Optimizing BDDs for Time-Series Dataset Manipulation

Stergios Stergiou
Fujitsu Labs of America
Sunnyvale, CA
stergiou@fla.fujitsu.com

Jawahar Jain
Fujitsu Labs of America
Sunnyvale, CA
jawahar@fla.fujitsu.com

***Abstract*—In this work we advocate the adoption of Binary Decision Diagrams (BDDs) for storing and manipulating Time-Series datasets. We first propose a generic BDD transformation which identifies and removes 50% of all BDD edges without any loss of information. Following, we optimize the core operation for adding samples to a dataset and characterize its complexity. We identify time-range queries as one of the core operations executed on time-series datasets, and describe explicit Boolean function constructions that aid in efficiently executing them directly on BDDs. We exhibit significant space and performance gains when applying our algorithms on synthetic and real-life biosensor time-series datasets collected from field trials.**

I. Introduction

A. Motivation

Improvements in processor energy efficiency have brought significant computation resources closer to the human body. At the same time, on-body sensors are becoming increasingly less intrusive, thus leading to their more widespread adoption. This has spurred the creation of new markets for continuous monitoring and processing of biomarkers for wellness and health applications. While computational power has grown to sufficient levels to enable real-time computation, mobile storage and communication resources have not scaled appropriately to support the data creation rates of biosensors, which can can be on the order of hundreds of megabytes per day. Time-series datasets obtained from biosensors can be noisy, out of sync, and have connectivity drop issues, but they do exhibit significant regularity as they represent signals such as cardiographs or breathing waveforms that are (at large) repeating. The potential of compressibility is therefore apparent. However, traditional compression techniques such as Lempel-Ziv-Welch or LZMA do not appear to be suitable replacements for data representation structures because of relatively low compression rates and, most importantly, because the data become inaccessible until they are decompressed. Binary Decision Diagrams [1], one of the cornerstone creations of the EDA community, were designed specifically for compactly representing large, highly regular bit-vectors, while at the same time allowing for random access and efficient manipulation. As such, they are potentially an attractive fit for the target application.

B. Contributions

In the context of this work, BDDs are mainly used for representing sets of integers S by means of their *Characteristic Function* (CF), a Boolean function $f^S(\vec{x})$ that evaluates to T iff $[\vec{x}] \in S$, where $[\vec{x}]$ denotes the integer that is formed by the boolean assignments of the variables on $\vec{x}$. Following, we informally enumerate our contributions.

1) Trace Based BDD Compaction: We propose a transformation that discards half of the edges present on any BDD. Let us assume that the nodes of a BDD are stored consecutively in memory in a pre-specified order, for example, as obtained by performing a depth-first traversal of the BDD following zero edges first. We make the following crucial observation: When a node d is visited for the first time during the traversal via an edge $e = (s, d)$, then the exact position of d relative to the position of s can be deduced independently of e. On a BDD of n nodes, $n - 1$ nodes will be discovered for the first time via an edge during the traversal and therefore $n - 1$ edges out of a total of $2n$ edges can be discarded without any loss of information.

2) Efficient Disjunctions with Cubes: We then proceed to optimize the core operation for adding data points to a BDD. Let $f^{S_t}(\vec{x})$ denote the characteristic function of all samples until time instance t and $g^{s_{t+1}}(\vec{x})$ denote the characteristic function of singleton $\{s_{t+1}\}$. Then $f^{S_{t+1}}(\vec{x}) = f^{S_t}(\vec{x}) \vee g^{s_{t+1}}(\vec{x})$. The BDD of g is a path. It is constructed for performing the disjunction and is discarded immediately thereafter. While the BDD of g is straightforward to construct, it is inefficient to do so for each new sample point, and garbage collect its nodes afterwards. We propose a new algorithm for executing binary operations directly between BDDs and minterms, without ever explicitly constructing a BDD for the latter, thus significantly speeding up the core operation of adding a new sample.

3) Range Queries: One of the most elementary operations for accessing data is the range query, whereby all data points collected within a particular time range $[t_1, t_2]$ are returned. We propose a new algorithm for constructing BDDs of *threshold* functions $TH_A(\vec{t})$, which evaluate to T iff $[\vec{t}] \geq A$. We show that such BDDs are paths and are efficient to construct. We then use threshold functions to construct range functions as $R_{[t_1,t_2]}(\vec{t}) = TH_{t_1}(\vec{t}) \wedge \neg TH_{t_2+1}(\vec{t})$.

II. Preliminaries

A *Reduced Ordered Binary Decision Diagram* is a directed acyclic graph. Its nodes are either *internal* or *terminal*. Internal nodes are labeled by a boolean variable identifier and have two outgoing edges, labeled *one* and *zero*. Terminal nodes have no outgoing edges and are either labeled *T* or *F*. The graph is free of nodes whose edges point to the same node and nodes that have the same variable id, one edge and zero edge. At most two uniquely labeled terminal nodes are present. There exists a single node with no incoming edges, named *root*. All paths from the root to a terminal node visit nodes whose variable identifiers are ordered in the same way. Each path includes at most one node labeled with the same variable identifier. Each ROBDD node represents a boolean function. Terminal node T (respectively F) corresponds to function 1 (respectively 0.) Internal node $(id, one, zero)$ corresponds to $f(\vec{x}) = x_{id} \cdot one + \overline{x_{id}} \cdot zero$. Layer i of an ROBDD comprises all nodes labeled with variable id i. ROBDDs are a canonical data structure in the sense that there exists a unique BDD for every Boolean function under the assumption of a fixed variable ordering. BDD libraries typically implement operations between ROBDDs in a depth-first manner [2]. They are based on the following recursive decomposition (where $\diamond$ is any 2-input function.)

$$f \diamond g = x_i(f|_{x_i=1} \diamond g|_{x_i=1}) + \bar{x}_i(f|_{x_i=0} \diamond g|_{x_i=0}) \quad (1)$$

We adopt NanoDDs [3], an implementation of ROBDDs whereby the node structure is adjusted dynamically based on the size of the BDDs, providing significant speedups and space savings over traditional BDD libraries with no loss in operation efficiency.

III. Trace Based BDD Compaction

In this section we describe the transformation for compactly maintaining BDDs. Given a BDD, the goal is to remove as much information as is possible without losing any information. The BDD

978-3-9815370-0-0/DATE13/©2013 EDAA

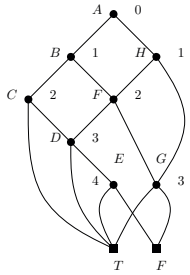

Fig. 1. Example BDD for trace based transformation. Nodes are labeled with the associated variable id. Letters denote the order of a depth-first traversal starting from the root node, following zero (dashed) edges first.

is typically stored as a sequence of nodes, each of which is represented by its associated variable id and two node ids corresponding to its two outgoing edges. We begin by breaking this logical separation into node structures, by storing the BDD as a sequence of variable and node ids, and proceed by discarding all redundant information.

We label nodes as they would be discovered by a depth-first traversal starting from the root node, following zero edges first. For example, let us examine the BDD on Fig. 1. Its nodes would be stored in memory as follows:

0BH 1CF 2tD 3tE 4tf 2DG 3tf 1FG

Let us consider an edge $e = (X, Y)$, where node Y has not been reached at a previous point during the traversal. Let as assume that e is a zero-edge. Then the node structure of Y will be stored in memory immediately after that of X. In the example, node B, described by "1CF" is stored immediately after node A ("0BH"). Let us now assume that e is a one-edge. Then the node structure of Y will be stored immediately after the memory footprint used to completely describe the BDD rooted by the target of the zero-edge of node X. In the example, the information for node H ("1FG") is stored immediately after the information that describes the BDD rooted at B ("1CF 2tD 3tE 4tf 2DG 3tf"). In both cases, the node id information provided by edge e can be deduced without prior knowledge of e and thus, e can be discarded without loss of information.

Continuing our example, we mark the edges that can be discarded in **bold**:

0**BH** 1**CF** 2t**D** 3t**E** 4tf 2D**G** 3tf 1FG

The optimized memory footprint is:

012t3t4tf2D3tf1FG

Lemma 3.1: A BDD of n nodes is uniquely described by n variable ids and $n + 1$ node ids.

Proof: As a BDD with n nodes has $2n$ edges and all nodes except the root will be discovered via one of those edges, exactly $n - 1$ edges can be discarded. □

We note that if the identity variable order is used on the BDD (for example by maintaining the actual order separately) then we can additionally delta-encode the variable ids, as it suffices to maintain the difference of the variable id of a node from the variable id of the parent node that first discovered it. Moreover, since a node structure no longer exists, the BDD manager needs to know whether a particular field is a variable or a node id. This can be performed either by merging the address spaces of the two fields, or using the most significant bit to discriminate between the two. In the first case for example, if the field is in the range $id \in [0..max\_var)$ then it denotes a variable, otherwise $id - max\_var$ denotes a node id.

More crucially, we observe that the complexity of arbitrary BDD operations is not altered when the BDDs are stored in the proposed trace-based format. Therefore, the transformation is useful in a more general context. Nevertheless, detecting whether a particular variable assignment evaluates to T becomes an $O(n)$ (as opposed to $O(\texttt{depth})$) operation.

```
1: BDD binary<OP>(BDD f, INT mt, INT var)
2: if var < 0 then
3:    return traditional_binary<OP>(f, T);
4: one = zero = f;
5: if f is nonterminal then
6:    if f.var == var then
7:       one = f.one; zero = f.zero;
8: if mt & 2^var then
9:    return bdd_node( var, binary<OP>(one, mt, var-1),
      traditional_binary<OP>(zero, F) );
10: else
11:   return bdd_node( var, traditional_binary<OP>(one, F),
      binary<OP>(zero, mt, var-1) );
```

Fig. 2. Implicit Minterm Algorithm.

```
1: BDD result = T;
2: INT bits = size of vector t⃗;
3: while bits do
4:    bits--;
5:    if value % 2 then
6:       result &= bdd_node(bits, T, F);
7:    else
8:       result |= bdd_node(bits, T, F);
9:    value »= 1;
```

Fig. 3. Threshold Algorithm.

IV. Implicit Minterm Operations

In this section we formalize our approach to storing time-series datasets as BDDs and optimize the core operation for adding data points to the BDD. Let $D^{q_t,q_s}(t) : \mathbb{N} \to \mathbb{N}$ denote a time-series dataset of q_t-bit time and q_s-bit sample quantization. Let $\vec{v}$ be a vector of boolean variables. Let $[\vec{v}]$ denote the integer represented by the binary representation on $\vec{v}$. For example $[(0, 1, 1, 0, 1)] = 13$. Let the CF of D be defined as $f^D(\vec{t}; \vec{s}) \equiv T$ iff $D^{q_t,q_s}([\vec{t}]) = [\vec{s}]$. $\vec{t}$ comprises q_t boolean variables while $\vec{s}$ comprises q_s boolean variables. Initially $f^D(\vec{t}; \vec{s})$ is set to F. Each new data sample (t_i, s_i) is added to f^D by performing a disjunction with the CF that represents the new sample. The time variables precede the sample variables in the ordering, while the most significant time bit is first in the order. The BDD of the sample CF is a path BDD. While the particular disjunction is an efficient operation both theoretically and practically, it is worth optimizing it further as this operation is repeatedly applied for each new data point. Let us identify the points of inefficiency with this core operation. First, the path BDD is created for each data point, and is subsequently discarded after the disjunction has been completed. For large datasets this imposes a significant overhead on the garbage collector. More concretely, for q_t-bit time and q_s-bit samples, adding P points will require creating and garbage collecting up to $P \cdot (q_t + q_s)$ nodes. Second, cache and node hash tables are "polluted" by the additional nodes. We propose instead to perform disjunctions with path BDDs implicitly. Specifically, we modify the traditional recursive execution of BDD ops such that one of the parameters is not a BDD but rather a minterm. As the second parameter represents a path BDD that includes a node for every variable, the minterm's topmost variable will always be the topmost during the recursion. If the i-th bit of the minterm is 1 (resp 0) then the implied node is $(i, *, 0)$ (resp $(i, 0, *)$.) This approach circumvents the creation and removal of all path BDD nodes while additionally avoids polluting the cache and node hash tables with them. The algorithm is depicted on Fig. 2.

V. Range Queries

In this section we describe our approach to querying time-series datasets modelled as BDDs. Data stored as BDDs can be queried in multiple ways. If the goal is to detect the presence of a particular (t, s) data point, we simply traverse the corresponding path on the

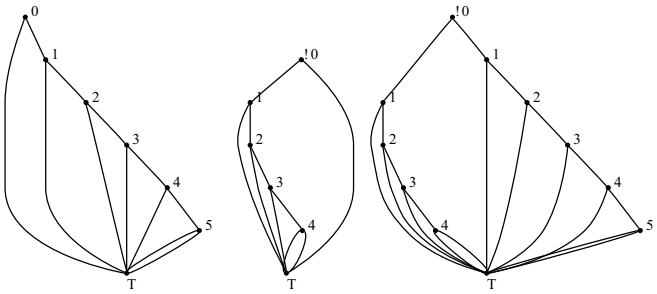

Fig. 4. Construction of range BDD for [5..37] that depends on 6 variables. (a) $TH_5(\vec{t})$, (b) $\neg TH_{38}(\vec{t})$, (c) $R_{[5..37]}(\vec{t}) = TH_5(\vec{t}) \wedge \neg TH_{38}(\vec{t})$. Solid edges denote one edges. Dashed edges denote zero edges. Dotted edges denote negated zero edges.

BDD, following the one edge when the related bit is one and zero otherwise. If terminal T is reached then the point exists in the dataset. If terminal F is reached at any point during the traversal, the point is not part of the dataset. If the goal is to detect the data point possibly present at a particular time instance, we traverse the corresponding path until we either reach terminal F or a sample point variable. At this point, we have detected that a data point is present and can decode its value. If the goal is to access all datapoints in the dataset sequentially, we perform a depth-first traversal starting from the root following zero edges first.

If we wish to access all data points within a particular time range, or collect all time stamps of data points that have values in a particular value range, our approach is to construct a boolean function that represents the corresponding range and perform a conjunction with the dataset CF. Let $TH_A(\vec{v})$ be a boolean function defined as $TH_A(\vec{v}) \equiv T$ iff $[\vec{v}] \geq A$. Let $R_{[A,B]}(\vec{v})$ be a boolean function defined as $R_{[A,B]}(\vec{v}) \equiv T$ iff $A \leq [\vec{v}] \leq B$. The BDD of TH_A is constructed by the algorithm on Fig. 3. Following, we prove the correctness of this algorithm.

Theorem 5.1: Algorithm 3 constructs the BDD of $TH_{value}(\vec{x})$.

Proof: Let q_v denote the number of bits of threshold integer $value$. We proceed inductively on q_v.

Terminal Case: For $q_v = 1$: If $value = 0$ then BDD T correctly represents $\{0, 1\}$. If $value = 1$ then BDD $(x_0, 1, 0)$ correctly represents $\{1\}$.

Induction: Let $q_v = k > 1$ and BDD $TH^{k-1}_{value\&(2^k-1)}$ has been properly constructed.

(a) If $value \ \& \ 2^k = 0$, the following holds: If $x_k = 1$ then $TH^k_{value\&(2^{k+1}-1)}$ is always true. If $x_k = 0$ then $TH^k_{value\&(2^{k+1}-1)} = TH^{k-1}_{value\&(2^k-1)}$. In both cases, it holds: $TH^k_{value\&(2^{k+1}-1)} = x_k + TH^{k-1}_{value\&(2^k-1)}$.

(b) If $value \ \& \ 2^k = 2^k$, the following holds: If $x_k = 0$ then $TH^k_{value\&(2^{k+1}-1)}$ is always false. If $x_k = 1$ the $TH^k_{value\&(2^{k+1}-1)} = TH^{k-1}_{value\&(2^k-1)}$. In both cases, it holds: $TH^k_{value\&(2^{k+1}-1)} = x_k \cdot TH^{k-1}_{value\&(2^k-1)}$.

□

The range function is constructed by utilizing threshold functions as follows: $R_{[A,B]}(\vec{v}) = TH_A(\vec{v}) \cdot \overline{TH}_{B+1}(\vec{v})$.

It is known than in general the complexity of a binary operation between two BDDs of sizes s_1 and s_2 is $O(s_1 \cdot s_2)$ [1]. In our particular case however, the structure of one of the two BDDs (representing CFs of sample points or threshold functions) is very restricted. Specifically, such BDDs are path BDDs with the additional property that each node has at least one edge pointing to a terminal node. We refer to such BDDs as *restricted path* BDDs. Following we show that the complexity of a binary operation between a BDD and a restricted path BDD is $O(s_1+s_2)$ even when complemented edges are allowed. This validates our intuition related to the predictability and efficiency of such operations. On Fig. 4 we show the two threshold functions and the range function for range [5..37].

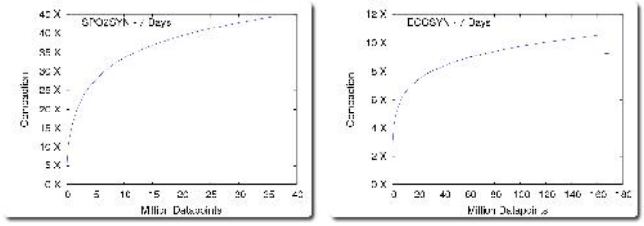

Fig. 5. Compaction Results for Synthetic Datasets

Theorem 5.2: A Boolean operation between an unrestricted BDD f and a restricted path BDD g of size s_1 and s_2 respectively is $O(s_1 + s_2)$ in time and space.

Proof: Let us consider the decomposition of (1). Since g is a restricted path BDD, at least one of $g|_{x_i=1}$, $g|_{x_i=0}$ is T or F. Let us assume that $g|_{x_i=1}$ is T or F. Subsequently, $f|_{x_i=1} \diamond g|_{x_i=1}$ will either be $T, F, f|_{x_i=1}$ or $\overline{f|_{x_i=1}}$, depending on the nature of the $\diamond$ operation, and the recursion will terminate on that branch, possibly generating a single node in the case of $\overline{f|_{x_i=1}}$ when complemented edges are not utilized. The recursion will follow the path of the non-terminal edges of g. Examining the decompositions bottom-up, we observe that the size of the subBDD will only increase by a single node if f does not depend on x_i. □

VI. Experimental Results

In this section, we evaluate the proposed algorithms on synthetic and real-life datasets. We are adopting two synthetic dataset generators, for blood oxygenation and electrocardiograph signals respectively. We selected these signals since their corresponding biosensors are the least intrusive while at the same time providing significant information. The first generator, SPO2SYN, was developed for the purposes of this work, while the second, ECGSYN [4] is employed to assess biomedical signal processing techniques which are used to compute clinical statistics from the ECG. Additionally, we deployed our biosensor data collection and processing platform on a small number of 8 participants for approximately 1-hour intervals per day over a period of 2 weeks, in a medically supervised hospital setting, in order to collect real-life datasets. Following, we present compaction and query execution experimental results, as well as the speedups obtained when adopting our implicit minterm BDD operations.

A. Compaction Results

We show the compaction obtained on 7-day long outputs of SPO2SYN and ECGSYN on Fig. 5. The uncompressed signals are being stored in binary format, allocating 32-bits for time and 8-bits (resp 10-bits) for SpO2 (resp. ECG) samples. We observe that the compaction rate is in general monotonically increasing and is approximately 45X for SpO2 and 10X for ECG. The discontinuity on the ECG signal compaction graph exists since at that point the size of the BDD nodes increased by 1 byte in order to support the growth of the CF BDD. We show the compaction results on all high-rate signals collected during the trial, specifically the SpO2, 3D Accelerometer, Breathing and ECG signals for the 8 participants,

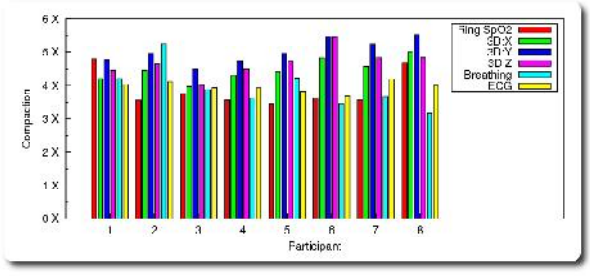

Fig. 6. Compaction Results on Real-Life Datasets from 8 Participants

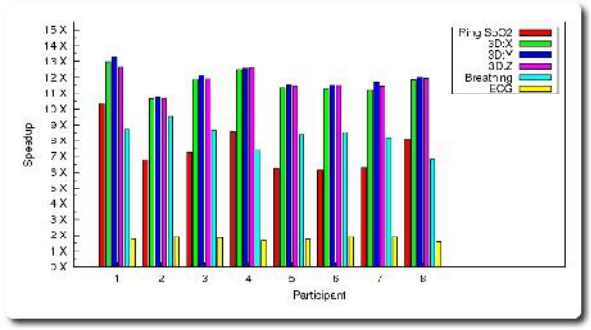

Fig. 7. Speedup Results on Real-Life Datasets from 8 Participants

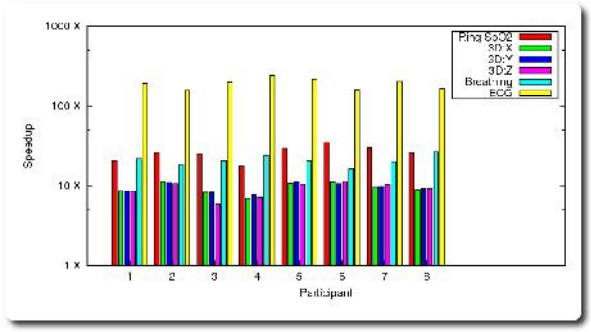

Fig. 8. Speedup Results for Queries

on Fig. 6. We observe that the obtained compaction rates are robust under minor signal variations and different participants. The aggregate recording duration per sensor is approximately twenty hours. Based on the results obtained from the synthetic datasets, we expect the compaction rates to increase on longer data recordings.

B. Implicit Minterm Operations Speedup Results

We obtain the execution times for building the BDD for a particular dataset by (1) using our implicit minterm operations and (2) using traditional BDD operations when adding new sample points. On the SPO2SYN dataset we obtain approximately 7X speedup while for the ECGSYN dataset we obtain approximately 1.7X speedup. We observe that the relative benefit is reduced as more data are added to the BDD. This is expected, since the implicit minterm operations avoid the construction of the path BDDs but they do not essentially affect the execution time of the disjunction operation, which becomes relatively slower as the CF BDD grows in size. We also observe speedups of 1.8X to 13X for the field-trial datasets, shown on Fig. 7. Similar to the synthetic datasets, the speedup on the ECG signals from real-life data is smaller, as these datasets are relatively larger because of higher sampling rate.

C. Query Results

We compare the execution times of range queries on BDDs and uncompressed datasets. For each dataset, we execute a query with a random starting point, whose range is approximately 20% of the size of the dataset. On the case of BDDs, we construct the BDD of the corresponding range function and perform the conjunction with the dataset BDD. On the case of uncompressed datasets, we perform a binary search in order to detect the starting point in the dataset and sequentially read the datapoints until the ending point is reached. The results are depicted on Fig. 8. The reader is cautioned not to interpret the results as speedups on actual overall application execution times, as the query times maybe a small percentage of the whole execution of some calculation.

D. Trace Storage Compression Results

In this section we explore the increase in compressibility obtained when runtime access to the dataset is not required. We first present results on the datasets from the 8 participants. We apply the trace-based

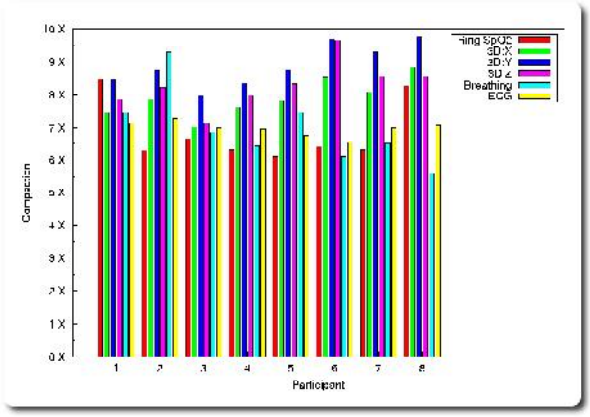

Fig. 9. Trace-Based Compression Results on Real-Life Datasets from 8 Participants. Trace-Based transformation is applied to each BDD after it has been constructed.

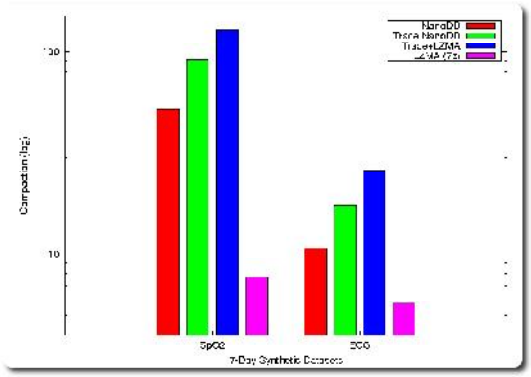

Fig. 10. Compression Results on Synthetic Datasets.

transformation on the resulting BDDs and compare, as previously, with the raw binary dataset. The results are depicted on Fig. 9. As expected, we observe an 1.8X increase in compaction compared to the results on Fig. 6. Subsequently, we present compression results on the synthetic datasets on Fig. 10. For reference, we also show the compression rates obtained by using LZMA (as implemented in the open-source tool 7z), both on the raw binary dataset and on the output of the trace-based transformation. In this scenario, it may be appropriate to use traditional compression tools as random accessibility to the dataset is not required. We observe that, in all cases, the compression rates obtained when adopting BDDs are superior to the ones obtained by traditional compression alone.

VII. Conclusion

In this work we proposed adopting BDDs for lossless storage and manipulation of time-series datasets. We described an efficient algorithm for optimizing the core operation performed when adding new data points to a BDD. We analyzed efficient algorithms for performing range queries directly on BDDs. We introduced a new BDD transformation that allows for discarding half of the edges present on a BDD. We benchmarked our algorithms on synthetic datasets and real-life data collected on a field trial. We obtained significant speedups (up to 12X for manipulation and 150X for range queries) and compression rates (up to 120X.)

References

[1] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677 –691, 1986.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Design Automation Conference*, 1990, pp. 40–45.

[3] S. Stergiou and J. Jain, "Dynamically resizable binary decision diagrams," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, 2010, pp. 185–190.

[4] P. McSharry, G. Clifford, L. Tarassenko, and L. Smith, "A dynamical model for generating synthetic electrocardiogram signals," *Biomedical Engineering, IEEE Transactions on*, vol. 50, no. 3, pp. 289 –294, March 2003.