Materialization and Decomposition of Dataspaces for Efficient Search

Shaoxu Song, *Student Member*, *IEEE*, Lei Chen, *Member*, *IEEE*, and Mingxuan Yuan

Abstract—Dataspaces consist of large-scale heterogeneous data. The query interface of accessing tuples should be provided as a fundamental facility by practical dataspace systems. Previously, an efficient index has been proposed for queries with keyword neighborhood over dataspaces. In this paper, we study the materialization and decomposition of dataspaces, in order to improve the query efficiency. First, we study the views of items, which are materialized in order to be reused by queries. When a set of views are materialized, it leads to select some of them as the optimal plan with the minimum query cost. Efficient algorithms are developed for query planning and view generation. Second, we study the partitions of tuples for answering top-k queries. Given a query, we can evaluate the score bounds of the tuples in partitions and prune those partitions with bounds lower than the scores of top-k answers. We also provide theoretical analysis of query cost and prove that the query efficiency cannot be improved by increasing the number of partitions. Finally, we conduct an extensive experimental evaluation to illustrate the superior performance of proposed techniques.

Index Terms—Dataspaces, materialization, decomposition.

1 Introduction

Dataspaces are recently proposed [1], [2] to provide a co-existing system of heterogeneous data. The importance of dataspace systems has already been recognized and emphasized in handling heterogeneous data [3], [4], [5], [6], [7]. In fact, examples of interesting dataspaces are now prevalent, especially on the Web [3].

For example, Google Base[1] is a *very large, self-describing, semistructured, heterogeneous* database. We illustrate several dataspace tuples with attribute values in Fig. 1 as follows: each entry T_i consists of several attributes with corresponding values and can be regarded as a tuple in dataspaces. Due to the heterogeneity of data, which are contributed by users around the world, the data set is extremely sparse. According to our observations, there are total 5,858 attributes in 307,667 tuples (random samples), while most of these tuples only have less than 30 attributes individually.

Another example of dataspaces is from Wikipedia,[2] where each article usually has a tuple with some attributes and values to describe the basic structured information of the entry. For instance, a tuple describing the Nikon Corporation may contain attributes like (founded:Tokyo Japan 1917), (industry: imaging), (products: cameras) ...}. Such interesting tuples could not only be found in article entries but also mined by advanced tools such as Yago [8] in the DBPedia project.[3] Again, the attributes of tuples in different entries are various, while each tuple may only contain a limited number of attributes. Thereby, all these tuples from heterogeneous sources form a huge dataspace in Wikipedia.

Due to the heterogeneous data, there exist matching correspondences among attributes in dataspaces. For example, the matching correspondence between attributes manu and prod could be identified in Fig. 1, since both of them specify similar information of manufacturer of products. Such attribute correspondences are often recognized by schema mapping techniques [9]. In dataspaces, a pay-as-you-go style [5] is usually applied to gradually identify these correspondences according to users' feedback when necessary.

Once the attribute correspondences are recognized, the keywords in attributes with correspondences are said *neighbors* in schema level. For example, keywords Apple in attributes manu and prod are neighbor keywords, since manu and prod have correspondence. Consequently, a query with keyword neighborhood in schema level [10] should not only search the keywords in the attributes specified in the query, but also match the neighbor keywords in the attributes with correspondences. For example, a query predicate (manu : Apple) should search keyword Apple in both the attributes manu and prod, according to the correspondence between manu and prod.

To support efficient queries on dataspaces, Dong and Halevy [10] utilize the encoding of attribute-keywords as *items* and extend the inverted index to answer queries. Specifically, each distinct attribute name and value pair is encoded by a unique item. For instance, (manu : Apple) is denoted by the item I_1. Then, each tuple can be represented by a set of items. Similarly, the query input can also be encoded in the same way. Since the data are extremely

1. http://base.google.com/.
2. http://www.wikipedia.org/.
3. http://dbpedia.org/.

• *The authors are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong.*
E-mail: {sshaoxu, leichen, mingxuan}@cse.ust.hk.

Manuscript received 4 Dec. 2009; revised 20 Apr. 2010; accepted 11 June 2010; published online 26 Oct. 2010.
Recommended for acceptance by N. Bruno.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2009-12-0821.
Digital Object Identifier no. 10.1109/TKDE.2010.213.

We consider a dataspace with following tuples,

T_1 : {(name : iPod), (color : red), (manu : Apple Inc.), (tel : 567),
(addr : Infinite Loop, CA), (website : itunes.com)};

T_2 : {(name : iPod), (color : cardinal), (prod : Apple), (tel : 123),
(post : Infinite Loop, Cupert), (website : apple.com)};

T_3 : {(name : iPad), (color : white), (manu : Apple Inc.),
(post : Infinite Loop), (website : apple.com), (phn : 567)}.

where manu denotes an attribute of manufacturer, prod is producer, and addr denotes address. The element (manu : Apple Inc.) in T_1 denotes the value of attribute manu is Apple Inc. in tuple T_1.

Fig. 1. Example of dataspaces.

sparse, the inverted index can be built on items to support the efficient query answering.

In this paper, from a different aspect of query optimization, we study the materialization and decomposition of dataspaces. The idea of improving query efficiency with keyword neighborhood in schema level follows two intuitions: 1) the reuse of contents of a query, and 2) the pruning of contents for a query.

Motivated by the neighbor keywords that are queried together, we study the materialization of views of items in order to reuse the computation. Intuitively, due to the correspondence of attributes, keywords in neighborhood in schema level are always searched together in a same predicate query. For example, a query on (manu : Apple) will always search (prod : Apple) as well. Therefore, we can cache the search results of (manu : Apple) and (prod : Apple), as a materialized view in dataspaces. Such view results could be reused in different queries. When multiple views are available, it leads us to the problem of selecting the optimal query plans on materialized views.

To answer the top-k query, we study the pruning of unqualified partitions of tuples. Specifically, tuples in dataspaces are divided into a set of nonoverlapping groups, namely, *partitions*. When a query comes, we develop the score bounds of the tuples in partitions. After processing the tuples in some partitions, if the current top-k answers have higher scores than the bounds of remaining partitions, then we can safely prune these remaining partitions without evaluating their tuples.

1.1 Contribution

To our best knowledge, this is the first work on studying materialization and decomposition of dataspaces for efficient search. Following the previous work by Dong and Halevy [10], the attribute-keyword model is also utilized in this study. Although our techniques are motivated by queries with keyword neighborhood in schema level in dataspaces, the proposed idea of materialization and decomposition is also generally applicable to attribute-keyword search over structured and semi-structured data. Our main contributions in this paper are summarized by:

1. We study the query planning on item views that are materialized in dataspaces. The materialization scheme in dataspaces is first introduced, based on which we can select a plan with minimum cost for a query. The optimal planning problem can be formulated as an *integer linear programming* problem. Thereby, we investigate greedy algorithms to select the near optimal query plan, with relative error bounds on the query cost.
2. We discuss the generation of item views to minimize the query costs. Obviously, the more the materialized views are, the better the query performance is. However, real scenarios usually have a constraint on the maximum available disk space for materialization. Thereby, we also study greedy heuristics to generate views that can possibly provide low cost query plans.
3. We propose the decomposition of dataspaces to support efficient top-k queries. The decomposition scheme in dataspaces is first introduced, where tuples are divided into nonoverlapping partitions. The score bounds for the tuples in a partition to the query are theoretically proved. Safe pruning is then developed based on these score bounds in partitions. It is notable that we are not proposing a new top-k ranking method. Instead, our partitioning technique is regarded as a complementary work to the previous merge operators. Thereby, advanced merge methods, such as TA family methods [11], [12], can be cooperated together with our approaches as presented in experiments.
4. We develop a theoretical analysis for the cost of querying with partitions. We provide the analysis of pruning rate and query cost by using the self-similarity property, which is also verified by our experimental observations. According to the cost analysis, we cannot always improve the query efficiency by increasing the number of partitions. The generation of partitions is also discussed according to the cost analysis.
5. We report an extensive experimental evaluation. Both the materialization of item views and the decomposition of tuple partitions are evaluated in querying over real data sets. Especially, the decomposition techniques can significantly improve the query time performance. Moreover, the hybrid approach which combines views and partitions together can always achieve the best performance and scales well under large data sizes. In addition, the experimental results also verify our conclusions of cost analysis, that is, we can improve the query performance by increasing the number of views but not that of partitions.

The remainder of this paper is organized as follows: first, we introduce the preliminary of this study in Section 2. Section 3 develops the planning of queries with materialization on views of items. In Section 4, we propose the pruning on partitions for merging and answering top-k queries. Section 5 reports our extensive experimental evaluation. We discuss the related work in Section 6. Finally, Section 7 concludes this paper.

2 Preliminary

In this section, we introduce some preliminary settings of existing work, including the query and index of dataspaces. The notations frequently used in this paper are listed in Table 1.

TABLE 1
Notations

Symbol	Description
T	A tuple instance from dataspaces
$\mathcal{I}$	The set of all the items in dataspaces
I_i	The i-th item in $\mathcal{I}$, $I_i \in \mathcal{I}$
$\mathcal{Q}$	Query log
Q	Query predicates of a query in $\mathcal{Q}$
$\hat{Q}$	Neighbor predicates of a query Q
$\mathcal{V}$	View scheme
V_i	The i-th view in $\mathcal{V}$, $V_i \in \mathcal{V}$
$\mathcal{H}$	Partition scheme
H_i	The i-th partition in $\mathcal{H}$, $H_i \in \mathcal{H}$
$\mathcal{P}$	Query plan of views
s_i	Length of the i-th list
c_i	Cost of retrieving the i-th list
v_{ij}	Indicate whether partition V_j contains item I_i

2.1 Data

We first introduce the model to represent the data. As the encoding system presented in [10], we can use pairs of (attribute : keyword) to represent the content of a tuple. For example, the attribute value (manu : Apple Inc.) can be represented by {(manu : Apple), (manu : Inc.)}, if each word is considered as a keyword. Let item I be a unique identifier of a distinct pair. We can represent each tuple T as a set of items, that is, $T = \{I_1, I_2, \ldots, I_{|T|}\}$.

Assume that $\mathcal{I}$ is the set of all the items in dataspaces. We use the vector space model [13] to logically represent the tuples.

Definition 2.1 (Tuple Vector). *Given a tuple T, the corresponding* tuple vector $\mathbf{t}$ *is given by*

$$\mathbf{t} = (t_1, t_2, \ldots, t_{|\mathcal{I}|}), \tag{1}$$

where t_i denotes the weight of item I_i in the tuple T, having $0 \leq t_i \leq 1$.

For example, the weight $t_i = 1$ of item I_i denotes $I_i \in T$; otherwise 0 means $I_i \notin T$. Advanced weight schemes, such as *term frequency* and *inverse document frequency* [13] in information retrieval, can also be applied. Without loss of generality, we adopt the *tf*idf* score in this work.

2.2 Attribute Correspondence

The correspondence between two attributes (e.g., manu versus prod) is often recognized by schema mapping techniques [9] in data integration. The main principles of techniques include data instances matching, linguistic matching of the schema element names, schema structural similarities, and domain knowledge including user feedback (see [9] for a survey). In dataspaces, the matching correspondence between attributes are often incrementally recognized in a pay-as-you-go style [5], e.g., gradually identified according to users' feedback when necessary.

Let A_i, B_i be two attributes with matching correspondence, denoted by $A_i \leftrightarrow B_i$. Any keywords w_i appearing in A_i, B_i are said *neighbors*. For instance, we consider a matching correspondence of attributes $\text{manu} \leftrightarrow \text{prod}$. It states that keywords w_i appearing in manu and prod are said neighbor keywords, e.g., (manu : Apple) and (prod : Apple). Since the correspondence between the same attribute is straightforward, a keyword can always be regarded as a neighbor to itself, such as (prod : Apple) and (prod : Apple).

2.3 Query

In this paper, we consider queries with a set of attribute and keyword predicates, e.g., (manu : Apple) and (post : Infinite). Thus, the query inputs can be represented in the same way as tuples in dataspaces.

As discussed in [10], the query with keyword neighborhood in schema level over dataspaces should not only consider tuples with matched keywords on the attributes specified in the query, but also extend to the attributes with correspondence according to the keyword neighborhood.

For example, we consider a query

$$Q = \{(\text{manu} : \text{Apple}), (\text{post} : \text{Infinite})\}.$$

The query evaluation searches not only in the manu and post attributes specified in the query, but also in the attributes prod and addr according to the attribute correspondences $\text{manu} \leftrightarrow \text{prod}$ and $\text{addr} \leftrightarrow \text{post}$, respectively.

Definition 2.2. *A* disjunctive query with keyword neighborhood in schema level, $Q = \{(A_1 : w_1), \ldots, (A_{|Q|} : w_{|Q|})\}$, *specifies a set of attribute-keyword predicates. It is to return all the tuples T in dataspaces with neighbor keywords to Q with respect to attribute correspondence, i.e., for an attribute A_i of Q, we can find a B_i of T, such that $A_i \leftrightarrow B_i$ and $(B_i : w_i) \in T$.*

Obviously, there may exist multiple attributes B_i associated to an attribute A_i according to $A_i \leftrightarrow B_i$. The disjunctive query only needs that one of them is true, i.e., considering "OR" logical operator between different attribute matching correspondences

$$\bigvee_{(B_i : w_i) \in T} A_i \leftrightarrow B_i.$$

For instance, we consider the above $Q = \{(\text{manu} : \text{Apple}), (\text{post} : \text{Infinite})\}$. According to the attribute matching correspondences $\text{manu} \leftrightarrow \text{prod}$ and $\text{addr} \leftrightarrow \text{post}$, a tuple T is considered as a candidate answer, if T either contains keyword Apple in manu or prod or contains Infinite in post or addr. Therefore, we can evaluate the query by finding all tuples T such that

$$\begin{aligned}&((\text{manu} : \text{Apple}) \in T \vee (\text{prod} : \text{Apple}) \in T)\\ &\quad \bigvee ((\text{post} : \text{Infinite}) \in T \vee (\text{addr} : \text{Infinite}) \in T).\end{aligned}$$

Let $\hat{Q}$ be the *neighbor predicates* of a query Q, i.e., a set of items with keyword neighborhood in schema level to the query Q,

$$\hat{Q} = \{(B_i : w_i) | B_i \leftrightarrow A_i, (A_i : w_i) \in Q\}.$$

The disjunctive query returns tuples T that match at least one predicate in $\hat{Q}$. For example, we have neighbor predicates $\hat{Q}$ for the above query $Q = \{(\text{manu} : \text{Apple}), (\text{post} : \text{Infinite})\}$ as follows:

$$\hat{Q} = \{(\text{manu} : \text{Apple}), (\text{prod} : \text{Apple}), (\text{post} : \text{Infinite}), (\text{addr} : \text{Infinite})\}.$$

Let $\mathbf{q}$ be the corresponding tuple vector of neighbor predicates $\hat{Q}$ of a query Q, where $q_i = 1$ for $I_i \in \hat{Q}$ and 0 otherwise. Then, the ranking score between any tuple T and the query Q can be computed by score aggregation functions on neighbor predicates. Without loss of generality, we should support any scoring function that satisfies monotonicity [11]. For example, we can consider the intersection of vectors of the tuple T and neighbor predicates $\hat{Q}$, which is widely used in keyword search studies [14]

$$score(\hat{Q}, T) = \|\mathbf{q} \cdot \mathbf{t}\| = \sum_{i=1}^{|\mathcal{I}|} q_i t_i = \sum_{I_i \in \hat{Q}} t_i. \quad (2)$$

Therefore, to evaluate the ranking score between Q and T, we are essentially required to compute $\sum_{I_i \in \hat{Q}} t_i$ with respect to neighbor predicates $\hat{Q}$.

It is notable that different attribute correspondences of an attribute require an OR operator. This "OR" semantics for the query with keyword neighborhood in schema level is different from the CompleteSearch [15]. Specifically, CompleteSearch indicates an "AND" operator of predicates. For example, in CompleteSearch, the query (manu : Apple), (prod : Apple) will return tuples containing both (manu : Apple) and (prod : Apple). Those tuples, which contain one of predicates or none of them, can be directly ignored. Instead, in dataspace query with OR operator, tuples having only part of the predicates will also be considered and ranked as candidates. Such OR logical semantics are necessary for querying with keyword neighborhood on attributes with correspondence, since more than one attribute may be associated to an attribute according to attribute correspondence and the query only needs that one of them is matched with respect to neighbor keywords.

2.4 Index

Indexing of dataspaces has been studied by Dong and Halevy [10], which extends inverted index for dataspaces. The *inverted index*, also known as *inverted files* or *inverted lists* [16], [17], [18], consists of a vocabulary of items $\mathcal{I}$ and a set of inverted lists. Each item I_i corresponds to an inverted list of tuple IDs, usually sorted in a certain order, where each ID reports the item weight t_i in that tuple.

In Fig. 2, we use an example to illustrate the index framework. The data set consists of 10 tuples (denoted by 1-10) with an item vocabulary $\mathcal{I} = \{(A : a), (B : a), \ldots, (L : g)\}$, having $|\mathcal{I}| = 12$. In the inverted lists, for each item (an attribute and keyword pair such as (manu : Apple)), we have a pointer referring to a specific list of tuple IDs, where the item appears. For instance, Fig. 2b shows an example of the inverted lists of item $(D : d)$, which indicates that the keyword d appears in the attribute D of tuples 2, 3, 5, 8, 10. In the real implementation, each tuple ID in the list is associated with a weight value t_i.

Definition 2.3. *Consider the neighbor predicates $\hat{Q}$ of a query Q. Let $\mathcal{L}$ be the set of lists corresponding to the items in neighbor predicates $\hat{Q}$, respectively. The* merge operator $\oplus$ *returns a new list of tuples, by merging the lists in $\mathcal{L}$, with $score(\hat{Q}, T)$ on each tuple T.*

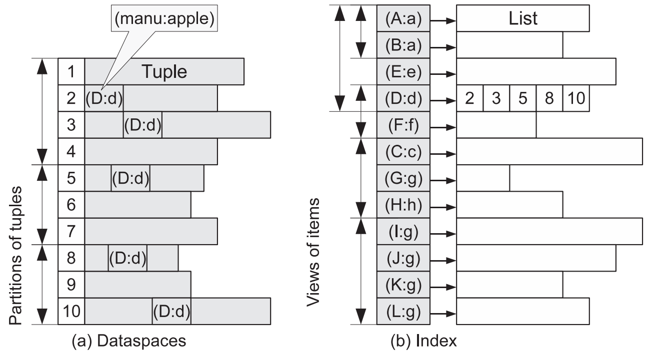


Fig. 2. Indexing dataspaces.

For example, consider a query $Q = \{(A : a), (C : c), (D : d)\}$. Suppose that there exists attribute correspondence $A \leftrightarrow B$. Thereby, we have the neighbor predicates $\hat{Q} = \{(A : a), (B : a), (C : c), (D : d)\}$. The merge operator computes $\sum_{I_i \in \hat{Q}} t_i$ for each tuple T that appears in the lists of items $(A : a), (B : a), (C : c), (D : d)$.

Let $c_i = O(s_i) = \pi s_i + r$, where π is a constant, s_i is the size of the list of item I_i and r is a constant time of random access. Then, the *merging cost* can be estimated by $\sum_{I_i \in \hat{Q}} c_i$.

Advanced merge methods, such as *threshold algorithm* (TA) [19], *combined algorithm* (CA) [11] or IO-Top-K [12], can be applied to merge inverted lists. When such TA-family methods are utilized, the above $\sum_{I_i \in \hat{Q}} c_i$ is then an upper bound of estimated merge cost. In the following, instead of proposing a new merge operator for inverted lists, we focus on advanced techniques that are built upon the available merge operators to further improve the query efficiency.

3 Planning with Materialization

According to the attribute correspondence, e.g., addr $\leftrightarrow$ post, each query predicate on attribute addr has to conduct a search on attribute post as well. In other words, those neighbor keywords on addr, post are often searched together in predicate queries with keyword neighborhood in schema level. Intuitively, we would like to cache these query results for reuse.

In relational databases, a view consists of the result set of a query, which can be materialized to optimize queries. Similarly, in this study, we introduce the view on a set of items (query predicates) in dataspaces to speed up the query processing. Specifically, the merge results of item sets are materialized, and then queries can utilize these materialized views. It raises two questions 1) how to select the views of items to materialize, and 2) how to utilize the materialized views to minimize the query cost.

Note that the techniques discussed as follows are also applicable in attribute-keyword search over structured and semi-structured data, since the data in dataspaces are modeled by attribute-keyword as well. However, due to the OR operator of predicates that should be considered for queries with keyword neighborhood in schema level, our current techniques can only support general attribute-keyword queries with OR operator.

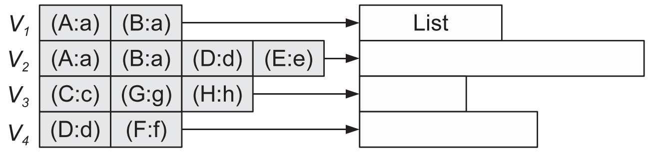


Fig. 3. Materialization on views of items.

3.1 Materialization

We first introduce the concept of materialization. Let *view* V be a set of items (attribute-keyword pairs). By applying the merge operation $\oplus$, we get a new list of tuples with corresponding $score(V, T)$. This list is stored in the disk as the materialization of view V.

In Fig. 3, we show an example of materialized lists of item views. For instance, we have neighbor keywords a in $(A : a)$ and $(B : a)$ according to the attribute correspondence $A \leftrightarrow B$. The first view, denoted by $V_1 = \{(A : a), (B : a)\}$, materializes the merge results of lists corresponding to items $(A : a)$ and $(B : a)$ in the example of Fig. 2. Those items appearing together frequently may also be materialized as well, e.g., $V_4 = \{(D : d), (E : e)\}$ where $(D : a), (E : e)$ may frequently appear together in tuples or queries.

Let $\mathcal{V}$ denote the view scheme, i.e., the set of views that are materialized. Since all the original items are already stored, we can treat each item as a single size view, that is, $\mathcal{I} \subseteq \mathcal{V}$.

3.2 Standard Query Plan

Given a query Q, a query plan $\mathcal{P}$ of Q is a set of views, having $\mathcal{P} \subseteq \mathcal{V}$, which can be used to evaluate $score(\hat{Q}, T)$ for each possible tuple T. Since various views are available in a view scheme $\mathcal{V}$, it leads to study the selection of optimal plan $\mathcal{P}$ with the minimum query cost.

3.2.1 Formalization

We first formalize the definition of query plan $\mathcal{P}$. Let $v_{ij} = 1$ denote that view V_j contains item I_i; otherwise, $v_{ij} = 0$ means not containing, having $i = 1, \ldots, |\mathcal{I}|$ and $j = 1, \ldots, |\mathcal{V}|$. Let $q_i = 1$ denote that item I_i is contained in the neighbor predicates $\hat{Q}$ of a query Q; otherwise, $q_i = 0$, having $i = 1, \ldots, |\mathcal{I}|$.

Definition 3.1. *Given a query Q, a feasible* standard plan *$\mathcal{P}$ is a subset of all views, $\mathcal{P} \subseteq \mathcal{V}$, having*

$$\sum_j v_{ij} x_j = q_i, \quad i = 1, \ldots, |\mathcal{I}|,$$

$$x_j = \begin{cases} 1, & \text{if } V_j \in \mathcal{P}, \\ 0, & \text{if } V_j \notin \mathcal{P}, \end{cases} \quad j = 1, \ldots, |\mathcal{V}|.$$

During the query evaluation, lists corresponding to the views in $\mathcal{P}$ are merged by using the merge operator $\oplus$ in Definition 2.3. It is notable that a feasible plan requires $\sum_j v_{ij} x_j = q_i$ as illustrated in Definition 3.1, i.e., no duplicate items in $\mathcal{P}$. As presented in the following, such requirement is necessary for computing the ranking scores of tuples. We first prove that the standard plan can evaluate $score(\hat{Q}, T)$ for each possible tuple T.

Lemma 1. *Let $\mathcal{P}$ be a feasible query plan for a query Q. For any tuple T, we have $score(\hat{Q}, T) = \sum_{V \in \mathcal{P}} score(V, T)$.*

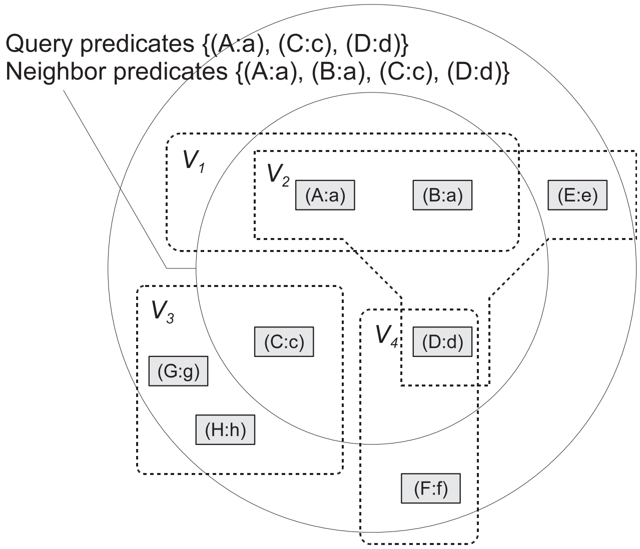


Fig. 4. Plan selection.

Proof. According to the score function in (2), we have

$$\begin{aligned} \sum_{V \in P} score(V, T) &= \sum_j x_j score(V_j, T) \\ &= \sum_j x_j \sum_i v_{ij} t_i \\ &= \sum_i \sum_j x_j v_{ij} t_i \\ &= \sum_i q_i t_i = score(\hat{Q}, T), \end{aligned}$$

where $i = 1, \ldots, |\mathcal{I}|$ and $j = 1, \ldots, |\mathcal{V}|$. □

For example, we consider a query $Q = \{(A : a), (C : c), (D : d)\}$ in Fig. 4. According to the attribute correspondence, $A \leftrightarrow B$, we have

$$\hat{Q} = \{(A : a), (B : a), (C : c), (D : d)\}.$$

Suppose that the view $V_1 = \{(A : a), (B : a)\}$ is materialized. A feasible standard plan can be $\mathcal{P} = \{V_1, (C : c), (D : d)\}$. The formula $\sum_j v_{ij} x_j = q_i$ semantically denotes that the union of views $V_j \in \mathcal{P}$ is exactly the neighbor predicates $\hat{Q}$ of query Q, i.e., $\bigcup_{V_j \in \mathcal{P}} V_j = \hat{Q}$. In fact, we can further develop the following properties of feasible plans.

Lemma 2. *For any view V in a feasible standard plan $\mathcal{P}$, we have $V \subseteq \hat{Q}$.*

Proof. Assume that there exists an item I_i having $I_i \in V$ but $I_i \notin \hat{Q}$. Thus, we have $\sum_j v_{ij} x_j \geq 1 > q_i = 0$, which contradicts the definition of feasible $\mathcal{P}$. □

Lemma 3. *For any two views V_1 and V_2 in a feasible standard plan $\mathcal{P}$, we have $V_1 \cap V_2 = \emptyset$.*

Proof. Assume that there exists an item I_i having $I_i \in V_1 \cap V_2$. Thus, we have $\sum_j v_{ij} x_j \geq 2 > 1 \geq q_i$, which contradicts the definition of feasible $\mathcal{P}$. □

3.2.2 Optimal Plan

First, recall that all the original items are already materialized, i.e., $\mathcal{I} \subseteq \mathcal{V}$. Therefore, given any query Q, a feasible plan always exists, that is, $\mathcal{P} = \hat{Q}$. Next, we study the selection of query plans with the minimum cost. Let c_j be the cost of retrieving the materialized list of view V_j as defined in Section 2. Then, the cost of plan $\mathcal{P}$ can be estimated by $\sum_j c_j x_j$.

Definition 3.2. *The problem of selecting the optimal standard query plan is to determine the* $\mathbf{x}$ *for* $\mathcal{P}$, *having*

$$\begin{aligned} \text{minimize} \quad & \sum_j c_j x_j \\ \text{subject to} \quad & \sum_j v_{ij} x_j = q_i, \quad i = 1, \dots, |\mathcal{I}|, \\ & x_j \in \{0, 1\}, \quad j = 1, \dots, |\mathcal{V}|, \end{aligned}$$

which is a 0-1 integer programming *or* binary integer programming *problem.*

Unfortunately, the *0-1 integer programming* problem with nonnegative data is equivalent to the *set cover* problem, which is NP-complete [20]. Therefore, we explore the approximate solutions by greedy algorithm.[4]

3.2.3 Greedy Algorithm

Intuitively, in each step of adding a view into $\mathcal{P}$, we can greedily select the view V_j with the minimum cost of each item unit, i.e., the minimum ratio $\frac{c_j}{|V_j|}$, where c_j denotes the cost of V_j and $|V_j|$ means the size of view V_j, such as $|V_j| = \sum_{I_i \in V_j} s_i$.

According to Lemma 2, not all the views $V_j \in \mathcal{V}$ should be considered for a specific Q. Instead, as presented in line 3 of Algorithm 1, we only need to evaluate the views that are contained by neighbor predicates $\hat{Q}$, i.e., $V_j \subseteq \hat{Q}$. Moreover, since any two views in a feasible plan are nonoverlapping (Lemma 3), we can remove the items of the currently selected view V_k from $\hat{Q}$ in each step and stop till $\hat{Q} = \emptyset$.

Algorithm 1. Standard Planning **SP**(Q)

1: $\mathcal{P} := \emptyset$
2: $\hat{Q} :=$ neighbor predicates of Q according attribute correspondence
3: **while** $\hat{Q} \neq \emptyset$ **do**
4: $\quad k := \arg\min_j \frac{c_j}{|V_j|}, V_j \subseteq \hat{Q}$
5: $\quad \mathcal{P} := \mathcal{P} \cup V_k$
6: $\quad \hat{Q} := \hat{Q} \setminus V_k$
7: **return** $\mathcal{P}$

Let d be the size of the largest view $V_j \subseteq \hat{Q}$ and H_d be the dth *harmonic number*, having $H_d = \sum_{k=1}^{d} \frac{1}{k}$. Then, the relative error of greedy approximation is bounded as follows:

Corollary 1. *[22] The cost of the plan returned by the greedy algorithm SP is at most H_d times the cost of the optimal plan.*

3.3 General Query Plan

Note that the standard plan only contains views that are subsets of the neighbor predicates $\hat{Q}$ of a query Q. However, the views with items not in $\hat{Q}$ can be used in the query evaluation as well. For example, in Fig. 4, we can also utilize the view V_2 by removing the item $(E : e)$ (not requested by $\hat{Q}$) from V_2. Moreover, if both V_2 and V_4 are considered, then the item $(D : d)$ will be counted twice in the score function which contradicts to the correctness in Lemma 1. Therefore, we need to deduct the items such as nonrequested $(E : e)$ or duplicate $(D : d)$. The operator of removing corresponding lists is defined as follows, namely, the *negative merge operator*.

Definition 3.3. *Consider a set of items, e.g., V. Let $\mathcal{L}$ be the set of lists corresponding to the items in V, respectively. The* negative merge operator $\ominus$ *returns a new list of tuples, by merging the lists in $\mathcal{L}$, with negative-score(V, T) on each tuple T.*

It is notable that the negative merge employs negative score values, which still satisfy the monotonicity of score functions. Therefore, TA family methods [11] can still be utilized for negative merge. As illustrated in the following, lists with both positive scores and negative scores in a general query plan are merged together by using a TA-style algorithm.

We define the general query plan with both the merge operator $\oplus$ and the negative merge operator $\ominus$. Let v_{ij} and q_i have the same semantics as the standard plan in Definition 3.1.

Definition 3.4. *Given a query Q, a* general plan *$\mathcal{P}$ consists of two subsets of all views, $\mathcal{P}^{\oplus} \subseteq \mathcal{V}$ and $\mathcal{P}^{\ominus} \subseteq \mathcal{V}$, having*

$$\begin{aligned} & \sum_j v_{ij}(x_j^+ - x_j^-) = q_i, && i = 1, \dots, |\mathcal{I}|, \\ & x_j^+ = \begin{cases} 1, & \text{if} \quad V_j \in \mathcal{P}^{\oplus}, \\ 0, & \text{otherwise}, \end{cases} && j = 1, \dots, |\mathcal{V}|, \\ & x_j^- = \begin{cases} 1, & \text{if} \quad V_j \in \mathcal{P}^{\ominus}, \\ 0, & \text{otherwise}. \end{cases} && j = 1, \dots, |\mathcal{V}|, \end{aligned}$$

Similar to Lemma 1, we can also prove that $score(\hat{Q}, T) = \sum_{V \in \mathcal{P}} score(V, T)$ for the general plan $\mathcal{P}$. During the query evaluation, the merge operator $\oplus$ and negative merge operator $\ominus$ are then conducted on $\mathcal{P}^{\oplus}$ and $\mathcal{P}^{\ominus}$, respectively. For example, a general plan for the query Q can be $\mathcal{P} = \{V_2^{\oplus}, (C : c)^{\oplus}, (E : e)^{\ominus}\}$. The above definition specifies a constraint that the union of view in $\mathcal{P}^{\oplus}$ minus the union of views in $\mathcal{P}^{\ominus}$ is exactly $\hat{Q}$ of the query Q, i.e., $(\bigcup_{V_j \in \mathcal{P}^{\oplus}} V_j) \setminus (\bigcup_{V_j \in \mathcal{P}^{\ominus}} V_j) = \hat{Q}$. In fact, the standard plan is a special case of the general plan, where $\mathcal{P}^{\ominus} = \emptyset$.

3.3.1 Optimal Plan

We then introduce the problem of selecting the optimal general plan with the minimum cost. The only difference between two kinds of merge operators is their outputs of scores, while the cost of the negative merge operator is actually the same as the merge operator. Thereby, we can estimate the cost of a general plan $\mathcal{P}$ by $\sum_j c_j x_j^+ + c_j x_j^-$.

Definition 3.5. *The problem of selecting the optimal general query plan is to determine the $\mathbf{x}^+$ for $\mathcal{P}^{\oplus}$ and the $\mathbf{x}^-$ for $\mathcal{P}^{\ominus}$, having*

4. Advanced approximation approaches on solving the binary integer programming problem can also be adopted [21], which is not the focus of this paper.

$$\begin{aligned}\text{minimize}\ & \sum_j c_j x_j^+ + c_j x_j^- \\ \text{subject to}\ & \sum_j v_{ij} x_j^+ - v_{ij} x_j^- = q_i, \quad i = 1, \ldots, |\mathcal{I}|, \\ & x_j^+ \in \{0,1\}, \quad j = 1, \ldots, |\mathcal{V}|, \\ & x_j^- \in \{0,1\}, \quad j = 1, \ldots, |\mathcal{V}|,\end{aligned}$$

which is exactly the 0-1 integer programming *problem. However, the coefficients of variables* x_j^- *are negative.*

Lemma 4. *For an optimal general plan* $\mathcal{P}$, *we have* $\mathcal{P}^{\oplus} \cap \mathcal{P}^{\ominus} = \emptyset$.

Proof. Assume that there exists a view $V_j \in \mathcal{P}^{\oplus} \cap \mathcal{P}^{\ominus}$. Then, we can build another feasible general plan $\mathcal{P}_1$, $\mathcal{P}_1^{\oplus} = \mathcal{P}^{\oplus} \setminus V_j$ and $\mathcal{P}_1^{\ominus} = \mathcal{P}^{\ominus} \setminus V_j$, whose cost is less than $\mathcal{P}$. □

As proved in by Dobson [23], when there are negative entries, it is unlikely that we can guarantee the existence of a polynomial approximation scheme with relative error bounds. Therefore, we study the greedy heuristics.

3.3.2 Heuristics

First, we introduce a virtual (empty) view V_0 with cost $c_0 = 0$. Then each $V_j \subseteq \hat{Q}$ can be represented by $\{V_j^{\oplus}, V_0^{\ominus}\}$. Next, we consider possible pairs of $\{V_j^{\oplus}, V_l^{\ominus}\}$ that can be used by the query $V_j \setminus V_l \subseteq \hat{Q}$, where $j = 1, \ldots, |\mathcal{V}|$ and $l = 0, \ldots, j-1, j+1, \ldots, |\mathcal{V}|$, having $l \neq j$ according to Lemma 4. Let the ratio be $\frac{c_j + c_l}{|V_j \setminus V_l|}$, which denotes the average cost of retrieving each unit of items. As presented in Algorithm 2, similar to the SP algorithm, we can greedily select the view pair $\{V_{k_1}, V_{k_2}\}$ with the minimum ratio in each step. The view V_{k_1} is considered to be in $\mathcal{P}^{\oplus}$, while V_{k_2} is added into $\mathcal{P}^{\ominus}$.

Algorithm 2. General Planning **GP**(Q)
1: $\mathcal{P}^{\oplus} := \mathcal{P}^{\ominus} := \emptyset$
2: $\hat{Q} :=$ neighbor predicates of Q according attribute correspondence
3: **while** $\hat{Q} \neq \emptyset$ **do**
4: $(k_1, k_2) := \arg\min_{j,l} \frac{c_j + c_l}{|V_j \setminus V_l|}, V_j \setminus V_l \subseteq \hat{Q}$
5: $\mathcal{P}^{\oplus} := \mathcal{P}^{\oplus} \cup V_{k_1}$
6: $\mathcal{P}^{\ominus} := \mathcal{P}^{\ominus} \cup V_{k_2}$
7: $\hat{Q} := (\hat{Q} \cup V_{k_2}) \setminus V_{k_1}$
8: **return** $\mathcal{P}^{\oplus}, \mathcal{P}^{\ominus}$

Corollary 2. *The cost of the general plan returned by GP algorithm is at least no greater (worse) than the cost of the standard plan returned by SP algorithm.*

Proof. The worst case is $\mathcal{P}^{\ominus} = \emptyset$, i.e., $x_j^- = 0$ for all j, which is exactly the solution of the SP algorithm. □

3.4 Generating Views

Now we present how to generate the view scheme $\mathcal{V}$. Obviously, the larger the number of views in $\mathcal{V}$ is, the better the query performance will be. Let $\mathcal{S}$ be the space of all possible views on the item set $\mathcal{I}$. The ideal scenario is to materialize all the possible views, i.e., $\mathcal{V} = \mathcal{S}$, when the space of materialization is not limited. However, real applications usually have a constraint on the maximum available disk space, say M, for materialization. The problem we address is to determine a $\mathcal{V} \subseteq \mathcal{S}$ with disk space less than M.

When there is no query log available in the beginning, we can randomly generate views as $\mathcal{V}$. Let $x_j = 1$ denotes that the view $V_j \in \mathcal{S}$ is selected to materialize in the view scheme $\mathcal{V}$; otherwise $x_j = 0$. Then the disk space cost can be $size(\mathcal{V}) = \sum_j s_j x_j$. The random generation of view stops when the space cost $size(\mathcal{V})$ exceeds the limitation M.

After processing a batch of queries, we can rely on the query log to select views for materialization. Let $\mathcal{Q}$ be a set of query tuples, i.e., the query log. The straightforward strategy is to materialize the views of item sets V_j that appear most frequently in the query log $\mathcal{Q}$. This interesting intuition leads us to the famous frequent itemset mining algorithms [24], [25]. Each query log can be treated as a transaction with a set of items. Then, we can select the frequent k-itemsets as materialized views. Existing efficient algorithms can be utilized, such as Apriori [24] or FP-growth [25], which are not the focuses of this paper. However, this frequency-based strategy fails to offer optimal views due to the favor of views with smaller sizes, e.g., the frequency of $\{(C:c), (D:d)\}$ is always not less than $\{(C:c), (D:d), (E:e)\}$.

3.4.1 Cost-Based Generation

We seek the view scheme that can minimize the cost of the query log. Let $q_{ik} = 1$ denote that the item I_i is contained in the neighbor predicates $\hat{Q}_k$ of a query Q_k; otherwise, not containing. Let $y_{jk} = 1$ denote that the view $V_j \in \mathcal{S}$ is expected to be used in the query tuple Q_k, no matter V_j is selected in $\mathcal{V}$ ($x_j = 1$) or not.

Definition 3.6. *The problem of generating the optimal view scheme* $\mathcal{V}$ *is to determine a feasible* $\mathbf{x}$ *having,*

$$\begin{aligned}\text{minimize}\quad & \sum_{j,k} c_j x_j y_{jk} \\ \text{subject to}\quad & \sum_j v_{ij} x_j y_{jk} = q_{ik},\ i = 1, \ldots, |\mathcal{I}|,\ k = 1, \ldots, |\mathcal{Q}| \\ & \sum_j s_j x_j \leq M \\ & x_j \in \{0,1\}, \quad j = 1, \ldots, |\mathcal{S}|, \\ & y_{jk} \in \{0,1\}, \quad j = 1, \ldots, |\mathcal{S}|, \quad k = 1, \ldots, |\mathcal{Q}|.\end{aligned}$$

Then, $\mathcal{V} = \{V_j \mid x_j = 1, V_j \in \mathcal{S}\}$.

Similar to the query planning, we also study greedy heuristics to solve this problem. Let $f_j = \sum_k y_{jk}$ be the frequency of V_j in the neighbor predicates of query log $\mathcal{Q}$. Similarly, we can develop the greedy heuristic by the ratio

$$\frac{c_j}{|V_j| \sum_k y_{jk}} = \frac{c_j}{|V_j| f_j}.$$

In each greedy step, we select the view V_j to $\mathcal{V}$ which has the minimum ratio, or equivalently, the V_j that can cover maximum number of items ($|V_j| f_j$) by each unit of cost $\frac{1}{c_j}$.

The cost-based view generation algorithm is developed as follows: for the initialization from lines 2-5 in Algorithm 3, we assume that each view $V_j \subseteq \hat{Q}_k$ can possibly be used, i.e., assigning $y_{jk} = 1$. Therefore, we have $f_j{+}{+}$ in line 5 when $V_j \subseteq \hat{Q}_k$. During each greedy step in lines 6-12, once we decide to select a view (say V_l) into $\mathcal{V}$, then all the other views V_j (having $V_j \subseteq \hat{Q}_k$ and $V_j \cap V_l \neq \emptyset$ according to Lemmas 2 and 3) are impossible to be used, i.e., assigning $y_{jk} = 0$. In other words, we should remove such $y_{jk} = 1$ from f_j by $f_j --$ in line 12. The program terminates when $\mathcal{Q} = \emptyset$ or the disk space $size(\mathcal{V})$ exceeds the limitation M.

Algorithm 3. View Generation $\mathbf{VG}(\mathcal{Q})$

1: $f_j := 0$
2: **for** $j : 1 \rightarrow |\mathcal{S}|$ **do**
3: **for** $k : 1 \rightarrow |\mathcal{Q}|$ **do**
4: **if** $V_j \subseteq \hat{Q}_k$ **then**
5: $f_j ++$
6: **while** $\mathcal{Q} \neq \emptyset$ and $size(\mathcal{V}) \leq M$ **do**
7: $l := \arg\min_j \frac{c_j}{|V_j| f_j}$
8: $\mathcal{V} := \mathcal{V} \cup V_l$
9: **for** $k : 1 \rightarrow |\mathcal{Q}|$ **do**
10: **if** $V_l \subseteq \hat{Q}_k$ **then**
11: $\hat{Q}_k := \hat{Q}_k \setminus V_l$
12: $f_j --$ for V_j that $V_j \cap V_l \neq \emptyset$
13: **return** $\mathcal{V}$

Corollary 3. *For any $\mathcal{V}$ generated by the VG algorithm, the total cost of the queries in $\mathcal{Q}$ by using the optimal standard plan for each query tuple is no worse than the cost of $\sum_{j,k} c_j x_j y_{jk}$.*

According to the greedy strategy of selecting views, the first chosen ones can have more effectiveness in reuse, while those views ranked lower in the generation may benefit the queries less. Note that some of the views have extremely low frequency when considering the entire view scheme space. Although the view generation is offline processing, we can select a candidate subset of views from the entire space as $\mathcal{S}$, e.g., with frequency greater than a threshold in the query log.

3.4.2 Updates

We mainly have two aspects of updates in dataspaces, i.e., updates of tuples and updates of attribute correspondences. For the updates of tuples, we consider the inserting and deleting tuples. During the updating, inverted lists of both original items and their corresponding materialized views should be addressed. Efficient approaches have already been developed for updating inverted lists [26], which can be applied as well.

It is notable that the attribute correspondences between attributes in dataspaces are often incrementally recognized in a pay-as-you-go style [5]. The neighbor predicates with respect to neighbor keywords of query workload evolve as well. Consequently, it leads to the updates of views in $\mathcal{V}$. For the frequency-based view scheme, it is easy to update the frequency statistics, remove low frequency item patterns in updated query predicates and add high frequency item patterns to $\mathcal{V}$. Those views whose frequencies decrease may be replaced by new frequent views. For the cost-based view scheme, however, we can only rely on batch updates, due to the maintenance of y_{jk} in f_j for each specific view V_j. Note that if the updates have to be conducted online, the cost of updating should be considered as well. Consequently, there will be a trade-off between the view update cost and the query cost with respect to workload. As presented, the generation of views has already been shown hard. Therefore, it is highly nontrivial to find optimal updates of views with respect to the balance of update cost and query cost.

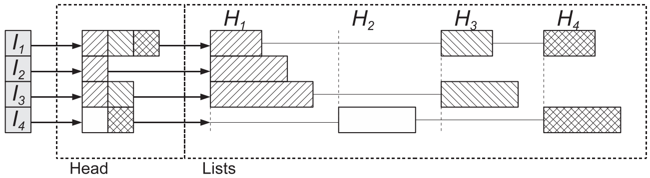


Fig. 5. Decomposition on partitions of tuples.

4 Merging with Decomposition

Instead of searching data in the entire space, we often decompose the data into partitions for efficient top-k queries. During the query processing, those partitions of tuples with low scores to the query are then pruned directly without evaluation. In this section, we also study the decomposition of dataspaces for efficient query. Again, we have to address two questions: 1) how to prune the partitions of tuples on top-k answers, and 2) how to generate the partitions of tuples which will have less query cost.

Recall that during the evaluation of a query, we rely on the merge operator to merge the lists referred by the query plan. That is, given a set of lists,[5] we study the techniques for efficiently merging the lists to return top-k answers. Since all the tuples are decomposed into a set of partitions, and the merge operator is then applied on each partition of tuples, respectively. Given a query, we can develop the bound of scores of the tuples in each partition. Therefore, those partitions whose score bounds are lower than the top-k answers can be pruned.

It is notable that we utilize the previous merge operators such as TA family methods [11] to rank the answers in each partition. Instead of proposing a new top-k ranking method, our partitioning technique is regarded as a complementary work to the previous merge operators. Thereby, advanced merge methods, such as IO-top-k [12], can be cooperated together with our decomposition.

4.1 Decomposition

Let $\mathcal{H}$ denotes a partition scheme, i.e., a set of m nonoverlapping partitions of all the tuples. In other words, each tuple T is assigned to one and only one partition $H_i \in \mathcal{H}, i = 1, \ldots, m$.

Thereby, each list can be decomposed to a set of nonoverlapping sublists of tuples according to the partitions of tuples. For example, as illustrated in Fig. 5, each list can be decomposed into at most $m = 4$ partitions. Some partitions might be empty in a specific list. For instance, the list of item $(A : a)$ say I_1 in Fig. 2b is decomposed into three partitions, H_1, H_3, and H_4, while H_2 is empty. It states that the item I_1 does not appear in any tuple in partition H_2.

In order to compute the score bounds of the tuples in each partition, we introduce a *head* structure for each list. The head stores the following information: 1) partition ID, 2) the bound of item weights in the partition, and 3) the pointer of start and offset of the partition in the list. Both the head and

5. Corresponding to either original items or views. For simplicity, in the remainder of this section, we use the example of original items, which is the same for views.

the lists of tuple partitions of an item are stored in continuous disk blocks and can be retrieved in one random access as the original lists.

4.1.1 Updating

During the updating (insertion or deletion of tuples), both the lists and the head information should be updated, including the bound of weight and also the pointers to the partitions.

4.2 Pruning Top-k Answers

4.2.1 Bound

In order to evaluate the score bounds of the tuples in a partition, we first introduce the formal representation of partitions. Note that each partition also describes a set of items, which appear in the tuples of this partition. Therefore, similar to the tuple vector, each partition can be logically represented by a *partition vector* of items.

Definition 4.1 (Partition Vector). *Let H be a partition in $\mathcal{H}$. The corresponding* partition vector *is defined by,*

$$\mathbf{h} = (h_1, h_2, \ldots, h_{|\mathcal{I}|}), \tag{3}$$

where h_i is the bound of weight of the item I_i in the partition H. Specifically, let T be any tuple in the partition H. We have

$$h_i = \max_{T \in H}(t_i), \tag{4}$$

where t_i is the weight of item I_i in the tuple T.

Given a query Q, we can compute an intersection score between any partition H and the neighbor predicates $\hat{Q}$ of Q,

$$score(\hat{Q}, H) = \|\mathbf{q} \cdot \mathbf{h}\| = \sum_{I_i \in \hat{Q}} h_i.$$

As presented in the following, this $score(\hat{Q}, H)$ is exactly the upper bound of scores of the tuples in the partition H.

Lemma 5. *Let T be any tuple in a partition H, we have*

$$score(\hat{Q}, H) \geq score(\hat{Q}, T), \tag{5}$$

where Q is the query.

Proof. According to the definition of partition vector in (4), for any item I_i, we have $h_i \geq t_i$. Therefore,

$$score(\hat{Q}, H) = \sum_{I_i \in \hat{Q}} h_i \geq \sum_{I_i \in \hat{Q}} t_i = score(\hat{Q}, T).$$

In other words, $score(\hat{Q}, H)$ is the bound of scores of tuples $T \in H$ to the query Q. □

When a list of item I_i is manipulated by the negative merge operator, e.g., in a general query plan, we can assign $h_i = 0$ for each partition H. For the tuple $T \in H$ containing item I_i, we have $t_i > 0$, i.e., $-t_i < 0$ in negative merge. Moreover, for the tuple $T' \in H$ that does not contain the item I_i, we have $t_i = -t_i = 0$. Therefore, we can assign $h_i = 0$ for this item I_i for simplicity.

4.2.2 Pruning

Next, we can order the partitions in decreasing order of their upper score bounds to the query. After processing the first g partitions with the highest bounds, we obtain a current top-k answer, say K. The following theorem specifies the condition of pruning the next $g+1$ partition.

Theorem 1. *Let K_k be the kth tuple with the minimum score in the top-k answers in the previous g steps. For the next $g+1$ partition, if we have*

$$score(\hat{Q}, K_k) \geq score(\hat{Q}, H_{g+1}), \tag{6}$$

then the partition H_{g+1} can be safely pruned.

Proof. According to Lemma 5, for any tuple T in the partition H_{g+1}, we have $score(\hat{Q}, K_k) \geq score(\hat{Q}, H_{g+1}) \geq score(\hat{Q}, T)$. Since K_k is the tuple in the current top-k results with the minimum ranking score, in other words, the tuples in partition H_{g+1} will never be ranked higher than K_k and can be pruned safely without further evaluation. □

For the remaining partitions $H_{g+2}, H_{g+3}, \ldots$, since the partitions are in the decreasing order of score bounds, we have

$$score(\hat{Q}, K_k) > score(\hat{Q}, H_{g+1}) > score(\hat{Q}, H_{g+x}),$$

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

For example, we consider the query Q with neighbor predicates $\hat{Q} = \{I_1, I_2, I_3, I_4\}$ in Fig. 5. Suppose that the partitions are ordered by the bounds as follows, H_1, H_3, H_4, H_2. After processing the first partition H_1, if the current kth answer K_k has a score higher than the bound of next the partition H_3, then we can prune all the remaining partitions H_3, H_4, and H_2 without evaluating their lists of tuples.

4.2.3 Algorithm

Given a query Q and an integer k, the query algorithm is described in the following Algorithm 4. During the initialization, the PARTITIONS($\hat{Q}$) function returns a set of partitions $\mathcal{H}$ ranked in descending order of score bounds to the query Q. Recall that the bound h_i of item I_i of each partition H is recorded in the head structure as illustrated in Fig. 5. Thus, the bound of scores of each partition can be efficiently computed by merging the heads of all the items referred in the neighbor predicates $\hat{Q}$ of Q.

Algorithm 4. Merge Top-k **MT**(Q, k)

$\hat{Q} :=$ neighbor predicates of Q according attribute correspondence
$\mathcal{H} := \mathrm{PARTITIONS}(\hat{Q})$
$K := \emptyset$
for $j : 1 \rightarrow \mathcal{H}.size$ **do**
  if $score(\hat{Q}, K_k) \geq score(\hat{Q}, H_j)$ **then**
    break
  else
    $K' := \mathrm{MERGE}(H_j.lists)$
    $K := \mathrm{RANK}(K, K')$
return K

Let $score(\hat{Q}, K_k)$ be the kth largest score in the current top-k answers K. For the partition H_j, if the $score(\hat{Q}, K_k)$ is larger than the bound of the tuple scores of partition

$score(\hat{Q}, H_j)$, then the partition H_j and all the remaining partitions can be pruned. Otherwise, we merge and rank all the tuples in the partition H_j. The MERGE$(\cdot)$ function is the implementation of the merge operator introduced in Definition 2.3 or Definition 3.3.

4.3 Cost Analysis

Suppose that we have $m = |\mathcal{H}|$ partitions on the n tuples in the dataspaces. We analyze the cost of disk space and query time.

4.3.1 Space Cost

Let $O(n)$ be the space cost of the original inverted lists of all the n tuples. We have $\frac{n}{m}$ tuples in each partition on average. Thus, the space cost introduced by the partition information can be estimated by $\frac{m}{n}O(n)$. The total space cost with partitions is $(1+\frac{m}{n})O(n)$. Since the number of partitions is always less than tuples, $m \le n$, the space cost is at most twice the cost of original inverted lists.

4.3.2 Time Cost

Again, let $O(n)$ be the time cost of merging on the entire space of n tuples. Suppose that the pruning is conducted after processing the first g partitions. Then, we can estimate the time cost of the query with pruning as follows:

Lemma 6. *The cost of processing first g partitions can be estimated by*

$$\left(\frac{m}{n} + \frac{g}{m}\right)O(n).$$

Proof. The merge cost with pruning partitions consists of two aspects, from the merge of partitions and tuples respectively. First, $\frac{m}{n}O(n)$ denotes the merge of partitions, in order to calculate the bounds of all the m partitions. Moreover, according to the pruning, all the remaining $m-g$ partitions can be ignored without evaluation. That is, the merge cost of tuples would be $\frac{g}{m}$ of the original cost without partition pruning $O(n)$. □

Now, we study the estimation of the number of partitions that are processed before pruning, i.e., the g value. Intuitively, we want to estimate the probability $\frac{g}{m}$ that a partition is processed in a query, i.e., the probability of a partition having bound greater than the top-k answers. Therefore, we introduce the concept of *correlation integral* [27] $C(\epsilon)$ which denotes the mean probability that two objects from two sets, respectively, are similar (with similarity greater than ϵ). Let $|X|$ be the size of object set X and $\mathbf{x}_i$ be an object in X. Let $|Y|$ be the size of object set Y and $\mathbf{y}_i$ be an object in Y. Then, for a specific similarity value ϵ, the correlation integral $C(\epsilon)$ can be approximated by the correlation sum

$$C(\epsilon) = \frac{1}{|X|\cdot|Y|}\sum_{i=1}^{|X|}\sum_{j=1}^{|Y|}\Theta(\|\mathbf{x}_i \cdot \mathbf{y}_j\| - \epsilon), \tag{7}$$

where $\Theta(\cdot)$ is the *heaviside function*, $\Theta(x) = 1$ for $x \ge 0$ and 0 otherwise, and $\|\cdot\|$ is the intersection similarity.

Let X be the set of query tuples and Y be the set of data tuples in dataspaces. Then, the correlation integral is the probability that the similarity between a query and a tuple T is greater than ϵ, denoted by $C_t(\epsilon)$. Moreover, if we define the set Y to be the set of partition vectors in dataspaces, then the correlation integral, say $C_h(\epsilon)$, means the probability that a query has high similarity (greater than ϵ) to the bound of a partition H. During the computation, if a query workload is provided, then we can directly use query tuples as X. However, if query workload is not available, then we can only rely on the data itself, i.e., using data tuples as X as well.

According to the fractal and self-similarity features, which have been observed in various applications including the high dimension spaces [28], [29], [30], there exists a constant, known as *correlation dimension* [27] D,

$$D = \frac{\partial \log C(\epsilon)}{\partial \log \epsilon}. \tag{8}$$

We observe the D_t corresponding to $C_t(\epsilon)$ of tuples and D_h corresponding to $C_h(\epsilon)$ of partitions in real data sets of dataspaces. As presented in Figs. 9, 10, and 11, a straight line can be fit in each plot, respectively, whose slope is exactly the constant D according to the above definition.

According to the constant D in (8), we can represent the relationship among ϵ, D and $C(\epsilon)$ as follows:

$$C(\epsilon) \propto \epsilon^D, \tag{9}$$

where $\propto$ stands for *proportional*, i.e., follows the power law. Let $C(\epsilon) = (\phi\epsilon)^D$, where ϕ is a constant and can be observed together with slope D in Figs. 9, 10, and 11.

Lemma 7. *The number of processed partitions g can be estimated by*

$$g \approx m\left(\frac{\phi_h}{\phi_t}\right)^{D_h}\left(\frac{k}{n}\right)^{\frac{D_h}{D_t}}.$$

Proof. Recall that $C(\epsilon)$ denotes the mean probability that the similarity is greater than ϵ. Let ϵ_t be the minimum similarity of the top-k answers. Then, we have $\frac{k}{n} = C_t(\epsilon) = (\phi_t\epsilon_t)^{D_t}$. Moreover, let ϵ_h be the bound of similarity scores of the gth partition. Thus, we also have $\frac{g}{m} = C_h(\epsilon) = (\phi_h\epsilon_h)^{D_h}$.

According to the pruning condition of partitions, we have the similarity score $\epsilon_t \approx \epsilon_h$, that is,

$$\frac{1}{\phi_t}\left(\frac{k}{n}\right)^{\frac{1}{D_t}} \approx \frac{1}{\phi_h}\left(\frac{g}{m}\right)^{\frac{1}{D_h}}.$$

In other words, we have

$$g \approx m\left(\frac{\phi_h}{\phi_t}\right)^{D_h}\left(\frac{k}{n}\right)^{\frac{D_h}{D_t}}.$$

The lemma is proved. □

Combining Lemmas 6 and 7, we have the following conclusion. Let γ be

$$\gamma = \left(\frac{\phi_h}{\phi_t}\right)^{D_h}\left(\frac{k}{n}\right)^{\frac{D_h}{D_t}}. \tag{10}$$

Corollary 4. *The cost of merging with pruning on partitions can be estimated by*

TABLE 2
Observations of ϕ, D of $C(\epsilon)$

	Base		Wiki	
	ϕ	D	ϕ	D
Tuple	0.12	-3.4	8	-1.5
Random	0.15	-2.5	0.3	-1.5
Feature-based	0.25	-2.5	3	-1.5

$$\left(\frac{m}{n} + \gamma\right) O(n). \tag{11}$$

Given a data set, the values of D_t and ϕ_t are then fixed according to their definitions. For example, as we observed in Fig. 9, we can find an ideal line $C_t(\epsilon) = (\phi_t \epsilon_t)^{D_t}$ having $D_t = -3.4$ and $\phi_t = 0.12$, which can approximately fit the observed Base data set. Recall that the slope D_h and the corresponding ϕ_h in (10) of γ can also be observed as constants on a large enough partition scheme. For example, in Fig. 10, we can observe $C_h(\epsilon) = (\phi_h \epsilon_h)^{D_h}$ having $D_h = -2.5$ and $\phi_t = 0.15$, which fits the Base data set with random partitions. In other words, γ will be a constant which is independent with respect to the processed number of partitions g and total number of partitions m. Therefore, according to Corollary 4, we cannot further improve the query efficiency by increasing the number of partitions m. Our experimental evaluation also verifies this conclusion.

4.4 Generating Partitions

Now, we discuss the generation of tuple partitions. The partition scheme is preferred which has lower query cost according to the above theoretical analysis.

4.4.1 *Random Partition*

The straightforward partition scheme is to assign a tuple to a partition at random. The distribution of tuples in different partitions tends to be the same. In other words, each partition shows a similar partition vector. Therefore, the prune power is low by conjecture.

Recall that the correlation dimension has $D_h < 0$. According to Lemma 7, the smaller the ϕ_h is, the larger the number of processed partitions g would be. In fact, as we observed in Table 2, the random partition scheme shows a small ϕ_h, e.g., $\phi_h \approx 0.3$ in the Wiki data set. Thus, theoretically, the query efficiency based on random partitions is low. Our experimental evaluation also verifies the unsuitability of the random partition scheme.

4.4.2 *Feature-Based Partition*

The feature-based partitioning is developed by the intuition that the tuples in the same partition share similar contents of items (features). There are various algorithms to partition tuples according to their similarities [31], which is not the focus of this paper. For example, we can employ a classifier to make decisions based on item features of partitions. Or we can use the clustering algorithms to group the tuples into m clusters without a supervised classifier.

Intuitively, in a feature-based partition scheme, the tuples in the same partitions share similar item features, while the tuples in different partitions often have various item features. Consequently, the partition vectors of different partitions are various as well. For a specific query, the bounds of partition will be more distinguishable. In other words, more irrelevant tuples in those partitions with low bounds can be possibly pruned.

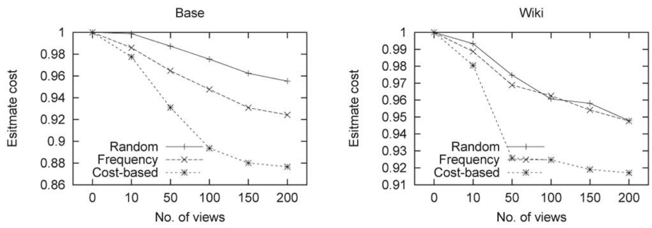


Fig. 6. Estimated cost of a query.

In fact, as we observe, the feature-based partition scheme has a large ϕ_h value, such as $\phi_h \approx 3$ in the Wiki data set in Table 2. Thus, given a specific k value, the feature-based partition has a smaller constant γ than the random approach (see details in Section 5.2). According to Corollary 4, the time cost of queries on feature-based partitions should be small as well. Therefore, in our experiments, the feature-based partition shows better query time performance.

5 Experiments

This section reports the experimental evaluation of proposed techniques. We evaluate the following approaches: the *baseline* approach with extended inverted lists [10], the planning with materialization of *views*, the merging with decomposition of *partitions*, and the *hybrid* approach with both views and partitions. In the implementation, we use the Combined Algorithm [11] as the state-of-art merge operator of inverted lists. Moreover, the successful idea of inverted block-index in IO-Top-K [12] is also applied, i.e., divide each inverted list into blocks and use score-descending order among blocks but keep the tuple entries within each block in the order of tuple IDs. Such block idea in CA is complementary to our proposed materialization and decomposition techniques. The main evaluation criterion is query time cost. We run the experiments in two real data sets, Google Base (Base), and Wikipedia (Wiki). There are 7,432,575 tuples crawled from Google Base web site, in size of 3.06 GB after preprocessing. The data of Wikipedia consists of 3,493,237 tuples, in size of 0.82 GB after preprocessing. Items of attribute-keywords are associated with *tf*idf* weight scores [13]. During the evaluation, we randomly select 200 tuples from the data set as a synthetic workload of queries.[6] The average response time of these queries are reported when different approaches are applied. The experiment runs on a machine with Intel Core 2 CPU (2.13 GHz) and 2 GB of memory.

5.1 Evaluating Materialization

In this experiment, we mainly test the performance of different planning approaches under various disk space limitations. Let $\bar{s}$ be the average size of lists. Then, the limitation of space M is actually the limitation of the number of views, say $\frac{M}{\bar{s}}$. Since the lists of single-item views (i.e.,

6. We explore the correspondences of attributes in dataspaces by using instance-level matching [9].

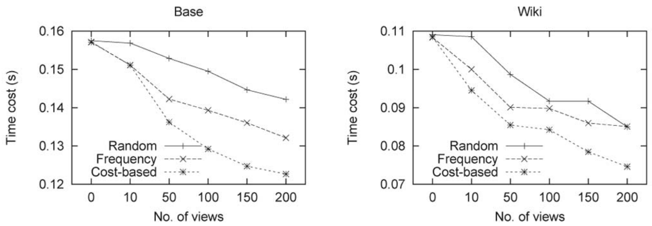


Fig. 7. Time cost of a query.

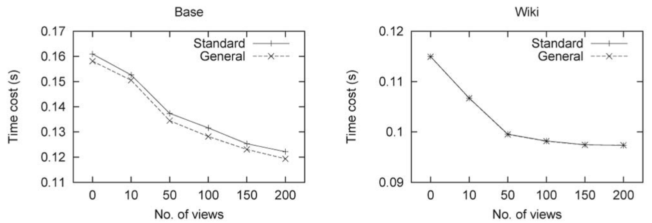


Fig. 8. Time cost of a query by different plans.

original items) are already stored by the index, we mainly check the extra space cost introduced by materializing the views with multiple items, i.e., $m = |\mathcal{V}| - |\mathcal{I}|$. In the following experiments, the number of views denotes m multiitem views by default. When the view number $m = 0$, it means that no multiitem views are materialized, i.e., equivalent to the baseline approach.

In Fig. 6, we present the estimated cost of query plan $\mathcal{P}$, i.e., $\sum_j c_j x_j$ in Definition 3.1, when different total numbers of views m are available. With the increase of materialized views m, the query plan $\mathcal{P}$ can choose more effective views with less estimated cost. Obviously, a randomly generated view has rare chance to be effective for a specific query. Thus, the query may have to retrieve the original items with higher cost, since the randomly generated views could be useless. The frequency-based view scheme materializes those views with frequent items according to the historical query log. Queries can reuse these views and consequently have query plans with less estimated cost. Finally, we also report the estimated cost of queries on cost-based view scheme, which is smaller than the other ones.

Fig. 7 reports the corresponding query time cost of various view schemes. As shown in figures, the time cost is roughly proportional to the estimated cost of corresponding query plans. The more materialized views m are, the better the query time performance is. According to the above observation of estimated cost, the frequency-based approach has more chance to utilize effective materialization than the random one. Therefore, the time cost of queries on frequency-based views is lower as well. However, the frequency-based approach favors small views as we mentioned in Section 3.4, while the cost-based scheme can generate useful views according to the cost estimation. Thus, queries on cost-based views have even lower time cost. Note that if there is no proper views available in large size, cost-based strategy will generate similar views as frequency one. Consequently, the difference between these two generation strategies may not be large, e.g., 10 views in Fig. 7. Nevertheless, the cost-based approach will not generate significantly worse views than frequency one.

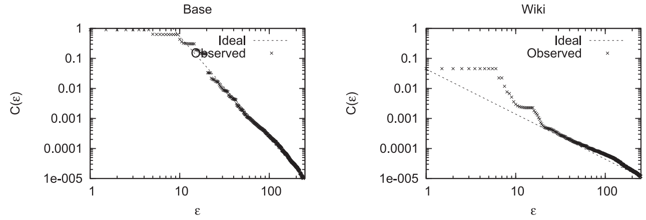


Fig. 9. $C(\epsilon)$ Observation of tuples.

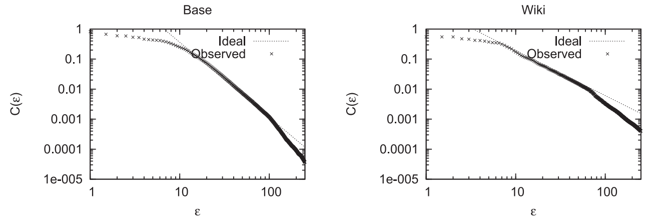


Fig. 10. $C(\epsilon)$ Observation of random partitions.

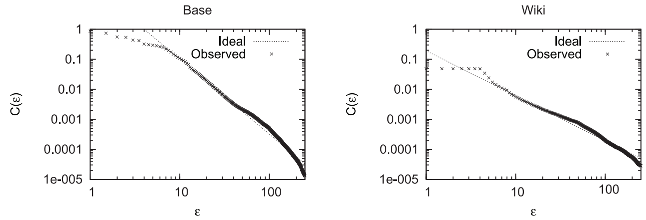


Fig. 11. $C(\epsilon)$ Observation of feature-based partitions.

According to our view selection strategies in query plan and view generation, we always first choose those most effective views that can contribute to the query at most. Thereby, with the increase of views, e.g., from 150 to 200 in Figs. 6 and 7, the achieved improvement may not as significant as first chosen ones like 1-50.

In Fig. 8, we evaluate the standard and general query planning. As we presented in Corollary 2, the worst case of an optimal general plan is the corresponding optimal standard plan without any negative merge operation. Therefore, as presented in Fig. 8, in Wiki data set, the performance of general plan is generally not worse than the standard one. When proper negative merge is applicable, e.g., in Base data set, the general plan can achieve better performance.

5.2 Evaluating Decomposition

In this experiment, we first observe that the constant D in (8) exists in real dataspace examples, since our cost estimation is based on the assumption of the existence of this D. Specifically, we collect the $C(\epsilon)$ values of tuples, random partitions, and feature-based partitions, which are reported in Figs. 9, 10, and 11, respectively. Each point $(\epsilon, C(\epsilon))$ denotes the observation of probability $C(\epsilon)$ with similarity score ϵ in the corresponding data set. We also plot an ideal $C(\epsilon) = (\phi\epsilon)^D$ in each data set that can fit our observations. We can record the constants ϕ and D of ideal $C(\epsilon) = (\phi\epsilon)^D$ as the estimation of real observations, which are presented in Table 2. For example, we have $D_h = -2.5$ and $\phi_h = 0.25$ for feature-based partitions in Base data set, according to the ideal $C(\epsilon) = (0.25\epsilon)^{-2.5}$ that fits the observed data in Fig. 11. According to the definition of γ in (10), for the random partitions in Base, we have

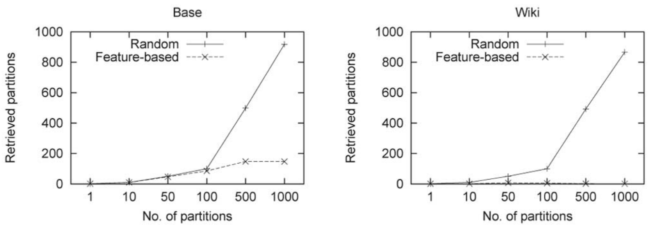


Fig. 12. Retrieved partitions of a query.

$$\gamma_{\text{random}} = \left(\frac{\phi_h}{\phi_t}\right)^{D_h} \left(\frac{k}{n}\right)^{\frac{D_h}{D_t}} = \left(\frac{0.15}{0.12}\right)^{-2.5} \left(\frac{k}{n}\right)^{\frac{-2.5}{-3.4}} = 0.572\left(\frac{k}{n}\right)^{\frac{-2.5}{-3.4}}.$$

For the feature-based scheme, we have

$$\gamma_{\text{feature}} = \left(\frac{0.25}{0.12}\right)^{-2.5} \left(\frac{k}{n}\right)^{\frac{-2.5}{-3.4}} = 0.159\left(\frac{k}{n}\right)^{\frac{-2.5}{-3.4}}.$$

Obviously, the constant γ_{feature} is less than γ_{random}. According to Corollary 4, the queries on feature-based partitions should have lower time cost than the random approach.

Similarly, we can also observe the constant γ in Wiki

$$\gamma_{\text{random}} = \left(\frac{0.3}{8}\right)^{-1.5} \left(\frac{k}{n}\right)^{\frac{-1.5}{-1.5}} = 137.706\frac{k}{n},$$

$$\gamma_{\text{feature}} = \left(\frac{3}{8}\right)^{-1.5} \left(\frac{k}{n}\right)^{\frac{-1.5}{-1.5}} = 4.354\frac{k}{n}.$$

That is, we have $\gamma_{\text{feature}} < \gamma_{\text{random}}$ as well. So far, according to the above observation and analysis on both data sets, queries on feature-based partition should have better performance than that of the random one. Next, we show that this case holds for the real query evaluation.

Specifically, we observe the time performance of top-k[7] queries under different number of partitions m. Note that when the partition number $m = 1$, it means that all the tuples are in one partition, which is equivalent to the baseline approach.

In Fig. 12, we study the number of retrieved partitions g that have to be processed before the pruning can be applied. First, compared with feature-based partition scheme, the random approach has much more retrieved partitions g, which also confirms our analysis of the random scheme in Section 4.4. Moreover, as we analyzed, the constant γ approximately denotes the rate of processed partitions $\frac{g}{m}$. In the above observation, we find that the γ_{feature} is much less than γ_{random} in Wiki ($\frac{4.354}{137.706}$), compared with those of Base ($\frac{0.159}{0.572}$). Thus, in Fig. 12, the pruning power of featured-based partitions on Wiki is stronger than that in Base.

Fig. 13 shows the time cost of queries with corresponding partitions. When the number of partitions m is small, e.g., 10 partitions in Fig. 13, the overlap of items (features) among partitions might be large. That is, in terms of our cost analysis, we cannot accurately observe the constant γ about D and ϕ. Thus, the bounds of partitions are not distinguishable for a specific query. Consequently, the pruning is not ensured, and the cost will be high. With the increase of m, e.g., from 10 to 100 partitions in Fig. 13, the constant γ can be observed. In other words, the bounds of partitions can effectively identify and prune those low score tuples, thereby the cost drops. Moreover, with the further increase of partition numbers m, e.g., 1,000 partitions in Fig. 13, the performance cannot be improved any more, since the fractal property has already been clearly observed with a constant γ. Consequently, the time cost increases with the number of partitions, which confirms our conclusion in Corollary 4.

7. $k = 5$; similar results are observed with $k = 1, 10$, etc.

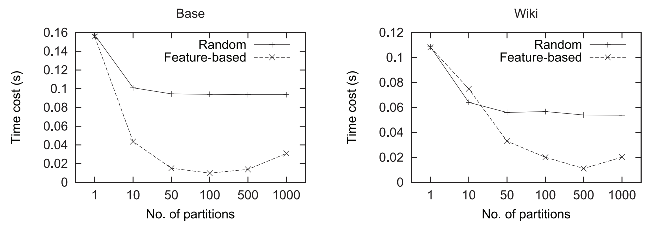


Fig. 13. Time cost of a query.

5.3 Scalability

Finally, we combine our proposed views and partitions together, called hybrid approach with materialization+decomposition. For the view materialization, we use the cost-based view generation and the standard query plan. The partition decomposition uses the feature-based partition generation. The state-of-art CA method is utilized as baseline approach where materialization and decomposition are not applied. We mainly test the performance of approaches under different data sizes, in order to evaluate scalability.

As presented in Fig. 14, the time cost of all the approaches increases linearly as the data size. Both our materialization and decomposition approaches can improve the time performance in various data scale, compared with the baseline approach. The hybrid one can always achieve the best performance and scales well under large sizes.

6 Related Work

The concept of dataspaces is proposed in [1] and [2], which provides a coexisting system of heterogeneous data. Due to the huge amount of increasing data especially from the Web, the importance of dataspace systems has already been recognized and emphasized [3], [4]. Recent work [5], [6], [7] 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, which also plays a fundamental role in practical dataspace systems.

The problem of indexing dataspaces is first studied by Dong and Halevy [10] to support efficient query. Based on the encoding of attribute label and value as items, the

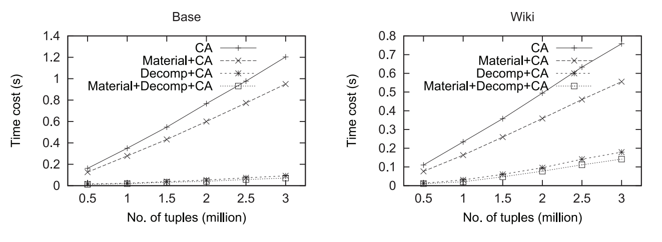


Fig. 14. Scalability.

inverted index is extended to dataspaces, which is also considered as the baseline approach in our work. In fact, inverted index [16], [17] has been studied for decades as efficient access to sparse data. Zobel and Moffat [18] provide a comprehensive introduction of key techniques in the text indexing literature. Li et al. [32] also study the indexing for approximately retrieving the sparse data, which extends inverted lists as well. The difference between dataspaces and XML is also discussed by Dong and Halevy [10]. Specifically, since XML techniques rely on encoding the parent-child and ancestor-descendant relationships in an XML tree, which do not fit in dataspaces, the query processing of related items in XML [33] is not directly applicable to the query processing with keyword neighborhood in dataspaces [10]. Once the data are indexed, our approaches introduce materialization in dataspaces and provide (near) optimal query planning based on the materialized data. For high dimensional data, Sarawagi and Kirpal [34] propose the grouping of items to materialize their corresponding inverted lists. However, the proposed algorithm is developed for set similarity joins, and does not address the generation of optimal query plans based on the available materialized data.

Various strategies for storing sparse data are also proposed. Chu et al. [35] use a big wide-table to store the sparse data and extract the data incrementally [36]. Rather than the predominant positional storage with a preallocated amount of space for each attribute, the wide-table uses an interpreted storage to avoid allocating spaces to those null values in the sparse data. Agrawal et al. [37] 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. [38] 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. [39], [40] 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 [39] and the recent Star Schema Benchmark of data warehousing [40]. Chaudhuri et al. [41] study a similarity join operator (SSJoin [41], [42]) 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 [43]), which are stored separately in different tuples, respectively. Our work is independent with the storage of dataspaces, instead we develop the materialization and decomposition of dataspaces upon the indexing framework.

For the merge operator, the inverted lists are usually sorted by the tuple IDs, then efficient merging algorithm can be applied [18]. When the inverted lists are sorted by the weight of each tuple, the threshold algorithm [19] can return the top-k answers efficiently. Advanced TA family methods such as combined algorithm [11] can also be used as merge operator of inverted lists. Moreover, inverted block-index is also proposed in IO-Top-K [12], i.e., divide each inverted list into blocks and use score-descending order among blocks but keep the tuple entries with in each block in tuple ID order. Arjen P. de Vries et al. [44] also study the k-NN search by avoiding merging all the dimensions referred by query items. In our study, we first decompose the tuples into a set of partitions, then the above merging techniques can be applied in each partition, respectively. Thus, our focus is the pruning of partitions rather than the merge operator.

Materialized views in relational databases are often utilized to find equivalent view-based rewritings of relational queries [45], such as conjunctive queries or aggregate queries in databases. Similar problem is also studied in index selection [46]. Specifically, given a workload of SQL statements and a user-specified storage constraint, it is to recommend a set of indexes that have the maximum benefit for the given workload. Greedy heuristics are often used in such selection, e.g., in SQL Server [47] and DB2 [48]. Chirkova and Li [49] study the generation of a set of views that can compute the answers to the queries, such that the size of the view set is minimal. Heeren et al. [50] consider the index selection with a bound of available space, upon which the average query response time is minimized. Instead of considering materialized views and index separately, Aouiche and Darmont [51] take view-index interactions into account and achieve efficient storage space sharing. All these previous works focus on rewriting relational queries based on materialized views in relational databases. In our study, we extend the concept of materialization to dataspaces and explore the corresponding query optimization.

Partitioning-based approaches for efficient access are studied as well. Lester et al. [52] propose the partitioning index for efficient online index. To make documents immediately accessible, the index is divided into a controlled number of partitions. Nikos Mamoulis [53] 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 [34] 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. The idea of cracking databases into manageable pieces is developed recently to organize data in the way users request it. Rather than dataspaces, Idreos et al. [54], [55] mainly study the cracking of relational databases. The cracking approach is based on the hypothesis that index maintenance should be a byproduct of query processing, not of updates.

7 Conclusions

In this paper, we study the materialization and decomposition of dataspaces in order to improve the efficiency of queries with keyword neighborhood in schema level. Since neighbor keywords are always queried together, we first propose the materialization of neighbor keywords as views of items. Then, the optimal query planning is studied on the item views that are materialized in dataspaces. Due to the NP-completeness of the problem, we study the greedy approaches to generate query plans. Obviously, the more materialized views there are, the better the query performance is. The generation of views is then discussed with limitation of materialization space. Moreover, we also study the decomposition of dataspaces into partitions of tuples for top-k queries. Efficient pruning of tuple partitions is developed during the top-k query processing. We propose a theoretical analysis for the cost of querying with partitions and find that the pruning power cannot be improved by increasing the number of partitions. The generation of partitions is also discussed based on the cost analysis.

Finally, we report an extensive experiment to illustrate the performance of proposed methods. In the method of materialization, the general query plans show no worse performance than the standard query plans. When proper negative merge is applicable, the general plan can achieve better performance. The hybrid approach with both views and partitions can always achieve the best performance. Furthermore, the experimental results also verify our conclusions of cost analysis, that is, we can improve the query performance by increasing the number of views but not that of partitions.

Acknowledgments

Funding for this work was provided by Hong Kong RGC GRF 611608, NSFC Grant Nos. 60736013, 60970112, and 60803105, and Microsoft Research Asia Gift Grant, MRA11EG05. The authors also thank the support from HKUST RFID center.

References

[1] M.J. Franklin, A.Y. Halevy, and D. Maier, "From Databases to Dataspaces: A New Abstraction for Information Management," *SIGMOD Record,* vol. 34, no. 4, pp. 27-33, 2005.

[2] A.Y. Halevy, M.J. Franklin, and D. Maier, "Principles of Dataspace Systems," *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '06),* pp. 1-9, 2006.

[3] M.J. Franklin, A.Y. Halevy, and D. Maier, "A First Tutorial on Dataspaces," *Proc. VLDB Endowment,* vol. 1, no. 2, pp. 1516-1517, 2008.

[4] J. Madhavan, S. Cohen, X.L. Dong, A.Y. Halevy, S.R. Jeffery, D. Ko, and C. Yu, "Web-Scale Data Integration: You can Afford to Pay as You Go," *Proc. Conf. Innovative Data Systems Research (CIDR),* pp. 342-350, 2007.

[5] S.R. Jeffery, M.J. Franklin, and A.Y. Halevy, "Pay-As-You-Go User Feedback for Dataspace Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* pp. 847-860, 2008.

[6] A.D. Sarma, X. Dong, and A.Y. Halevy, "Bootstrapping Pay-As-You-Go Data Integration Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08),* pp. 861-874, 2008.

[7] M.A.V. Salles, J.-P. Dittrich, S.K. Karakashian, O.R. Girard, and L. Blunschi, "Itrails: Pay-As-You-Go Information Integration in Dataspaces," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 663-674, 2007.

[8] F.M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A Core of Semantic Knowledge," *Proc. 16th Int'l Conf. World Wide Web (WWW '07),* pp. 697-706, 2007.

[9] E. Rahm and P.A. Bernstein, "A Survey of Approaches to Automatic Schema Matching," *Int'l J. Very Large Data Bases,* vol. 10, no. 4, pp. 334-350, 2001.

[10] X. Dong and A.Y. Halevy, "Indexing Dataspaces," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 43-54, 2007.

[11] R. Fagin, "Combining Fuzzy Information: An Overview," *SIGMOD Record,* vol. 31, no. 2, pp. 109-118, 2002.

[12] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "Io-Top-k: Index-Access Optimized Top-k Query Processing," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06),* pp. 475-486, 2006.

[13] G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, 1989.

[14] F. Liu, C.T. Yu, W. Meng, and A. Chowdhury, "Effective Keyword Search in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '06),* pp. 563-574, 2006.

[15] H. Bast and I. Weber, "The Completesearch Engine: Interactive, Efficient, and Towards IR& DB Integration," *Proc. Conf. Innovative Data Systems Research (CIDR),* pp. 88-95, 2007.

[16] R.A. Baeza-Yates and B.A. Ribeiro-Neto, *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[17] I.H. Witten, A. Moffat, and T.C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images,* second ed. Morgan Kaufmann, 1999.

[18] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," *ACM Computing Surveys,* vol. 38, no. 2, pp. 1-55, 2006.

[19] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. 20th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '01),* 2001.

[20] D. Peleg, G. Schechtman, and A. Wool, "Approximating Bounded 0-1 Integer Linear Programs," *Proc. Second Israel Symp. Theory and Computing Systems,* pp. 69-77, 1993.

[21] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., 1982.

[22] V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," *Math. Operations Research,* vol. 4, no. 3, pp. 233-235, 1979.

[23] G. Dobson, "Worst Case Analysis of Greedy Heuristics for Integer Programming with Non-Negative Data," *Math. Operations Research,* vol. 7, no. 4, pp. 515-531, 1982.

[24] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB '94),* pp. 487-499, 1994.

[25] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery,* vol. 8, no. 1, pp. 53-87, 2004.

[26] L. Lim, M. Wang, S. Padmanabhan, J.S. Vitter, and R.C. Agarwal, "Efficient Update of Indexes for Dynamically Changing Web Documents," *J. World Wide Web,* vol. 10, no. 1, pp. 37-69, 2007.

[27] P. Grassberger and I. Procaccia, "Measuring the Strangeness of Strange Attractors," *Physica D: Nonlinear Phenomena,* vol. 9, nos. 1/2, pp. 189-208, 1983.

[28] A. Belussi and C. Faloutsos, "Estimating the Selectivity of Spatial Queries Using the "Correlation" Fractal Dimension," *Proc. 21th Int'l Conf. Very Large Data Bases (VLDB '95),* pp. 299-310, 1995.

[29] B.-U. Pagel, F. Korn, and C. Faloutsos, "Deflating the Dimensionality Curse Using Multiple Fractal Dimensions," *Proc. 16th Int'l Conf. Data Eng.,* pp. 589-598, 2000.

[30] F. Korn, B.-U. Pagel, and C. Faloutsos, "On the "Dimensionality Curse" and the "Self-Similarity Blessing,"" *IEEE Trans. Knowledge and Data Eng.,* vol. 13, no. 1, pp. 96-111, Jan./Feb. 2001.

[31] J. Han and M. Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufmann, 2000.

[32] B. Li, M. Hui, J. Li, and H. Gao, "Iva-File: Efficiently Indexing Sparse Wide Tables in Community Systems," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '09),* pp. 210-221, 2009.

[33] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "Xsearch: A Semantic Search Engine for Xml," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* pp. 45-56, 2003.

[34] S. Sarawagi and A. Kirpal, "Efficient Set Joins on Similarity Predicates," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* pp. 743-754, 2004.

[35] E. Chu, J.L. Beckmann, and J.F. Naughton, "The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 821-832, 2007.

[36] E. Chu, A. Baid, T. Chen, A. Doan, and J.F. Naughton, "A Relational Approach to Incrementally Extracting and Querying Structure in Unstructured Data," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 1045-1056, 2007.

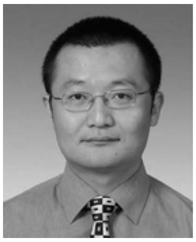
[37] R. Agrawal, A. Somani, and Y. Xu, "Storage and Querying of e-Commerce Data," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB '01),* pp. 149-158, 2001.

[38] J.L. Beckmann, A. Halverson, R. Krishnamurthy, and J.F. Naughton, "Extending RDBMSs to Support Sparse Datasets Using an Interpreted Attribute Storage Format," *Proc. 22nd Int'l Conf. Data Eng. (ICDE '06),* p. 58, 2006.

[39] D.J. Abadi, A. Marcus, S. Madden, and K.J. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 411-422, 2007.

[40] D. Abadi, S. Madden, and N. Hachem, "Column-Stores Vs. Row-Stores: How Different are They Really?" *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '08)*, 2008.
[41] S. Chaudhuri, V. Ganti, and R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning," *Proc. 22nd Int'l Conf. Data Eng. (ICDE '06)*, p. 5, 2006.
[42] A. Arasu, V. Ganti, and R. Kaushik, "Efficient Exact Set-Similarity Joins," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06)*, pp. 918-929, 2006.
[43] E. Ukkonen, "Approximate String Matching with q-Grams and Maximal Matches," *Theoretical Computer Science—Selected Papers of the Combinatorial Pattern Matching School*, vol. 92, no. 1, pp. 191-211, 1992.
[44] A.P. de Vries, N. Mamoulis, N. Nes, and M.L. Kersten, "Efficient k-NN Search on Vertically Decomposed Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02)*, pp. 322-333, 2002.
[45] G. Gou, M. Kormilitsin, and R. Chirkova, "Query Evaluation Using Overlapping Views: Completeness and Efficiency," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pp. 37-48, 2006.
[46] S. Chaudhuri, M. Datar, and V.R. Narasayya, "Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution," *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no. 11, pp. 1313-1323, Nov. 2004.
[47] S. Agrawal, S. Chaudhuri, and V.R. Narasayya, "Automated Selection of Materialized Views and Indexes in Sql Databases," *Proc. 26th Int'l Conf. Very Large Data Bases (VLDB '00)*, pp. 496-505, 2000.
[48] G. Valentin, M. Zuliani, D.C. Zilio, G.M. Lohman, and A. Skelley, "DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes," *Proc. 16th Int'l Conf. Data Eng.*, pp. 101-110, 2000.
[49] R. Chirkova and C. Li, "Materializing Views with Minimal Size to Answer Queries," *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '03)*, pp. 38-48, 2003.
[50] C. Heeren, H.V. Jagadish, and L. Pitt, "Optimal Indexing Using Near-Minimal Space," *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '03)*, pp. 244-251, 2003.
[51] K. Aouiche and J. Darmont, "Data Mining-Based Materialized View and Index Selection in Data Warehouses," *J. Intelligent Information Systems*, vol. 33, no. 1, pp. 65-93, 2009.
[52] N. Lester, A. Moffat, and J. Zobel, "Fast On-Line Index Construction by Geometric Partitioning," *Proc. 14th ACM Int'l Conf. Information and Knowledge Management (CIKM '05)*, pp. 776-783, 2005.
[53] N. Mamoulis, "Efficient Processing of Joins on Set-Valued Attributes," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03)*, pp. 157-168, 2003.
[54] S. Idreos, M.L. Kersten, and S. Manegold, "Database Cracking," *Proc. Conf. Innovative Data Systems Research (CIDR)*, pp. 68-78, 2007.
[55] S. Idreos, M.L. Kersten, and S. Manegold, "Updating a Cracked Database," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07)*, pp. 413-424, 2007.

Shaoxu Song is currently working toward the PhD degree in the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong. His research interests include data quality and data dependency. He is a student member of the IEEE.



Lei Chen received the BS degree in computer science and engineering from Tianjin University, China, in 1994, the MA degree from Asian Institute of Technology, Thailand, in 1997, and the PhD degree in computer science from University of Waterloo, Canada, in 2005. He is now an associate professor in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology. His research interests include uncertain databases, graph databases, multimedia and time series databases, and sensor and peer-to-peer databases. He is a member of the IEEE.

Mingxuan Yuan received the BSc degree in 2002 and MSc degree in 2006, both in computer science and engineering, from Xi'an Jiaotong University, China. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology, China. His research interests include privacy problems of social networks.