PIDS: Attribute Decomposition for Improved Compression and Query Performance in Columnar Storage

Hao Jiang
University of Chicago
hajiang@cs.uchicago.edu

Chunwei Liu
University of Chicago
chunwei@cs.uchicago.edu

Qi Jin
University of Chicago
qijin@uchicago.edu

John Paparrizos
University of Chicago
jopa@cs.uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

ABSTRACT

We propose PIDS, Pattern Inference Decomposed Storage, an innovative storage method for decomposing string attributes in columnar stores. Using an unsupervised approach, PIDS identifies common patterns in string attributes from relational databases, and uses the discovered pattern to split each attribute into sub-attributes. First, by storing and encoding each sub-attribute individually, PIDS can achieve a compression ratio comparable to Snappy and Gzip. Second, by decomposing the attribute, PIDS can push down many query operators to sub-attributes, thereby minimizing I/O and potentially expensive comparison operations, resulting in the faster execution of query operators.

PVLDB Reference Format:
Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, Aaron J. Elmore. PIDS: Attribute Decomposition for Improved Compression and Query Performance in Columnar Storage. *PVLDB*, 13(6): 925-938, 2020.
DOI: https://doi.org/10.14778/3380750.3380761

1. INTRODUCTION

Due to effective compression for minimizing storage and query latency, columnar systems are critical for modern data-intensive applications that rely on vast amounts of data generated by servers, applications, smartphones, cars, and billions of Internet of Things (IoT) devices. By persisting the same attribute from different records consecutively, columnar systems enable fast scan operations by minimizing I/O and efficient compression by keeping similar data physically close. Since compressed data must be decoded each time a query is executed, many columnar stores [4, 2, 3] allow for lightweight encoding schemes over traditional byte-oriented compression algorithms, such as Gzip and Snappy [1]. They provide significant size reduction at the cost of high overhead for decoding before reading data. Lightweight encoding is a family of compression algorithms applicable to data streams of the same type, and has less compression/decompression overhead [1] and *in situ* data filtering without decoding [15], potentially at the cost of reduced compression.

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380761

MIR−00880−33BB1−512
MIR−00C80−33FB1−512
MIR−00000−337F1−4096
MIR−040C0−373F1−512
MIR−00000−337F1−4096
MIR−00C00−33F71−4096

(a) Machine partition

138A211	162
180B161	1126
120B181	771
228B149	550
177B153	362
183B109	507

(b) Ref ID

0101000020E61000000CFD083315C852C0F070116B33054440
0101000020E6100000AC88531328C352C028556E1E1E094440
0101000020E610000010A23DDD87C752COE0159AEE6C034440
0101000020E6100000CC490E29FDCE52C0289F52833B094440
0101000020E61000006495B3267FC752C048F580C01D004440
0101000020E61000004896C0D141C752C008D6E7CO6BFF4340
0101000020E6100000F426379E30CB52COCCO6FC67A34F64340
0101000020E61000005C0170DFEFBE52COA0C125BBAA094440

(c) Log ID

Figure 1: Sample attributes with identifiable patterns.

In evaluating popular lightweight encoding algorithms and Gzip on a large corpus of string attributes, we observe that even with the best lightweight encoding applied, Gzip can still further compress the encoded data on a large number of attributes. By examining these attributes, we observe that many of these attributes contain repetitive substrings across values. An example is shown in Figure 1a, where all rows contain "MIR"; "33F71" and "4096" also occur multiple times. These substrings are captured by Gzip, but not by lightweight encoding as compression is applied to the entire attribute value. This difference leads to the performance gap in the compression ratio between the two approaches.

We see that in many such attributes, these substring repetitions can be captured by a simple pattern. In Figure 1, we show excerpts from three attributes. It is obvious that `Machine Partition` follows a pattern MIR-{hex(5)}-{hex(5)}-{int}, where hex(x) represents any hexadecimal number of x digits, and `Ref ID` follows pattern {hex(7)}{int}. The pattern for column `Log ID` is less obvious, but a closer look shows that all records have the same length, a common header consisting of a 17-digit hexadecimal number, "52C0" in the middle, and a common tail "40". We can use these patterns to split string attributes into smaller components that we call *sub-attributes*. By extracting the sub-attributes and encoding them, we can potentially close the gap between lightweight encoding and Gzip.

In this paper, we propose an innovative storage method, ***P**attern **I**nference **D**ecomposed **S**torage (PIDS)* to exploit

patterns in string attributes to improve compression and query performance. PIDS employs an unsupervised algorithm to infer a *pattern* automatically from an input attribute, if applicable, stores rows that do not match the pattern as *outliers*, extracts *sub-attributes* from the matched rows using the pattern, and compresses them independently. PIDS is transparent to the user. While the sub-attributes are physically stored separately, PIDS provides a logical view that is identical to the original string attributes. PIDS rewrites query operations on the logical view to operate on sub-attributes to speed up execution.

The pattern inference algorithm in PIDS works by collecting a set of samples from the input attribute and uses a Programming-By-Example approach to extract patterns. Besides lexical similarities, such as common symbols, it also captures the semantic similarity within string attributes, allowing observing more hidden patterns. The inference algorithm provides a classifier recognizing whether or not a string attribute contains a valid pattern, which helps preclude them from the potential costly inference. PIDS also provides an intermediate language to describe the pattern, allowing it to quickly adapt to other inference algorithms or use patterns provided by the end-user.

PIDS enables more efficient compression in two ways. Exploring patterns from string attributes allows common substrings to be eliminated, such as removing `MIR-` from every instance in Figure 1a. Additionally, the extracted sub-attributes and outliers are stored as physically separated columns, on which different encoding schemes can be applied to improve compression efficiency. PIDS thus can provide a compression ratio that is comparable to Gzip, while supporting efficient encoding and decoding operations that are comparable to lightweight encoding. We empirically evaluate PIDS on a extensive collection of string attributes, showing that a large portion of them contain a valid pattern and get a compression benefit from PIDS.

PIDS uses patterns to gain insights on the attribute and facilitates efficient execution of common query operators (including equality, less and wildcard search predicates), and materialization. Given a predicate `"Machine Partition"= "ABC"`, we know immediately that it does not match any data that follows the pattern MIR-{hex(5)}-{hex(5)}-{int}. PIDS also enables a query framework to "push down" the predicates to the sub-attribute level, and potentially skip data not matching the criteria to save disk I/O and decoding effort. This is especially beneficial for wildcard queries. For example, knowing that the first sub-attribute of machine partition is a 5-digit hexadecimal, we can push down the predicate `"Machine Partition" = "MIR-00880%"` to its sub-attributes, and get an equivalent query *sub_attr*_1 = `00880`. Compared to the original query, which performs a wildcard match on the entire string, the new query only needs to execute an equality check on one numeric sub-attribute, saving both I/O and computation effort. This brings up to 30x performance boost for operator execution.

The contribution of PIDS includes:

- An algorithm for discovering common patterns in string type columnar datasets.
- An intermediate language, PIDS IR, for pattern description and compiling efficient ad-hoc code for sub-attribute extraction and predicate execution.
- A PIDS prototype based on the Apache Parquet format supporting the automatic inference of column pattern, sub-attribute extraction, and query operators on the sub-attributes.

In the rest of the paper, we discuss related work (Sec. 2), present an overview of PIDS (Sec. 3), discuss the inference algorithm (Sec. 4), describe sub-attribute extraction and compression in PIDS (Sec. 5), explain how query operators work with PIDS (Sec. 6), and present experiments on our PIDS prototype (Sec. 7).

2. RELATED WORK

In this section, we discuss relevant background from two areas, namely, data extraction and data compression.

Data Extraction: In general, inferring patterns from samples is a program synthesis technique known as Programming-by-Example (PBE). A PBE algorithm automatically analyzes existing examples and generates programs that can be applied to new examples. PBE has many applications in data processing tasks that involve large amounts of input data with indeterminate formats, such as in structured data extraction [5, 22, 10, 21, 27, 11], table transformation [13, 6, 16], and entity augmentation [29, 31].

In PBE tasks, users often provide both input and output example pairs [21, 6, 16]. By treating input and output examples as states in a search space and by defining data transformation operations as transitions between the states, such a problem can efficiently be converted to a search problem. As the number of available states is usually exponentially large, pruning techniques [21] and heuristics [16] are usually employed to facilitate the search process.

For a large input dataset, as is in our case, it is often impractical for a user to provide output examples corresponding to each input. The algorithms introduced by PADS [10] and Datamaran [11] both follow a search-rank-prune pattern process to automate pattern inference. Similarly to the input-output example case, input examples are mapped to a search space as the source state and transitions are defined between states. However, instead of searching for a given target, the algorithm computes all reachable states from the source, ranks them with a custom scoring function, and prunes the states with the lowest scores. This process is repeated until a reasonably good target state is discovered.

Unfortunately, such search-rank-prune processes are often time-consuming due to a large number of potential states. Instead of searching all possible states, in PIDS, we use a greedy search approach where, with the use of a heuristic function, we evaluate all possible transitions from the current state, choose the transition with the maximum gain, and ignore the rest. By carefully designing the transition rules and heuristic function, our approach becomes very efficient while achieving good accuracy performance. While previous algorithms extract structures from ad-hoc unstructured log data, they have to make many assumptions on the input data. Instead, PIDS targets data from the same attribute in a relational database. Besides, PIDS employs a classifier to filter out input data that is unlikely to contain a structure. This allows PIDS to make fewer assumptions on the input data and extract structures that are omitted by the previous methods.

Table 1 compares the assumptions made by PADS, Datamaran, and PIDS. We briefly introduce the assumptions here; the detailed definitions can be found in Datamaran [11]. *Coverage threshold* assumes that the generated pattern matches at least a certain percentage of samples. PIDS does not

Table 1: Assumptions made by Extraction Algorithms.

Assumption	PADS	Datamaran	PIDS
Coverage Threshold	✘	✔	✘
Non-overlapping	✔	✔	✘
Structural Form	✔	✔	✔
Boundary	✔	✘	✔
Tokenization	✔	✘	✘

force a coverage threshold as it will generate a pattern that always covers the entire sample set. *Non-overlapping* assumes that the alphabets used in the pattern and field values are not overlapping. In other words, a character c is either part of a pattern or part of the extracted data, but not both. In PIDS, any word can be either part of the pattern or the extracted data. *Structural Form* assumes that the pattern is a tree-like structure with each node as a sub-pattern. This assumption is shared by all approaches. *Boundary* assumes that the boundary of records can be easily identified beforehand. As PIDS processes input data from an attribute, the input is well-bounded. *Tokenization* assumes that each character can be tokenized as a part of the pattern or the extracted data beforehand. PIDS does not make this assumption and determines the role of each character during inference. In Section 7, we empirically compare PIDS against Datamaran.

Compression and Encoding: Lightweight encoding, such as dictionary encoding, and LZ77 [32] are two popular families of compression algorithms adopted in columnar stores. Lightweight encoding features high throughput and *in situ* data filtering. LZ77 features higher compression ratios but requires decompression before evaluation. By extracting sub-attributes and applying a lightweight encoding, we achieve a compression ratio comparable to LZ77 compression, while preserving all the benefits of lightweight encoding.

Popular byte-level compression techniques, such as Gzip and Snappy, both belong to the LZ77 family, which utilizes a sliding window on an input data stream, looking for strings that contain a recurring prefix, and encodes each string as a reference to the previous occurrence. Gzip applies Huffman encoding to the reference stream for a better compression ratio [9] and Snappy skips that step for higher throughput. LZ77 [32] has a theoretical guarantee for a good compression ratio. However, it is a sequential algorithm and needs to decompress an entire data block before the original data can be accessed. This brings a high latency for accessing the compressed data.

Lightweight encoding is a family of entry-level compression techniques. It transforms each data entry (an integer, a line of text, etc.) in the input to a shorter representation. Popular lightweight encoding schemes include dictionary encoding, bit-packed encoding, delta encoding, run-length encoding, and their hybrids [28, 7, 33, 18]. Lightweight encoding has very low CPU consumption as the operations are usually simple and can often be performed in parallel [15]. Lightweight encoding maintains entry boundaries during compression, allowing access of compressed entries without decoding the entire data block, and direct predicate execution on compressed data, skipping decompression [15].

Another related research area is string dictionary compression [24, 19]. Research here applies dictionary encoding to the data, then compresses the dictionary entries using methods such as prefix-coding and delta encoding. Such approaches also address repetitions of substrings in a string attribute, but they help little for improving query execution. Additionally, these methods rely on sorting the string dictionary for prefix-coding. In Section 7, we compare PIDS with BRPFC [19], a state-of-the-art technique in this area.

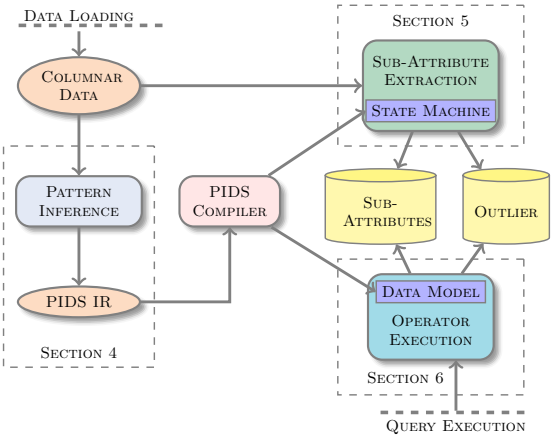


Figure 2: PIDS System Architecture.

Concurrent with our work, B. Ghita et al. [12] propose a vision similar to PIDS: white-box compression. Here, the authors propose learning patterns to split or merge attributes, and apply compression on the new columns. Our work differs in an emphasis on fast extraction, a simple grammar-based pattern inference system, and a tight coupling and evaluation of query operators.

3. OVERVIEW

In this section, we describe PIDS by using an example to walk through its components. Figure 2 shows the major components and execution steps in PIDS and their corresponding sections in the paper. The system takes a columnar string dataset as input, sampling data from each column to determine if there exists a common pattern in it, and generates a pattern expressed in PIDS IR, the intermediate representation used by PIDS to describe a pattern. The generated pattern, together with the input attribute, is sent to `Sub-Attribute Extraction`, which splits the data record into sub-attributes. To improve system efficiency, it employs the PIDS compiler to create a state machine for the pattern matching and substring extraction tasks. If PIDS cannot extract a pattern from the input columnar dataset, PIDS treats the data column as a single sub-attribute and uses existing lightweight encoding compression techniques, such as Dictionary Encoding, to compress it.

The extracted sub-attributes are then exported to external storage as separate columns, which are stored and compressed independently. As the pattern is inferred from samples, there will be a chance that some rows are not included in the sample and thus not described by the pattern. PIDS addresses this problem by maintaining an outlier store, which is separate from the sub-attribute columns. Rows that do not match the pattern are considered outliers and are stored in the outlier store in its original string form.

When executing a query operator, PIDS sends the request to `Operator Execution`, which is responsible for pushing down the operator to the sub-attribute columns, including the outlier column. The `Operator Execution` also uses the PIDS compiler to compile PIDS IR into data models and code for data loading.

```
pattern := token | union | seq
token   := const(val) // constant typed literal
        |  int        // int of arbitrary length
        |  hex        // hexadecimal number
        |  rangeint(min, max)  // int with min, max
        |  flint(len) // fixed-length integer
        |  flhex(len) // fixed-length hex integers
        |  str        // string of Unicode letters
        |  flstr(len) // fixed-length string
        |  sym(char)  // non-letter/digit characters
        |  empty      // no character
union   := union pattern
        |  pattern
seq     := seq pattern
        |  pattern
```

Figure 3: PIDS IR Grammar.

4. PATTERN INFERENCE

In this section, we describe the pattern inference used in PIDS, which is a PBE problem with input-only examples. PIDS uses a heuristic-based search algorithm to infer patterns from examples. It treats each pattern as a state and defines a series of transformation rules between patterns. The algorithm starts from the pattern that is the enumeration of all input examples, and searches for patterns that are reachable from the starting state via the transitions. When the search ends, the pattern with the highest heuristic score is the final output.

First, we introduce PIDS IR, the intermediate language used to described a pattern in Section 4.1, then we describe the transition rules in detail in Section 4.2.

4.1 PIDS IR

PIDS IR is a concise language optimized for describing common patterns in string datasets. The grammar of PIDS IR is shown in Figure 3. The basic building blocks of a pattern are tokens, unions, and seqs. The token family includes basic types, such as `const`, `int`, `hex`, `str`, and `sym`. Instead of hard-coding a list of symbols [11], PIDS marks all characters that are not Unicode letters or digits as symbols, to be adaptive to multilingual applications. It also provides two collection types, `union` and `seq`. A `union` represents a set of patterns, of which at least one appears. A `seq` represents a list of patterns that all appear in order. As an example, the pattern shown in Figure 1a can be written in PIDS IR as

```
seq( const('MIR') sym('-') flhex(5) sym('-')
     flhex(5) sym('-') int )
```

Similar IRs used in a previous work [10] often target datasets with more complex structures, such as a text corpus and log data, and provide support for nested data structures (i.e., dictionaries and arrays) and common data types (i.e., dates and timestamps). PIDS chooses not to support nested data structures. As many systems [4, 23] already provide native support for nested data structures, users who need these structures are more likely to directly leverage the native format, instead of packing the structure into a string column. It also does not include these common complex types. If these structures were to appear in the target dataset, they can easily be captured by the inference algorithm and represented in PIDS IR. This minimizes the number of built-in terms in the language, making it more concise. In Section 7 we show this also facilitate efficient queries.

Patterns written in PIDS IR can be easily transformed into other descriptive formats or machine code for pattern processing. For our prototype evaluation, we develop a library to generate regular expressions from PIDS IR and a compiler to directly generate Java bytecode from PIDS IR for query operator evaluation and sub-attribute extraction. The same approach can also be generalized to other languages and platforms, such as translated into LLVM [20].

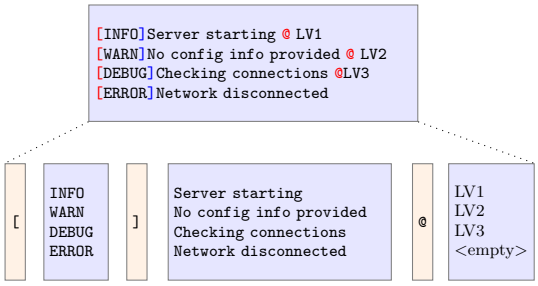


Figure 4: Splitting a union with common symbols.

4.2 Pattern Inference Algorithm

As described above, the inference algorithm works as a heuristic-based search, starting with generating an initial pattern, which is simply a union of all input examples, each as a `const` token, and iteratively applying transition rules to it until no further optimization can be done. We categorize the transition rules into two phases: Splitting and Pruning.

Splitting: In the splitting phase, PIDS performs a depth-first-scan on the pattern tree and looks into each union it encounters, searching for common tokens in union members. PIDS splits the union members into "columns" using these common tokens, and represents each column as a shorter union. This converts the original union into a seq of smaller unions and common tokens. We show an example in Figure 4, where the algorithm discovers "[", "]", and "@" as common tokens, and converts the original union into a seq containing three symbols and three shorter unions. In the splitting phase, PIDS applies three rules iteratively on the union to discover common patterns from its members.

The first rule, `CommonSymbol`, looks for symbols that are non-alphanumeric characters, such as hyphen, brackets, commas, and colons. These symbols commonly serve as separators between different parts of the input [10, 11]. For this reason, PIDS prioritizes the CommonSymbol rule by a "majority-take-all" approach. PIDS recognizes a symbol as a common one as long as it appears in a majority of the members (e.g., 80%). As shown in Figure 4, where the symbol "`@`" is extracted even if the last line does not contain it. An `empty` is left for the member without the symbol.

The second rule, `SameLength`, targets unions that have members with the same number of characters. In practice, it is fairly common that values from the same column have the same length, as in Figure 1c, which can serve as an indicator that some pattern exists for the data. This rule assumes that characters at the same position of each union member are the same type, and learns a character's type from all members. For example, given a union containing three const "ABDEAABD", "PA305402", and "UP25CE38", we can determine that the first two characters are letters and the remaining six characters are hex digits. This union can thus be split by the SameLength rule into two smaller unions.

Finally, if none of the above rules apply, PIDS uses the `CommonSeq` rule to look for common sequences of tokens among `union` members. This rule employs a dynamic pro-

gramming algorithm to look for the longest common subsequence from two sequences. By applying this algorithm to the first two union members, we obtain a list of common subsequences. These subsequences are then compared against the next union member, leaving us with the common subsequences among the first three members. Repeating this process gives us the common subsequences among all union members, which can then be used to split the union.

`CommonSeq` also uses word2vec to recognize similar words in text and uses them as separators to extract patterns. Due to a lack of training corpus, PIDS does not train the word2vec model itself. Instead, it utilizes Glove [26], a pre-trained word2vec model, to convert the words in an input attribute into vectors. PIDS then goes through each record in the attribute, looking for a set of words whose pairwise cosine similarity is greater than a user-defined threshold. For example, when dealing with a string attribute of U.S. postal addresses, PIDS recognizes that the set of words "Road", "Street",and "Avenue" has a large pairwise cosine similarity. PIDS uses these words as separators to split input records.

Pruning: In the pruning step, PIDS cleans up the pattern generated in the splitting phase by removing redundant structures and merging adjacent tree nodes. This step creates a concise tree and allows the next iteration to be executed more efficiently.

PIDS executes three rules to prune the pattern tree. The `Squeeze` rule removes all unnecessary or duplicated structures, such as seqs and unions containing only one member. We list part of the transforming rule below:

```
seq(a) => a
seq(a, empty, b) => seq(a, b)
union(a) => a
```

where "a","b" represent arbitrary patterns.

`MergeAdjacent` searches for adjacent tokens of the same type in a `seq`, and merges them together if possible.

The `Generalize` rule replaces a `union` of `const`s with generalized tokens such as `str` or `int`. For example, `union(213, 42, 442)` can be rewritten as either `int` or `rangeint(42, 442)`. Since `flint`, `rangeint`, and `flstr` contain more information about the underlying data, PIDS priorities them and only falls back to more general `int` and `str` upon failure.

By repeatedly executing these two phases, PIDS gradually constructs a concise pattern from the given samples. We use Figure 1a as an example to show how this works. In the splitting phase, PIDS utilizes the `CommonSymbol` rule to discover hyphens in the records as separators and obtains the following pattern.

```
seq( const('MIR') sym('-') union(00880, 04C80, ...)
     sym('-') union(33BB1,33FB1,...) sym('-')
     union(512,1024,...)  )
```

In the pruning phase, PIDS executes the `Generalize` rule on the unions, converting them into generalized tokens, and obtains the final result.

```
seq( const('MIR') sym('-') flhex(5) sym('-')
     flhex(5) sym('-') int )
```

5. SUB-ATTRIBUTE EXTRACTION AND COMPRESSION

In this section, we present how PIDS extracts and stores sub-attributes. Section 5.1 describes the data extraction algorithm, Section 5.2 introduces how PIDS handles outliers, and Section 5.3 discusses how PIDS stores and compresses data and how it can improve compression efficiency.

5.1 Sub-Attribute Extraction

PIDS uses a state-machine based algorithm to extract sub-attributes from a target attribute. It randomly samples n rows for the target attribute and applies the pattern inference algorithm described in the previous section to generate a pattern from the samples. Let $s = seq(p_1, p_2, \ldots, p_n)$ be the pattern generated from the samples. Each p_i that is not a `const` or `sym` represents an extractable sub-attribute. We construct a directed acyclic deterministic finite state machine (DA-DFS) that describes s as follows.

1. Every token can be represented by a DA-DFS. `const` and `sym` correspond to DA-DFSs that accepting the string literals they hold. `int, hex, str` and their fixed length version correspond to DA-DFSs that takes the required number (can be infinite) of digits or letters.
2. If s_1, s_2 can be represented by DA-DFS $d(s_1)$, and $d(s_2)$, a DA-DFS for $seq(s_1, s_2)$ can be constructed by merging the last state of $d(s_1)$ with the first state of $d(s_2)$.
3. If s_1, s_2 can each be represented by DA-DFS $d(s_1)$, and $d(s_2)$, a DA-DFS for $union(s_1, s_2)$ can be constructed by merging the starting states of $d(s_1)$ and $d(s_2)$, and merging all states reachable from the starting state via the same transition sequence.

PIDS generates a DA-DFS from the pattern and marks the states representing the start/stop of each sub-attribute. When the state machine processes an input string, it will extract the sub-strings corresponding to each sub-attribute when the execution reaches these marked states.

State machines are usually implemented using a 2D array as a lookup table to store the transitions. As a result, each state transition involves several memory access operations. Since the size of the lookup table is the product of the number of states and alphabet size, for large alphabets, the size of the lookup table easily exceeds the L1 cache size as the number of states increases, making state transitions less efficient.

As the transition table is immutable during state machine execution, and the number of transitions from a single state is usually small, most entries in the 2D array table are empty. PIDS implements the state machine efficiently by hard-coding state transitions. It only uses `switch` and `if` statements to implement state transitions and sub-attribute extraction, eliminating memory access to improve efficiency.

We use the PIDS compiler to generate ad-hoc code given a pattern instance. In our prototype developed in Java, the compiler builds the DA-DFS from PIDS IR in memory, then uses the ASM library [8] to generate bytecode equivalent to the state machine. Finally, it loads the generated bytecode into JVM and applies it to the target attribute for sub-attribute extraction.

5.2 Handling Outliers

From Section 4 we know that the pattern generated by the inference algorithm is guaranteed to match all samples. However, some records may exhibit a different pattern that are not included in the samples. These records fail to be matched by the state machine and are called outliers. An

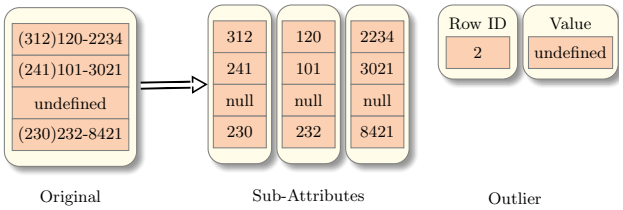


Figure 5: Sub-attributes are ordered as the original records and outliers are stored separately with an explicit row ID.

example is shown in Figure 5, where the dataset contains two types of records, valid phone numbers and a constant value "undefined". PIDS matches the phone numbers, and treats the "undefined" records as outliers.

As outliers cannot be split into sub-attributes, we store them independently from where the sub-attributes are stored. To retain the offset information for each record, which is often used as a join key when columnar stores materialize multiple attributes, we put the records in the sub-attribute table at the same offset as they are in the original table, inserting `null` at locations corresponding to outliers. Note that a null value in the original attribute will be treated as an outlier, and can be easily distinguished from null values that serve as placeholders for outliers. In the outlier store, we note the offset and the value explicitly as two columns.

We make this design decision based on two observed facts:

- The number of outliers should be small compared to the total number of records. PIDS targets datasets that can be described by a single pattern. If the number of outliers exceeds a certain level, then the dataset is not a good fit for PIDS.
- Most operators need to access either the sub-attribute table, or the outlier table, but not both. When an operator matches the pattern, we know it either targets the sub-attribute table or the outlier table. Only a limited number of operators, such as sorting and materialization need to access both tables.

As the number of outliers tends to be small, storing a row ID explicitly for them will save space, while still allowing us to restore outliers to their correct position during materialization. We execute operators that need to access both tables on each table separately, and merge the result. More details on operator execution can be found in Section 6.

5.3 Storage and Compression

Our PIDS prototype utilizes the Apache Parquet [4] storage format for its popularity and flexibility. A Parquet file consists of multiple row groups, which serve as a horizontal split of the columns. A row group contains several column chunks, each containing the data of one column in that row group. Data in a column chunk are stored continuously on disk and can be loaded efficiently with a sequential read. The data in column chunks are organized into pages, with each page as the unit for encoding and compression.

PIDS stores and compresses each sub-attribute independently as a column, using a dictionary and bit-pack-run-length hybrid encoding. It maintains an independent dictionary for each column chunk that translates distinct entries in the target sub-attribute to an integer code, then use run-length and bit-packing to compress the integer codes.

In practice, we notice that there are two types of attributes on which PIDS does not work well. The first type is attributes with no patterns, such as attributes with single word or natural language text. The second type is attributes with low cardinality. If an attribute already has a low cardinality, the extracted sub-attributes will likely to have similar cardinality. For such attributes, directly compressing the original attribute using a single dictionary is more efficient than compressing each sub-attribute using separated dictionaries. PIDS uses a classifier to recognize these attributes and compresses them as a single sub-attribute. To efficiently recognize and exclude these attributes, PIDS employs a k-nn classifier with Euclidean distance that weights nearest neighbors using their inverse squared distance. We use our public dataset as the training set and label attributes as positive if PIDS encodes smaller than the best encoding, and negative otherwise. We evaluate the accuracy of the model using 5-fold cross-validation (fitcknn with kfoldPredict in Matlab) with the following features from a string attribute:

- Ratio of Distinct Values $\frac{\text{cardinality}}{\text{num of records}}$
- Mean and variance of record length
- Mean of Shannon Entropy of each record

6. OPERATOR EXECUTION

To support efficient query execution directly on sub-attributes, we describe how PIDS supports common query operators, including predicate filteringand materialization. Many operators can be "pushed down" to sub-attributes, such that the operator can be decomposed into several independent operators on each sub-attribute, and the results from each sub-attribute can be combined to obtain the final output. Pushing down operators to sub-attributes also enables incremental execution, where we execute operators on the sub-attributes one at a time, skipping records on sub-attributes for which previous executions have determined that an attribute cannot satisfy the operator. For example, when executing equality predicate on a phone number, if the first sub-attribute (area code) does not match the predicate constant on some rows, we know that these rows do not match without examining the other sub-attributes. When scanning the remaining sub-attributes, we can skip these rows, saving I/O and decoding effort, thus speeding up the execution. Query rewriting from string operations (e.g. `like`, $=$, $\neq$, or $<$) to sub-attribute operations is done by PIDS and is transparent to the user. For operators that cannot be pushed down to sub-attributes, PIDS materializes sub-attributes into an in-memory data structure before applying the operator. For exposition we start with an assumption that all sub-attributes are of fixed length. In Section 6.4, we introduce how PIDS handles comparisons on sub-attributes of variable length.

6.1 Efficient Data Skipping

PIDS implements data skipping by using a bitmap to mark the positions of rows that need to be evaluated on the next sub-attribute, and update the bitmap after each sub-attribute evaluation. For our Parquet-based prototype, data skipping occurs at three levels.

- Column Chunk Level. PIDS first consults the bitmap to see if a column chunk contains rows to be accessed, then uses zone map information to determine if a column chunk can be skipped for a given operator. Skipping a column chunk saves disk I/O.
- Page Level. Similar to column chunk level, PIDS uses both bitmap and zone map to perform page-level skipping. Skipping pages saves decompression effort.

- Row Level. When scanning data on a sub-attribute, PIDS uses a bitmap to locate the next row to be read, skipping all rows in between. If the rows are encoded using lightweight encoding, PIDS skips the bytes without decoding them. Skipping rows thus saves decoding effort.

6.2 Predicate Filtering

We define a predicate as a tuple (OP, a), where OP is the operation and a is the constant. For example, (less, 5) on column x evaluates to true for all values where $x < 5$. PIDS implements three relational predicates `equal`, `less`, and `like`. Other predicates, such as `greater-equal`, can be obtained through logical combinations of the existing ones.

All three predicates support pushing down the operators to sub-attributes. When executing a predicate, PIDS first matches the constant against the pattern. If they do not match, execution terminates with no disk access involved. Otherwise, we use the match result to push down the predicate to the sub-attributes. We use x to represent the target column, which consists of sub-attributes $x_1, x_2, \ldots, x_m$, and assume that the constant a of the predicate has a match $(a_1, a_2, \ldots, a_m)$ to the pattern.

6.2.1 Equality Predicate

The equality predicate (equal, a) can be decomposed as $x = a \iff \wedge_{i=1}^{m}(x_i = a_i)$, that is, $x = a$ if and only if the equality $x_i = a_i$ holds for all sub-attributes x_i.

To skip as many rows as possible, PIDS uses a histogram to estimate the selectivity of $x_i = a_i$, and scans x_i in increasing order of selectivity. For example, if $x_1 = a_1$ is expected to match 10% of rows and $x_2 = a_2$ is expected to matches 5% of rows, we first execute $x_2 = a_2$, then $x_1 = a_1$ to allow more rows to be skipped. PIDS uses a bitmap to mark the positions where all sub-attributes x_i scanned so far satisfy $x_i = a_i$ and the equality check on the next sub-attribute is performed only on the marked positions.

When performing an equality check on sub-attributes, PIDS utilizes the encoding dictionary to translate the predicate constant into an integer code and performs the equality check on encoded data directly [15] to save decoding effort. If the constant is not in the dictionary, the entire sub-attribute can be skipped.

6.2.2 Less Predicate

When the sub-attributes are all fixed length, a less predicate $(less, a)$ can be pushed down to sub-attributes as a combination of $x_i < a_i$ and $x_i = a_i$. If $x_1 < a_1$, we have $x < a$. Otherwise if $x_1 = a_1$, we proceed to check x_2. Again, if $x_2 < a_2$, we have $x < a$. Otherwise if $x_2 = a_2$, we proceed to x_3. This process is repeated until all x_i have been processed. Formally, this can be written as

$$x < a \iff \vee_{i=1}^{m}[\wedge_{j=1}^{i-1}(x_j = a_j) \wedge (x_i < a_i)]$$

PIDS maintains two bitmaps, `result` for the positions satisfying $x < a$ so far, and `posToScan` for the positions to scan on next sub-attribute. They are updated as follows in the i-th iteration.

$$\begin{aligned} \texttt{posToScan}_i &= \texttt{posToScan}_{i-1} \wedge (x_i = a_i) \\ \texttt{result}_i &= \texttt{result}_{i-1} \vee (\texttt{posToScan}_{i-1} \wedge (x_i < a_i)) \end{aligned} \quad (1)$$

PIDS iterates through all sub-attributes, computes $x_i = a_i$ and $x_i < a_i$ on rows marked by `posToScan`, and updates the bitmaps using Equation (1). When the iteration ends, `result` stores all records satisfying $x < a$.

6.2.3 Like Predicate

Some like predicates, such as prefix or suffix search, may only need to access a limited set of sub-attributes, which is identifiable by matching the predicate to the pattern. For example, assume we have a pattern for phone numbers as `seq( sym('(') flint(3) sym(')') flint(3) sym('-') flint(4))`. A prefix search (like, '(345)44%') has a unique match on the phone number pattern as (345, 44%, %), and can be pushed down to sub-attributes as $x_1 = 345 \wedge x_2 \sim$ 44%, where we use $\sim$ to denote the like predicate. Similarly, a suffix search (like '%442') has a match (%, %, %442) and can be pushed down as $x_3 \sim$ %442. These predicates are then evaluated using a similar approach as in the equality case. We first sort sub-attributes based on their selectivity and iterate through each x_i, executing the predicates. This allows PIDS to greatly simplify the execution of many prefix and suffix queries as it can directly skip sub-attributes that are not included in the predicates. This approach also applies to some wildcard queries. For example, (like, %4242%) can be pushed down as $x_3 = 4242$ when we discover that only sub-attribute x_3 contains four-digit numbers.

A challenge to this approach is that in some cases the constant containing '%' can have multiple matches on the pattern. For example, "(123)%432%" has two matches against the phone number pattern, (123, 432, %), and (123, %, %432%). They lead to different execution results, and both need to be included in the final result.

PIDS solves this by collecting all possible matches, pushing each of them down to the sub-attributes, and merging and simplifying the generated expression. The example above can be written as follows when pushing down to sub-attributes: $(x_1 = 123 \wedge x_2 = 432 \wedge x_3 \sim \%) \vee (x_1 = 123 \wedge x_2 \sim \% \wedge x_3 \sim$ %432%). This is simplified to $x_1 = 123 \wedge ((x_2 = 432) \vee (x_3 \sim$ %432%)). PIDS first executes $x_1 = 123$, getting a bitmap, and uses that bitmap to skip rows when scanning x_2 and x_3.

6.3 Materialization

PIDS implements two types of materialization operations, string materialization and fast materialization. String materialization reads fields from all sub-attributes, and composes them back to the original string format according to the pattern. This is applied to a column when a projection is performed. However, sometimes we materialize a column not for output, but only to execute operators that cannot be pushed down to sub-attributes, such as joins or hashing for group-by aggregations. In these cases, we only read fields from each sub-attribute and keep the values in an in-memory structure and not do convert the values into strings. It also excludes the content of the pattern. We call this *fast materialization*.

PIDS executes both types of materialization by reading out each sub-attribute in order and storing the values in an in-memory structure. When performing string materialization, PIDS further converts each field in the structure to strings and injects them into the proper position in the pattern. Although the algorithm is straightforward, it can become a performance bottleneck since each sub-attribute brings overhead for decoding and formatting, and this overhead accumulates as the number of sub-attributes increases.

PIDS applies many optimization techniques to mitigate overhead, including a fast algorithm to convert an integer to string, and a cache-friendly implementation to read the sub-attributes in blocks. In the prototype we developed using Java, we create a sizeable native memory as a buffer to avoid the instantiation of too many string objects and relieve the overhead brought to JVM garbage collection.

6.4 Sub-Attribute of Variable Length

In operators involving comparisons, such as less predicate and sorting, we push down the operators to sub-attributes based on the assumption that the ordering of the original attribute is uniquely determined by the ordering of its sub-attributes. For example, on an attribute x with two sub-attributes x_1, x_2, $x_1 < a_1 \vee (x_1 = a_1 \wedge x_2 < a_2) \implies x < a$. This is true when the sub-attributes only contain rows of the same length, but the situation becomes more complicated when some sub-attributes contain rows of variable length. An example is shown below on the left, where x_2 has variable size of 2 to 4.

	x_1 x_2	Padding x_2		Add Length
e_1	313-3195-T	→ 3195	→	31954
e_2	313-42-T	→ 4200	→	42002
e_3	313-420-T	→ 4200	→	42003

By simply extracting the sub-attributes and comparing them, we have $e_2.x_2 = 42 < 3195 = e_1.x_2 \implies e_2 < e_1$, while the right order should be $e_1 < e_2$ under string comparison. To correct this, we pad the data to restore their correct order. This padding is applied to data on the fly when performing comparisons and has no impact on data stored on disk.

In the example above, we see that the symbol following x_2 is a dash, which has a smaller ASCII code than the digits. Thus, we right-pad x_2 with 0 (marked in red, underlined) to length 4, which restores the correct order of e_1 and e_2. This padding now makes e_2 and e_3 indistinguishable, for the correct order of $e_2 < e_3$. As shorter entries in x_2 are smaller, we append the entries with their original length (marked in blue, underlined) to break the tie. After the padding, the entries in x_2 satisfy $e_i.x_2 < e_j.x_2 \implies e_i < e_j$. When the following separator is greater than a digit, the algorithm right-pads the entry with 9 instead of 0. And as shorter entries are larger when the separator's ASCII code is larger than a digit, e.g., '42**:**'>'429**:**', the algorithm appends maxLen-len(value) in this case, where maxLen is the maximal length of the target sub-attribute.

If a sub-attribute of variable length is a string type, we pad the first symbol following that sub-attribute to the value, and perform natural string comparison. For example, to compare addresses with two sub-attributes "Chicago, IL" and "Milwaukee, WI", we compare "Chicago," and "Milwaukee," by including the comma.

6.5 Handling Outliers

As described in Figure 5, PIDS stores outliers in a separate location in the original string format, along with its row ID. When we choose a valid pattern, the number of outliers should be relatively small, and we reasonably assume that the entire outlier table can be materialized in memory.

As outliers are stored in their original string format, all operators can be applied directly. When executing operators, we merge the outlier result with the result from the main table. Depending on the type of operator, different merging strategies are adopted.

For predicate execution, we generate a bitmap sized to the original data, marking the row ID of outlier records that satisfy the predicate, then perform a logical OR operation between the main data bitmap and the outlier bitmap to get the final result. For equality predicate, we notice that if the predicate matches the pattern, the result must be either be in the main table or the outlier table, but not both, so we only need to query one table, saving the logical OR operation.

For materialization, we need to merge the result from the main table with the outliers. We first materialize the outlier table as a memory buffer. When materializing data from the main table, we check null values that indicate the occurrence of outliers, and use the null value's position as the row ID to look up the memory buffer for the corresponding value. The value is inserted into the results from the main table.

7. EXPERIMENTS

In this section, we present the experiment results showing that PIDS improves both compression and query efficiency in a columnar store. We develop a prototype of PIDS in Java and Scala, using the Apache Parquet columnar storage format [4]. Our experimental platform is equipped with 2x Xeon Silver 4116 CPU, 192G Memory, and a NVMe SSD. It runs Ubuntu 18.04 LTS, OpenJDK 1.8.0_191, and Scala 2.12.4. For all throughput results, we report the average throughput of ten runs of a fixed duration after warm-up. We run the experiments on target attributes with uniformly randomly generated predicates for each execution.

Datasets: We use two datasets in the experiments. To justify that PIDS is widely applicable, we use a dataset consisting of 9124 string attributes, collected from various real-world data sources, such as open government sites, machine learning, social networks, and machine logs. The data sources details can be found at https://github.com/UCHI-DB/comp-datasets.

To evaluate operator execution efficiency, we choose four representative string attributes: Phone Number, Timestamp, IPv6, and Address. We generate IPv6 and Phone Number data uniformly, Timestamp data uniformly in a 10-year span, and Address data using a TPC-DS data generator [25]. For each attribute, we target a number of records that require 128 MB of space in PIDS. Table 2 shows examples of these attributes and the number of records.

Baselines: In our experiments, we compare PIDS against popular string encoding/compression algorithms, including Parquet with *no encoding*, Parquet with lightweight encoding, Snappy, Gzip, and a Block Re-Pair Front-Coding (BRPFC) dictionary [19]. BRPFC encodes an attribute using a dictionary, sorts the dictionary entries, then applies Front Coding and Re-Pair [18] on the entries. We implement BRPFC in the Apache Parquet framework and verify that our implementation has comparable performances to the original version. Due to a lack of support for SIMD in Java, we implement a scalar version of BRPFC, and use the SIMD improvement factors reported in the original paper to estimate the performance of a SIMD version [19].

In these baseline algorithms, the attributes are stored as string types in Parquet. For lightweight encoding, we empirically choose the encoding schemes in Parquet with the

Table 2: Representative attributes used in our evaluation.

Attribute	Sub Attrs	Rows (million)	Example
Phone	3	35	(312)414-4125
Timestamp	8	20	2019-07-25 12:30:01 3424.24232
IPv6	8	10	3D4F:1342:4524:3319:8532:0062 :4224:53BF
Address	8	13	121 Elm St., Suite 5, Chicago, Cook County, IL 60025

Table 3: PIDS achieves the best compression ratio on both the entire dataset of 9124 String Attributes (All), and the set of attributes filtered by the classifier (Cls).

	Raw	PIDS	Best Enc.	Snappy	Gzip	BRPFC
All	106G	14.6G	18.8G	33.1G	18.2G	26.6G
Cls	45G	4.2G	8.4G	9.8G	5.1G	7.8G

smallest compressed file size on the target attribute, referred to as *Best Encoding* in the rest of the paper.

Many DBMS have specific data types for timestamp data, usually backed by 64-bit integers. Storing timestamps as integers helps achieve better compression, but requires extra effort if users want to query partial fields, such as month or date. We include this approach in the baseline, referred to as *64-bit Int* in the experiments, to show that PIDS facilitates more efficient query operators and comparable storage benefits.

7.1 Compression Efficiency

In this section, we evaluate the compression efficiency of PIDS. In Table 3 we show the compressed size and compression ratio of PIDS against the baseline methods on the dataset of 9124 string attributes, and on the attributes recognized by the classifier. We see that in both cases PIDS has achieved the best compression ratio, and is 20% smaller than Gzip, which is the second best. The classifier recognizes 2868 attributes as compressable and achieves an accuracy of 91.2%. PIDS infers valid patterns from 4,596 (50.73%) attributes. Of these attributes, 3,214 fully match the pattern (no outliers), and the overall average outlier percentage is 0.6%. The average length of attributes with a pattern is 19, and the average number of sub-attributes is 7. We extracted 32,105 sub-attributes, with 50% integer and 50% string .

In the top sub-figure of Figure 6, we evaluate PIDS and the baseline methods on the four attributes used in operator evaluation. For compression, PIDS outperforms all other approaches on the 4 attributes and achieves a 2-3 times size reduction compared to Snappy, a popular compression method used in many columnar stores. Figure 6 also shows that Timestamp compressed with PIDS has similar size as that stored in 64-bit Integers. In PIDS, all sub-attributes are bit-packed, and the total number of bits they occupy is no larger than a 64-bit integer.

The bottom sub-figure of Figure 6 shows the throughput of end-to-end compression for all approaches, which measures the time consumption including pattern inference, extraction, encoding, and persistence. BRPFC has a throughput that is 10x slower than others largely due to the recursive pairing step, making it almost invisible in the figure. PIDS's throughput varies by attribute. On the phone attribute, PIDS is slightly faster than Snappy, while on address and IPv6, PIDS is about half as fast as Snappy. To understand the factors impacting PIDS's encoding performance, we study how much time each step consumes. As pattern

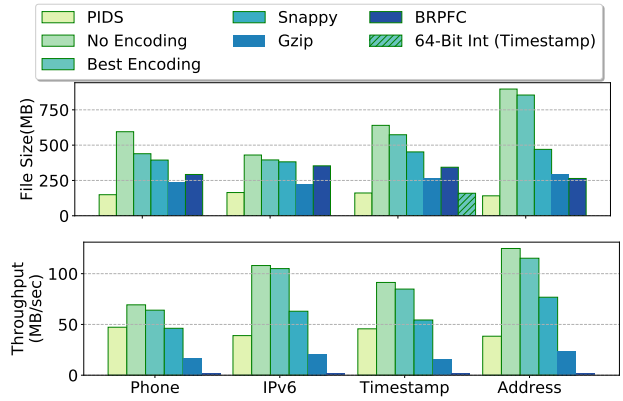
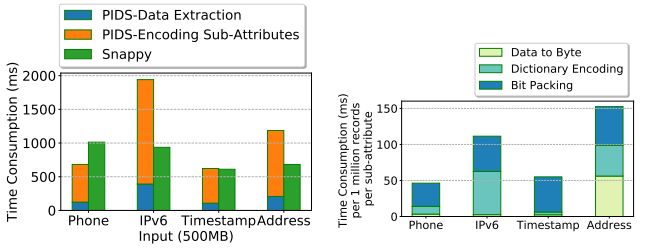


Figure 6: Comparing PIDS compression performance in both size and end-to-end compression throughput.



(a) PIDS spends most time on encoding sub-attributes. **(b)** Average Time Consumption Per Sub-Attribute

Figure 7: Compression time breakdown. PIDS spends more time on attributes with more sub-attributes, higher cardinality, and more string sub-attributes.

inference takes a constant amount of time, and disk IO is 10-20x faster compared to the overall throughput, we focus on the data extraction and encoding steps. In Figure 7a, we see that PIDS spends about 80% on the encoding step, which is also the source of the variance in the throughput.

As encoding is done on each sub-attribute independently, in Figure 7b we study the decomposition of time consumption to process one sub-attribute on the four attributes. IPv6 and Address spend significant time on dictionary encoding because their sub-attributes have a larger cardinality and need to maintain a larger dictionary. The average cardinality for the sub-attributes in IPv6 is 65,536, and for Address it is 100,000. Whereas Phone is 4,000, and Timestamp is 3,784. Moreover, Figure 7b shows that Address spends much time on converting data to bytes, which the other three attributes do not. Table 2 shows that while Timestamp, IPv6 and Address all have 8 sub-attributes, all sub-attributes in Timestamp and IPv6 are integers, while 6 of 8 sub-attributes in Address are strings. Encoding strings to bytes takes much more time than encoding integers. The analysis above suggests that PIDS performs better on sub-attributes with low cardinality and few string types.

7.2 Operator Execution

In this section, we show that PIDS accelerates common query operators. Comparing the same operators on the same dataset stored in string format, PIDS brings at least 2 times performance improvement to all predicate execution tasks. In some tasks, such as prefix search, the improvement can be over 30 times.

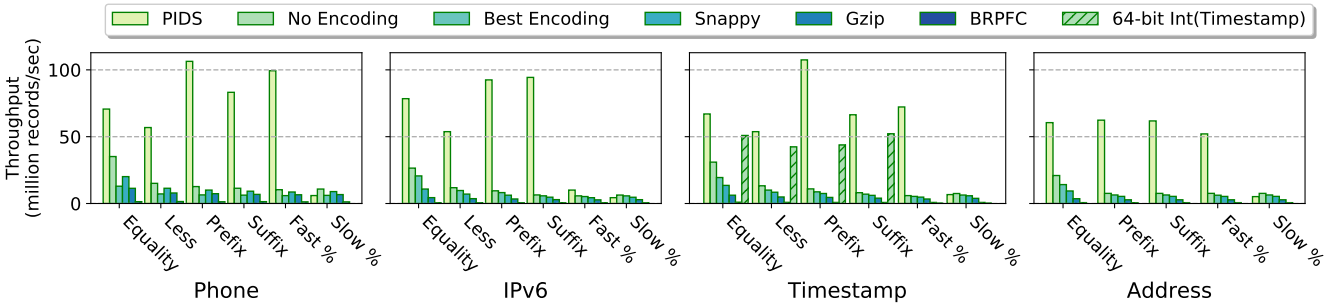


Figure 8: Predicate Execution. PIDS is 2-30x faster than all baselines on all four attributes for Equality, Less, Prefix and Suffix, and most Fast Wildcard Predicates. PIDS is also 20% faster than Timestamp stored as 64-bit int.

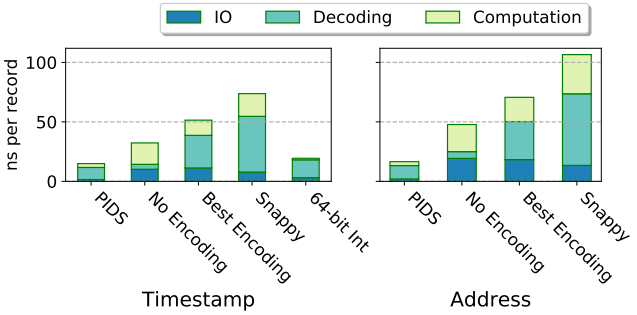


Figure 9: Equality cost breakdown (Gzip and BRPFC are excluded for clarity due to too high of time cost).

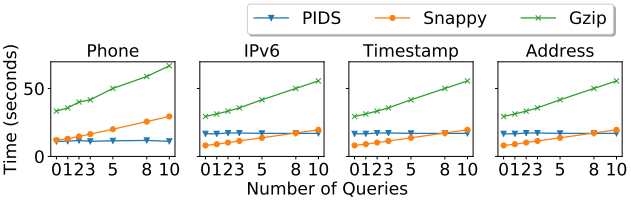


Figure 10: Time consumption of compressing an attribute and performing multiple equality queries.

7.2.1 *Predicate Filtering*

We compare the performance of PIDS on various predicates against the baselines. In addition to the straightforward equality and less predicates, we also test prefix, suffix, and wildcard predicates. Examples of these queries are given in Table 4. We experiment with two types of wildcard predicates. "Fast Wildcard" is a wildcard predicate that has a single match against the pattern. When executing a fast wildcard predicate, PIDS can push down the predicates to only the involved sub-attributes, ignoring other sub-attributes, and speed up the execution. For example, "%33:27%" is a fast wildcard predicate for the timestamp attribute, as it only matches the minute and second sub-attributes. A "Slow Wildcard" is a wildcard predicate that has more than one match against the pattern. In the worst case, PIDS needs to scan all sub-attributes when executing these predicates. "%12%" is a slow wildcard for the timestamp attribute as it can potentially match any sub-attribute for timestamp. To execute this predicate, all eight sub-attributes and the whole data file need to be accessed.

The experimental results are shown in Figure 8. We do not test the less predicate on the address attribute since it makes no practical sense. On the equality and less predicates, PIDS beats all string-based competitors by 2-10 times. This improvement primarily comes from data skipping by progressively filtering sub-attributes. For example, when executing an equality predicate on the IPv6 attribute, PIDS scans on average only 2.006 sub-attributes per execution, and accesses 24.35% of the whole data file.

Table 4: Examples of wildcard queries.

	Prefix	Suffix	Fast Wildcard	Slow Wildcard
Phone	(377)62%	%524	%42-47%	%24%
IPv6	24BD:52%	%1B2D	%243B%	%1D%
Timestamp	2017-09%	%24389	%33:27%	%16%
Address	121 Elm St.%	%IL,32036	%Chicago,IL%	%Cook%

On prefix, suffix, and most fast wildcard predicates, PIDS beats all string-based competitors by 10-30 times. As we have seen in Section 6.2.3, we can convert these predicates to equality or like predicates on one or two sub-attributes, making it even more efficient than equality predicates. The only case where fast wildcard does not have obvious performance improvement is on the IPv6 attribute, where all sub-attributes have the same length. A wildcard predicate such as $x \sim$%1A2B% can match any sub-attribute, requiring executing equality predicates on all eight sub-attributes. However, as an equality comparison is faster than a wildcard search, PIDS manages to obtain 2 times throughput compared to its fastest competitor. The only case where PIDS does not work well is the slow wildcard, where it needs to execute like predicates on all sub-attributes. Nevertheless, PIDS still yields similar performance to Snappy.

For Timestamp stored as 64-bit integers, we implement prefix query with range predicate, and suffix query with modular operation to compare against PIDS. We see that on equality, less, prefix and suffix queries, PIDS is consistently 20% faster than 64-bit integer, due to data skipping. In addition, 64-bit int performs poorly on wildcard predicates, which require converting 64-bit Int to date string. The numbers are too small to be visible in the figure.

In Figure 9, we show a cost breakdown of equality predicates on the timestamp and address attributes, to help us understand how PIDS achieves its performance boost. We do not include Gzip and BRPFC in the figure as they consume much more time compared to others. Including them makes the details of other results hard to interpret. Not surprisingly, both Gzip and BRPFC spends the majority of the time in decompression. We see that the string-based competitors spend a considerable amount of time on I/O and incur large overhead in computation (primarily string comparison). Best (Lightweight) Encoding and Snappy both spend a significant amount of time on decoding/decompression. PIDS has a smaller file size, which saves I/O and it uses a dictionary to translate all predicates into integer comparisons, saving computation effort. By pushing down the

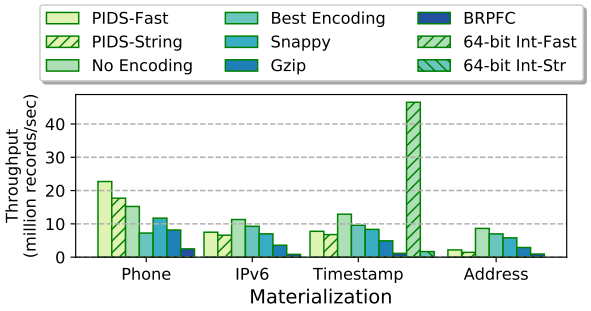


Figure 11: PIDS uses an optimized integer-to-string algorithm, and achieves a throughput at least as good as Snappy when doing string materialization.

predicates to sub-attributes, and performing data skipping, PIDS further reduces I/O and decoding overhead.

PIDS executes the pattern inference and data extraction task only once when it persists an attribute as a collection of sub-attributes. These steps in PIDS are similar to the compression step in Snappy and Gzip. When PIDS executes a query, it accesses each sub-attribute directly by pushing down the predicate. The query operation thus involves no pattern inference or data extraction operation and is transparent to the user. In Figure 10, we compare the end-to-end time consumption to read 500MB textual records from disk file, perform compression and persist to disk, then conduct multiple equality queries operations against the compressed file with Snappy, Gzip, and PIDS. Each query performs data decompression before execution. While PIDS takes slightly longer time when compressing data on some attributes, the overall time consumption is compensated by the fast query execution. Only after 8 queries, PIDS has a shorter total time on all four attributes. This is more promising, considering that PIDS also has a better compression ratio.

7.2.2 *Materialization and Outliers*

Figure 11 shows an experiment with the two types of materialization operators: fast materialization, which loads sub-attributes into an in-memory data structure that can be used by other operators, and string materialization, which generates string results for output. We denote them by PIDS-Fast and PIDS-String respectively. For the 64-bit Int representation of timestamp data, we also show 64-bit Int-Fast, which reads 64-bit integers into memory, and 64-bit Int-String, which uses Apache Commons' `FastDateFormat` to format 64-bit integers into timestamp strings.

Since materialization requires access to the whole dataset and no data can be skipped, it is not surprising that PIDS no longer beats competitors by a large margin. Nevertheless, we see that it still outperforms all other competitors on the phone attribute. On the IPv6 and timestamp attributes, PIDS outperforms Gzip, and has a similar throughput to Snappy. Only on the address attribute, does PIDS have a slightly worse throughput than Gzip. A cost breakdown shows that when the number of sub-attributes increases, more time is spent on decoding each sub-attribute, and decoding string sub-attributes is slower than decoding integer sub-attributes, primarily due to decoding bytes to UTF-8.

We notice that for the timestamp attribute, although 64-bit Int-Fast is 4 times faster than other approaches, 64-bit Int-String is so slow that it is almost invisible in the figure. We saw similar results when executing wildcard predicates on 64-bit integers. The profiling result shows that over 40% of the time is spent on formatting integers to strings. In PIDS, we introduce an optimized integer to string algorithm inspired by the integer constant division algorithm [30], which formats an integer to string 37 times faster than `String.format` in Java. With this algorithm, PIDS-String is only 10% slower than PIDS-Fast and remains competitive against other approaches.

In Figure 12, we test the impact of outliers on the string materialization of phone numbers by controlling the percentage of outliers. We show the throughput normalized against having no outliers. We observe a minimal impact on the throughput when the percentage of outliers is less than 1% (shown in the small box). Additional increases to the percentage of outliers create a proportional impact on throughput. Considering that most attributes we observe have less than 1% outliers, this result shows that outliers have a negligible impact.

7.3 Pattern Inference and Data Extraction

Applying PIDS on an attribute requires executing the pattern inference algorithm on the attribute, and using the inferred pattern to extract sub-attributes. In this section, we show that PIDS accomplishes these tasks efficiently.

In Figure 13, we show the time used to infer a pattern from the attributes while varying the sampling size. We see with sample size of 2000, the inference latency is around 1 second. This latency is negligible for large data loading tasks as it is an one-off operation.

We also study how the sample size affects accuracy in random sampling, measured by the coverage of a pattern on an attribute. Assuming the data in an attributes can be covered by non-overlapping patterns $p_1, p_2, \ldots, p_n$, each with coverage c_i. The pattern PIDS generates will cover p_i when the sample includes at least one record from p_i. With uniform sampling, a sample size $\mathbb{E}(S) = \max(\frac{1}{1-c_i})$ is sufficient to cover all p_i. We experimentally verify that a sample size 500 is sufficient to achieve coverage of 99%, and sample size 2000 can reach coverage of 99.95%, on the 4596 string attributes that PIDS extracts a valid pattern. To further improve coverage, other sampling methods such as adaptive and biased sampling [17] can also be employed to guarantee instances with small numbers also appear in the sample, at the cost of higher inference latency.

It is crucial to have an efficient data extraction algorithm for attribute decomposition. In Figure 14, we show a micro-benchmark comparing PIDS's sub-attribute extraction algorithm with the widely used regular expression-based algorithm and a state machine-based algorithm based on recent work on extracting structures from relational attributes [14]. We varied the number of sub-attributes, with each sub-attribute containing five numerical digits and split by a comma. When the number of sub-attributes is smaller than 10, PIDS achieves over 2 times throughput compared to regular expression, and 50% improvement compared to a state machine. The throughput of PIDS diminishes gradually when the attribute length increases, but still outperforms the two competitors by 30-50%.

Next, we compare PIDS's pattern inference and data extraction algorithm against Datamaran [11], a state-of-the-art solution for extracting structural information from log-like datasets. We conduct the comparison from the perspective of both `correctness`: being able to generate accurate

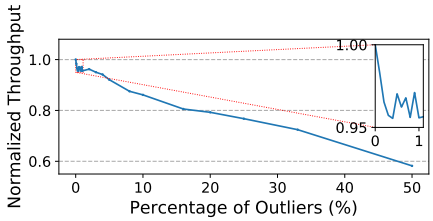


Figure 12: Phone string materialization with increasing the percentage of outliers.

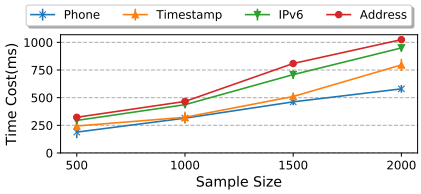


Figure 13: Pattern inference latency.

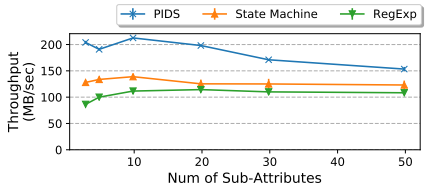


Figure 14: Sub-attributes extraction performance.

Table 5: Comparison of PIDS and Datamaran on inference accuracy. PIDS finds more patterns than Datamaran on the entire dataset of 9124 string attributes.

Category	Entire Dataset	After Classifier
Both find a pattern	1758(19.04%)	566 (19.73%)
Both find no pattern	4597(49.89%)	17 (0.59%)
Only PIDS finds a pattern	2841(30.83%)	2285 (79.67%)
Only Datamaran finds a pattern	18(0.22%)	0(0%)

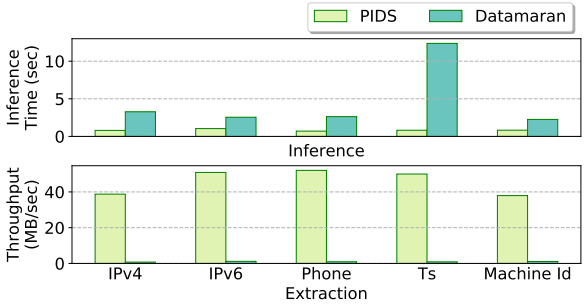


Figure 15: Comparison of PIDS and Datamaran on time efficiency. PIDS is 2-15x faster than Datamaran on inference and 20-40x on data extraction.

patterns from given attributes, and `time efficiency`: better throughput on pattern inference and data extraction. We evaluate an open-source implementation from the authors.

To compare the accuracy of pattern inference, we apply both PIDS and Datamaran to 1) the entire dataset of 9124 string attributes, 2) the dataset of attributes marked as sound by the classifier we introduced in Section 5.3. We mark an extracted pattern as valid if it (1) contains more than one sub-attribute and (2) has more than 50% coverage. In Table 5, we show that PIDS manages to find a pattern on 2841 attributes(30% of total) that Datamaran does not, while Datamaran only find 18 patterns that PIDS does not. After applying the classifier, PIDS works even better to recognize 2285 patterns that Datamaran does not. Besides, this result also cross-validates the effectiveness of the classifier: it manages to filter out most attributes that neither PIDS nor Datamaran extract patterns. We note that Datamaran was designed to target log-like data files, which may contribute to some of its inferior performance.

We are also interested in the attributes on which PIDS and Datamaran do not agree. We categorize the 31% data columns on which PIDS finds patterns, but Datamaran does not, into three types, and explain how PIDS handles them.

- **Type 1**: Attributes contains symbols not recognized by Datamaran. Datamaran relies on a hard-coded symbol table to split the records. PIDS overcomes this by inferring common separators from context and works in a truly unsupervised manner.
- **Type 2**: Attributes contain English words or phrases. Some attributes, such as addresses, contain English words or phrases. Datamaran recognizes all spaces in these attributes as separators and generates erroneous patterns. PIDS can recognize words and phrases, and treats them as integral components.
- **Type 3**: Attributes with optional sub-attributes. For example, if an attribute consists of 50% date string, and 50% timestamp string, PIDS recognizes the date sub-attributes as mandatory and the time ones as optional.

There are 18 attributes on which PIDS infers no pattern, but Datamaran does. We manually inspect these attributes and see that these attributes contain only one distinct value. PIDS infers a pattern consisting of only constant from these attributes, and consider such pattern as invalid. We believe this also shows PIDS is more effective in the sense of recognizing meaningful pattern.

We then compare the time efficiency of pattern inference and data extraction between PIDS and Datamaran, using 5 representative string attributes, each containing 1 million records. Datamaran samples 2000 records from the target attribute for inference, and we configure PIDS to follow the same setting. In Figure 15, we see that PIDS is 2-15x faster than Datamaran in pattern inference. In the data extraction task, PIDS achieves 20-40x throughput comparing to Datamaran. Considering that PIDS is implemented in Java and Datamaran in C++, this performance boost is significant.

8. CONCLUSION

In this paper, we introduce PIDS, a technique to extract sub-attributes from relational string attributes in columnar stores, and execute query operators on them. We build a prototype of PIDS based on the Apache Parquet storage format to show that PIDS can improve both compression ratio and query operator execution efficiency. When executing query operators, PIDS is 2-30 times faster than execution on the original string attributes. In the future, we plan to extend PIDS to support more features, including using cross-validation to recognize extractable patterns, multi-pattern support for attributes, and supervised pattern inference algorithm. We are also interested in exploring how PIDS can improve other database operators such as joins, hashing, and aggregation, and integrate PIDS into a workload-aware and compression-aware cost optimizer.

9. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful feedback. This work was supported by a Google DAPA Research Award, and gifts from the CERES Center for Unstoppable Computing, NetApp, Cisco Systems, Exelon, and FutureWei.

10. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[2] Apache Foundation. Apache CarbonData. https://carbondata.apache.org/.

[3] Apache Foundation. Apache ORC. https://orc.apache.org.

[4] Apache Foundation. Apache Parquet. https://parquet.apache.org/.

[5] A. Arasu and H. Garcia-Molina. Extracting Structured Data from Web Pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 337–348, New York, NY, USA, 2003. ACM.

[6] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 218–228, New York, NY, USA, 2015. ACM.

[7] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. 2005.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[9] L. P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996.

[10] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. *SIGPLAN Not.*, 43(1):421–434, jan 2008.

[11] Y. Gao, S. Huang, and A. Parameswaran. Navigating the Data Lake with DATAMARAN: Automatically Extracting Structure from Log Datasets. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 943–958, New York, NY, USA, 2018. ACM.

[12] B. Ghită, D. Tomé, and P. Boncz. White-box compression: Learning and exploiting compact table representations. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, Jan. 2020.

[13] S. Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[14] A. Ilyas, J. M. F. da Trindade, R. Castro Fernandez, and S. Madden. Extracting syntactical patterns from databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 41–52, April 2018.

[15] H. Jiang and A. J. Elmore. Boosting Data Filtering on Columnar Encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 6:1–6:10, New York, NY, USA, 2018. ACM.

[16] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. Foofah: Transforming Data By Example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 683–698, New York, NY, USA, 2017. ACM.

[17] G. Kollios, D. Gunopulos, N. Koudas, and S. Berchtold. Efficient biased sampling for approximate clustering and outlier detection in large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1170–1187, Sep. 2003.

[18] N. J. Larsson and A. Moffat. Offline Dictionary-Based Compression. In *Proceedings of the Conference on Data Compression*, DCC '99, pages 296–, Washington, DC, USA, 1999. IEEE Computer Society.

[19] R. Lasch, I. Oukid, R. Dementiev, N. May, S. S. Demirsoy, and K.-U. Sattler. Fast & strong: The case of compressed string dictionaries on modern cpus. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, pages 4:1–4:10, New York, NY, USA, 2019. ACM.

[20] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[21] V. Le and S. Gulwani. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA, 2014. ACM.

[22] K. Lerman, L. Getoor, S. Minton, and C. Knoblock. Using the Structure of Web Sites for Automatic Segmentation of Tables. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 119–130, New York, NY, USA, 2004. ACM.

[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *PVLDB*, 3(1-2):330–339, 2010.

[24] I. Müller, C. Ratsch, and F. Färber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*, 2014.

[25] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.

[26] J. Pennington, R. Socher, and C. D. Manning. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[27] M. Raza and S. Gulwani. Automated Data Extraction using Predictive Program Synthesis. Association for the Advancement of Artificial Intelligence, February 2017.

[28] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. ONeil, and et al. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[29] P. Venetis, A. Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering

Semantics of Tables on the Web. *PVLDB*, 4(9):528–538, 2011.

[30] H. S. Warren. Integer division by constants. In *Hackers delight*, chapter 10, pages 190–192. Addison-Wesley, 2 edition, 2013.

[31] M. Zhang and K. Chakrabarti. InfoGather+: Semantic Matching and Annotation of Numeric and Time-varying Attributes in Web Tables. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 145–156, New York, NY, USA, 2013. ACM.

[32] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[33] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.