World Wide Web (2013) 16:141–170
DOI 10.1007/s11280-012-0163-7

# Indexing dataspaces with partitions

**Shaoxu Song · Lei Chen**

Received: 12 December 2011 / Revised: 17 March 2012 /
Accepted: 5 April 2012 / Published online: 1 May 2012

**Abstract** Dataspaces are recently proposed to manage heterogeneous data, with features like partially unstructured, high dimension and extremely sparse. The inverted index has been previously extended to retrieve dataspaces. In order to achieve more efficient access to dataspaces, in this paper, we first introduce our survey of data features in the real dataspaces. Based on the features observed in our study, several partitioning based index approaches are proposed to accelerate the query processing in dataspaces. Specifically, the *vertical partitioning index* utilizes the partitions on tokens to merge and compress data. We can both reduce the number of I/O reads and avoid aggregation of data inside a compressed list. The *horizontal partitioning index* supports pruning partitions of tuples in the top-k query. Thus, we can reduce the computation overhead of irrelevant candidate tuples to the query. Finally, we also propose a *hybrid index* with both vertical and horizontal partitioning. The extensive experiment results in real data sets demonstrate that our approaches outperform the previous techniques and scale well with the large data size.

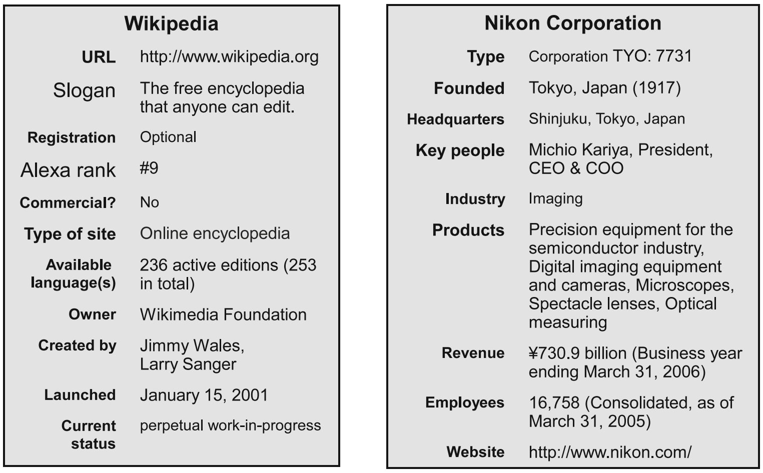
**Keywords** partitioning index · dataspaces

## 1 Introduction

Data records extracted from Web are often heterogeneous and loosely structured [23]. The concept of dataspaces is proposed in [16, 17], which provides a co-existing

S. Song (✉)
Key Laboratory for Information System Security, Ministry of Education;
TNList; School of Software, Tsinghua University, Beijing, China
e-mail: sxsong@tsinghua.edu.cn

L. Chen
Department of Computer Science and Engineering, The Hong Kong University of Science
and Technology, Clear Water Bay, Hong Kong
e-mail: leichen@cse.ust.hk



| Wikipedia | |
|---|---|
| URL | http://www.wikipedia.org |
| Slogan | The free encyclopedia that anyone can edit. |
| Registration | Optional |
| Alexa rank | #9 |
| Commercial? | No |
| Type of site | Online encyclopedia |
| Available language(s) | 236 active editions (253 in total) |
| Owner | Wikimedia Foundation |
| Created by | Jimmy Wales, Larry Sanger |
| Launched | January 15, 2001 |
| Current status | perpetual work-in-progress |

| Nikon Corporation | |
|---|---|
| Type | Corporation TYO: 7731 |
| Founded | Tokyo, Japan (1917) |
| Headquarters | Shinjuku, Tokyo, Japan |
| Key people | Michio Kariya, President, CEO & COO |
| Industry | Imaging |
| Products | Precision equipment for the semiconductor industry, Digital imaging equipment and cameras, Microscopes, Spectacle lenses, Optical measuring |
| Revenue | ¥730.9 billion (Business year ending March 31, 2006) |
| Employees | 16,758 (Consolidated, as of March 31, 2005) |
| Website | http://www.nikon.com/ |

**Figure 1** Example of 2 tuples from Wikipedia.

system of heterogeneous data. Instead of traditionally integrating the heterogeneous data into an uniform interface in advance, dataspaces offer a best-effort integration in a *pay-as-you-go* style [21, 27, 30]. In particular, the semantic schema of dataspaces may not be fully explored or identified. Therefore, dataspace systems rely on schema-less services to access the data before identifying semantic relationships [14]. Examples of these interesting dataspaces are now prevalent, especially on the Web [18].

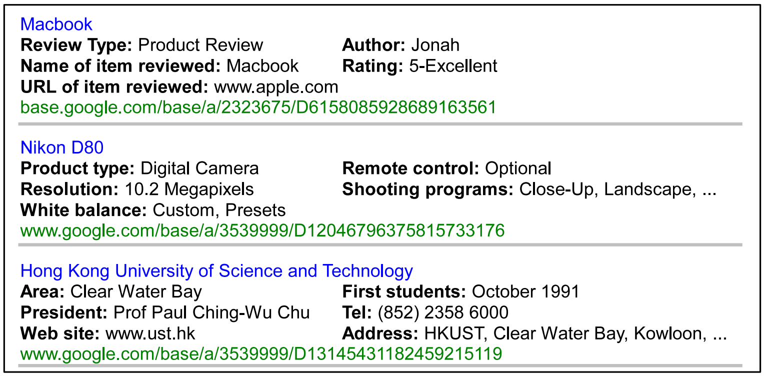
In Wikipedia,[1] each article usually has a tuple with some attribute value pairs to describe the basic structured information of the entry. For instance, as shown in Figure 1, a tuple describing the Nikon Corporation may contain attributes like {**founded:** Tokyo Japan (1917); **industry:** imaging; **products:** cameras ...}. The attributes of tuples in different entries are various (i.e., heterogeneous), while each tuple may only contain a limited number of attributes (i.e., sparse). Thus, all these tuples form a huge dataspace in Wikipedia.

Given another example, Google Base[2] is a *very large, self-describing, semi-structured, heterogeneous* database. As shown in Figure 2, each tuple consists of several attribute value pairs and can be regarded as a tuple in the dataspaces. According to our observations (see Section 2), there are total 5,858 attributes in 307,667 tuples (random sample items), while most of these tuples only have less than 30 attributes individually. Therefore, the dataspace is extremely sparse.

The traditional tables in a relational database mainly focus on specific domains, with a limited number of attributes or columns, while dataspaces are in universal agents like the Web, with loose or even no limitations on the attributes. Therefore,

[1] http://www.wikipedia.org/

[2] http://base.google.com/



Macbook
**Review Type:** Product Review **Author:** Jonah
**Name of item reviewed:** Macbook **Rating:** 5-Excellent
**URL of item reviewed:** www.apple.com
base.google.com/base/a/2323675/D6158085928689163561

Nikon D80
**Product type:** Digital Camera **Remote control:** Optional
**Resolution:** 10.2 Megapixels **Shooting programs:** Close-Up, Landscape, ...
**White balance:** Custom, Presets
www.google.com/base/a/3539999/D12046796375815733176

Hong Kong University of Science and Technology
**Area:** Clear Water Bay **First students:** October 1991
**President:** Prof Paul Ching-Wu Chu **Tel:** (852) 2358 6000
**Web site:** www.ust.hk **Address:** HKUST, Clear Water Bay, Kowloon, ...
www.google.com/base/a/3539999/D13145431182459215119

**Figure 2** Example of 3 tuples in Google Base.

comparing with the traditional tables, we intuitively conjecture the characteristics of dataspaces.

- *Heterogeneous*. Rather than a specific domain, the items come from universal areas. The contents, e.g., attributes and values in the tuples, are various.
- *Sparse*. Although the entire dataspace is in a high dimensions space of attributes, each single tuple may only have a very small set of attributes.
- *Large scale*. The data might be contributed from the Web around the world. Up till 2008, there have been 2,330,427 articles in English Wikipedia.

The queries over dataspaces can be abstracted to two typical operators, i.e., AND / OR queries. The AND query returns the candidate answers that satisfy *all* the query predicates. For example, a query may specify predicates like industry $=$ imaging and products $=$ cameras. The returned tuples should contain these two values in the corresponding attributes, respectively. Moreover, the query over dataspaces may not only searches the specified attributes, e.g., the Website attribute in a query Website=www.wikipedia.org, but also the heterogeneous attributes correlated to the specified attributes,[3] such as URL. That is, the query returns tuples with either Website=www.wikipedia.org or URL=www.wikipedia.org. This OR query returns the candidate answers that may only satisfy *some* predicates. Finally, the returned candidate tuples are then ranked according to their (similarity) scores to the query. We have previously studied the optimization of dataspace queries, e.g., by using materialization [32] or semantic data dependencies [31]. As a complementary aspect of query optimization, in this paper, we focus on the indexing of dataspaces.

According to our survey in real data in Section 2, the attribute frequency in the dataspaces follows the Zipf law [37] like distribution. This interesting observation motivates us to explore the successful inverted index [7, 33, 35, 38, 39] in information retrieval to manipulate the dataspaces. As presented in the previous indexing of dataspaces [14], each pair of attribute label and value maps to a token, then the inverted index can be extended to manipulate dataspaces. However, this basic

[3]As mentioned, the correlations of heterogeneous attributes are often identified by pay-as-you-go approaches [21, 27, 30], which is out the scope of this study.

technique, which also serves as a baseline approach in this study, falls short in the efficiency consideration. Given a query, all the referred tuple lists have to be merged for ranking, which is quite expensive in terms of I/O and CPU cost. In fact, the main bottleneck of inverted list like index is the time cost to merge the tuple lists of large sizes [29]. Rather than the whole bunch of result tuples, a typical query may only be interested in the top-k answers in the real application. Thus, the aggregating and ranking for those low score tuples are wasting.

To address efficient accessing of dataspace, we develop a framework of indexing on partitioning. Our contributions in this study are organized as follow:

*First* We propose a *vertical partitioning index*. According to our observations in the real data, we find that some attributes always appear together in the same tuples. In other words, tokens in these attributes share similar inverted lists of tuples. We can merge the similar tuple lists of these tokens as compression. When a query with these tokens comes, we can directly utilize the compressed tuple lists. Following the convention of representing tuples by token vectors, the groups of tokens in certain compressed lists indeed form the partitioning over the token vector space. Thereby, we name this token relationship based approach *vertical partitioning index*. We benefit from this technique in the following aspects: the storage is saved by merging similar tuple lists; the number of I/O reads is reduced (e.g., two query tokens refer to one same vertical partition list); the cost is avoided for merging the lists of two query tokens in a same vertical partition list.

*Second* We propose a *horizontal partitioning index*. Recall that the previous index [14] cannot tell the most relevant answers until aggregating all the tuples in the inverted lists that are referred by query tokens. The top-k answers are the tuples similar to the query with the highest ranking scores, while most of the remaining candidate tuples are irrelevant to the query with low scores. Our approach is dedicated to reduce the computation of these irrelevant candidate tuples. According to our survey in real data, the tuples in a specific domain usually share similar contents, while the contents of tuples in different domains are various. Thus, we can partition the tuples into groups of similar contents. Given a query, we can find an upper-bound of ranking scores for the tuples in each partition. Then, we first process the partitions with the highest probability of containing top-k answers. If the current top-k answers already have higher scores than the upper-bounds of the remaining partitions, then these partitions of tuples can be safely pruned. Since this technique partitions tuples, we name this tuple relationship based approach *horizontal partitioning index*.

*Third* We propose a *hybrid index* which cooperates with both the vertical and horizontal partitioning strategies. Specifically, we present a framework for indexing based on both token and tuple partitions. Then, the query processing can conduct pruning on the horizontal partitions, and utilize the compressed vertical partition lists in each horizontal partitions.

*Finally* We report the experimental study on evaluating the proposed approaches. By using two real datasets, Google Base and Wikipedia, we compare the proposed approaches of vertical partitioning index, horizontal partitioning index, hybrid index, as well as the previous index without partitioning [14] as the baseline approach. Our approaches, especially the hybrid index, outperform the previous technique and scale well with large data sizes.

The rest of this paper is organized as follows. Section 2 introduces the basic index on dataspaces and presents our observations of data features in the real dataspaces. In Section 3, we present the vertical partitioning index based on the token partitions. Section 4 introduces the horizontal partitioning approach and discusses the query processing with efficient pruning on the tuple partitions. We present the hybrid index in Section 5. Section 6 reports the experimental results of our approaches. Finally, we discuss the related work in Section 7 and conclude this paper in Section 8.

## 2 Basic index and observations

In this section, we discuss the features of dataspaces, and introduce the basic framework for indexing and querying on the sparse data. More importantly, we also present advanced observations on the real data to further motivate the efficient indexing.

### 2.1 Sparse features

We use the real data, Google Base, to illustrate the sparse features. In this test case, there are total 5,858 attributes observed in 307,667 tuples (items). However, most of these tuples only have less than 30 attributes, thus, the table is extremely sparse. Specifically, in Figure 3, we present the statistics of attribute frequency in the dataspaces. The Y-axis denotes the number of tuples that contains a certain attribute $x$, i.e. the frequency of attribute $x$. The X-axis represents all the attributes, which are ranked according to their corresponding frequency values.

As shown in Figure 3, there are only about 10 attributes that appears in more than 1%($10^{-2}$) tuples. In other words, for the rest huge amount attributes, the frequencies of appearance in tuples are very low, for example, less than 0.1% ($10^{-3}$) for those attributes ranked start from $10^2$ to 5,858. In other words, most of the attributes

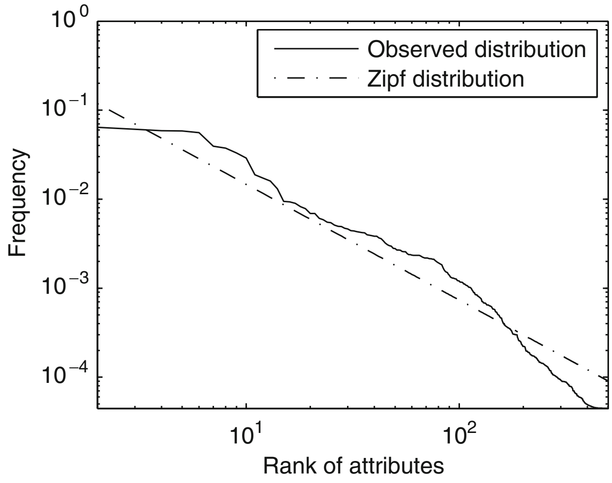


**Figure 3** Attribute frequency distribution.

appear in a small number of tuples. This observation demonstrates the extreme sparse characteristic in the real dataspaces.

Moreover, according to the observation in Figure 3, the attribute frequency distribution follows the Zipf law [37] like style. This interesting observation motivates us to explore the successful inverted index [7, 22, 25, 33, 35, 38, 39] used in information retrieval to manipulate the dataspaces.

## 2.2 Indexing

### *2.2.1 Data modeling*

To extend inverted index for dataspaces [14], the attribute label and value data are encoded by tokens. Specifically, we use a pair of (attribute label, attribute value), denoted by $(a, v)$, to represent the content of a tuple. Thus, a tuple in dataspaces can be represented by a vector

$$\{(a_1, v_1), (a_2, v_2), \ldots, (a_{|t|}, v_{|t|})\}. \tag{1}$$

For example, {area: clear water bay} can be represented by {(area:clear), (area:water), (area:bay)}, if the word token is considered. Let token $u$ be a unique identifier of the pair $(a, v)$, e.g., can be simply the string area:bay. Then, the tuple is actually a vector of tokens

$$\{u_1, u_2, \ldots, u_{|\mathbf{D}|}\}. \tag{2}$$

where the domain $\mathbf{D}$ denotes all the unique identifiers of distinct $(a, v)$ pairs in the whole dataset.

**Definition 1** (tuple vector) We use the vector space model [28] to represent the attribute label and value appearing in a tuple $t$.

$$t = \left(w_1, w_2, \ldots, w_{|\mathbf{D}|}\right), \tag{3}$$

where $w_i$ denotes the weight of token $u_i$ in the dataset.

Here, the weight $w_i$ of token $u_i$, i.e., pair $(a_i, v_i)$, can be the number of occurrences in the tuple $t$, i.e., *term frequency*. Advanced weight schemes, such as *idf* [28] in information retrieval, can also be applied.

### *2.2.2 Inverted index*

The *inverted index*, also known as *inverted files* or *inverted lists*, consists of a vocabulary with domain $\mathbf{D}$ and a set of inverted lists. Each token $e$ corresponds to an inverted list of tuple IDs, where each ID reports the occurrence of token $e$ in that tuple.

Since the tuples in dataspaces are transformed into vectors of tokens, the inverted index is applicable [14]. As shown in Figure 4, we use an example to illustrate the index framework. The dataset consists of 11 tuples with the vocabulary domain size $|\mathbf{D}| = 12$. In the inverted lists, for each token (an attribute label and value pair), we have a pointer referring to a specific list of tuple IDs, where the token appears. For instance, Figure 4b shows an example of the inverted lists of token d, which indicates that the token d appears in the tuples 2, 3, 5, 8, 11. In the real implementation, each

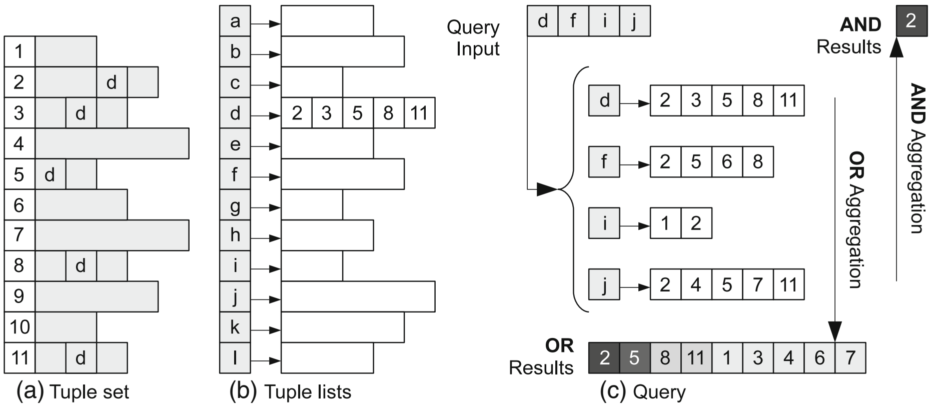


**Figure 4** Indexing dataspaces.

tuple ID in the list is associated with a value, which denotes the weight of the token d in that tuple.

## 2.3 Query processing

In this paper, we consider the queries that are also a set of attribute label and value predicates, e.g., industry = imaging, products = cameras. Thus, the query inputs can be represented in the same way as we model the tuples in dataspaces.

We mainly consider two logical connective conditions, i.e., the AND condition query and the OR condition query. Given a query $Q$, for example $Q$={d, f, i, j}, we want to find the tuples containing all the tokens (AND condition) or tuples containing the most tokens (OR condition). As shown in Figure 4c, we first load all the inverted lists corresponding to the query tokens. For the OR aggregation, tuple 2 has the highest frequency of 4 in the query $Q$, and is ranked first in the results (assume each token with weight 1). For the AND query, there is only one tuple 2 that contains all the query tokens and will be ranked in the first place in the returned result.

### *2.3.1 Ranking function*

During the aggregation of all the tuple lists referred by the query, by the AND or OR condition, the candidate tuples are ranked according to their ranking scores to the query. In this work, we evaluate the ranking score based on the content of matching tokens, i.e., (attribute label, attribute value) pairs, between the query and tuple. We use the ranking function as follows.

**Definition 2** (ranking function) Consider any tuple $t = \{w_1, w_2, \ldots, w_{|\mathbf{D}|}\}$ and the query $Q = \{q_1, q_2, \ldots, q_{|\mathbf{D}|}\}$ with domain $\mathbf{D}$. The *score* of the *ranking function* is defined by the intersection of these two vectors.

$$score(t, Q) = \sum_{1 \leq i \leq |\mathbf{D}|} \min(t.w_i, Q.q_i). \tag{4}$$

where $w_i$ and $q_i$ denote the weight of token $u_i$ in the tuple $t$ and the query $Q$ respectively.

### *2.3.2 Algorithm without partitioning*

The first algorithm evaluates the query directly on the basic index without partitioning [14]. As shown in Algorithm 1, given a query $Q$, we first read all the inverted tuple lists corresponding to $Q$ by the function TUPLE-LISTS($Q$). Let $T$ be all the tuple lists referred by the query as candidates. For each tuple list $T_i.list$, we merge and aggregate all the tuples in the list with their corresponding weights, and rank the scores in the current top-k answers. Note that in this algorithm, we cannot find out which tuple will be ranked high in the final result until all the candidate tuples are aggregated and ranked.

**Algorithm 1** Top-*k* query without partitioning

```
1: procedure QUERY($Q, k$)
2:     $T$ = TUPLE-LISTS($Q$)
3:     $K = \emptyset$
4:     for $i \leftarrow 1, T.size$ do
5:         $K$=AGGREGATE($K, T_i.list$)
6:     return $K$
```

The only difference between the AND query and the OR query is the different version of AGGREGATE() function. The OR query aggregates any tuple that appears in any list in $T$, while the AND query only aggregates the tuple appearing in all the tuple lists in $T$. Once the candidate tuples are generated, the same ranking function is applied to these candidates in both AND and OR queries.

### *2.3.3 Problems*

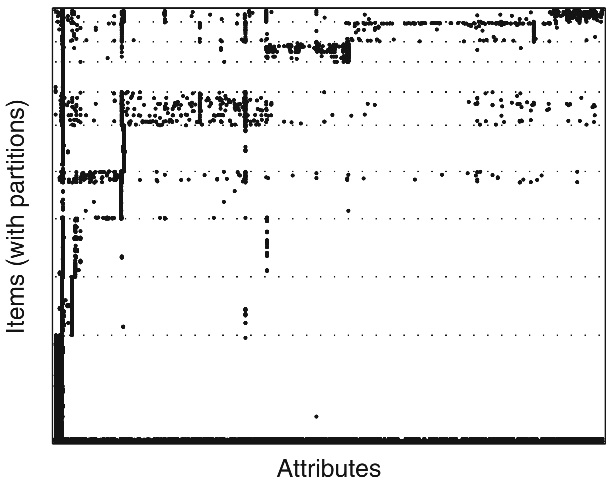
In this framework, with the increase in data sizes, the lengths of the inverted lists will be large, which will increase the high cost in merging. In addition, this basic framework does not take the user's interests into consideration. For example, very often, users may only be interested in the top-k ranked answers to their queries. However, the basic framework cannot tell the most relevant answers directly.

Therefore, the straight forward question is whether we can predicate some important candidates, or whether we can prune some irrelevant tuples to avoid aggregating the entire inverted lists. It is difficult to make such a quick judgment by the first glance of the traditional full text data, where the data are totally unstructured. Thus, we study the advanced features of distributions in the real dataspaces and utilize them to construct efficient index.

## 2.4 Observations

We study the features and patterns existing in the real dataspaces, Google Base. First, we observe the attribute distribution on different categories of tuples. As shown in Figure 5, the X-axis denotes all the attributes. A point ($x$, $y$) in the figure denotes the attribute $x$ appears in the tuple $y$. Then, each tuple is represented by a vector of attributes (feature points) in the figure. The tuples between two horizontal dash lines

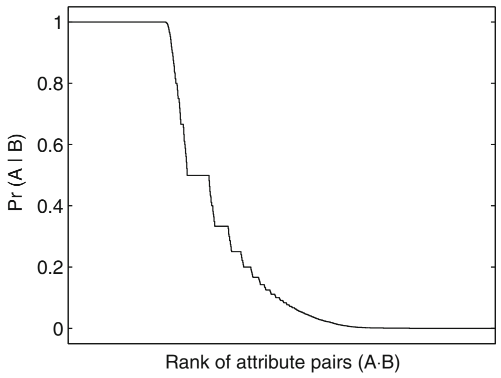
**Figure 5** Item attribute distribution.



have the same category. This observation illustrate two important hints in studying the dataspaces: first, *the tuples within the same category shares similar attributes*; second, *the attributes in different categories are often different from each others.*

The interesting block structures arouse our curiosity on the relationship between the attributes. Thus, we partition all the attributes into attribute pairs. In Figure 6, the X-axis denotes all the possible pairs of attributes. For each attribute pair ($A \cdot B$), we study the conditional probability of attribute $A$ on $B$, that is, $\Pr(A \mid B) = \frac{\Pr(A \cdot B)}{\Pr(B)}$. The semantic meaning is the probability of $A$'s appearance if $B$ has appeared in a tuple, which can be regarded as the relationship between attribute $A$ and $B$. The attribute pairs are sorted according to their conditional probability values. In Figure 6, there are about 20% attribute pairs with the probability of 1. In other words, all these attributes always appear together in the same tuples. This property could be utilized in the compressing of dataspace index. Moreover, based on those pairs with high probability of appearing together, we might introduce heuristic strategies in accelerating the dataspace retrieval.

**Figure 6** Attribute pair distribution.



## 3 Vertical partitioning index

According to the above observation, the tuples in the same domains might share similar contents of describing tokens (attributes and values). In the view of tokens, we can utilize the token relationships of appearing together. Specifically, we consider two tokens $u_1$ and $u_2$. Suppose that we observe these two tokens appearing together frequently in the tuples. Now, a query with $\{u_1, u_2\}$ is posed. Recall that each tokens $u$ corresponds to an inverted list in the disk. During the query processing, we have to read and merge these two inverted lists, to get the ranked tuple answers. Based on the observation of $u_1$ and $u_2$, we can first compress the index by merging parts of the results in order to accelerate the query processing. Following the vector space based description of tuples as shown in Figure 4a, the tokens $\{u_1, u_2\}$ form a vertical partition of tokens in the token vector space. We name these token relationship based approaches *vertical partitioning*.

### 3.1 Indexing

#### *3.1.1 Compression*

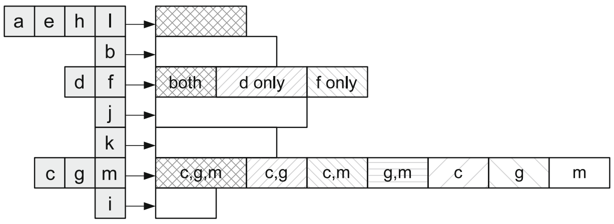
Recall that the heavy cost arises from two aspects during the query processing: first, the I/O cost of reading inverted lists from disks; second, the aggregation of these inverted lists for ranking results. However, there are many tokens that always appearing together, that is, $P(A \mid B) = 1$. Therefore, rather than separately storing their corresponding inverted lists with duplicates, we can compress the index.

For example, in Figure 7, suppose that the tokens a, e, h, l always appear together in the same tuples. Thus, the inverted lists of these tokens are exactly the same, i.e., duplicates. In the compressed index, all these four tokens point to the same inverted list. Note that each inverted list is stored in the continuous blocks in the disk. Without compression, we need three random disk accesses separately for the query {e, h, l}. In the compressed index, only one random disk access is needed for the same query. Moreover, besides reducing the I/O cost, we also avoid the aggregation of these four inverted tuple lists in the query.

Moreover, instead of exactly the same appearance, two tokens may also share similar inverted lists. We can also compress the index similarly.

For instance, in Figure 7, the inverted lists of token d, f are combined together as a big single list. We have several sections in each vertical partition list, such as the tuple list with both d and f, the tuple list with only d, and the tuple list with only f. Again, when a query covers {d, f}, we can both reduce the I/O cost of random disk access and the aggregation cost of inverted lists. Note that the compression case is

**Figure 7** Vertical partition on tokens.



a special case of materialization, where there is only one section of tuple list whose tuples contain all the tokens in the vertical partition list.

We call these compressed lists, which share the same tokens or almost the same tokens, the *vertical partition lists*.

### *3.1.2 Crack vertical partitioning*

Now the problem is to select the tokens to merge their corresponding tuple lists. Intuitively, the tokens with similar (or same) tuple lists are preferred to be merged to save the space and reduce the aggregation cost in potential queries. In other words, we want to merge those tokens that always appear together in the same tuples. This interesting intuition leads us to the famous frequent itemset mining algorithms [4, 20]. The tuples can be treated as transactions, while each token is an item. Then, we can select the frequent k-itemsets (token sets) with the highest supports as the vertical partitions. It is notable that the indexing requires the partitions to be disjoint. Thereby, we can consider the frequent k-itemsets in the descending order of their supports. In each iteration, we select the currently highest support token set as a valid partition, if it does not overlap with the previously selected partitions; otherwise, discard it. The existing efficient algorithms can be utilized, such as Apriori [4] or FP-growth [20], which are not the focuses of this paper. Note that when query workload is available, the exploration of token relationships can be conducted on the query log as well.

## 3.2 Query processing

Next, we study the matching of tuples to a given query $Q$ in vertical partition lists. In the real implementation, each *vertical partition list* consists of two parts, the *head* and *tuple lists*. As the example in Figure 8, the *head* part stores all the tokens associated to the *vertical partition list*, that is, {c, g, m}. The *tuple lists* part stores all the lists of tuple IDs with various token appearances. We use three binary digits xxx to represent the appearance of these three tokens. Specifically, the list mapping to 010 includes all the tuples with token g but not c or m. The tuples in list 101 contain both token c

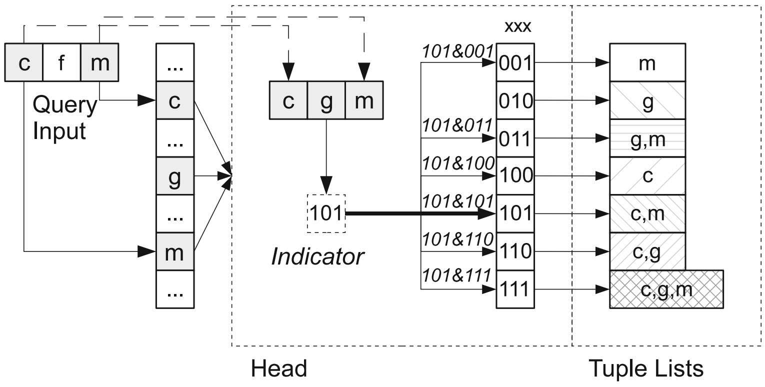


**Figure 8** Matching vertical partition lists.

and m but not f. There may have $2^m - 1$ tuple lists at most in a vertical partition list with *m* tokens.

Given a query, for example *Q*={c, f, m}, we first read the vertical partition lists that are referred by the query. Then, the query is matched with the tokens in the head of the vertical partition list to generate an *indicator*. The indicator denotes all the tokens in the vertical partition list that are referred in the query. For example, the indicator 101 in Figure 8 denotes that token c and m in this vertical partition list are requested in the query. Finally, we return and aggregate those tuple lists matching with the indicator. For the AND query, we return the tuple list with all the tokens represented in the indicator, i.e. xxx&101 $=$ 101. The OR query returns the tuple lists that contains any token in the query, i.e., xxx&101 $> 0$.

### 3.2.1 Algorithm with vertical partitioning

The query algorithm with vertical partitioning index is described in the following Algorithm 2. We consider a query *Q* and an integer *k*. Let $V_i, i \in [1, V.size]$ be all the vertical partition lists referred by the query *Q*. Let $V_i.T_j$, $j \in [1, V_i.size]$ be all the tuple lists in the vertical partition list $V_i$. Let $V_i.T_j.tokens$ denotes the tokens associated to the tuple list $T_j$ in the vertical partition list $V_i$. Those tuple lists whose *tokens* are matched with the query *Q* are aggregated in the results.

**Algorithm 2** Top-*k* query with vertical partitioning

```
1: procedure QUERY($Q, k$)
2:     $V =$ VERTICAL-LISTS$(Q)$
3:     $K = \emptyset$
4:     for $i \leftarrow 1, V.size$ do
5:         for $j \leftarrow 1, V_i.size$ do
6:             if MATCH$(Q, V_i.T_j.tokens)$ then
7:                 $K$=AGGREGATE$(K, V_i.T_j.list)$
8:     return $K$
```

Again, different AGGREGATE() functions are used in AND and OR queries, which are the same as those in Algorithm 1 in the basic inverted index. Moreover, as mentioned, we have different matching condition in AND and OR queries as well. The MATCH() function for AND query can be $Q = V_i.T_j.tokens$, while it is $Q \cap V_i.T_j.tokens \neq \emptyset$ for the OR query.

### 3.2.2 Cost evaluation

Consider a vertical partition list *v* with *M* tokens, which corresponds to *M* inverted tuple lists in the basic approach. Thus, there may be $2^M - 1$ of tuple lists at maximum in *v*. Let *l* be the average length of the tuple lists. Suppose that a query *Q* covers all the tokens in *v*. Then, the cost to aggregate the tuples in *v* can be

$$O(v) = M \cdot l \cdot O(A)\ , \tag{5}$$

where $O(A)$ is the cost of aggregating one tuple ID in the top-k ranking.

Let $B = \{B_i \mid i \in [1, M]\}$ be all the tuple lists corresponding to these *M* tokens in the basic index. According to the definition of vertical partitioning, the vertical partition list *v* is a compression of all the tuples in *B*. Let $v.T_j$ be the *j*-th tuple list in *v* with *m* tokens. Then, $v.T_j$ must be the sub-list of some lists in *B*, that is, existing a $B' = \{B_i \mid v.T_j \subseteq B_i, B_i.token \in v.T_j.tokens, i \in [1, M]\}$ having $|B'| = m$. In other words, each $v.T_j$ maps to $m$ sub-lists in the basic index, where $m$ is the number of tokens associated to $v.T_j$. Thereby, the cost of aggregating these tuples in the basic index can be $l \cdot m \cdot O(A)$. To sum up, the cost of aggregating all the tuples in $v$ in the basic index is

$$O(B) = \sum_{m=1}^{M} l \cdot m \cdot O(A) \cdot C_M^m = M \cdot 2^{M-1} \cdot l \cdot O(A) ,$$

which is higher than the cost of vertical partitioning approach in (5) if $M > 1$. When there is only one token in each vertical partition, i.e., $M = 1$, the vertical partitioning approach is equivalent to the basic inverted index. According to the above cost analysis, our vertical partitioning index can improve the query efficiency, which is also verified in the experimental evaluation in Section 6.

## 4 Horizontal partitioning index

Recall that, in our observations, block style structures exist in dataspaces. The tuples from different domains or categories might have distinct contents of tokens (attributes and values). On the other hand, rather than covering all the universal areas, a certain user query probably concentrates in specific topics in one or several categories. Due to the above mentioned characteristics of dataspaces, during the query processing, there might be some tuples (probably from not so close categories) matching part of the query tuples. The matching size will not be large, thus, these tuples do not rank highly in the results but affect the query efficiency. In this section, we develop an index based on the partitions of tuples, for the efficient query of top-k answers. The proposed techniques are applicable to both the AND / OR queries.

### 4.1 Indexing

Intuitively, the categories or partitions of tuples in the dataspaces can be utilized to prune the unqualified results. That is, among a large number of candidate tuples in the inverted tuple lists, it is desirable to first identify those groups of tuples that are most related to the query. Then, after testing these high probability candidates, we can prune some remaining groups of tuples with low chances to be the answers.

#### *4.1.1 Physical partition*

We partition all the tuples into a set of groups according to the tuples similarities. For example, as shown in Figure 9a, we group the tuple set into four partitions. Let A, B, C, D be the four partitions of the tuple set. For each partition, we build an inverted index individually in Figure 9b. For example, there are five inverted tuple lists, corresponding to tokens a, b, c, e, k, for the tuples in the partition A.

Note that the partitioning of tuples is exclusive. In other words, if a tuple $t$ appears in the tuple lists of the partition A, then this tuple $t$ will never appear in the tuple lists of other partitions again. Based on the exclusive property, we can develop the upper-bound of scores of the tuples in a partition when given a query. Those tuples in the partitions with low upper-bounds of scores are then identified

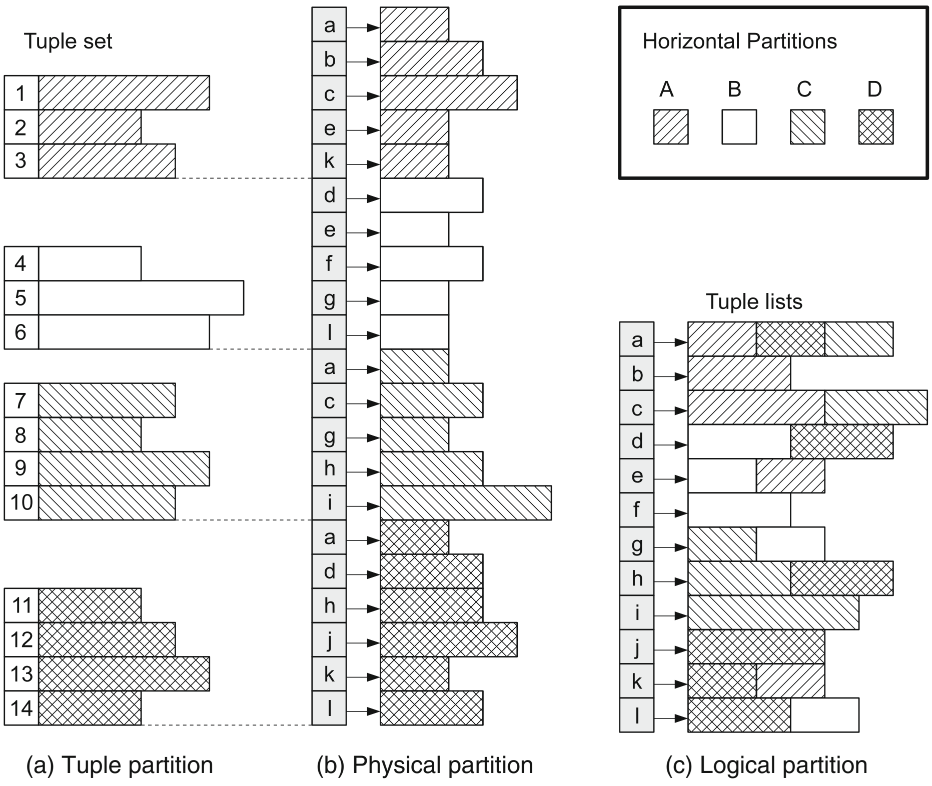


**Figure 9** Horizontal partition on tuples.

as irrelevant answers to the query. After processing some partitions, we can safely prune these partitions whose upper-bounds are lower than the scores of the current top-k answers. Consequently, we implement the intuition of pruning tuples in the horizontal partitioned index.

In real implementation with large size data, the vocabulary of the inverted index is indexed with an efficient access, for example the B$^+$-tree structure. The corresponding inverted tuple list of each token, due to the large sizes, is disk resident. We need one random access of disk block to read the inverted list of a token. When building the inverted index for each partition in Figure 9b, we increase the total number of inverted lists. There are duplicate tokens among different partitions, such as token e in both partition A and B. If the pruning rate is low, then we have to read almost all the tuple lists for one token. That is, the total number of random disk accesses increases, thus the query processing cost is increased as well. In order to reduce the disk access costs, we propose the following advanced partitioning strategy.

### *4.1.2 Logical partition*

In order to address the above problem of duplicate tokens in different partitions, we do not partition and store the inverted index physically for each partition. Instead, as shown in Figure 9c, we partition the tuples inside the inverted lists. Then, given a query token, we do not read the inverted lists for different partitions separately. By using the logical partition strategy in Figure 9c, the tuples are still partitioned with the same groups. But for each token, we only need to read the random disk address once. This is the reason we call this partition the logical implementation of the physical partition.

Each inverted list is stored in sequential disk blocks when the list is larger than the block size. Therefore, even when the inverted list is larger than one disk block, the costs of sequential read in logical partitions are smaller than the random seeks of physical partitions in the disk. The pruning still works to avoid aggregating those tuple lists that are located in not-so-relevant partitions.

### *4.1.3 Hierarchy implementation*

Given a query, now the problem is how to quickly determine the partitions having the highest relevance to the query. The brute force approach is to compare the query input with each partition, and pick the partition with the highest upper-bounds of tuple scores in each step. However, if the number of partitions is much larger than the query inputs, $m \gg |Q|$, it is inefficient to compare the partitions one by one.

Therefore, we build an index over both the partitions and tuples, namely *horizontal partition lists*. There are two parts in each horizontal partition list, the *head* and the *tuple lists*. Since the number partitions are less than the original number of tuples, thus, as shown in Figure 10, we store the inverted partition lists in the head part for efficient partition selection before conducting the query on the inverted tuple lists. In fact, each partition $P$ in the inverted partition lists of a token $e$ maps to a subsequence of the corresponding inverted tuple lists (i.e., all the tuples in the partition $P$ with the occurrence of token $e$). Furthermore, each partition in the partition list is also associated with a weight of the corresponding token in this partition for the computation of score bounds.

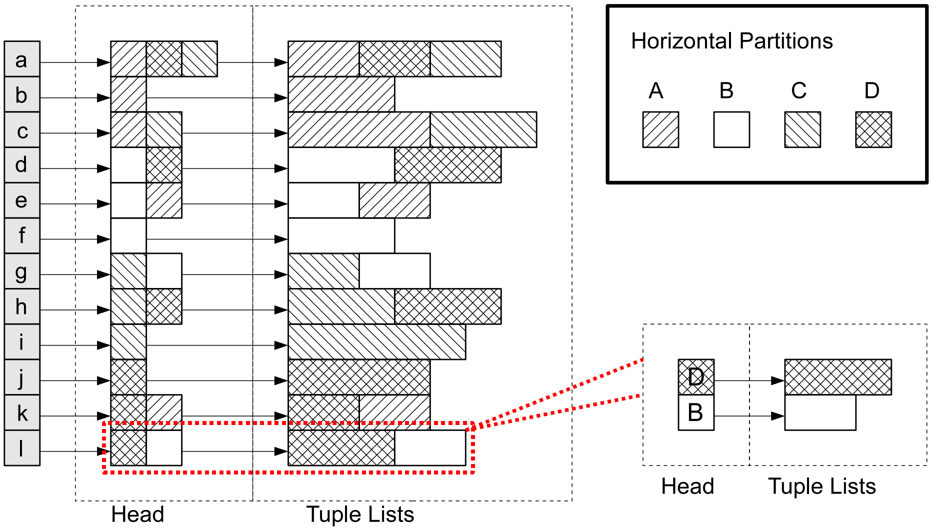


**Figure 10** Horizontal partition lists.

Then, given a query $Q$, we can fast identify the partitions having the most relevance to the query. For example, consider a query $Q$ ={d, e, f, g} in Figure 10. Since the partitions are associated with weights of corresponding tokens, partition B has the highest bound of similarity scores to the query (i.e., covers 4 query tokens assuming weight 1 for each). Thus, partition B will be queried first, while the tuples in partition A, P and D are probably pruned if the returned results in B are high in ranking score.

So far, we have presented the framework for indexing with horizontal partitions. Next, we discuss the mechanism and correctness of the query with pruning on horizontal partitions. Moreover, we will also present the implementation of query processing on the horizontal partitions.

## 4.2 Pruning top-k candidates

First of all, we discuss the representation of partitions. A partition $P$ consists of a number of tuples, while each tuple contains a set of tokens. Thus, the partition can also be represented by a set of tokens that appear in the tuples of this partition. We denote a partition by a *partition vector*.

**Definition 3** (partition vector) Let $t$ be any tuple in the partition $P$. Then, the partition $P$ can be represented by a *partition vector*,

$$P = (w_1, w_2, \ldots, w_{|\mathbf{D}|}) \tag{6}$$

where $w_j$ is the weight of the token $u_j$ in the partition $P$. The weight $P.w_j$ is defined by

$$P.w_j = \max_{t \in P} (t.w_j). \tag{7}$$

where $t$ is any tuple in the partition $P$, and $t.w_j$ is the weight of token $u_j$ in the tuple $t$.

For example, in Figure 10, we consider the weight of token l in the partition D, i.e., $\mathsf{D}.w_{\mathsf{l}}$. Let $\mathsf{l.D}.list$ be the list of tuple IDs in partition D that contains token l. Then, we have

$$\mathsf{D}.w_{\mathsf{l}} = \max_{t \in \mathsf{l.D}.list} (t.w_{\mathsf{l}}) \ ,$$

where $t.w_{\mathsf{l}}$ is the weight of token l of any tuple $t$ in the list of $\mathsf{l.D}.list$.

Once we have represented the partition $P$ by a vector as the query $Q$, a score of partition $P$ can also be measured by using the score function in formula 4. Given a query $Q$ and a partition $P$, the *partition score* is given by the intersection of these two vectors

$$score(P, Q) = \sum_{1 \le i \le |\mathbf{D}|} \min(P.w_i, Q.w_i), \tag{8}$$

where $|\mathbf{D}|$ is the domain size of all the tokens. In fact this $score(P, Q)$ is exactly the upper-bounds of the tuple scores in the partition $P$.

**Theorem 1** *Let t be any tuple in a partition P, then we have a upper-bound of ranking score(t, Q), i.e.*

$$P._{bound(Q)} = score(P, Q) \geq score(t, Q). \tag{9}$$

*where Q is the query.*

*Proof* Let $u_i$ be any token in the domain **D**. According to the definition of partition vector in formula 7, for any tuple $t$ in the partition $P$, we have $P.w_i \geq t.w_i$. Given a query $Q$, the weight of $Q.w_i$ is fixed. Thereby, we have $\min(P.w_i, Q.w_i) \geq \min(t.w_i, Q.w_i)$. To sum up all the tokens of $u_i$, we finally get $score(P, Q) \geq score(t, Q)$. □

For the AND condition query, it is required that the returned tuples should contain all the query tokens. Therefore, if one partition vector $P$ does not cover all the query tokens, then the tuples inside this partition cannot contain all the query tokens either. In other words, the partition $P$ can be pruned without further computing the upper-bound according to $P$'s partition vector. We can directly set $P._{bound(Q)} = 0$ to avoid aggregating the tuples in this partition $P$.

In each query processing step, we pick up the next partition $P'$ with the next highest $P._{bound(Q)}$ to the query. Now, the problem is to determine whether the next $g + 1$ partition and all the other remaining partitions can be safely pruned, after $g$ steps of querying on the first $g$ partitions with highest upper-bounds.

**Theorem 2** *Let $t'$ be the tuple with the minimum score in the top-k answers in the previous g steps. For the next $g + 1$ partition, if we have*

$$score(t', Q) > score(P_{g+1}, Q), \tag{10}$$

*then the partition $P_{g+1}$ can be safely pruned in the query.*

*Proof* According to Theorem 1, for any tuple $t$ in the partition $P_{g+1}$, we have $score(t', Q) > score(P_{g+1}, Q) \geq score(t, Q)$. Since $t'$ is the tuple in the current top-$k$ results with the minimum ranking score, in other words, the tuples in partition $P_{g+1}$ will never be ranked higher than $t'$ and can be pruned safely without further evaluation. □

For the remaining partitions $P_{g+2}, P_{g+3}, \ldots,$ since each query step selects the partition with the highest upper-bounds of scores, we have

$$score(t', Q) > score(P_{g+1}, Q) > score(P_{g+x}, Q),$$

where $x = 2, 3, \ldots$. Therefore, we can prune all the remaining partitions, starting from $P_{g+1}$.

## 4.3 Query processing

Intuitively, the query proceeding with pruning of tuples can be conducted as follows. Let $P_i.tokens$ be the set of tokens in the partition $P_i$, where each token is associated with a weight discussed in formula 7. When a top-k query comes, we first develop the upper-bound of similarity scores of the tuples in partition $P_i$ to the query $Q$, i.e., $P_i.bound(Q) = score(Q, P_i)$. Then the first evaluated partition is $P' = \arg\max_i P_i.bound$. By conducting the query on the tuple lists of partition $P'$, we get a set of ranked tuples $K$ as the top-k answers in $P'$. Suppose that the minimum top-k ranking score in the $K$ is $K[k] = \min_{t \in K} score(t, Q)$. Let $P''$ be the partition with the next highest upper-bound of scores. If $K[k] > P''.bound$, then the query processing can prune the candidate tuples in $P''$. Otherwise, we repeat the processing recursively.

### *4.3.1 Algorithm with horizontal partitioning*

Assume that the horizontal partitioning index has been built in the preprocessing. Given a query $Q$ and an integer $k$, the query algorithm is described in the following Algorithm 3.

**Algorithm 3** Top-$k$ query with horizontal partitioning

```
1:  procedure QUERY($Q, k$)
2:      $H$ = HORIZONTAL-LISTS($Q$)
3:      $P$ = PARTITIONS($H$)
4:      $K = \emptyset$
5:      for $i \leftarrow 1, P.size$ do
6:          if $K[k].score \geq P_i.bound$ then
7:              break
8:          else
9:              for $j \leftarrow 1, H.size$ do
10:                 $K$=AGGREGATE($K, H_j.P_i.list$)
11:     return $K$
```

In this algorithm, the operator PARTITIONS($H$) is conducted on the horizontal partition lists $H$ referred by the query. A set of ranked partitions $P$ are returned in descending order of score upper-bounds to the query $Q$. Let $K[k].score$ be the $k$-th largest score in the current top-k answers. For the partition $P_i$, if the current $K[k].score$ is larger than the upper-bound of the tuple scores of partition $P_i.bound$, then the partition $P_i$ and all the remaining partitions can be pruned. Otherwise, we aggregate and rank all the tuples in the partition $P_i$.

Again, the AGGREGATE() functions are different for the AND and OR query types, which have been presented in Algorithms 1 and 2. Furthermore, in computing of the upper-bounds of partitions in PARTITIONS(), the AND query directly set the bounds of those partitions to 0, where the partition vectors do not contain all the query tokens.

### *4.3.2 Cost evaluation*

Let $O(B)$ be the cost of aggregating all the tuples in the basic index for a query $Q$. Suppose that there are $P$ partitions covered by the query $Q$. Let $E(P)$ be the cost estimation of evaluating the upper-bounds of $P$ partitions. Then, the cost of querying on the horizontal partitioning index can be estimated by $O(P) = (1 - r) \cdot O(B) + E(P)$, where $r$ is the pruning rate. The higher pruning rate $r$ is, the lower aggregation cost is.

### *4.3.3 Crack horizontal partitioning*

So far, we have presented the approach of indexing and querying the dataspaces with horizontal partitioning. The partitions of tuples are determined by the intuition that the tuples in the same partition share similar contents of tokens. Thus, the tuples can be grouped based on the tuple similarities. For example, we employ a classifier to make decisions by going through all tuples. In this study, including the following experimental settings, we employ the k-means clustering algorithm [19] to partition the tuples into $m$ clusters without a supervised classifier.

Here, the interesting problem is the selection of the number of partitions $m$. Recall that the processing of partitions affects the query cost, i.e., $E(P)$. Obviously, the number of partitions is an essential issue of $E(P)$, during computing upper-bounds and ranking the partitions. In fact, the more the partitions are, the higher the cost $E(P)$ is. In the extreme, if each partition corresponds to exact one tuple, then the cost $E(P)$ of ordering the horizontal partitions is similar to the cost of querying on the basic index $O(B)$. On the other hand, if the number of partitions is too small, e.g., only one partition, then the horizontal partitioning index are equivalent to the basic one.

## 5 Hybrid partitioning index

Now, we have two strategies of partitioning index, vertical partitioning (on tokens) and horizontal partitioning (on tuples), separately. Next, we present a hybrid index with both vertical and horizontal partitions.

### 5.1 Indexing

As shown in Figure 11, the *hybrid partition index* consists of two parts as well, the *head* and the *tuple lists*. Each hybrid partition list contains one or several tokens and records the horizontal partitions in the head. The tuples are first divided into blocks according to the horizontal partitions, and in each block they are further divided into

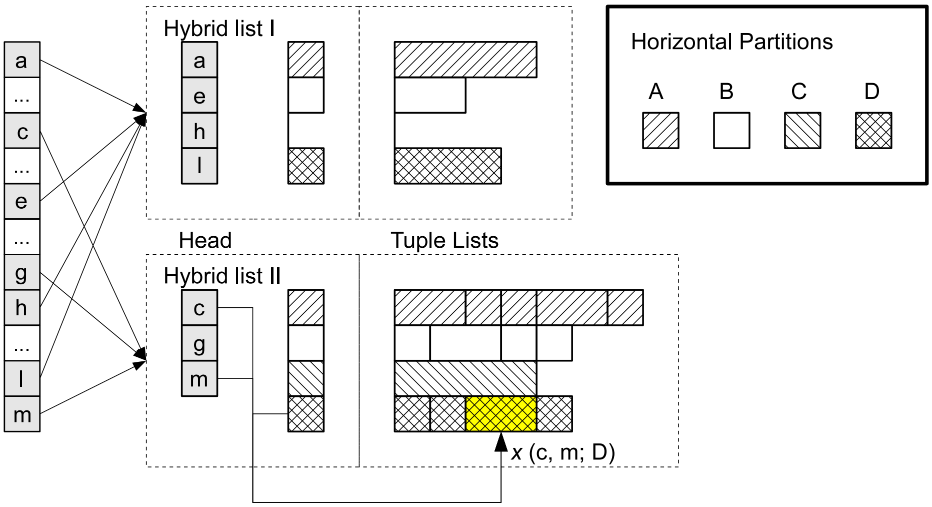


**Figure 11** Hybrid partition list.

sub-tuple lists on the vertical partitions. If one block is generated by compression, then there is no further division in each horizontal partition. For example, the hybrid list I in Figure 11 contains four tokens a, e, h, l that are always appearing together in the same tuples. Thus, there is no vertical partition in each horizontal partition list. For the compressed data, each horizontal partition is further divided into sections, according to different token appearances. For example, in the hybrid list II in Figure 11, the cell x denotes the list of all the tuples in the partition D with tokens c, m except f. Although there are different partitions, vertically and horizontally, all the tuples in one hybrid partition list are stored in continuous disk spaces and can be read in one I/O operation.

## 5.2 Query processing

### *5.2.1 Algorithm with hybrid partitioning*

Given a query $Q$ and an integer $k$, the query algorithm is described in the following Algorithm 4. Assume that the hybrid lists $Y$ are loaded into the memory by HYBRID-LISTS($Q$). Let $P$ be all the partitions, which are sorted in descending order of the upper-bounds of scores to the query $Q$. For the current partition $P_i$, we evaluate the sub-list $Y_j.P_i$ corresponding to $P_i$ in each list $Y_j$. In fact, the sub-list $Y_j.P_i$ is exactly a vertical partition list in the vertical partitioning index. Thus, similar to the Algorithm 2, the tuple list $Y_j.P_i.V_l$ in $Y_j.P_i$ is aggregated to the result $K$, according to the matching of the corresponding *tokens* and the query $Q$ (line 10–14 in Algorithm 4).

**Algorithm 4** Top-$k$ query with hybrid partitioning

```
1:  procedure QUERY($Q, k$)
2:      $Y$ = HYBRID-LISTS($Q$)
3:      $P$ = PARTITIONS($Y$)
4:      $K = \emptyset$
5:      for $i \leftarrow 1, P.size$ do
6:          if $K[k].score \geq P_i.bound$ then
7:              break
8:          for $j \leftarrow 1, Y.size$ do
9:              for $l \leftarrow 1, Y_j.P_i.size$ do
10:                 if MATCH($Q, Y_j.P_i.V_l.tokens$) then
11:                     $K$=AGGREGATE($K, Y_j.P_i.V_l.list$)
12:     return $K$
```

Again, the AGGREGATE() is the same function used in Algorithm 1. The MATCH() function has the same processing as Algorithm 2 for vertical partitioning. The PARTITIONS() function is adopted from Algorithm 3 for horizontal partitioning. All these functions have versions for AND and OR queries respectively, which are discussed above.

### *5.2.2 Cost evaluation*

Let $O(V)$ be the cost of querying on the vertical partitioning index. Let $E(P)$ be the cost of preparing the partitions' information in PARTITIONS($Y$). The cost of querying on the hybrid partitioning index can be estimated by $O(Y) = (1-r) \cdot O(V) + E(P) = \frac{(1-r)}{2^{M-1}} O(B) + E(P)$, where $r$ is the pruning rate, $M$ is the average

**Table 1** Cost comparison of approaches.

| Approach | Cost |
|---|---|
| Baseline index | $O(B)$ |
| Vertical index | $\frac{1}{2^{M-1}}O(B)$ |
| Horizontal index | $(1-r)\cdot O(B)+E(P)$ |
| Hybrid index | $\frac{(1-r)}{2^{M-1}}O(B)+E(P)$ |

number of tokens in each hybrid partitioning list, and $O(B)$ is the cost of querying on the basic inverted index.

Finally, to sum up, we compare the cost evaluation of four index approaches in this paper as follows (Table 1). Although the improvement of the vertical partitioning approach is large, it depends on whether the $M$ tokens appear together frequently in the real query workload. For the horizontal partitioning, we have a balance of pruning rate $r$ and the partition numbers. If the number of partitions is not large and the pruning rate is high, the $E(P)$ will be much smaller than the pruned query cost, i.e., $E(P) \ll r \cdot O(B)$, and the total cost can be improved. The hybrid partition index benefits from both. If the number of partitions is moderated large, the hybrid approach shows lower cost than the other ones, which is also verified in our experiments in Section 6.

## 6 Experimental evaluation

This section reports the experimental evaluation of querying over the various index approaches. We compare the following approaches in this study, including the baseline approach of indexing dataspaces [14], the index with vertical partitioning, the index with horizontal partitioning, and the hybrid index with both vertical and horizontal partitioning. The main evaluating criterion is the query time cost of these approaches, i.e., query efficiency. We run the experiments in two real data sets, Google Base and Wikipedia. We crawl 7,432,575 tuples (items) from the Google Base web site, in size of 3.06 GB after preprocessing. The dataspace of Wikipedia consists of 3,493,237 tuples, in size of 0.82 GB after preprocessing. 100 tuples are randomly selected from the dataspaces as the query workload for each dataset. In the evaluation, we conduct these queries and observe the average metrics.

### 6.1 Evaluating vertical partitioning index

The first experiment evaluates the performance of querying on the vertical partitioning index. Table 2 presents the total number of vertical partition lists that are retrieved by these queries, the average number of disk I/O operations, and the average query time cost of OR and AND queries respectively. We observe the vertical partitions with different frequent $k$ sets of tokens, $k = 1$ to 5.

As shown in Table 2, the vertical lists of tuples referred by the OR and AND queries are the same under the same query input predicates. Compared with the OR query, the AND query only aggregates the tuples with all the query tokens, thus has lower

**Table 2** Query evaluation on vertical partitioning index.

| Frequent *k* set | # vertical list | # disk I/O | OR query time cost (ms) | AND query time cost (ms) |
|---|---|---|---|---|
| Base | | | | |
| 1 | 1546 | 16.58 | 144.07 | 30.94 |
| 2 | 1139 | 12.41 | 133.29 | 21.56 |
| 3 | 1018 | 11.19 | 132.65 | 19.53 |
| 4 | 964 | 10.64 | 132.04 | 19.06 |
| 5 | 927 | 10.27 | 131.25 | 18.75 |
| Wiki | | | | |
| 1 | 1707 | 17.79 | 212.68 | 18.15 |
| 2 | 1237 | 12.97 | 195.33 | 15.79 |
| 3 | 1115 | 11.70 | 193.13 | 13.43 |
| 4 | 1038 | 10.93 | 168.91 | 12.82 |
| 5 | 1006 | 10.61 | 187.03 | 12.97 |

time cost. These results also verify the conjecture that the aggregation of tuples is costly, especially in the OR query with many candidate tuples.

The frequent 1 set, where each token is a partition, is equivalent to the basic inverted list. Recall that a partitioning of frequent *k* set denotes that the tuple lists of the tokens in the same set are compressed. For example, a vertical partition list of

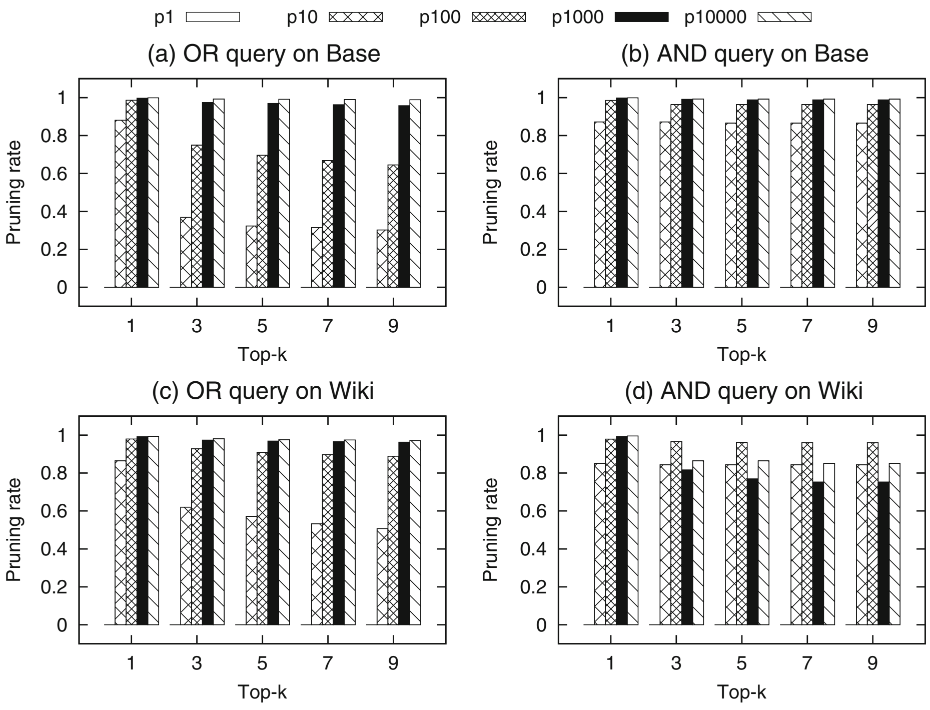


**Figure 12** Pruning rate evaluation on horizontal partitioning index.

frequent 3 set index may contain 3 tokens at most and their corresponding tuple lists are merged. Therefore, with increase of $k$ in Table 2, the total number of vertical partition lists retrieved by the queries decreases. In other words, for each query, we need a less number of vertical partition lists that are read from the disk as well. Since the vertical partition lists compress the tuple list of the tokens in the frequent sets, the query evaluation on these compressed tuple lists can reduce the tuple aggregation time. As shown in Table 2, the query time cost reduces with the increase of $k$.

With the further increase of $k$, however, the query time cost may not be improved. Suppose that a query $Q$ only covers 1 token of all the 5 tokens in a (frequent 5 set) vertical partition list. The I/O cost of reading the information for the other 4 tokens is wasted. Thus, the time cost of $k = 5$ is higher than the $k = 4$ case's in the Wiki evaluation in Table 2.

## 6.2 Evaluating horizontal partitioning index

Next, we study the query performance on the horizontal partitioning index. The first metric we used is the pruning rate, i.e., the rate of partitions that are pruned in the query processing. We also present the corresponding time cost in different top-k queries. Figures 12 and 13 show the results of both two data sets with various partition numbers (p1 to p10000).

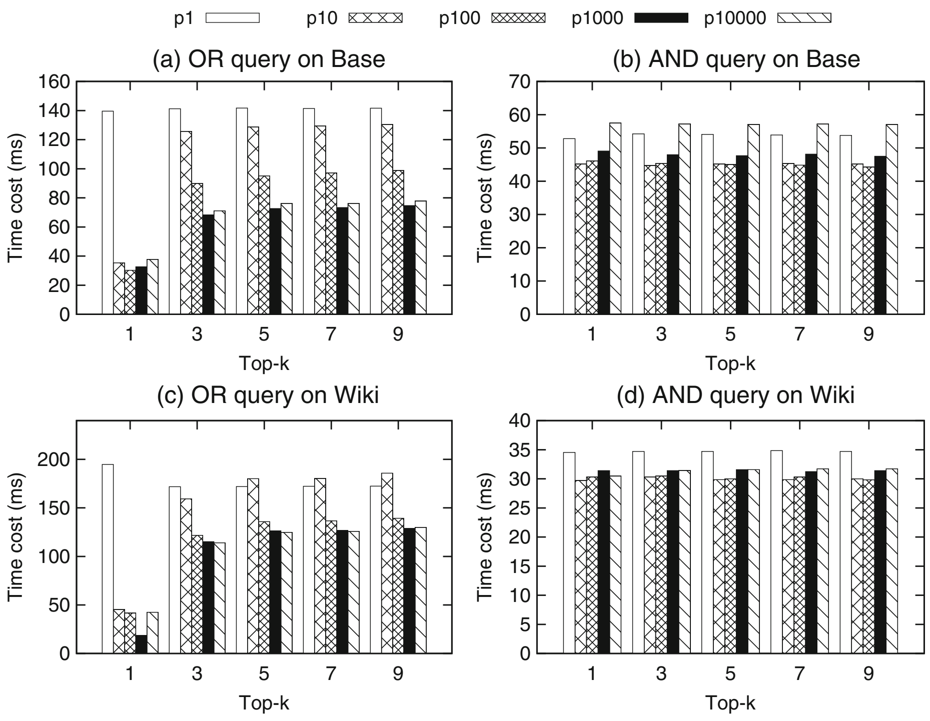


**Figure 13** Time cost evaluation on horizontal partitioning index.

The partition p1 means that all the tuples are grouped in 1 partition, which is equivalent to the basic inverted index. According to the query strategies in the horizontal partitioning index, there is no partition that can be pruned, that is, the pruning rate of p1 is 0. As shown in Figure 12, with the increase of partition numbers, the upper-bounds of scores in the partitions are more accurate, thus the pruning rate is improved as well. The corresponding time cost, in Figure 13, decreases when more tuples are pruned in higher pruning rate. With the further increase of partition numbers, the overhead of processing partition information grows, such as computing the upper-bounds of scores of the partitions for the query. Therefore, when the partitions are extremely large like p10000, the time performance may not be improved although the pruning rate is still high. The time cost of p10000 is a bit higher than the query on p1000. Nevertheless, according to our empirical study, even a small partition number such as 10 or 100 can already improve the performance largely.

Note that, in OR query, the top-1 query has a significant higher performance than the others. The reason is that we choose the tuples from the dataset as the queries. The top-1 answer has a high score after processing the partition which contains the query. Thereby, the next remaining partitions whose upper-bounds of scores may be lower than the high top-1 score can be pruned. On the other hand, for the top-9 query for example, the top-9 score is not as high as the top-1 answer's, thus the remaining partitions may have a higher upper-bounds than the top-9 answer and cannot be pruned. Moreover, in the AND query, there may be less than k candidate answers in

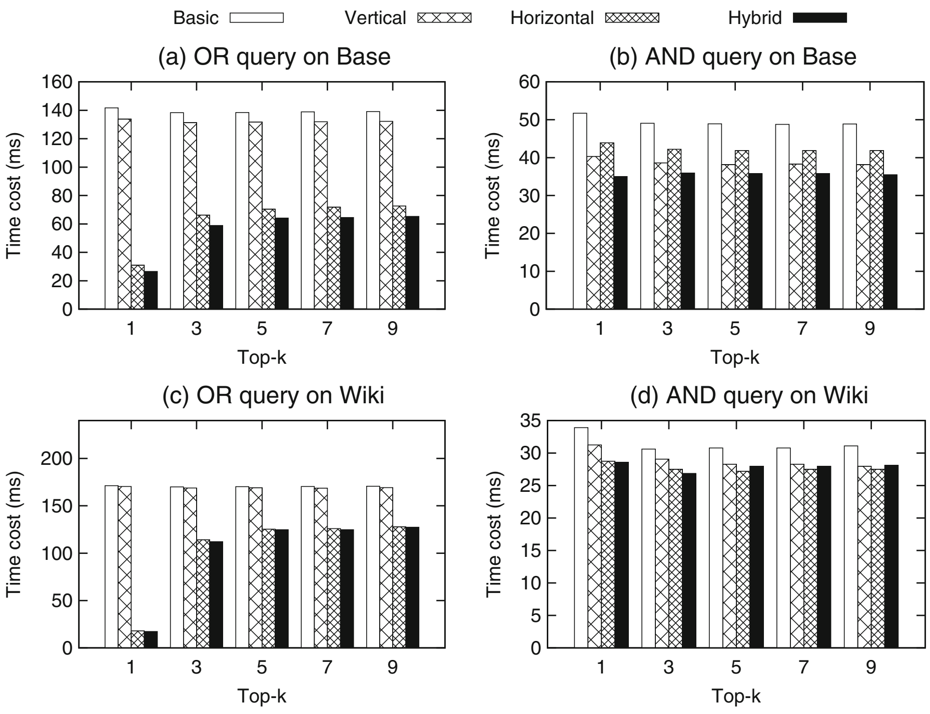


**Figure 14** Comparison of approaches on top-k.

the whole dataset that can cover all the query tokens. Thus, with the increase of k, the query result does not change much. Nevertheless, as shown in Figure 13, the query on the horizontal partitioning index (such as p1000) achieves improvement of time performance comparing with the approach without partitioning (i.e. p1).

## 6.3 Comparison study

Now, we compare all the four approaches, including the basic inverted index, the vertical partitioning index, the horizontal partitioning index and the hybrid index. Again, we first present the time cost evaluation of four approaches in different top-k queries, in Figure 14. Then, the scalability of these approaches is also studied in different data sizes. We present the scalability with top-1 and top-10 in Figures 15 and 16, respectively. The vertical partitioning index uses the frequent 4 set as the token partitions. The horizontal partitioning index processes 1000 partitions of tuples, i.e., p1000. The hybrid index combines these two partitions together, with frequent 4 set and p1000 settings.

In the OR query, comparing with the basic approach, the improvement of the vertical partitioning index is not as significant as the horizontal one. Although many tokens and their tuple lists are compressed in the vertical partitioning index, a certain query may only covers parts of the compressed token sets. For those tokens not included in the frequent set, we have to process as the basic inverted lists. In the same manner, the hybrid approach shows a small improvement comparing with the

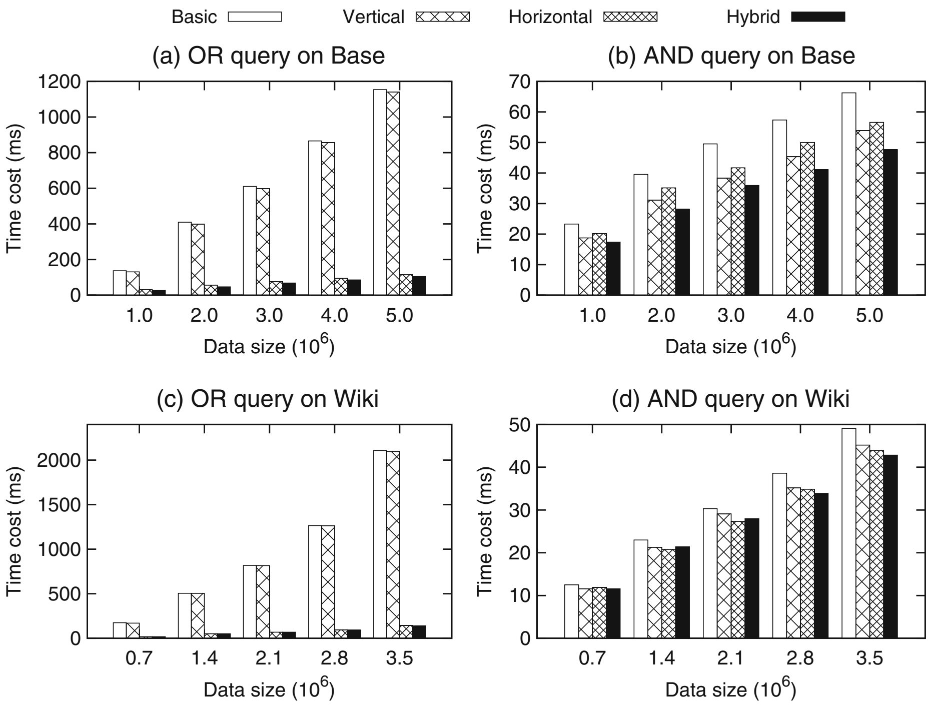


**Figure 15** Comparison of approaches on top-1 scalability.

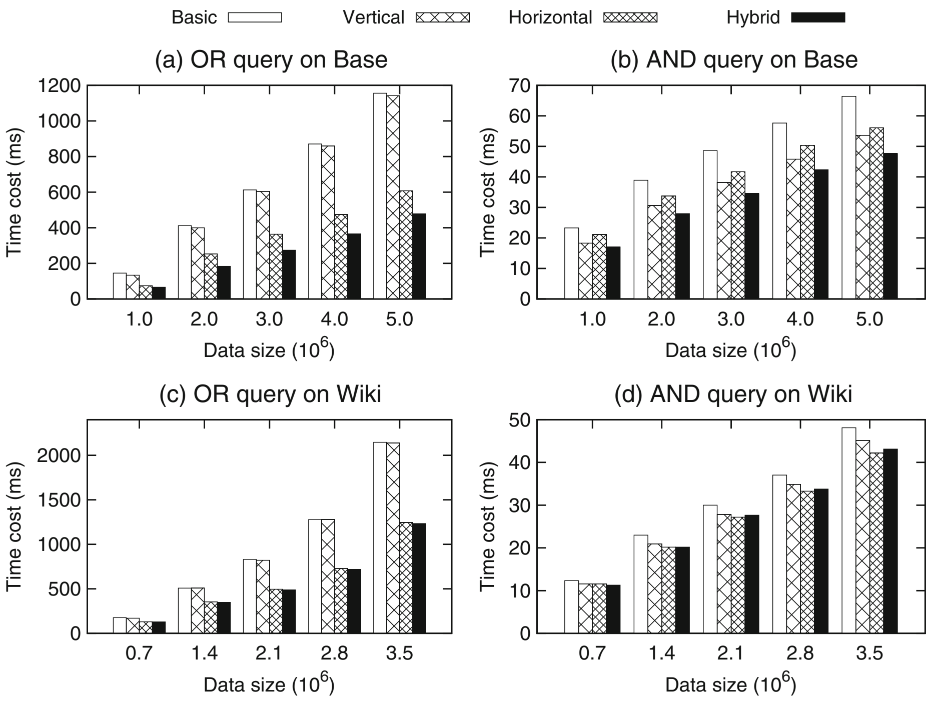


**Figure 16** Comparison of approaches on top-10 scalability.

pure horizontal partitioning approach. Moreover, our partitioning based approaches show good scalability in different data sizes. Especially, in the top-1 OR query, the hybrid index achieves about one tenth of the time cost in the basic inverted index. As discussed above, since there may be less than k answers in the top-k AND query, the result of top-1 and top-10 are quite similar in Figures 15 and 16. When the improvements of both the vertical and horizontal partitioning approaches are limited, the corresponding hybrid approach can barely further reduce the time cost, e.g., in Figures 14d, 15d and 16d. Indeed, the hybrid method needs to coordinate both the partitions with more processing cost than that of each single partitioning index. If such processing cost is larger than the pruned cost, the overall performance can even be a little bit worse than the single partitioning methods, such as case 9 in Figure 14d and case 3.5 in Figure 16d. Nevertheless, according to the results, the hybrid approach can achieve significantly better results in most cases, and have only a little bit higher cost than the single partitioning methods in the other cases.

*Summary* According to the experimental evaluation, we find the following. (1) The vertical partitioning index shows better time performance than the previous dataspaces index [14] without partitioning. (2) Our horizontal partitioning index can significantly reduce the time cost of top-k queries, especially on the top-1 queries. (3) The hybrid index can achieve the lowest time cost in most cases, on both the OR and AND queries. (4) Our proposed approaches scale well under various large data sizes, with millions of tuples in a dataspace.

## 7 Related work

The concept of dataspaces is proposed in [16, 17], which provides a co-existing system of heterogeneous data. Recent work [21, 27, 30] is mainly dedicated to offering best-effort answers in a *pay-as-you-go* style in view of integration issues. In our study, instead, we concern the efficiency issues on accessing dataspaces.

The problem of indexing dataspaces is first studied by Dong and Halevy [14] to support efficient query. Based on the encoding of attribute label and value as tokens, the inverted index is extended to dataspaces, which is also considered as the baseline approach in our work. In fact, inverted index [7, 35] has been studied for decades as efficient access to sparse data. Zobel and Moffat [38] provide a comprehensive introduction of key techniques in text indexing literature. However, as mentioned, the straight-forward extension for indexing dataspaces falls short in the efficiency consideration, while our approaches consider the partitioning-based index motivated by the observation on real data.

Partitioning-based index is studied as well. Lester [22] partitioning the index for efficient on-line index. To make documents immediately accessible, the index is divided into a controlled number of partitions. Nikos Mamoulis [25] studies the efficient joins on set-valued attributes, by using inverted index. Different from our large number of attributes, the join predicates are evaluated between two attributes only. Sarawagi and Kirpal [29] also propose an efficient algorithm for indexing with a data partitioning strategy. All these efficient techniques are dedicated to the single attribute problem, while the dataspaces contain various attributes. Although, the queries over multiple attributes in a relation are studied [24], the original inverted index is directly applied. In this sense, our partition-based indexing is complementary to the query optimization techniques. Partitioning of blocks is natural and can be applied in Web search as well [9]. In particular, to index and query with partitions, Web pages are segmented into several blocks according to their contents, such as titles, articles, images, etc. It is notable that our vertical and horizontal partitioning of tokens and tuples in dataspaces is conducted based on the data distribution in space vectors as shown in Figure 5, and thus is different from the Web page segmentation. The partitioning idea is also applied in accelerating XML queries [6, 26, 36]. Instead of dividing correlated tokens as studied in our work, the partitioning (and compressing) in XML data is conducted over path patterns with structural relationships. Thereby, the previous XML techniques are not applicable in the dataspaces scenario.

Various storage strategies of sparse data are also proposed. Chu et al. [12] use a big wide-table to store such sparse data and extract the data incrementally [11]. Rather than the predominant positional storage with a pre-allocated amount of space for each attribute, the wide-table uses a interpreted storage to avoid allocating spaces to those null values in the sparse data. Agrawal et al. [1] study a vertical format storage of the tuples. Specifically, a 3-ary vertical scheme is developed with columns including tuple identifier, attribute name, and attribute value. Beckmann et al. [8] extend the RDBMS attributes to handle the sparse data as interpreted fields. A prototype implementation in the existing RDBMS is evaluated to illustrate the advanced performance in dealing with sparse data. Abadi et al. [2, 3] provide comprehensive studies of the column-based store comparing with the row-based store. The column store with vertical partitioning shows advanced performance in many applications, such as the RDF data of the Semantic Web [3] and the recent

Star Schema Benchmark of data warehousing [2]. To efficiently access these attribute values, further partitioning store techniques are studied. Chaudhuri et al. [10] study a similarity join operator (SSJoin [5, 10]) on text attributes, which are also organized in a vertical style. Specifically, each value of text attributes is converted to a set of tokens (words or q-grams [34]), which are store separately in different tuples respectively, similar to the attribute partitioning. Our work is independent with storage of the dataspace, instead we build the index over the data and the query processing efficiency depends on the index structures only.

In the top-k aggregation query [15], optimization with efficient pruning strategies are developed. In their scenario, a sorted list is maintained for each attribute. The top-k answers are determined under a fixed monotone aggregation function of all the attributes. Thus, it is not directly applicable to our dataspace cases with large number of sparse attributes. Instead, we develop the partitions of data in the inverted index, and utilize the monotone aggregation features to do efficient pruning. de Vries et al. [13] also study the partitioning of data for efficient k-NN search in sparse data objects but without attributes. In our study, we process the tuples with sparse attributes whose values are sparse as well.

## 8 Conclusions and discussion

In this paper, we study the partition-based indexing approaches for efficiently accessing dataspaces. First, we start from a survey of real dataspaces. Due to the extremely sparse feature, a basic index with extensions on inverted lists is introduced, together with the discussion of corresponding problems. Based on the observations of token relationships in the dataspaces, we develop the vertical partitioning index. The tokens that share similar tuples lists are merged together and compressed. Then the query on these compressed vertical partition lists can reduce both the number of I/O reads and the cost of aggregating the tuples inside the vertical partition list. Moreover, according to the relationships of tuples, we propose the horizontal partitioning index. Given a query, we can prune the tuple partitions according to their upper-bounds of tuple scores. Finally, we present the hybrid index with both vertical and horizontal partitioning. The extensive experiment results demonstrate the performance of the proposed index in real dataspaces. Our approaches, especially the hybrid index, outperform the previous technique and scale well with large data sizes. These experimental results also verify our theoretical analysis of cost of different approaches.

As an indexing technique, the proposed partitioning index is complementary to the query optimization approaches with materialized views in dataspaces [32]. The indexing with compressed lists requires partitions to be disjoint, while the materialized views allow overlapping and share some common tokens. It is not surprising that compressing inverted lists in the index could lead to the adaption of the corresponding materialized views and affect the query performance. Unfortunately, to coordinate these indexing and materialization is highly non-trivial, as finding a good scheme of materialized views is already hard. Therefore, we leave it as a future work to generate a synchronized plan of partitioning-based index and materialization.

**Acknowledgements** This work is supported by China NSFC Project No. 61073005, Hong Kong RGC GRF Grant under Project No. 611411, Microsoft Research Asia Theme Grant MRA11EG0, HP IRP Project Grant 2011 and HP Catalyst Program 2011, HKUST RPC grant RPC10EG13 and HKUST SSRI11EG01.

## References

1. Agrawal, R., Somani, A., Xu, Y.: Storage and querying of e-commerce data. In: VLDB, pp. 149–158 (2001)
2. Abadi, D., Madden, S., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: SIGMOD Conference (2008)
3. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: VLDB, pp. 411–422 (2007)
4. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: VLDB, pp. 487–499 (1994)
5. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB, pp. 918–929 (2006)
6. Arion, A., Bonifati, A., Manolescu, I., Pugliese, A.: Path summaries and path partitioning in modern xml databases. World Wide Web **11**(1), 117–151 (2008)
7. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press / Addison-Wesley (1999)
8. Beckmann, J.L., Halverson, A., Krishnamurthy, R., Naughton, J.F.: Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In: ICDE, p. 58 (2006)
9. Bruno, E., Faessel, N., Glotin, H., Maitre, J.L., Scholl, M.: Indexing and querying segmented web pages: the blockweb model. World Wide Web **14**(5–6), 623–649 (2011)
10. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE, p. 5 (2006)
11. Chu, E., Baid, A., Chen, T., Doan, A., Naughton, J.F.: A relational approach to incrementally extracting and querying structure in unstructured data. In: VLDB, pp. 1045–1056 (2007)
12. Chu, E., Beckmann, J.L., Naughton, J.F.: The case for a wide-table approach to manage sparse relational data sets. In: SIGMOD Conference, pp. 821–832 (2007)
13. de Vries, A.P., Mamoulis, N., Nes, N., Kersten, M.L.: Efficient k-NN search on vertically decomposed data. In: SIGMOD Conference, pp. 322–333 (2002)
14. Dong, X., Halevy, A.Y.: Indexing dataspaces. In: SIGMOD Conference, pp. 43–54 (2007)
15. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS (2001)
16. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Record **34**(4), 27–33 (2005)
17. Halevy, A.Y., Franklin, M.J., Maier, D.: Principles of dataspace systems. In: PODS, pp. 1–9 (2006)
18. Franklin, M.J., Halevy, A.Y., Maier, D.: A first tutorial on dataspaces. PVLDB **1**(2), 1516–1517 (2008)
19. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2000)
20. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min. Knowl. Discov. **8**(1), 53–87 (2004)
21. Jeffery, S.R., Franklin, M.J., Halevy, A.Y.: Pay-as-you-go user feedback for dataspace systems. In: SIGMOD Conference, pp. 847–860 (2008)
22. Lester, N., Moffat, A., Zobel, J.: Fast on-line index construction by geometric partitioning. In: CIKM, pp. 776–783 (2005)
23. Li, Q., Chen, J., Wu, Y.: Algorithm for extracting loosely structured data records through digging strict patterns. World Wide Web **12**(3), 263–284 (2009)
24. Lu, W., Chen, J., Du, X., Wang, J., Pan, W.: Efficient top-*k* approximate searches against a relation with multiple attributes. World Wide Web **14**(5–6), 573–597 (2011)
25. Mamoulis, N.: Efficient processing of joins on set-valued attributes. In: SIGMOD Conference, pp. 157–168 (2003)
26. Ng, W., Lau, H.L., Zhou, A.: Divide, compress and conquer: Querying xml via partitioned path-based compressed data blocks. World Wide Web **11**(2), 169–197 (2008)

27. Salles, M.A.V., Dittrich, J.-P., Karakashian, S.K., Girard, O.R., Blunschi, L.: Itrails: pay-as-you-go information integration in dataspaces. In: VLDB, pp. 663–674 (2007)
28. Salton, G.: Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison-Wesley (1989)
29. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD Conference, pp. 743–754 (2004)
30. Sarma, A.D., Dong, X., Halevy, A.Y.: Bootstrapping pay-as-you-go data integration systems. In: SIGMOD Conference, pp. 861–874 (2008)
31. Song, S., Chen, L., Yu, P.S.: On data dependencies in dataspaces. In: ICDE, pp. 470–481 (2011)
32. Song, S., Chen, S., Yuan, M.: Materialization and decomposition of dataspaces for efficient search. IEEE Trans. Knowl. Data Eng. **23**(12), 1872–1887 (2011)
33. Tomasic, A., Garcia-Molina, H., Shoens, K.A.: Incremental updates of inverted lists for text document retrieval. In: SIGMOD Conference, pp. 289–300 (1994)
34. Ukkonen, E.: Approximate string matching with q-grams and maximal matches. Theor. Comput. Sci. **92**(1), 191–211 (1992)
35. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann (1999)
36. Wu, X., Theodoratos, D., Souldatos, S., Dalamagas, T., Sellis, T.K.: Evaluation techniques for generalized path pattern queries on xml data. World Wide Web **13**(4), 441–474 (2010)
37. Zipf, G.K.: Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology. Addison-Wesley (1949)
38. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2), Article 6 (2006). doi:10.1145/1132956.1132959
39. Zobel, J., Moffat, A., Sacks-Davis, R.: An efficient indexing technique for full text databases. In: VLDB, pp. 352–362 (1992)