Received October 26, 2019, accepted December 12, 2019, date of publication December 19, 2019,
date of current version December 31, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2960856

Indexing and Search of Order-Preserving Submatrix for Gene Expression Data

TAO JIANG[1], BOLIN CHEN[2], JUNTAO LI[3], AND GUOYU XU[1]
[1]School of Computer and Information Engineering, Henan University of Economics and Law, Zhengzhou 450046, China
[2]School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China
[3]School of Mathematics and Information Science, Henan Normal University, Xinxiang 453007, China

Corresponding author: Tao Jiang (jiangtao@mail.nwpu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61702161, Grant 61602386, Grant 61972320, Grant 91746115, and Grant 61602153, in part by the Key Research and Development and Promotion Program of He'nan Province of China under Grant 182102210213, Grant 182102210020, and Grant 172102210171, in part by the Key Research Fund for Higher Education of He'nan Province of China under Grant 18A520003, Grant 18A520015, Grant 19A413005, and Grant 18B510004, and in part by the Natural Science Foundation of Shaanxi Province of China under Grant 2017JQ6008.

ABSTRACT Bicluster pattern discovery plays a key role in analysis of gene expression data. One vital model of bicluster mining is Order-Preserving SubMatrix (OPSM), which finds similar tendency of some genes on some conditions. Most of the OPSM discovery methods are batch mining techniques and not suitable for low latency data query. To make data analysis efficient and effective, in this paper, we first propose a prefix-tree based indexing method *pfTree*, then give an optimization technique *pIndex* that employs row and column header tables to search the positive, negative and time-delayed OPSMs. Meanwhile, we present an online sharing query technique to accelerate the frequent searches. Finally, we conduct extensive experiments and compare our methods with the existing approaches. Experimental results demonstrate the efficiency and effectiveness of the proposed methods.

INDEX TERMS Gene expression data, online sharing queries, OPSM, pfTree, pIndex.

I. INTRODUCTION

Gene microarray technology gives the chances for monitoring of the expression level of huge genes on many experiments simultaneously. We always view the gene expression data, which probed on gene microarrays, as an $n \times m$ matrix with n rows (genes) and m columns (conditions), in which every entry indicates the expression level of a specific gene on a specific condition [1]–[7]. *Table* 1 shows an example matrix for gene expression dataset, which contains five genes and six experimental conditions. Existing clustering methods present not so well performance on analysing gene expression data, because most genes are closely co-expression only on some but not all conditions, and are not necessarily expression at the same or similar level. Hence, on this situation, biclustering grows and becomes an useful model to mine important clusters [8], [9]. Lately, one vital model in biclustering, i.e., *Order-Preserving SubMatrix* (OPSM) [10], has been accepted as a biologically meaningful cluster tool. Essentially, an OPSM is a submatrix, in which all the rows shows the same rising up and dropping down trend on the columns, for example, in *Table* 2, rows g_1 and g_2 show a growing expression level on columns c_1, c_2, c_3, c_4, and c_5. Meanwhile, OPSM clustering model concentrates on the relative order of conditions but not the absolute values. And OPSM consists of several types, such as positive, negative, time-delayed OPSMs. *Table* 2 presents some types of OPSMs mining from *Table* 1. And *Fig.* 1 illustrates the expression level (y-axis) of five genes under six experimental conditions (x-axis) in four graphs. These genes belong to different functional categories. As shown in the figures, the genes exhibit positive,

TABLE 1. (1) Raw gene expression data matrix. For clarity, we omit some values in certain cells. (2) Permutation of raw data matrix in Table 1. In essence, it first sorts the expression values of each gene increasingly. Then, it replaces each expression value with column label.

	c_1	c_2	c_3	c_4	c_5	c_6	Permutaion of columns
g_1	-3	-1	0.5	2	5		c_1, c_2, c_3, c_4, c_5
g_2	-4	-2	0	1.5	4		c_1, c_2, c_3, c_4, c_5
g_3	11	9.5	9	7.5	7		c_5, c_4, c_3, c_2, c_1
g_4		0	1	3	6	7	c_2, c_3, c_4, c_5, c_6
g_5		13	11	9.5	9	8	c_6, c_5, c_4, c_3, c_2

The associate editor coordinating the review of this manuscript and approving it for publication was Liangxiu Han.

TABLE 2. Order-Preserving SubMatrix (OPSM) mining from the data in Table 1. It only gives some biggest OPSMs in each type of OPSMs.

OPSM Types	Row No.	Column No.
Positive OPSMs	g_1, g_2	c_1, c_2, c_3, c_4, c_5
	g_1, g_2, g_4	c_2, c_3, c_4, c_5
Negative OPSMs	g_1, g_2	c_1, c_2, c_3, c_4, c_5
	$g3$	c_5, c_4, c_3, c_2, c_1
	g_1, g_2	c_2, c_3, c_4, c_5
	g_5	c_5, c_4, c_3, c_2
Time-Delayed Positive OPSMs	g_1, g_2	c_1, c_2, c_3, c_4, c_5
	$g4$	c_2, c_3, c_4, c_5, c_6
Time-Delayed Negative OPSMs	g_1, g_2	c_1, c_2, c_3, c_4, c_5
	$g5$	c_6, c_5, c_4, c_3, c_2

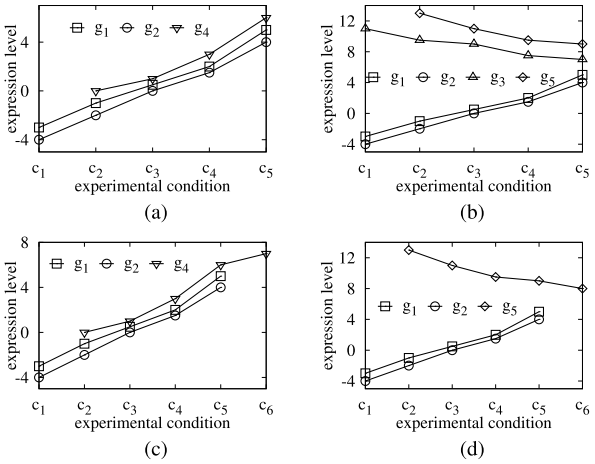


FIGURE 1. Diagram of OPSMs shown in Table 2. (a) Positive OPSMs, (b) Negative OPSMs, (c) Time-Delayed Positive OPSMs, (d) Time-Delayed Negative OPSMs.

negative, time-delayed expression patterns, even though the absolute expression levels under the same conditions are different. With the decreasing of gene expression analysis cost, it accumulates a lot of gene expression datasets and OPSM mining results, but these datasets are not efficiently employed by researchers, who have the experiences and can use specific row or column keywords to search supporting columns or rows. Thus, the problem *OPSM search or query is to retrieve some types of OPSMs based on column or row keywords from a given data matrix*, which plays a key part in inferring gene coregulated networks, designing new types of drugs, preventing diseases, and so on.

Most of the previous studies address the problem of OPSM mining [11]–[14], few work is studied for OPSM query. OPSM problem is first proposed by Ben-Dor [10]. Then, the researchers give some OPSM mining methods based on quantitative measures or qualitative measures [7]. And these methods find many types of OPSMs [15]–[18], such as OPSMs with constant values, or coherent values. Meanwhile, the scientists devise some OPSM mining tools, such as GPX [19] and BicAT [20]. However, these tools have a common feature that it uses an indirect way to give queried results, and the indirect way is not efficient. Therefore, we want to present a direct-way query tool. The most similar work with OPSM query is presented in work [19], Jiang et al give an interactive method, which can drill down and roll up, to facilitate OPSM search.

OPSM query is similar to string matching problem [21], [22], the similarity between the above two problems is to find if there is a pattern string in a given string. *KMP* [21] and *BM* [22] are two classic solutions in string problems. They work for string matching without gaps, but do not work well for OPSM query, which allows having gaps in a given string. Therefore, none of these methods can be used directly to solve the problem. However, OPSM clustering is based on the increasing or decreasing order of columns, it make the longest common subsequence method (*LCS*) [23] be the appropriate model to find the specific OPSMs. Clearly, it is necessary to build indexes to help OPSM queries. Prefix tree and suffix tree are two common basic models, because the former allows two strings to share the prefix, which saves the spaces, we choose it as the basic model.

It is a challenging work to devise a OPSM search tool on a direct way. The reasons are as following. First, there are huge amount of datasets. As the dropping of the expense on analysing gene expression, gene expression data are increasing at an unprecedented rate. Further, the analysis of gene expression data produces a large number of OPSM data. Second, how to design a general tool for two kinds of data. It is generally known that, on the one hand, the mining time of OPSM pattern on gene expression data is large, but the search time of OPSM pattern from OPSM data is small, on the other hand, the amount of gene expression data is small, and that of OPSM dataset is large. Last but not the least, memory capacity constraints make the index must be small enough, the growing of datasets makes index update should be efficient, and the quick response requirements of users make the queries on the index must be fast and scalable.

To solve these problems mentioned above, firstly we propose a naive method called *pfTree*, which indexes the datasets with prefix tree. Although the preliminary index reduces a lot of redundant data, its query efficiency is not high. In order to improve query efficiency, two header tables are added to the *pfTree* and named it as *pIndex*. Both of these structures can index two kinds of data, i.e., gene expression data and OPSM data, and OPSMs can be queried directly on them, thus it eliminates the process of mining OPSM from gene expression data. In this way, the advantages of both kinds of data are utilized to improve the query performance. Next, *pIndex* uses the row and column header tables to update the index and query OPSMs. To further improve query performance, two pruning methods are proposed to reduce the traversal of useless branches. To reduce the excessive candidate set of column keywords generated in the process of column fuzzy query, a first element rotation method *FIT* is proposed, which reduces the number of column keywords from $m!$ to m. Among the queries, we observe that some queries have been frequently processed before and are time-consuming, e.g., the time consumption of our test results in *Section VI*,

especially when executing fuzzy queries, take more than one second, thus an online sharing query technique is necessary to proposed to reduce the cost of frequent and time-consuming searches.

It applies two indexes *pfTree* and *pIndex* on two kinds of datasets, i.e., gene expression and OPSM datasets. At the same time, a lot of experiments have been done. The experimental results show that both indexes have good compression performance, and *pIndex* is more efficient in index update and query. Furthermore, the two methods are implemented on three platforms: single machine, Hadoop and Hama [24]. At the same time, they are also effective and scalable in index creation, query based on different keywords or nodes. The main contributions of this paper are as follows:

- It presents a basic method based on prefix tree, *pfTree*, and an optimization method named *pIndex* with two header tables of row and column.
- Index updating (insertion and deletion), multiple types of OPSM query methods, and online sharing query techniques are proposed. At the same time, the method named *FIT* reduces the number of candidate keyword sets, and several pruning methods for improving query performance are given.
- The validity and scalability of the proposed method are verified on three platforms: single machine, Hadoop and Hama. It is proved that *pIndex* with/without online sharing query method outperforms *pfTree* in processing cost and query accuracy.

The rest of paper is organized as follows: Section II gives preliminary concepts and presents the basic framework for Order-Preserving SubMatrix indexing and search. Section III presents a naive indexing method *pfTree*. Section IV illustrates the optimization indexing method *pIndex*, which contains building and updating method and how to construct header tables. Section V offers the exact and fuzzy queries using header table based search paradigm, proposes *FIT* strategy and pruning methods, describes the multi-types of OPSM query approaches, and shows the online sharing query strategy. We report empirical studies, and review related work in Section VI and VII, respectively. Section VIII concludes this study.

II. PRELIMINARIES

In this section, to keep the paper self contained, we introduce the preliminary concepts, such as positive, negative, and time-delayed OPSMs, and present some kinds of OPSM query methods, finally outline an indexing framework to address *Order-Preserving SubMatrix* query problems.

Throughout the paper, we use the following notations listed in *Table* 3. If there are no specific notifications, we use row and gene, column and experimental conditions interchangeably, due to they have the same meaning in this study.

Definition 1 (Order-Preserving SubMatrix (OPSM)): Given a dataset D, i.e., a data matrix M, $M_i(g, c)$ is a submatrix of D, and $g \subseteq G$, $c \subseteq C$. If M_i is an order-preserving submatrix, then for each row in g, there exists a permutation that preserves the order $e_{i1} \prec e_{i2} \prec \ldots \prec e_{ij} \prec \ldots \prec e_{ik}$ or $e_{i1} \succ e_{i2} \succ \ldots \succ e_{ij} \succ \ldots \succ e_{ik}$, where $(i1, \ldots, ij, \ldots, ik)$ is the permutation of the indexes of columns c, i.e., the data values e are monotonically increasing or decreasing with respect to the permutation of the indexes of columns c.

TABLE 3. Notations used in the paper.

Notation	Description
G	a set of genes
g	a subset of G
g_i	a gene
C	a set of conditions
c	a subset of C
c_i	a condition
$D(G, C)$ or D	a dataset
d	the delayed time
e	the values of one row in D
e_{ij}	an entry of D
δ	the size threshold of g
τ	the length threshold of c
$M(G, C)$ or M	a matrix
$M_i(g, c)$ or M_i	an OPSM

Definition 2 (Positive Order-Preserving SubMatrix (POPSM)): Given an OPSM $M_i(g, c)$, if M_i is a positive order-preserving submatrix, then for each row in g, there exists a permutation that preserves the order $e_{i1} \prec e_{i2} \prec \ldots \prec e_{ij} \prec \ldots \prec e_{ik}$ or $e_{i1} \succ e_{i2} \succ \ldots \succ e_{ij} \succ \ldots \succ e_{ik}$.

Example 1: In Fig. 1(a), genes g_1, g_2, g_4 have an increasing order under conditions c_2, c_3, c_4, c_5, thus we say $\{\{g_1, g_2, g_4\}, \{c_2, c_3, c_4, c_5\}\}$ is a Positive OPSM.

Definition 3: (Negative Order-Preserving SubMatrix (NOPSM)). Given an OPSM $M_i(g, c)$, if M_i is a negative order-preserving submatrix, then for each row in g_{up} ($g_{up} \subseteq g$), there exists a permutation that preserves the order $e_{i1} \prec e_{i2} \prec \ldots \prec e_{ij} \prec \ldots \prec e_{ik}$, and for each row in g_{down} ($g_{down} \subseteq g$), there exists a permutation that preserves the order $e_{i1} \succ e_{i2} \succ \ldots \succ e_{ij} \succ \ldots \succ e_{ik}$, where $g_{up} \cup g_{down} = g$.

Example 2: In Fig. 1(b), genes g_1, g_2 have an increasing order under conditions c_1, c_2, c_3, c_4, c_5, but gene g_3 has a decreasing order under conditions c_1, c_2, c_3, c_4, c_5, thus we say $\{\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}, \{\{g_3\}, \{c_5, c_4, c_3, c_2, c_1\}\}\}$ is a Negative OPSM (NOPSM).

Definition 4 (Time-Delayed Order-Preserving SubMatrix (DOPSM)): Given an OPSM $M_i(g, c)$, if M_i is a time-delayed order-preserving submatrix, then for each row in $g_{forward}$ ($g_{forward} \subseteq g$), there exists a permutation that preserves the order $e_{i1} \prec e_{i2} \prec \ldots \prec e_{ij} \prec \ldots \prec e_{ik}$, i.e., the data values e are monotonically increasing with respect to the permutation of the indexes of columns $c_{forward}(c_{i1}, \ldots, c_{ij}, \ldots, c_{ik})$ ($c_{forward} \subseteq c$), and for each row in g_{delay} ($g_{dealy} \subseteq g$), there exists a permutation that preserves the order $e_{i'1} \prec e_{i'2} \prec \ldots \prec e_{i'j} \prec \ldots \prec e_{i'k}$, i.e., the data values e are monotonically increasing with respect to the permutation of the indexes of columns $c_{delay}(c_{i'1}, \ldots, c_{i'j}, \ldots, c_{i'k})$ ($c_{delay} \subseteq c$),

where $c_{i1} - c_{i'1} = \ldots = c_{ij} - c_{i'j} = \ldots = c_{ik} - c_{i'k} = d$, *d is the delayed time. When the data values e are monotonically decreasing with respect to the permutation of the indexes of columns, the definition is also held. And the former one is Time-Delayed Positive OPSM (DPOPSM), the latter one is Time-Delayed Negative OPSM (DNOPSM).*

Example 3: In Fig. 1(c), genes g_1, g_2 *have an increasing order under conditions* c_1, c_2, c_3, c_4, c_5, *and gene* g_4 *has an increasing order under conditions* c_2, c_3, c_4, c_5, c_6, *but the latter one has a time point delay, thus we say* $\{\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}, \{\{g_4\}, \{c_2, c_3, c_4, c_5, c_6\}\}\}$ *is a Time-Delayed Positive OPSM (DPOPSM). Similarly, in Fig. 1(d),* $\{\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}, \{\{g_3\}, \{c_6, c_5, c_4, c_3, c_2\}\}\}$ *is a Time-Delayed Negative OPSM (DNOPSM).*

Definition 5 (Exact Query on Genes (EQ_g*)): Given a dataset D and a subset of genes* $g = (g_i, \ldots, g_j, \ldots, g_k)$, *exact query on g returns a subsets of conditions* $c = (c_x, \ldots, c_y, \ldots, c_z)$ *above the length threshold* τ *in OPMSs that contain all the items in g.*

Definition 6 (Exact Query on Conditions (EQ_c*)): Given a dataset D and a subset of conditions* $c = (c_x, \ldots, c_y, \ldots, c_z)$, *exact query on c returns a subsets of genes* $g = (g_i, \ldots, g_j, \ldots, g_k)$ *above the size threshold* δ *in OPMSs that contain all the items in c and keep the order of c.*

Definition 7 (Fuzzy Query on Genes (FQ_g*)): Given a dataset D and a subset of genes* $g = (g_i, \ldots, g_j, \ldots, g_k)$, *fuzzy query on g returns a subsets of conditions* $c = (c_x, \ldots, c_y, \ldots, c_z)$ *above the length threshold* τ *in OPMSs that contain a subset of the items in g above the size threshold* δ.

Definition 8 (Fuzzy Query on Conditions (FQ_c*)): Given a dataset D and a subset of conditions* $c = (c_x, \ldots, c_y, \ldots, c_z)$, *fuzzy query on c returns a subsets of genes* $g = (g_i, \ldots, g_j, \ldots, g_k)$ *above the size threshold* δ *in OPSMs that contain a subset of c above the length threshold* τ *and need not keep the order of c.*

Definition 9 (Query Similarity (S)): Given a query keyword list l and a query result r, the similarity between l and r, denoted as $S(l, r)$, *is given by*

$$S(l,r) = \begin{cases} \dfrac{|LCS(l,r)|}{|l|} & \text{(if } l \text{ has order constraint)} \\ \dfrac{|l \cap r|}{|l|} & \text{(otherwise)} \end{cases} \quad (1)$$

where $|LCS(l, r)|$ *is the number of items in the longest common subsequence between l and r,* $|l|$ *is the number of items in l, and* $|l \cap r|$ *is the number of common items between l and r. And it will be used in Section VI for the evaluation of accuracy.*

Problem definition: Given a dataset D, the delayed time d, the size threshold δ, the length threshold τ, multiple types of OPSM query is the process to search the OPSMs meeting *Definition* 1 to 8.

The process of OPSM query is mainly divided into the following three steps:

- *Index construction and update*: This is the most basic part. It uses prefix tree to load two different kinds of data, i.e., gene expression data and OPSM data. If several genes have the same prefix, they share the prefix in the tree and use the subsequent parts of these sequences as subbranches of the prefix. Index updates include data insertion and data deletion. It is an important task to make index updating more convenient and efficient.
- *Header table design*: This is an auxiliary data structure. It consists of two parts: (1) *Row Header Table* (*RHT* for short), which is used to facilitate deletion of pIndex index and OPSM query based on row keywords, and (2) *Column Header Table* (*CHT* for short), which is used to facilitate deletion of pIndex index and OPSM query based on column keywords.
- *Query processing*: It includes two sub-steps: (1) *Search*, which uses the row and column header tables to get the branches where the experimental conditions are located or the leaf nodes where the genes are located in a bottom-up way. (2) *Filter*, which mainly calculates the intersection of candidate sets in the previous step, and detects whether the candidate results are larger than the custom thresholds.

III. PFTREE

To design a compact and efficient index for OPSM queries, let us first examine an example *Example* 1. The row No. (gene name) and column No. (experimental condition) of OPSMs are listed in the first two and last two columns of *Table* 4, respectively.

TABLE 4. An OPSM dataset as running example.

Row No.	Column No.	Row No.	Column No.
g_1, g_2, g_5	$c_6, c_3, c_1, c_8, c_{16}$	g_4, g_8, g_{12}	c_3, c_2, c_{16}
g_3, g_6, g_9	c_6, c_3, c_1, c_2, c_8	g_4, g_6	$c_6, c_3, c_1, c_8, c_{16}$
g_7, g_{10}, g_{11}	c_6, c_2, c_3		

In the following, it takes the OPSM dataset as an example to illustrate how to create an index, because each row in the gene expression data can also be regarded as an OPSM. A compact index can be created based on the following observations:

- There are many repetitive fragments between OPSMs. If each duplicate fragment is stored only once, many unnecessary storage work can be avoided, and a lot of valuable memory space can be saved at the same time.
- If several OPSMs have exactly the same sequence of column labels, then these OPSMs can be merged into one, and only the row labels need to be put together in the merging process.
- If two OPSMs have the same prefix, then the same part can be made up of a common prefix and different parts can be made up of two branches.

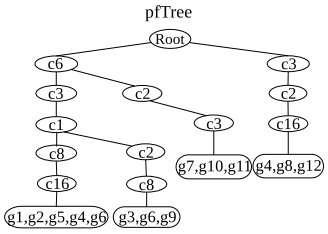


FIGURE 2. pfTree.

With these observations, we give an example to explain how to construct a prefix tree of OPSMs, called *pfTree*, which is a naive indexing method.

Example 4: Table 4 shows a sample OPSM dataset. This dataset will be used as our running example. The procedure of pfTree construction is plotted in Fig. 2.

In *Example* 4, initially, one may create the root of a tree, labelled with ''null''. Next, the OPSM data set is scanned, and the scanned first OPSM generates the first branch of the tree: $\{c_6, c_3, c_1, c_8, c_{16}\}$. Note that the inherent order of each column label must be maintained, as this sequence is a tendency for several genes that have a monotonically increasing or monotonically decreasing expression under these column labels. The row labels $\{g_1, g_2, g_5\}$ is then placed in the leaf node of the branch. For the second OPSM, since the column labels $\{c_6, c_3, c_1, c_2, c_8\}$ share the prefix with the column labels of the first OPSM, i.e., $\{c_6, c_3, c_1\}$, only the nodes $\{c_2\}$ and $\{c_8\}$ needs to be created, and are linked as child nodes of nodes $\{c_1\}$ and $\{c_2\}$, respectively. The row labels $\{g_3, g_6, g_9\}$ are then placed in the leaf node of the branch. For the column labels $\{c_6, c_2, c_3\}$ in the third OPSM, because it shares nodes $\{c_6\}$ with the first two branches, it is necessary to create two nodes $\{c_2\}$ and $\{c_3\}$ as child nodes of nodes $\{c_6\}$ and $\{c_2\}$, respectively. The row labels $\{g_7, g_{10}, g_{11}\}$ is then placed in the leaf node of the branch. For the forth OPSM $\{c_3, c_2, c_{16}\}$, since it does not share the prefix with the first three OPSMs, create a new branch $\{c_3, c_2, c_{16}\}$ and place the row labels $\{g_4, g_8, g_{12}\}$ in the leaf node of the branch. For the last OPSM $\{c_6, c_3, c_1, c_8, c_{16}\}$, because it is identical to the first OPSM, there is no need to create a new branch, just put the row labels $\{g_4, g_6\}$ in the leaf node of the branch which saves $\{g_1, g_2, g_5\}$. The method of creating *pfTree* refers to lines 6-7 in *Algorithm* 1.

From *Example* 4, we know that *pfTree* contains the complete information of dataset, i.e., *Lemma* 1, and its size and height are bounded by the amount of dataset and the longest OPSM, i.e., *Lemma* 2.

Lemma 1: Given a gene expression data or an OPSM dataset D, its corresponding pfTree contains the complete information of D.

Lemma 2: The size of pfTree is bounded by the amount of dataset D, and the height of the pfTree is bounded by the longest row in dataset D.

Lemma 3: The upper boundary of the space complexity of pfTree index is O(mn). The lower boundary of the space complexity of pfTree index is O(m).

Algorithm 1 pIndex Construction

Input: OPSM or gene expression dataset D

```
1  treeRoot ← null;
2  while (opsm ← D.nextLine()) ≠ null do
3      rows ← gene names in opsm; columns ←
       conditions in opsm; curNode ← treeRoot;
4      for it in columns do
5          linkFlag ← false;
6          if curNode does not have Child it then
7              curNode.addChild(it); linkFlag ← true;
8          curNode ← child it of curNode; gene frequency
           of curNode adds the gene name number in rows;
9          if columnHeadTable does not contain it then
10             put key-value < it, curNode > to
               columnHeadTable; set the back link of
               curNode to be null;
11         else
12             itNode ← get value by key it from
               columnHeadTable;
13             while itNode has forward Link do
14                 itNode ← forward Link of itNode;
15             if linkFlag = true then
16                 set forward link of itNode to be curNode;
                   set back link of curNode to be itNode;
17     set curNode to be final node of this branch; set the
       gene names in curNode to be rows;
18     for row in rows do
19         if rowHeadTable has the key row then
20             add the value searching by key row from
               rowHeadTable into nodeSet;
21         add curNode into nodeSet; put key-value
           < row, nodeSet > into rowHeadTable;
```

Proof: When there are n completely different branches, the space complexity is $O(mn)$. When n row data are constructed in the same branch, the space complexity is $O(m)$. □

Through experimental verification later, it is found that *pfTree*-based index deletion and OPSM query are not so efficient, although they have good performance in index creation and insertion. In order to improve its performance, an optimized indexing method *pIndex* is given in *Section IV*, which utilizes row and column header tables to facilitate the traversal of the prefix tree.

IV. PINDEX

pIndex construction includes three parts, which are *pfTree* building, column header table building, and row header table building. The *pfTree* building method has been shown in *Section* III, therefore we only give the column and row header table building approaches.

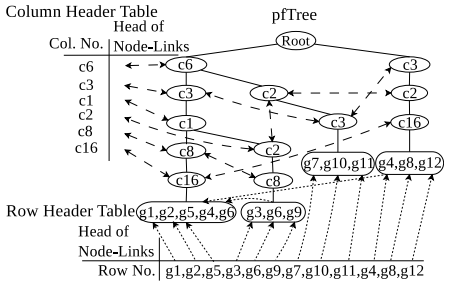


FIGURE 3. pIndex.

Example 5 (pIndex Building): We use Table 4 and Fig. 3 as an example. First, we create the root of pIndex. Then, scan the dataset, we get the OPSM $\{\{g_1, g_2, g_5\}, \{c_6, c_3, c_1, c_8, c_{16}\}\}$. *Create nodes* $\{c_6\}$, $\{c_3\}$, $\{c_1\}$,$\{c_8\}$, $\{c_{16}\}$, *and use the latter one as the child of the former one, respectively. Meanwhile, add each column label to the column header table, and link to the corresponding column node. Finally, add a leaf node* $\{g_1, g_2, g_5\}$. *In addition, we also add each row label to the row header table, and link to the corresponding row node. For the second OPSM* $\{\{g_3, g_6, g_9\}, \{c_6, c_3, c_1, c_2, c_8\}\}$, *due to it has prefix nodes* $\{c_6\}$, $\{c_3\}$, $\{c_1\}$ *with the first OPSM, it only needs to create nodes* $\{c_2\}$, $\{c_8\}$, *and uses* $\{c_8\}$ *as the child of* $\{c_2\}$. *Meanwhile, add column* c_2 *to CHT, and make* c_2 *in CHT and node* $\{c_2\}$ *link to each other. Due to* c_8 *is in CHT, we make the node* $\{c_8\}$ *in the first branch and node* $\{c_8\}$ *in the second branch link to each other. Finally, add the leaf node* $\{g_3, g_6, g_9\}$ *to node* $\{c_8\}$ *in the second branch. For the third one* $\{\{g_7, g_{10}, g_{11}\}, \{c_6, c_2, c_3\}\}$, *we create nodes* $\{c_2\}$ *and* $\{c_3\}$, *and make* $\{c_2\}$ *and* $\{c_3\}$ *as the child of* $\{c_6\}$ *and* $\{c_2\}$, *respectively. Meanwhile, let node* $\{c_2\}$ *in the second branch and node* $\{c_2\}$ *in this branch link to each other, and let node* $\{c_3\}$ *in the first branch and this branch link to each other. Finally,add a leaf node* $\{g_7, g_{10}, g_{11}\}$. *Next, it scans the forth OPSM* $\{\{g_4, g_8, g_{12}\}, \{c_3, c_2, c_{16}\}\}$, *we create nodes* $\{c_3\}$, $\{c_2\}$ *and* $\{c_{16}\}$, *and make the former one link to the latter one. Meanwhile, the nodes* $\{c_3\}$ *and* $\{c_2\}$ *in the third branch and* $\{c_{16}\}$ *in the first branch link to the nodes* $\{c_3\}$, $\{c_2\}$ *and* $\{c_{16}\}$ *in this branch, respectively. Finally, add a leaf node* $\{g_4, g_8, g_{12}\}$. *For the last OPSM* $\{\{g_4, g_6\}, \{c_6, c_3, c_1, c_8, c_{16}\}\}$, *we need not create new nodes and add new column label to CHT, only add the row No.* g_4 *and* g_6 *to the leaf node in the first branch. In addition, add row No.* g_4 *and* g_6 *to RHT, and make* g_4 *and* g_6 *link to node* $\{g_1, g_2, g_5, g_4, g_6\}$.

The method and rules for creating a column header table are as follows: the order of column labels appeared in the column header table is determined by the top-down, left-to-right order of the column labels in the prefix tree. Each element that appears in the column header table is pointed to its position in the first occurrence of the branch by a bidirectional link. Later, if it appears in other branches, the nodes in the former and latter branch will point to each other through bidirectional link. Continue in turn until no identical elements appear. When the entire OPSM data is scanned, the OPSM prefix tree index with column header table is established, which is shown in *Fig.* 3. The method of creating the column header table is shown in lines 9-16 of *Algorithm* 1. And column header table can be used in *pIndex* deletion and queries on conditions c.

The method and rules for creating row header table are as follows: the occurrence order of elements in row header table is similar to that in column header table, that is, it is determined by the occurrence order of row labels from left to right in the leaf nodes of prefix tree. In addition, linking manner between similar elements in row header table is unlike that in column header table, it places the nodes of the same element in the row header table into a hash set. For convenience of representation, one-way pointers are used in *Fig.* 3. Similarly, when the entire OPSM data is scanned, the OPSM prefix tree index with row header table is established, as shown in *Fig.* 3. The method for creating row header table is shown in rows 18-21 of *Algorithm* 1. And row header table can be used in *pIndex* deletion and queries on genes g.

In order to facilitate the update operation of *pIndex*, we give the insertion and deletion methods of *pIndex*. Because the insertion of *pIndex* is similar to the creation of *pIndex*, only the deletion methods of *pIndex* are introduced here, including deletion *by row* and *by column*.

Algorithm 2 pIndex Deletion by Rows

```
   Input: gene names rows
1  for row : rows do
2      nodeSet ← get nodes from rowHeadTable by key
       row;
3      for node in nodeSet do
4          if node number in nodeSet is more than 1 then
5              remove the node node in rowHeadTable by
               key row;
6          else if node number in nodeSet is 1 then
7              remove the value in rowHeadTable by key
               row;
8          else if node number in nodeSet is 0 then
9              break;
10         if gene name number in node is more than 1 then
11             delete the name from node by key row;
12         else if gene name number in node is 1 then
13             deleteNodesAndColums(node);
```

For *pIndex deletion by rows*, *Algorithm* 2 is given. First, it gets the keywords of rows (gene names). For each row keyword, it contains three operations: deleting from the row header table, deleting tree nodes, and deleting from the column header. For the first step, it gets the leaf nodes from the row header table (lines 1-2). Next, the number of leaf nodes obtained is detected. If the number of nodes is 0, the algorithm ends (lines 8-9). If the number of nodes is 1, the node is deleted from the row header table (lines 6-7). If the number of nodes is greater than 1, only the row (gene name) is deleted from the row header table (lines 4-5). For the other

two operations (lines 10-13), i.e., deleting nodes and deleting columns, refer to *Algorithm* 3.

Algorithm 3 Delete Nodes and Columns

```
   Input: nodes nodes
1  while node in nodes and node is not treeRoot do
2      decrease 1 form the frequency of node;
3      remove column ID in node from the child of its
       parent;
4      tmp ← node; node ← the parent of node;
5      set the parent of tmp to be null;
6      if both the back link and forward link of tmp are not
       null then
7          let the forward node of tmp and back node of
           tmp point to each other;
8      else if the back link of tmp is not null and the
       forward link of tmp is null then
9          let the back node of tmp do not point to tmp;
10     else if both the back link and forward link of tmp are
       null then
11         remove tmp from the column header table;
12     else if the back link of tmp is null and the forward
       link of tmp is not null then
13         set the back link of the forward node of tmp to
           be null; put the forward node of tmp to column
           header table instead of tmp;
14     if (1 < node.getFrequency) then break;
15 return pIndex;
```

Example 6 (pIndex deletion by rows): We use Table 4 and Fig. 3 as an example, and employ $\{g_7, g_{10}\}$ *and* $\{g_7, g_{10}, g_{11}\}$ *as the keywords respectively. When the keywords are* $\{g_7, g_{10}\}$*, we only need delete* g_7*,* g_{10} *from the leaf node and RHT. When the keywords are* $\{g_7, g_{10}, g_{11}\}$*, due to they are in one branch, so we fetches the leaf node* $\{g_7, g_{10}, g_{11}\}$ *from RHT. Since the support count of* g_7*,* g_{10}*,* g_{11} *are all one, we remove them from the RHT. Then, we directly remove the leaf node* $\{g_7, g_{10}, g_{11}\}$*, and remove nodes* $\{c_3\}$*,* $\{c_2\}$*,* $\{c_6\}$ *with a bottom-up manner. For node* $\{c_3\}$*, we first make the node* $\{c_3\}$ *in the first and forth branch link to each other, then remove it from the child of node* $\{c_2\}$*. And we use the same way to remove* $\{c_2\}$*. But for node* $\{c_6\}$*, we need not to remove it, due to it shares by other branches. Until now, we delete the branches that contain the keywords above.*

Next, the details of *Algorithm* 3 are introduced. First, it checks whether the node is the root of the tree. If so, the algorithm ends (line 1). Otherwise, the number of branches that sharing the node is reduced by 1 (line 2). Because the node to be deleted is in the leaf node, first the node is deleted from its parent node, and then the parent node of the node is set to *null* (lines 3-5). The deletion of the node from the column header table is divided into four cases. (1) If the node has forward and backward nodes, it is necessary to make the forward and backward nodes point to each other (lines 6-7). (2) If it only has a backward node, the forward pointer of the backward node and the backward pointer of the node need to be set to *null* (lines 8-9). (3) If it has neither a backward node nor a forward node, it only needs to delete the key-valve pair containing the node from the column header (lines 10-11). (4) If it has only the forward node, first set the backward pointer of the forward node to empty, and then place the forward node in the column header table to replace the current node (lines 12-13). Finally, if the number of branches of the parent node sharing the current node is greater than 1, the algorithm ends (line 14).

Algorithm 4 pIndex Deletion by Columns

```
   Input: columns columns
1  key ← the last item in columns;
2  node ← get node from column header table by key;
3  while node is not null do
4      pNode ← the parent of node;
5      count ← |columns| - 2;
6      while count is not smaller than 0 do
7          if the item in pNode is equal to the count item of
           column then
8              count ← count - 1;
9          if pNode is treeRoot then
10             break;
11         else pNode ← the parent of pNode;
12     if count is not smaller than 0 then
13         put node into key node set keyNodes;
14     node ← the forward node of node;
15     for one different node keyNode in keyNodes do
16         leafNodes ← the leaf nodes in branches
           containing keyNode;
17     for one different node leafNode in leafNodes do
18         for one gene name row in leafNode do
19             if gene name row is not in row header table
               then
20                 break;
21             else if the leaf node number containing row
               is 1 then
22                 remove row from row header table;
23             else if the leaf node number containing row
               is more than 1 then
24                 only remove the node leafNode;
25         deleteNodesColumns(leafNode); (Algorithm 3)
```

In order to delete pIndex by column keywords, it introduces *Algorithm* 4. Because it traverses the prefix tree from bottom to up, it first gets the last column keyword and the node position of the column keyword in the column header table (lines 1-2). According to the pointer chain containing the same list

of keywords, the related branches are detected whether they contain all keywords. If so, record the currently traversed nodes. Otherwise, the next branch is detected according to the keyword pointer chain of the above columns (lines 4-14). When the qualified traversed nodes are obtained, the leaf nodes of the related branches are obtained according to the above nodes (lines 15-16). Further, the elements in the relevant branches and row header table are deleted according to the leaf nodes in the bottom-up manner (lines 17-24). Removal of related branches is performed by calling *Algorithm* 3 (line 25). It should be noted that it is a recursive way to find leaf nodes. For more details, please refer to *Algorithm* 5.

Algorithm 5 Find Leaf Nodes

Input: tree node *node*
Output: leaf nodes *leafNodes*
1 **if** *node is leaf node* **then** add *node* into *leafNodes*;
2 **if** *node has child* **then**
3 $\quad$ *children* $\leftarrow$ the children of *node*;
4 $\quad$ **while** *child in children* **do** findLeafNodes(*child*);
5 **return** *leafNodes*;

Example 7 (pIndex deletion by columns): We use Table 4 and Fig. 3 as an example, and employ c_6, c_3, c_1 as keywords. And use pIndex deletion by columns to delete the data. Initially, it reverses the keywords, and gets c_1, c_3, c_6. Then, it searches column c_1 in CHT, and traces the first branch in a bottom-up way. And it finds c_1, c_3, c_6 in the branch. Further, it gets the leaf nodes $\{g_1, g_2, g_4, g_5, g_6\}$ and $\{g_3, g_6, g_9\}$. Finally, it firstly delete g_1, g_2, g_5, g_6, g_9. Due to g_4 in two leaf nodes, it only deletes the link from node $\{g_4, g_8, g_{12}\}$ to $\{g_1, g_2, g_4, g_5, g_6\}$. Next, change the link in the CHT, except $\{c_6\}$, for nodes $\{c_1\}$, $\{c_2\}$, $\{c_3\}$, $\{c_8\}$, $\{c_{16}\}$, it changes the links to null, $\{c_2\}$ in the third branch, $\{c_3\}$ in the third branch, null, $\{c_{16}\}$ in the forth branch, respectively. Then, it deletes column labels c_1 and c_8 from CHT. Finally, it deletes nodes $\{g_1, g_2, g_4, g_5, g_6\}$, $\{g_3, g_6, g_9\}$, $\{c_{16}\}$, $\{c_8\}$, $\{c_1\}$, $\{c_3\}$ from the first and second branches, respectively.

Lemma 4: The upper boundary of the space complexity of pIndex is $O(mn)$. The lower boundary of the space complexity of pIndex is $O(m + n)$.

Proof: The row and column header tables will consume $O(n)$ and $O(m)$ space, respectively. Based on *Lemma* 3, when there are n completely different branches, the space complexity is $O(mn)$, and when n row data are constructed in the same branch, the space complexity is $O(m + n)$. □

V. OPSM QUERIES

In this section, we explore the multiple types of OPSM queries, which include positive, negative, and time-delayed OPSM queries, based on *pIndex* with two header tables.

A. POSITIVE OPSM QUERIES

For row-based exact query EQ_g, the position of row keywords in the index is first determined by row header table (lines 2-4). Then, traverse the branches containing the row keywords in a bottom-up manner (lines 5-9). Further, calculate the longest common subsequence between branches that contain each keyword once and only once (lines 13-20). Finally, it returned the longest common subsequences that are larger than the threshold τ (lines 21-23). For more detailed information, please refer to *Algorithm* 6.

Algorithm 6 (EQ_g) Exact Query on Genes

Input: Gene name keywords g, Length threshold τ
Output: HashMap< g, set of conditions> *result*
1 **for** *name in g* **do**
2 $\quad$ **if** *keyword name is not in row header table* **then**
3 $\quad\quad$ **return** *null*;
4 $\quad$ *nodeSet* $\leftarrow$ get values from row header table by *name*;
5 $\quad$ **for** *node in nodeSet* **do**
6 $\quad\quad$ *colList* $\leftarrow$ the branch containing *node*;
7 $\quad\quad$ **if** (*colList.length* $\geq \tau$) **then**
8 $\quad\quad\quad$ add *colList* into *fstLists*;
9 $\quad$ put key-value <*name*, *fstLists*> to *hashMap*; clear the items in *fstLists*;
10 **if** *the gene number in g is* 1 **then return** *hashMap*;
11 *fstLists* $\leftarrow$ get values from *hashMap* by g_0;
12 *flag* $\leftarrow$ *false*;
13 **for** $i \leftarrow 1$ *to* $|g| - 1$ **do**
14 $\quad$ **if** *flag is true* **then**
15 $\quad\quad$ *fstLists* $\leftarrow$ *resLists*; clear *resLists*;
16 $\quad$ *flag* $\leftarrow$ *true*; *secLists* $\leftarrow$ get values by g_i from *hashMap*;
17 $\quad$ **for** *out in fstLists, in in secLists* **do**
18 $\quad\quad$ *lcs* $\leftarrow$ LongestCommonSubsequence(*out*, *in*);
19 $\quad\quad$ **if** *lcs.length* $\geq \tau$ **then**
20 $\quad\quad\quad$ *out* $\leftarrow$ *lcs*; add *lcs* into *resLists*;
21 **if** *resLists.size*() > 0 **then**
22 $\quad$ add key-value $<$ g, *resLists* $>$ into *result*;
23 **return** *result*;

Example 8 (Exact Query on Genes EQ_g): Take the data in Table 4 and the pIndex index in Fig. 3 as an example to illustrate the gene-based exact query (EQ_g) algorithm. Given the gene name $\{g_2, g_3, g_9\}$, and column threshold 3, the set of experimental conditions containing the above genes was queried. Firstly, the node links containing genes $\{g_2, g_3, g_9\}$ are found in the row header table. Then the branches $\{c_6, c_3, c_1, c_8, c_{16}\}$ containing gene g_2, and the branches $\{c_6, c_3, c_1, c_2, c_8\}$ containing genes g_3 and g_9 were obtained. Further, the longest common subsequence $\{c_6, c_3, c_1, c_8\}$ between the above branches is calculated. Since the length is greater than the threshold 3, it is the result.

Through the query process of *Example* 8, *Rule* 1 for pruning query results is found.

Rule 1 (Keyword No. based Pruning): For row and column keywords in exact queries, in exact queries based on experimental conditions, all column keywords must be included in the branches they find. In exact query based on genes, leaf nodes must contain all row keywords. If the condition is not satisfied, then checking of the branch or leaf node can be cancelled.

For column-based exact query EQ_c, the position of column keywords in the index is first determined by the column header table (lines 1-2). Then, starting with the branch node where the keyword is located, the branch is traversed in a bottom-up manner. It verifies that the branch contains all column keywords, and check the consistency in the order of these keywords in the branches (lines 4-12). If consistent, return the branches containing more than the threshold δ (lines 14-16). See *Algorithm* 7 for more details.

Algorithm 7 (EQ_c) Exact Query on Conditions

Input: Condition keywords c, Size threshold δ
Output: HashMap<c, set of genes> *result*

```
1  key ← the last item in c;
2  node ← get value from column header table by key;
3  if node is null then return null;
4  while node is not null do
5      pNode ← the parent of node; count ← |c| - 2;
6      while count ≥ 0 do
7          if the count item in c is equal to item in pNode
           then count ← count - 1;
8          if pNode is treeRoot then break;
9          else pNode ← the parent of pNode;
10     if count < 0 then add node into nodes;
11     node ← the forward node of node;
12 for inNode in nodes do
13     nameSet ← the leaf nodes in branches containing
       inNode;
14 if nameSet.size() ≥ δ then
15     put key-value < c, nameSet > into result;
16 return result;
```

Example 9 (Exact Query on Conditions EQ_c): Take the data in Table 4 and the pIndex index in Fig. 3 as an example to illustrate the column-based exact query algorithm (EQ_c). Given the experimental conditions c_6, c_3, c_1, and row threshold 3, query the gene sets containing the above experimental conditions. Firstly, the link of nodes containing experimental conditions c_1, c_3, c_6 (reverse order of input keywords) is found in the column header table. Then, branches $\{c_6, c_3, c_1, c_8, c_{16}\}$ and $\{c_6, c_3, c_1, c_2, c_8\}$ containing experimental conditions c_1 were obtained. These branches were then tested for inclusion of experimental conditions c_3 and c_6. In results, the above branches satisfy the conditions. Finally, gene set $\{g_1, g_2, g_3, g_4, g_5, g_6, g_9\}$ is larger than the threshold 3. It is the query result.

From *Example* 9, we find *Rule* 2 for pruning query results.

Rule 2. (Order based Pruning) For column keywords in exact queries, the branch queried must contain all column keywords, and the order of column keywords in the branch must also be consistent with the order in which keywords are entered. Otherwise, checking the branch is cancelled.

For fuzzy query based on genes FQ_g, the combination of row keywords larger than row threshold δ is calculated first (lines 1-2). Then, locate the branches containing each row keyword combination, and compute the longest common subsequence among the branches (lines 3-4). If it is greater than the column threshold τ, it is returned as a result (line 5). *Algorithm* 8 gives a more detailed query method.

Algorithm 8 (FQ_g) Fuzzy Query on Genes

Input: Gene name keywords g, Length threshold τ, Size threshold δ
Output: HashMap<subset of g, set of conditions> *result*

```
1  for i ← δ to |g| do
2      querySetLists ← the combination of i items in g;
3  for querySet in querySetLists do
4      result ← EQ_g(querySet, τ);
5  return result;
```

Example 10 (Fuzzy Query on genes FQ_g): Take the data in Table 4 and the pIndex index in Fig. 3 as an example to illustrate the gene based fuzzy query algorithm FQ_g. Given the gene name g_2, g_3, g_9, row threshold 2 and column threshold 3, the set of experimental conditions containing the above genes was queried. Firstly, the combinations of row keywords exceeding row threshold 2 were calculated, which are $\{g_2, g_3\}$, $\{g_2, g_9\}$, $\{g_3, g_9\}$, $\{g_2, g_3, g_9\}$, respectively. Then, the branch $\{c_6, c_3, c_1, c_8, c_{16}\}$ containing gene g_2, and the branch $\{c_6, c_3, c_1, c_2, c_8\}$ containing gene g_3 and g_9 was obtained. Further, the experimental condition sequence containing the above gene name combinations was calculated. The set of experimental conditions including $\{g_2, g_3\}$, $\{g_2, g_9\}$ and $\{g_2, g_3, g_9\}$ is $\{c_6, c_3, c_1, c_8\}$. The set of experimental conditions containing $\{g_3, g_9\}$ is $\{c_6, c_3, c_1, c_2, c_8\}$. As the lengths of the above experimental conditions are greater than the threshold 3, all of them are the results.

For column-based fuzzy query FQ_c, first flip the order of column keywords and take out the first element (lines 2-3). Then use the column header table to locate the branch containing this element, and detect whether these branches contain column keywords greater than the threshold τ (lines 4-20). If so, get the row label in the branch leaf node, and check whether the number of rows is greater than the threshold δ (lines 21-25). If so, return the column labels as keywords and the corresponding row labels as values (lines 26-28). Otherwise, continue to check whether other branches and other elements satisfy the above conditions when they are the first element (line 1). Note that the first element rotation method *FIT* is used to reduce the number of column keyword sets. Refer to *Algorithm* 9 for more details.

Algorithm 9 (FQ_c) Fuzzy Query on Conditions

Input: Condition keywords c, Size threshold δ, Length threshold τ
Output: HashMap<subset of c, set of genes> *result*

```
1  for i ← 0 to |c| − 1 do
2      key ← get value from column header table by c_i;
3      if key is null then return null;
4      while key is not null do
5          add c_i into list; node ← the parent of key;
6          no ← |c| − 2;
7          while no ≥ |c| − τ do
8              for it in c do
9                  if it is equal to the item in node then
10                     no ← no − 1; add it to list; break;
11             if node is treeRoot then break;
12             else node ← the parent of node;
13         if no < |c| − τ then
14             while node is not treeRoot do
15                 for it in c do
16                     if it is equal to the item in node then
17                         no ← no − 1; add it to list;
                           break;
18                 node ← the parent of node;
19             add key to nodes;
20         key ← the forward node of key;
21         for inNode in nodes do
22             nameSet ← getNamesInLeaves(inNode);
23         if result has the key list then
24             add the values getting from result by list to
               nameSet;
25         put key-value <list, nameSet> to result; clear
           nameSet and list;
26     for res in result do
27         if |res.value| < δ then
28             remove the value res from result;
29 return result;
```

Example 11 (Fuzzy Query on Conditions FQ_c): Take the data in Table 4 and the pIndex index in Fig. 3 as an example to illustrate the column-based fuzzy query algorithm (FQ_c). Given the experimental conditions c_6, c_3, c_1, column threshold 2 and row threshold 3, query the gene sets containing the above experimental conditions. Firstly, use c_1 as the first element, and get branch $\{c_6, c_3, c_1\}$ which is above the length threshold 2. Then, we fetch the gene name set $\{g_1, g_2, g_3, g_4, g_6, g_9\}$ which is above the size threshold 3. Further, we use c_3 as the first element, and get branch $\{c_6, c_3\}$ and $\{c_6, c_2, c_3\}$ which are above the length threshold 2. Finally, we fetch the gene name set $\{g_1, g_2, g_3, g_4, g_6, g_7, g_9, g_{10}, g_{11}\}$. When using c_6 as the first element, there are no results. Now, we return keyword sets and gene sets above as the results.

Lemma 5: The upper boundary of the time complexity of First Item of column keyword roTation method FIT is $O(mn)$. The lower boundary of the time complexity of First Item of column keyword roTation method FIT is $O(m)$.

Proof: Due to there are m number of first keywords and it uses column header table to locate the branches, when the column keywords are in n completely different branches, we get the space complexity is $O(mn)$. When column keywords are in the same branch, we get the space complexity is $O(m)$. □

Theorem 1: Let c_1 and c_2 be two subsequences of keywords in c such that c_1 is a subsequence of c_2. For any OPSMs in query results, support number $s(c_2) \leqslant s(c_1)$.

Proof: It is sufficient to show that the theorem is true for subsequences of column keywords whose length differ by 1, i.e., $|c_2| = |c_1| + 1$. We can repeat argument to prove the theorem for patterns of arbitray length. Let j be the column that is in c_2 but not in c_1. Each subsequence of OPSM's column part that matches c_1 can potentially be extended to match c_2 by inserting a column j. Therefore $s(c_2) \leqslant s(c_1)$. □

Corollary 1: Let c_1 and c_2 be two subsequences of keywords in c such that c_1 is a subsequence of c_2. c_2 is frequent only if c_1 is frequent.

Algorithm 10 (GEQ_c) General OPSM Exact Query on Conditions

Input: Condition keywords c, Size threshold δ, Delayed time d
Output: HashMap<c, set of genes>*result*

```
1  result.add(c, EQ_c(c, δ)); //POPSM
2  c_inverted ← the reverse order of c;
3  EQ_c(c_inverted, δ); //NOPSM
4  for i ← 1 to |d| do
5      col ← the sequence of c from c_i to the end;
6      EQ_c(col, δ); //DPOPSM
7      verificate and delete false positive results;
8      result.add(col, EQ_c(col, δ));
9  for i ← 1 to |d| do
10     col ← the sequence of c_inverted from c_i to the end;
11     EQ_c(col, δ); //DNOPSM
12     verificate and delete false positive results;
13     result.add(col, EQ_c(col, δ));
14 return result;
```

B. GENERAL OPSM QUERIES

Based on the Positive OPSM query method, we present a general query method for multiple types of OPSM search, *Algorithm* 10, which consists of Positive OPSM query, Negative OPSM query, and Time-delayed OPSM query. When searching the POPSMs, it invokes the *Algorithm* 7 EQ_c (line 1). For the NOPSM queries, it firstly reverses the order of keywords c, and gets inverted keywords $c_{inverted}$, then it invokes EQ_c (lines 2-3). For the DPOPSM queries, the delayed times

of which are less or equal to d, firstly, it fetches the keywords from c_i to the end one, where $0 \leqslant i \leqslant d$, then it invokes EQ_c (lines 4-8), finally it certificates and deletes the false positive results. For the DNOPSM queries, the delayed times of which are less or equal to d, firstly, it fetches the inverted keywords $c_{inverted}$ from c_i to the end one, where $0 \leqslant i \leqslant d$, then it invokes EQ_c (lines 9-13), finally it certificates and deletes the false positive results.

Example 12 (General Exact Query on Columns GEQ_c): Given dataset shown in Table 1, and the column query keywords c_1, c_2, c_3, c_4, please search POPSMs, NOPSMs, DPOPSMs and DNOPSMs using the pIndex and query method. And through searching, we get the POPSMs $\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}$, *NOPSMs* $\{\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}, \{\{g_3\}, \{c_5, c_4, c_3, c_2, c_1\}\}\}$, *DPOPSMs* $\{\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}, \{\{g_4\}, \{c_2, c_3, c_4, c_5, c_6\}\}\}$, *DNOPSMs* $\{\{\{g_1, g_2\}, \{c_1, c_2, c_3, c_4, c_5\}\}, \{\{g_5\}, \{c_6, c_5, c_4, c_3, c_2\}\}\}$. *The OPSMs are shown in Table 2.*

C. ONLINE SHARING QUERIES

Due to there are a lot of repeated OPSM queries, and some queries will consume a long response time and waste many computing resources. To accelerate the repeated and frequent searches, we introduce an online-sharing query method.

Firstly, we find the repeated queries Q_r. Then, we find the queries Q_t whose response time is above a threshold t. Further, we compute the intersection between Q_r and Q_t, denoted by Q. If there are enough spaces, we save all the results Q in the memory. If not, for each $q \in Q$, we find the *top-k* results and save in the memory. Based on the intuition that the more repeated times and the longer response time, the more likely to be stored, we give the criteria finding *top-k* results, shown in equation (2).

$$R(q) = \frac{Cnt_q}{Cnt_{max}} \times \alpha + \frac{Time_q}{Time_{max}} \times \beta \tag{2}$$

where $R(q)$ is the ranking score of query q, and the bigger score the higher rank; Cnt_q denotes the repeated times of query q, Cnt_{max} represents the maximum times in the repeated querys Q_r, $Time_q$ denotes the response time of query q, $Time_{max}$ represents the maximum time in the repeated querys Q_t, α and β are the weight in the score of ranking, and $\alpha + \beta = 1$.

VI. EXPERIMENTAL EVALUATION

This section mainly evaluates the effectiveness and scalability of the proposed indexing methods, query approaches, and optimization techniques with existing methods. We will compare the methods on a single machine and distributed processing systems. The experiments mainly verify the following aspects:

- The indexes *pIndex* and *pfTree* nearly have the same index size, and the compact ratio is close to 0.98 when the number of conditions is smaller.
- On single machine, although *pfTree* performs well on index building and index insertion, *pIndex* outperforms *pfTree* by 1 to 2 orders of magnitude in various cases on index deletion, EQ_g, EQ_c, FQ_g and FQ_c. Meanwhile, the exact and fuzzy query methods EQ_g, EQ_c, FQ_g and FQ_c with online sharing query technique have a better performance than those without online sharing query technique.
- On distributed processing systems, e.g., Hadoop and our modified Hama platform [24], the indexing method *pIndex* and query methods also show better scalability than existing methods.
- We also test the behaviours (scalability and accuracy) of multiple types of OPSM query method and online-sharing query technique.

TABLE 5. Details of the gene expression datasets.

Dataset	File Name	Rows	Columns
D_1	adenoma	12488	6
D_2	a549	22283	11
D_3	5q_GCT_file	22278	24
D_4	krasla	12422	50
D_5	bostonlungstatus	12625	94
D_6	bostonlungsubclasses	12625	202

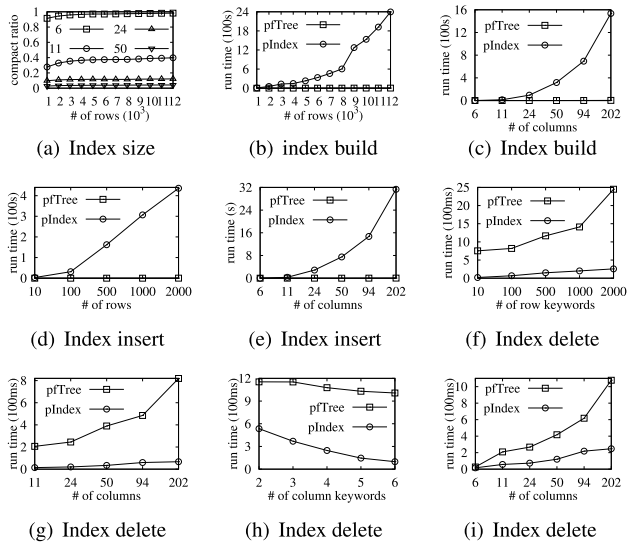
Datasets: We use two kinds of datasets in our experiments: real datasets [25] shown in *Table* 5 and a series of synthetic datasets. Most of our experiments have been performed on the real datasets since it is the source of real demand. Note we should sort the expression values in each row in an increasing or decreasing order, and replace each value with column label (index). In essence, it changes the real number to sequence data. Further, all the tests are based on the sequence data.

Platform: All our experiments are performed on 1.87GHz, 16GB memory, Inspur servers running Ubuntu 12.04. *pIndex*, *pfTree*, query methods and other techniques are implemented in Java language and complied with Eclipse 4.3. And the versions of Hadoop and Hama are 0.20.2 and 0.4.0, respectively.

A. EVALUATION ON SINGLE MACHINE

1) PINDEX VS PFTREE

First, we evaluate the size of *pfTree* and *pIndex*. As mentioned earlier, both *pfTree* and *pIndex* indexes index data based on prefix tree. Although *pIndex* index includes auxiliary data structure, i.e., row and column header tables, this auxiliary structure only needs a small amount of memory space, so there is a common compression ratio. *Fig.* 4(a) shows the compression ratio of the two indexes in the process of having four different columns (6, 11, 24, 50 columns) and increasing the number of rows from 1000 to 12000. The curves in the figure clearly show the fact that the fewer columns, the higher compression ratio. In addition, it also shows a hidden attribute: index compression is basically independent of the change of row number, that is, when the number of rows keeps growing, the index compression ratio does not change significantly.



(a) Index size (b) index build (c) Index build
(d) Index insert (e) Index insert (f) Index delete
(g) Index delete (h) Index delete (i) Index delete

FIGURE 4. Indexing performance on a single machine.

The size of indexes *pfTree* and *pIndex* has been verified. Here, the performance of *pfTree* and *pIndex* indexes in the creation process is evaluated. *Fig.* 4(b) and (c) show the creation time when two methods index datasets of different rows and columns, respectively. As shown in the figure, in the process of index creation, *pfTree* takes less time to index under all row and column conditions than *pIndex*, because *pIndex* takes extra time to create row and column header tables. Although the performance of *pIndex* on index creation is not as good as that of *pfTree*, the performance of *pIndex* is significantly better than that of *pfTree* in other subsequent tests.

Similar to the time-consuming situation of index creation, when inserting 10 to 2000 rows into *pfTree* and *pIndex* indexes with fixed columns (200 columns) and 6 to 202 columns with fixed rows (100 rows), *pfTree* also takes less time than *pIndex* indexes. For more details, refer to *Fig.* 4(d) and (e).

Fig. 4(f), (g), (h) and (i) show the scalable performance of removing some data from the *pfTree* and *pIndex* indexes. This test uses $10k$ rows of data in D_6 in *Table* 5 as index data. If not specified later, the number of rows and columns in the data used is $|G| = 10k$ and $|C| = 202$, respectively. First, test the time consumption of deleting 5 gene (row) sets from the index. As can be seen from *Fig.* 4(f), the time consumption of *pfTree* index on deletion increases from 754*ms* to 2449*ms*, while that of *pIndex* increases from 20*ms* to 256*ms*. The growth trend and time consumption of *pIndex* are significantly smaller than that of *pfTree*. Next, we delete the index by 100 row keywords on 5 different datasets ($|G| = 10k$, varying column numbers $|C|$). As shown in *Fig.* 4(g), the run time of *pfTree* dramatically increases from 206*ms* to 819*ms*, while that of *pIndex* increases from 14*ms* to 67*ms*. Similarly, when five sets of experimental conditions (columns) are deleted from the index, as shown in *Fig.* 4(h), the deletion time of *pfTree* index is reduced from 1154*ms* to 1007*ms*, while that of *pIndex* is decreases from 535 MS to 99 Ms. The decreasing trend of *pIndex* is obviously larger than that of *pfTree*. Finally, we delete the index by 4 column keywords on 6 different datasets ($|G| = 10k$, varying column numbers $|C|$). In *Fig.* 4(i), the run time of *pfTree* increases from 26*ms* to 1077*ms*, while that of *pIndex* increases from 16*ms* to 247*ms*. The increasing speed of the latter one is slower than the former, thus the proposed method has good performance.

2) EQ/FQ ON PFTREE/PINDEX/SQUERY

Next, we evaluate the scalability performance of *EQ* and *FQ* on *pfTree*, *pIndex*, and online sharing query technique (*sQuery* for short). In the following, we call the three methods above as *EQ*/*FQ*-*pfTree*, *EQ*/*FQ*-*pIndex*, and *EQ*/*FQ*-*sQuery*, respectively.

The first test is the exact query performance based on row (gene) keywords (EQ_g). Three compared methods are EQ_g-*pfTree*, EQ_g-*pIndex*, and EQ_g-*sQuery*. As shown in *Fig.* 5(a), the running times of three methods are nearly three horizontal lines, but the running time of EQ_g-*pfTree* is more than 30 times of the other methods. Further, the running time of EQ_g-*pIndex* is nearly 2 times of EQ_g-*sQuery*. When the same query is performed on six different datasets (that is, the index data is six different data), as shown in *Fig.* 5(b), the running time of EQ_g-*pfTree* is more than 35 times of EQ_g-*pIndex*. And the running time of EQ_g-*sQuery* is only half of that of EQ_g-*pIndex*. The test demonstrates the efficiency of row header table and the scalability of EQ_g-*pIndex* and EQ_g-*sQuery*.

The second experiment is the fuzzy query performance based on row (gene) keywords (FQ_g). Three compared methods are FQ_g-*pfTree*, FQ_g-*pIndex*, and FQ_g-*sQuery*. *Fig.* 5(c) and (d) show the performance of fuzzy query based on row (gene) keywords on different number of row keywords and datasets respectively. The running time of FQ_g-*pfTree* increases greatly, while the running times of FQ_g-*pIndex* and FQ_g-*sQuery* are both basically a horizontal line. And FQ_g-*sQuery* has better performance than FQ_g-*pIndex*. Overall, in both cases, FQ_g-*pIndex* performs 70 to 360 times (and 8 to 130 times) better than FQ_g-*pfTree*. And FQ_g-*sQuery* only consumes half of that of FQ_g-*pIndex*. The test demonstrates the efficiency of row header table and the scalability of FQ_g-*pIndex* and FQ_g-*sQuery*.

The third test is the exact query performance based on column (experimental condition) keywords (EQ_c). Three compared methods are EQ_c-*pfTree*, EQ_c-*pIndex*, and EQ_c-*sQuery*. As shown in *Fig.* 5(e), when the index data is D_6 and the number of column keywords increases from 2 to 6, the performances of EQ_c-*pIndex* and EQ_c-*sQuery* are much better than EQ_c-*sQuery* except for the case of keyword 2. The potential reason is that when there are fewer keywords, EQ_c-*pfTree* traverses fewer tree nodes, so the performance is better than the other two methods when the number of keywords is 2. Then we test the performance when indexing six different kinds of data and executing the same exact query based on columns. As shown in *Fig.* 5(f), the

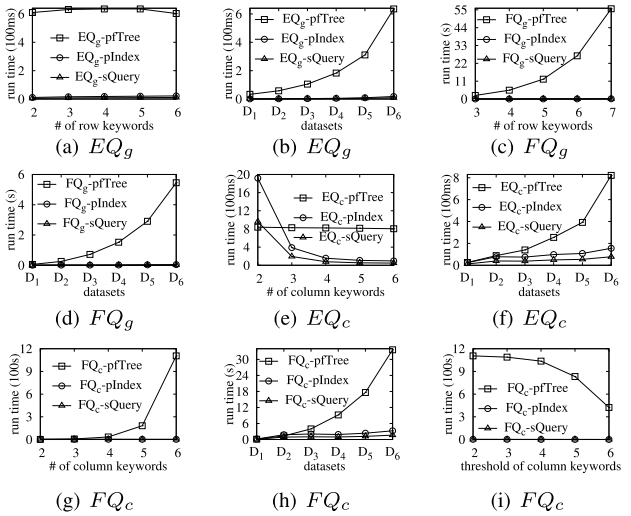


FIGURE 5. Query performance on a single machine.

performance of EQ_c-*pIndex* is 1 to 9 times that of EQ_c-*pfTree*. And the performance of EQ_c-*sQuery* is nearly 2 times that of EQ_c-*pIndex*.

The fourth test is the fuzzy query performance based on column (experimental condition) keywords (FQ_c). Three compared methods are FQ_c-*pfTree*, FQ_c-*pIndex*, and FQ_c-*sQuery*. As shown in *Fig.* 5(g), when the index data is D_6 and the number of column keywords increases from 2 to 6, the running time of FQ_c-*pfTree* increases sharply, but the running times of FQ_c-*pIndex* and FQ_c-*sQuery* are both basically a horizontal line. Then we test the performance when indexing six different data and executing the same column based fuzzy query. In *Fig.* 5(h) the response time of FQ_c-*pfTree* increases from 120*ms* to 33589*ms*, while the time of FQ_c-*pIndex* increases from 176*ms* to 3265*ms*, and that of FQ_c-*sQuery* increases from 88*ms* to 1633*ms*. The growth trend and execution time of FQ_c-*pIndex* and FQ_c-*sQuery* are far less than that of FQ_c-*pfTree*. Finally, the performance of the same fuzzy query based on six column keywords is tested when only the threshold of column keywords is changing, on dataset D_6, and the same fuzzy query based on six column keywords is executed. As shown in *Fig.* 5(i), the performance of FQ_c-*pIndex* is 461 to 759 times that of FQ_c-*pfTree*. And FQ_c-*sQuery* outperforms nearly 2 times that of FQ_c-*pIndex*. This test proves that the column header table and the first element rotation method (*FIT*) play an important role in the query process, and also shows that FQ_c-*pIndex* and FQ_c-*sQuery* have good scalability in the query based on column keywords.

3) GEQ VS EQ

This subsection tests the behaviours of five query methods. The query methods include (1) exact query of positive OPSM with condition keywords based on *pfTree* index (EQ_c-*pfTree*), (2) fuzzy query of positive OPSM with condition keywords based on *pIndex* index (EQ_c-*pIndex*), (3) general query of negative OPSM with condition keywords based on *pIndex* index (GEQ_c-*nega*), (4) general query of positive and delay OPSM with condition keywords based on *pIndex* index (GEQ_c-*posi-delay*), (5) general query of negative and delay OPSM with condition keywords based on *pIndex* index (GEQ_c-*nega-delay*). Due to general query of positive OPSM with condition keywords based on *pIndex* index (GEQ_c-*posi*) is nearly same with EQ_c-*pIndex*, so we do not give the results of the GEQ_c-*posi* method. When testing the GEQ_c-*posi-delay* and GEQ_c-*nega-delay*, we use $d = 1$ as the step length of delayed time.

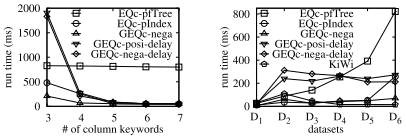


FIGURE 6. Performance of GEQ_c on a single machine.

First, we give the search times of all the query methods when varying the numbers of column keywords. Note that the indexing dataset is D_6, which has 10*k* rows. The response time is shown in *Fig.* 6(a). When the numbers of column keywords increase from 3 to 7, the search times of EQ_c-*pfTree* decrease from 830*ms* to 800*ms*, the response times of EQ_c-*pIndex* reduce from 480*ms* to 45*ms*, that of GEQ_c-*nega* drop from 210*ms* to 45*ms*, that of GEQ_c-*posi-delay* decrease from 270*ms* to 50*ms* (except the condition 3, that is 1900*ms*), that of GEQ_c-*nega-delay* reduce from 210*ms* to 50*ms* (except the condition 3, that is 1800*ms*). Because the keyword number of GEQ_c-*posi-delay* and GEQ_c-*nega-delay* is less than other methods, which leads more candidates, it will spend more time. However, this situation becomes less obvious with the increasing of column keyword number. This experiment demonstrates that the behaviours of the methods based on *pIndex* is better than that based on *pfTree*.

Next, we give the search times of all the query methods on 6 different datasets, when conduct the same query (4 column keywords, except *KiWi* [12] method). Note that the numbers of 6 different datasets are 10*k*. The response time is shown in *Fig.* 6(b). With the increasing of column keyword number, the response time of EQ_c-*pfTree* query method increases exponentially, while the other four methods remain in a certain time range. The details are as follows: the response time of EQ_c-*pfTree* query method increased rapidly from about 25*ms* to about 820*ms*, the query response time of EQ_c-*pIndex* and GEQ_c-*nega* methods is basically the same (from about 10*ms* to about 70*ms*. On dataset D_6, the response time of the former is slightly higher, about 260*ms*.), the query response time of GEQ_c-*posi-delay* and GEQ_c-*nega-delay* methods is basically the same, and slightly higher than that of EQ_c-*pIndex* and GEQ_c-*nega* methods, which gradually increases from about 20*ms* to 220*ms*. Because *KiWi* approach is a method of mining OPSMs from gene expression data in batches, there is no list of keyword options, so we adopt its default setting. *KiWi* runs the longest time on D_2

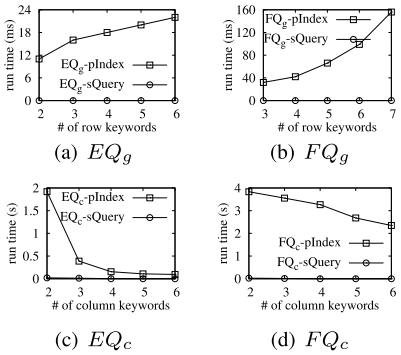


FIGURE 7. Sharing query performance on a single machine.

datasets (109 ms), shorter on D_3 and D_1 datasets (47*ms* and 31*ms*, respectively), and shortest on D_4, D_5 and D_6 datasets (about 16*ms*). The experiment proves that the methods proposed in this paper have better behaviours on efficiency and scalability.

4) EQ/FQ WITH ONLINE-SHARING TECHNIQUE VS EQ/FQ WITHOUT ONLINE-SHARING TECHNIQUE

This subsection evaluates the performances of two class of methods, query based on online-sharing technique and query without online-sharing technique, on exact or fuzzy query. We call these methods *EQ$_g$-sQuery*, *EQ$_g$-pIndex*, *FQ$_g$-sQuery*, *FQ$_g$-pIndex*, *EQ$_c$-sQuery*, *EQ$_c$-pIndex*, *FQ$_c$-sQuery*, *FQ$_c$-pIndex* for short. To show the results more clearly, we assume that the memory is large enough to test the optimal performance of the proposed methods. Thus, it can cache all the repeated and time-consuming results.

Firstly, we compare the behaviours of *EQ$_g$-sQuery* and *EQ$_g$-pIndex*. As shown in *Fig.* 7(a), we vary the numbers of row keywords from 2 to 6, the run time of *EQ$_g$-pIndex* increases from 11*ms* to 22*ms*, but that of *EQ$_g$-sQuery* is nearly one horizontal line, it spends about 0*ms*.

Secondly, we test the behaviours of *FQ$_g$-sQuery* and *FQ$_g$-pIndex*. As shown in *Fig.* 7(b), we vary the numbers of row keywords from 3 to 7, the run time of *FQ$_g$-pIndex* increases from 32*ms* to 156*ms*, but that of *FQ$_g$-sQuery* is nearly one horizontal line, it also spends about 0*ms*.

Thirdly, we evaluate the behaviours of *EQ$_c$-sQuery* and *EQ$_c$-pIndex*. As shown in *Fig.* 7(c), we vary the numbers of column keywords from 2 to 6, the run time of *EQ$_c$-pIndex* decreases from 1919*ms* to 95*ms*, but that of *EQ$_c$-sQuery* drops from 19*ms* to 0*ms*, which is nearly one horizontal line.

Finally, we give the behaviours of *FQ$_c$-sQuery* and *FQ$_c$-pIndex*. As shown in *Fig.* 7(d), we vary the numbers of column keywords from 2 to 6, the run time of *FQ$_c$-pIndex* drops from 3835*ms* to 2348*ms*, but that of *FQ$_c$-sQuery* decreases from 27*ms* to 0*ms*, which also is nearly one horizontal line.

B. EVALUATION ON SINGLE MACHINE, HADOOP AND HAMA

In the next experiments, we take *pIndex* method as an example to show its performance on single machine (SM for short), Hadoop and Hama (2 nodes) platforms. Because the performance of *pIndex* on a single machine has been given in detail, this section only shows its performance in index creation, exact query based on column keywords (EQ_c), fuzzy query based on column keywords (FQ_c), etc.

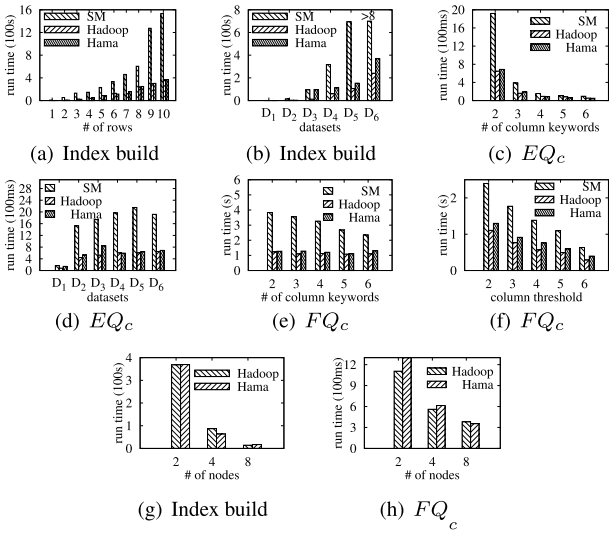


FIGURE 8. Performance on SM, Hadoop and Hama.

Fig. 8(a) and (b) show the time taken by *pIndex* building on varying rows and columns, respectively. As shown in the two figures, the performance of *pIndex* on Hadoop and Hama platforms is 2 to 6 times of that on SM platforms, no matter whether the row number changes or the column number changes. In *Fig.* 8(b), the performance of *pIndex* on Hadoop is significantly better than that on Hama. The reason is that the data has the chance to be skewed on Hama platform, and Hadoop can use its own small file mechanism to solve this problem.

Fig. 8(c) and (d) show the time consumption of exact query based on column keywords (EQ_c) under different number of keywords and different data. When the index data is D_6 and the number of column keywords increases from 2 to 6, the performance of column keyword based exact query (EQ_c) on Hadoop and Hams platforms is more than twice that of single machine (SM). When testing its performance of executing the same query on six different kinds of data, the performance of column keyword based exact query (EQ_c) on Hadoop and Hama platforms is also two to three times of that on a single machine.

Fig. 8(e) and (f) show the scalability of fuzzy query based on column keywords (FQ_c) on three platforms. When the index data is D_6 and the number of column keywords increases from 2 to 6, as shown in *Fig.* 8(e), the performance of column keyword based fuzzy query (FQ_c) on Hadoop and Hama platforms has the same superior performance, which is 2 to 3 times of that on a single machine (SM). Next, we continue to test the performance of FQ_c, when only the threshold of the column keywords is changing, on dataset D_6 and executing the same fuzzy query based on six column

keywords. As shown in *Fig.* 8(f), similar to the former experiment, the performance on Hadoop and Hama platforms has the same superior performance, both of which are two to three times of the performance on a single machine.

Finally, the scalability of index and query methods based on *pIndex* is evaluated under different cluster nodes. As shown in *Fig.* 8(g), when the number of cluster nodes increases from 2 to 8, the creation time of index in the case of 4 (or 8) nodes is one forth times of that in the case of 2 (or 4) nodes, and Hadoop and Hama platforms show almost the same excellent performance. Similarly, as shown in *Fig.* 8(h), when it performs a fuzzy query of six column keywords (FQ_c), its query performance in the case of 4 (or 8) nodes is twice that in the case of 2 (or 4) nodes. Through this experiment, we know that *pIndex* has good scalability. Although *pIndex* takes more time to create indexes than *pfTree*, it can be solved by using distributed parallel platform.

C. ACCURACY

The motivation of this test is to check weather the proposed methods can find the biggest OPSMs based on row/column keywords. We test the search performance on two real datasets [26]. And the evaluation criteria is defined in *Definition* 9. One real dataset is *Arabidopsis Thaliana* dataset, the other one is *Saccharomyces Cerevisiae* dataset. On the former one dataset, we conduct 14 exact queries based on genes. Based on the criteria, we get the recall ratios of conditions, which is shown in *Table* 6. The biggest ratio is 86.7% for the 3×15 size of OPSM, i.e., it gets 13 conditions from 15 ones. The smallest recall ratios is 20.0%. From the 14 searches, the recall rate of 3 queries is more than 80.0%, that of 7 queries is more than 70.0%, and that of of 10 queries is more than 50.0%. On the latter one dataset, we conduct 12 exact queries based on genes and get the recall ratios of conditions, which is shown in *Table* 7. Based on the criteria, we get the recall ratios of conditions. The biggest ratio is 94.1% for the 6×17 size of OPSM, i.e., it gets 16 conditions from 17 ones. The smallest recall ratios is 10.0%. From the 12 searches, the recall rate of 3 queries is more than 70.0%, that of 4 queries is more than 40.0%.

TABLE 6. Search on *Arabidopsis Thaliana* dataset.

True result size	Search result size	Recall ratio of conditions
5×13	5×11	84.6%
6×12	6×9	75.0%
9×11	9×8	72.7%
12×10	12×7	70.0%
14×9	14×5	55.6%
21×8	21×3	37.5%
33×7	33×4	57.1%
55×6	55×3	50.0%
97×5	97×1	20.0%
172×4	172×1	25.0%
302×3	302×1	33.3%
3×16	3×13	81.3%
3×15	3×13	86.7%
4×14	3×10	71.4%

TABLE 7. Search on *Saccharomyces Cerevisiae* dataset.

True result size	Search result size	Recall ratio of conditions
7×16	7×14	87.5%
11×15	11×7	46.7%
23×14	23×3	21.4%
41×13	41×2	15.4%
55×12	55×3	25.0%
73×11	73×3	27.3%
118×10	118×1	10.0%
164×9	164×2	22.2%
258×8	258×1	12.5%
387×7	387×1	14.3%
4×18	4×13	72.2%
6×17	6×16	94.1%

VII. RELATED WORK

Query-based biclustering method comes from the field of bioinformatics, and its application object is gene expression data [27]. Firstly, users provide functional related or co-expressed seed genes based on experience, and then use the seed to guide the search space pruning or biclustering mining search. Jiang et al. [19] design an interactive and visual tool called Gene Pattern eXplorer (GPX) for gene pattern mining, which can drill down and roll up. And it facilitates the OPSM search from the massive results. Dhollander *et al.* [28] propose a Bayesian query-driven biclustering framework named QDB, to answer the specific question of interest to a biologist. They recruit genes with similar expression profiles as the seeds to find a significant subset of experimental conditions. Zhao *et al.* [29] improve the QDB framework, and develop ProBic, a query-based biclustering strategy based on probabilistic relational models to extract high quality biclusters even in the presence of noise or when dealing with low quality seed sets. To answer specific questions of interest and incorporate prior knowledge and expertise from the user, Alqadah *et al.* [30] introduce a novel Query Based Bi-Clustering algorithm, QBBC, using a new formulation that combines the advantages of low-variance biclustering techniques and Formal Concept Analysis. Wang *et al.* [31] apply the longest common subsequence (LCS) framework to selected pairs of rows in an index matrix derived from an input data matrix to locate a seed for each bicluster to be identified. Li and Su [32] introduce a biclustering algorithm called BicGO to recognize complicated biclusters submerged in large scale datasets (matrix). Jiang *et al.* [33] proposes two constrained OPSM query methods, which exploit user defined constraints (must-link, cannot-link, interval, count) to search relevant OPSMs from two kinds of indexes introduced. Further, to reduce the index size of the methods in [33] and obtain much more accurate query relevancy, they proposes two types of OPSM indexing and constrained query methods based on Signature and Trie in [34]. Jiang *et al.* [35] presents and implements a prototype system for OPSM queries, which is called OMEGA (Order-preserving sub-Matrix mining, indExinG and seArch tool for biologists).

It uses Butterfly Network based BSP model to mine OPSMs in parallel [24]. Further, it builds index based on prefix-tree associated with two header tables for gene expression data or OPSM mining results. Then, it processes exact and fuzzy queries based on keywords [36]. The main difference of our contribution with respect to (w.r.t.) our previous work [36] is that here we support queries of multiple types of OPSMs w.r.t. positive OPSM queries. The second difference is that our current proposal ensures, when needed, online sharing queries for the repeated and frequent searches. The third difference is that we rewrite the paper, use much more examples to explain the approaches, add some experiments, and prove and guarantee the validity of the proposed methods in theory. In this study, we study the problems of OPSM queries. Due to its challenging nature, few previous work has been done on this topic.

VIII. CONCLUSION

To tackle the problems that Order-Preserving SubMatrix mining methods have longer latency and do not support many kinds of OPSM mining, the paper designs a new method from the perspective of data management. We present an indexing method called *pIndex*, which uses two header tables to accelerate the search behaviours. Further, we give some query methods, such as exact/fuzzy query on rows/columns, which support the search of positive, negative, and time-delay OPSMs. To reduce the cost of the repeated and frequent queries, we explore the online sharing query strategy for the follow-up OPSM queries. Experiments demonstrate that these techniques are efficient to speed up the query process and can give accuracy results. Note that, although our experiments are run on gene expression data, the method proposed in this paper can be utilized widely in many application domains beyond gene expression data analysis.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their thoughtful comments, which helped to improve the quality of the article. This article was presented in part at the 20th International Conference on Database Systems for Advanced Applications (DASFAA 2015).

REFERENCES

[1] S. C. Madeira and A. L. Oliveira, "Biclustering algorithms for biological data analysis: A survey," *IEEE/ACM Trans. Comput. Biol. Bioinformat.*, vol. 1, no. 1, pp. 24–45, Jan./Mar. 2004.

[2] L. Parsons, E. Haque, and H. Liu, "Subspace clustering for high dimensional data: A review," *ACM SIGKDD Explor. Newslett.*, vol. 6, no. 1, pp. 90–105, Jun. 2004.

[3] S. Busygin, O. Prokopyev, and P. M. Pardalos, "Biclustering in data mining," *Comput. Oper. Res.*, vol. 35, no. 9, pp. 2964–2987, 2008.

[4] F. Yue, L. Sun, K. Wang, Y. Wang, and W. Zuo, "State-of-the-art of cluster analysis of gene expression data," *Acta Automatica Sinica*, vol. 34, no. 2, pp. 113–120, 2008.

[5] H. P. Kriegel, P. Kroger, and A. Zimek, "Clustering of high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering," *ACM Trans. Knowl. Discovery Data*, vol. 3, no. 1, pp. 1–58, 2009.

[6] K. Sim, V. Gopalkrishnan, A. Zimek, and G. Cong, "A survey on enhanced subspace clustering," *Data Mining Knowl. Discovery*, vol. 26, no. 2, pp. 332–397, 2013.

[7] T. Jiang and Z. Li, "A survey on local pattern mining in gene expression data," *J. Comput. Res. Develop.*, vol. 55, no. 11, pp. 2343–2360, 2018.

[8] J. A. Hartigan, "Direct clustering of a data matrix," *J. Amer. Statist. Assoc.*, vol. 67, no. 337, pp. 123–129, 1972.

[9] Y. Cheng and G. M. Church, "Biclustering of expression data," in *Proc. 8th Int. Conf. Intell. Syst. Mol. Biol.(ISMB)*, Edmonton, AB, Canada, 2000, pp. 93–103.

[10] A. Ben-Dor, B. Chor, R. M. Karp, and Z. Yakhini, "Discovering local structure in gene expression data: The order-preserving submatrix problem," in *Proc. 6th Annu. Int. Conf. Comput. Biol. (RECOMB)*, Washington, DC, USA, 2002, pp. 49–57.

[11] L. Ji and K. L. Tan, "Mining gene expression data for positive and negative co-regulated gene clusters," *Bioinformatics*, vol. 20, no. 16, pp. 2711–2718, 2004.

[12] B. J. Gao, O. L. Griffith, M. Ester, H. Xiong, Q. Zhao, and S. J. M. Jones, "On the deep order-preserving submatrix problem: A best effort approach," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 2, pp. 309–325, Feb. 2012.

[13] Q. Fang, W. Ng, J. Feng, and Y. Li, "Mining order-preserving submatrices from probabilistic matrices," *ACM Trans. Database Syst.*, vol. 39, no. 1, 2014, Art. no. 6.

[14] Y. Xue, Z. Liao, M. Li, J. Luo, Q. Kuang, X. Hu, and T. Li, "A new approach for mining order-preserving submatrices based on all common subsequences," *Comput. Math. Methods Med.*, vol. 2015, May 2015, Art. no. 680434.

[15] Y. Yin, Y. Zhao, B. Zhang, and G. Wang, "Mining synchronous and asynchronous co-regulated gene clusters from time series microarray data," *Chin. J. Comput.*, vol. 30, no. 8, pp. 1302–1314, 2007.

[16] Y. Zhao, G. Wang, Y. Yin, and G. Xu, "A novel approach to revealing positive and negative co-regulated genes," *J. Comput. Sci. Technol.*, vol. 22, no. 2, pp. 261–272, 2007.

[17] Y. Zhao, J. X. Yu, G. Wang, L. Chen, B. Wang, and G. Yu, "Maximal Subspace Coregulated Gene Clustering," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 1, pp. 83–98, Jan. 2008.

[18] G. Wang, L. Yin, Y. Zhao, and K. Mao, "Efficiently mining time-delayed gene expression patterns," *IEEE Trans. Syst., Man, Cybern. B. Cybern.*, vol. 40, no. 2, pp. 400–411, Apr. 2010.

[19] D. Jiang, J. Pei, and A. Zhang, "GPX: Interactive mining of gene expression data," in *Proc. 30th Int. Conf. Very Large Data Bases (VLDB)*, Toronto, ON, Canada, 2004, pp. 1249–1252.

[20] S. Barkow, S. Bleuler, A. Prelić, P. Zimmermann, and E. Zitzler, "BicAT: A biclustering analysis toolbox," *Bioinformatics*, vol. 22, no. 10, pp. 1282–1283, 2006.

[21] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, Jul. 1977.

[22] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[23] L. Bergroth, H. Hakonen, and T. L. Raita, "A survey of longest common subsequence algorithms," in *Proc. 7th Int. Symp. String Process. Inf. Retr. (SPIRE)*, A Curuna, Spain, Sep. 2000, pp. 39–48.

[24] T. Jiang, Z. Li, Q. Chen, Z. Wang, W. Pan, and Z. Wang, "Parallel partitioning and mining gene expression data with butterfly network," in *Proc. 24th Int. Conf. Database Expert Syst. Appl. (DEXA)*, Prague, Czech Republic, 2013, pp. 129–144.

[25] *BroadInstitute: Datasets.Rar and 5Q Gct File.Gct*. Accessed: Mar. 20, 2019. [Online]. Available: http://www.broadinstitute.org/cgi-bin/cancer/datasets.cgi

[26] *Arabidopsis Thaliana and Saccharomyces Cerevisiae*. Accessed: Oct. 21, 2019. [Online]. Available: https://sop.tik.ee.ethz.ch/bimax/

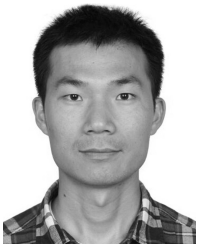
[27] Q. Zou, X. B. Li, W.-R. Jiang, Z.-Y. Lin, G.-L. Li, and K. Chen, "Survey of MapReduce frame operation in bioinformatics," *Briefings Bioinformat.*, vol. 15, no. 4, pp. 637–647, Jul. 2014.

[28] T. Dhollander, Q. Z. Sheng, K. Lemmens, B. D. Moor, K. Marchal, and Y. Moreau, "Query-driven module discovery in microarray data," *Bioinformatics*, vol. 23, no. 19, pp. 2573–2580, 2007.

[29] H. Zhao, L. Cloots, T. V. Bulcke, Y. Wu, R. D. Smet, V. Storms, P. Meysman, K. Engelen, and K. Marchal, "Query-based biclustering of gene expression data using probabilistic relational models," *BMC Bioinf.*, vol. 12, Dec. 2011, Art. no. S37.

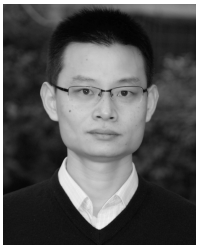
[30] F. Alqadah, J. S. Bader, R. Anand, and C. K. Reddy, "Query-based biclustering using formal concept analysis," in *Proc. SIAM Int. Conf. Data Mining (SDM)*, Anaheim, CA, USA, 2012, pp. 648–659.
[31] Z. Wang, G. Li, R. W. Robinson, and X. Huang, "UniBic: Sequential row-based biclustering algorithm for analysis of gene expression data," *Sci. Rep.*, vol. 6, no. 1, 2016, Art. no. 23466.
[32] G. Li and Z. Su, "BicGO: A new biclustering algorithm based on global optimization," unpublished.
[33] T. Jiang, Z. Li, X. Shang, B. Chen, W. Li, and Z. Yin, "Constrained Query of order-preserving submatrix in gene expression data," *Frontiers Comput. Sci.*, vol. 10, no. 6, pp. 1052–1066, 2016.
[34] T. Jiang, Z. Li, X. Shang, B. Chen, W. Li, and Z. Yin, "Constrained Query of order-preserving submatrix based on signature and trie," *J. Softw.*, vol. 28, no. 8, pp. 2175–2195, 2017.
[35] T. Jiang, Z. Li, Q. Chen, Z. Wang, K. Li, and W. Pan, "OMEGA: An order-preserving submatrix mining, indexing and search," in *Proc. Eur. Conf. Mach. Learn. Princ. Pract. Knowl. Discovery Databases (ECML/PKDD)*, Porto, Portugal, 2015, pp. 303–307.
[36] T. Jiang, Z. Li, Q. Chen, K. Li, Z. Wang, and W. Pan, "Towards order-preserving submatrix search and indexing," in *Proc. 20th Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, Hanoi, Vietnam, 2015, pp. 309–326.

TAO JIANG received the Ph.D. degree from Northwestern Polytechnical University, Xi'an, China. He is currently a Lecturer with the School of Computer and Information Engineering, Henan University of Economics and Law, Zhengzhou, China. His interests include biological data mining, big data management, and information retrieval.



BOLIN CHEN received the Ph.D. degree from the University of Saskatchewan, Saskatoon, Canada. He is currently an Associate Professor with the School of Computer Science, Northwestern Polytechnical University, China. His current research interests include bioinformatics, computational and systems biology, data mining, and data management.

JUNTAO LI received the B.S. and M.S. degrees in applied mathematics from Henan Normal University, China, in 2001 and 2004, respectively, and the Ph.D. degree in control theory and control engineering from Beihang University, China, in 2010. Since 2011, he has been an Associate Professor with Henan Normal University. His research interests include statistical machine learning, bioinformatics, and complex system modeling.



GUOYU XU received the Ph.D. degree from PLA Information Engineering University, Zhengzhou, China. He is currently a Lecturer with the School of Computer and Information Engineering, Henan University of Economics and Law, Zhengzhou. His interests include network information security and cryptography.