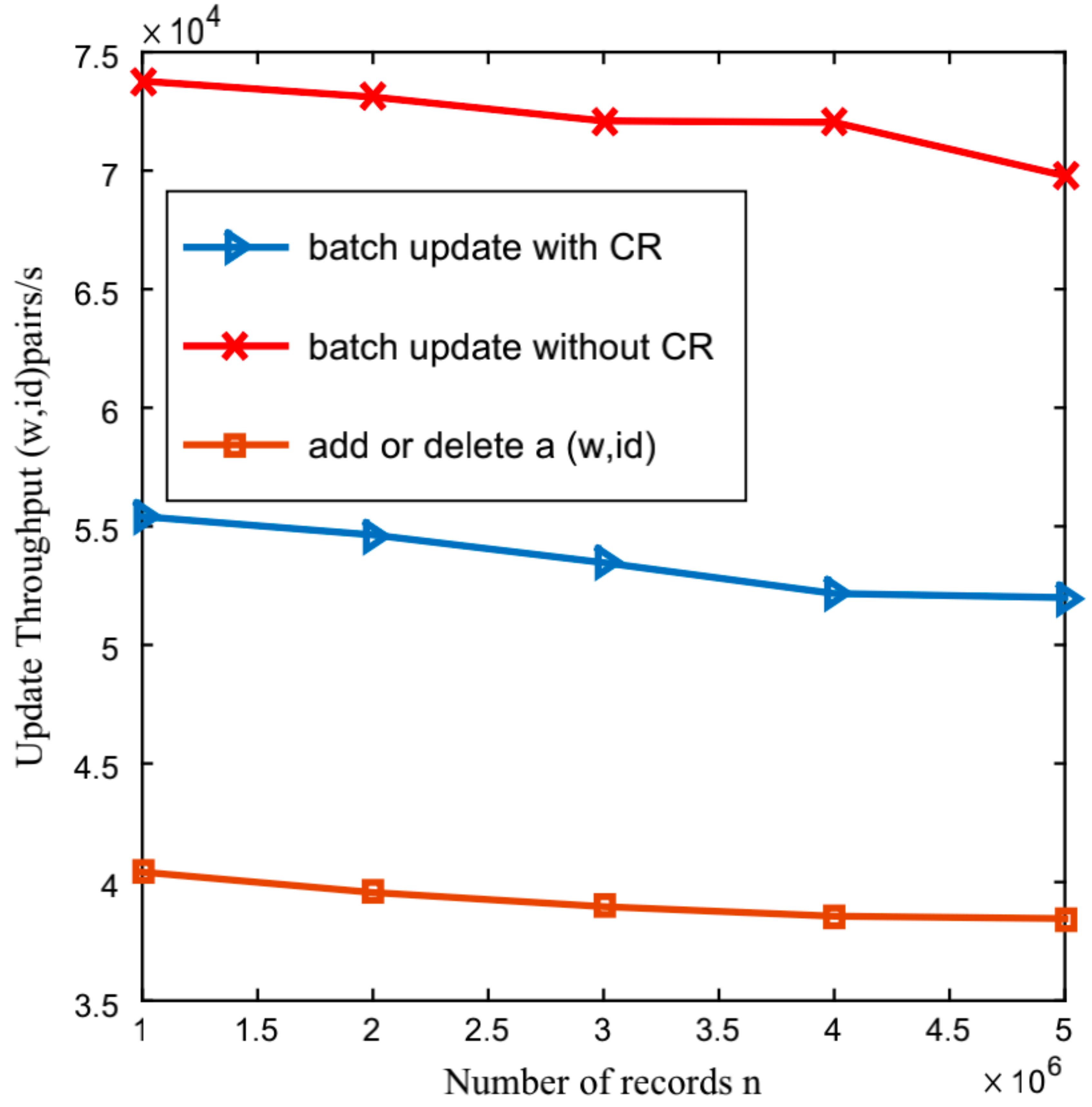


Fig. 11 Update efficiency

"With CR" denotes the update procedure including updating both CR and the tree. Each update to CR means a new version trapdoor generated.

The update complexity of VBTree is sub-linear.

Experiment - compare

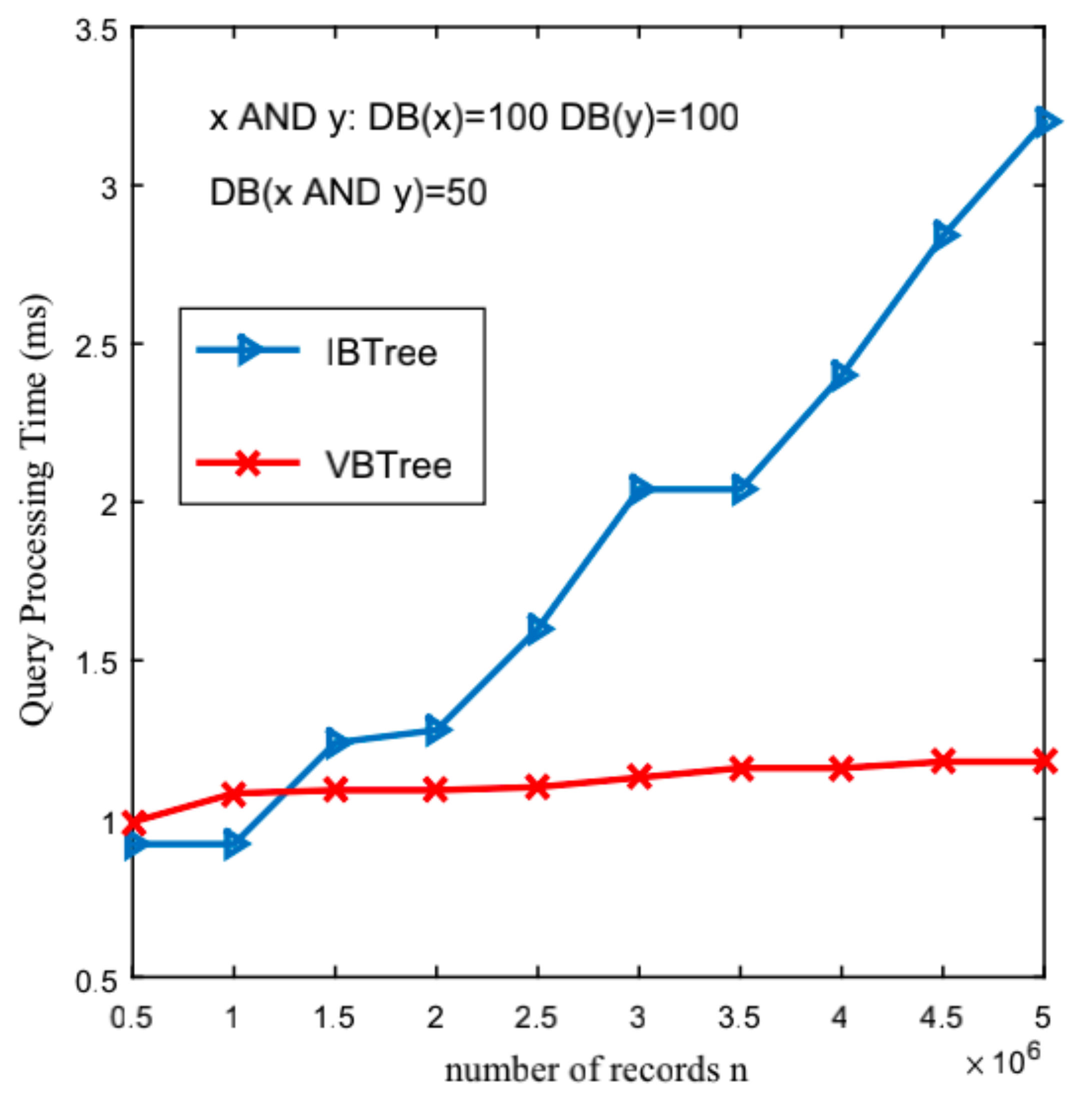


Fig. 13 Compared with IBTree (query time)

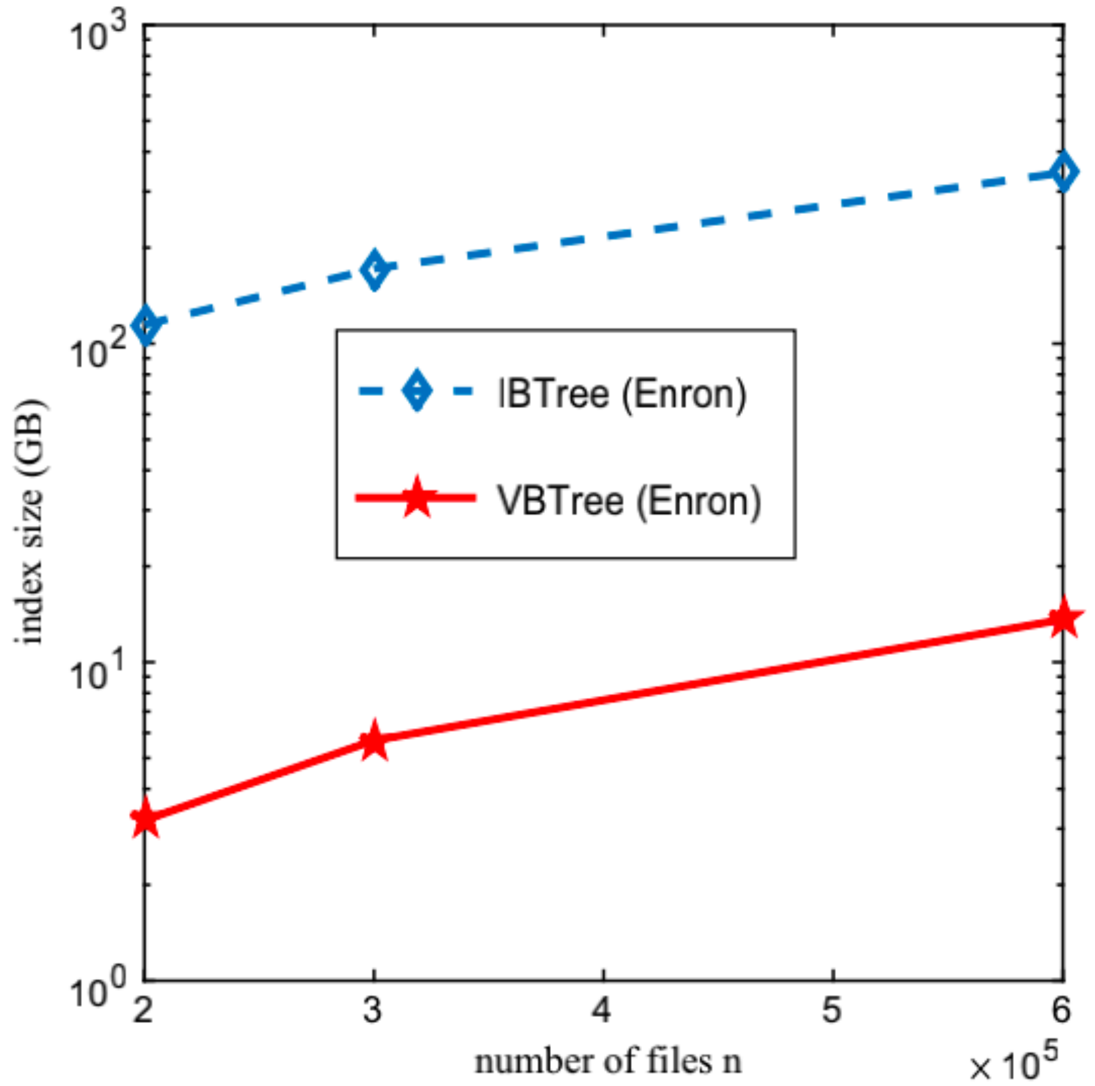


Fig. 14 Compared with IBTree (index size)

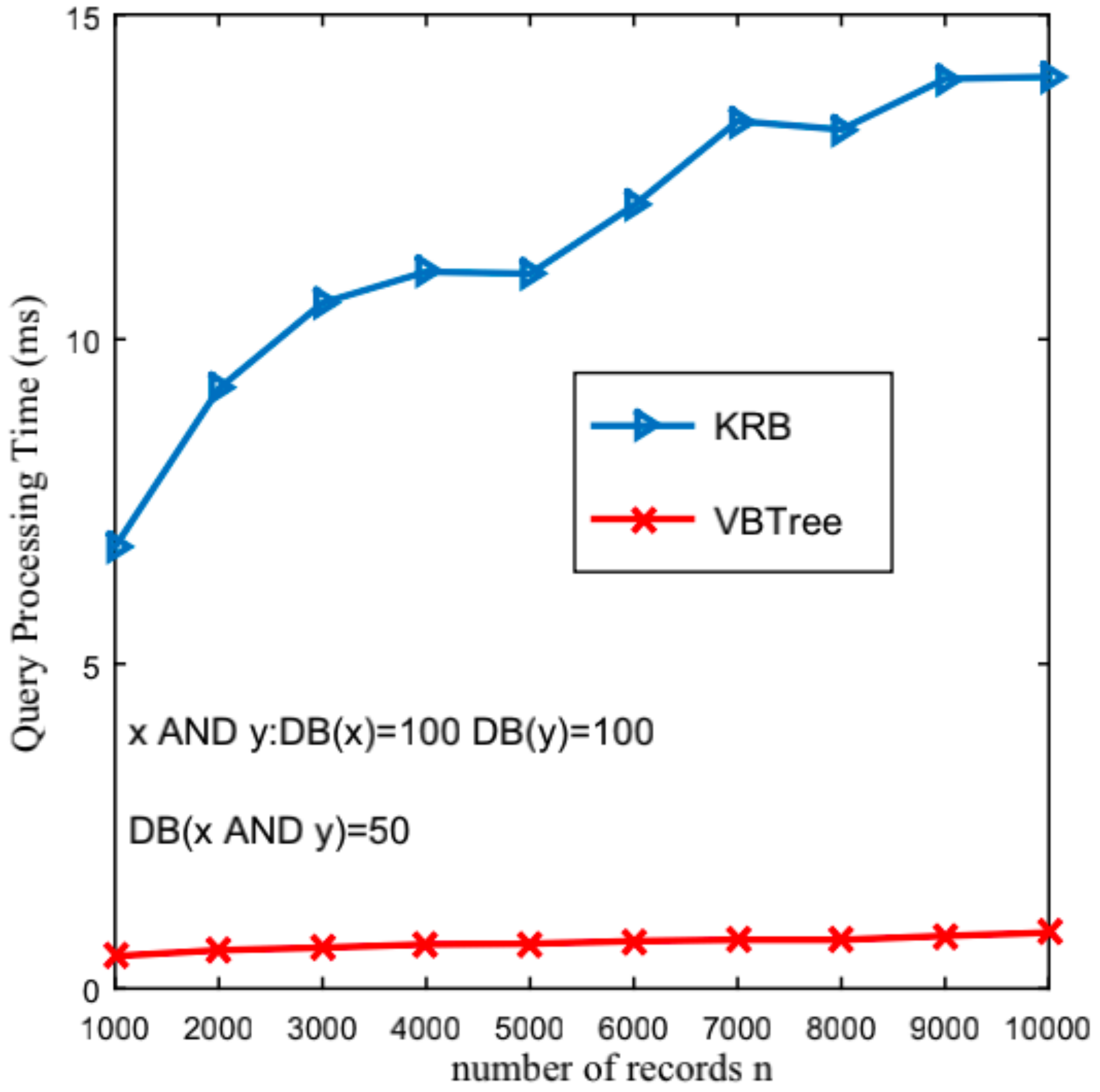


Fig. 15 Compared with KRB (query time)

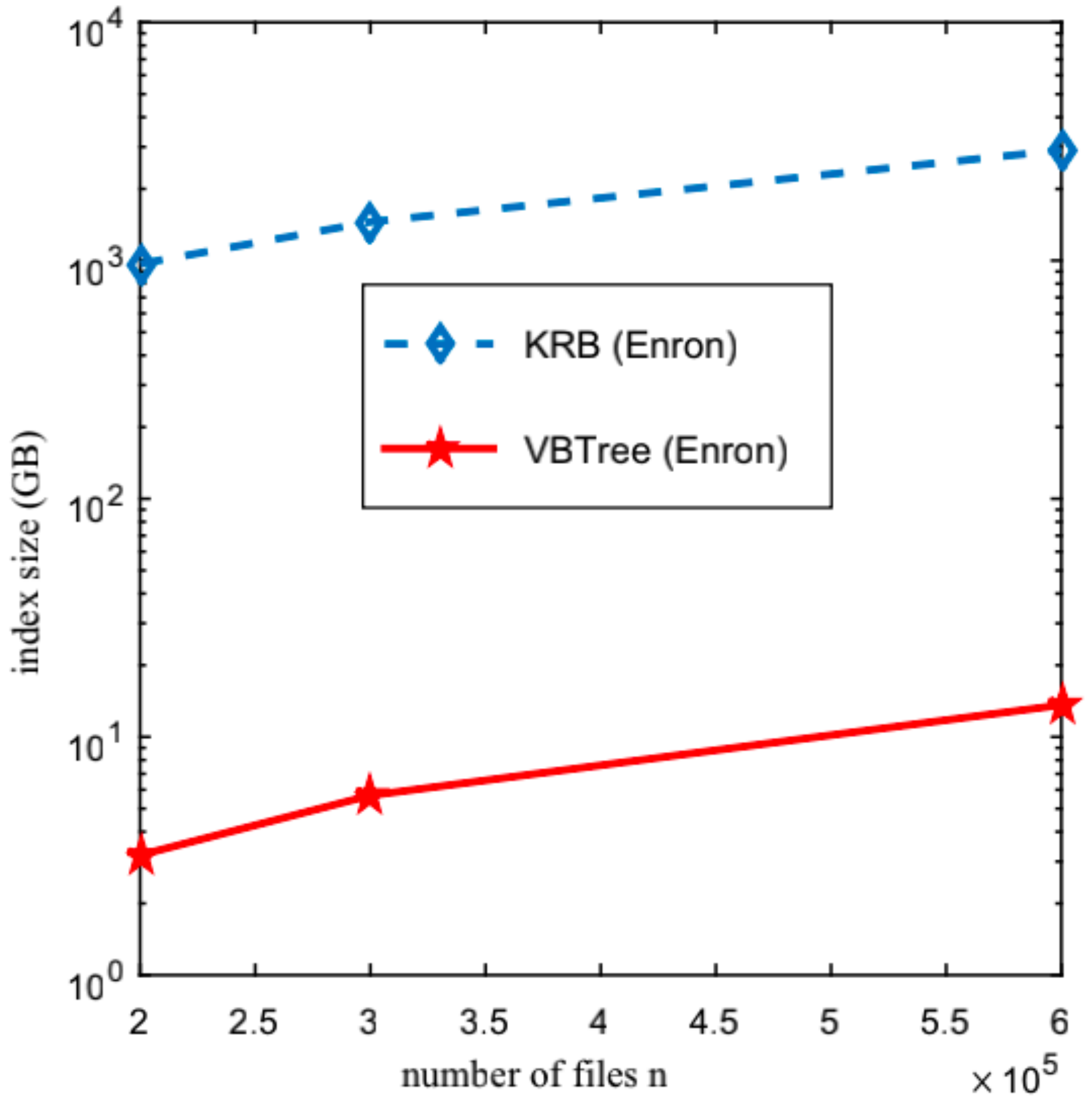


Fig. 16 Compared with KRB (index size)