

Lower Bounds on Retroactive Data Structures

Lily Chung  

Massachusetts Institute of Technology, Cambridge, MA, USA

Erik D. Demaine  

Massachusetts Institute of Technology, Cambridge, MA, USA

Dylan Hendrickson  

Massachusetts Institute of Technology, Cambridge, MA, USA

Jayson Lynch 

Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Abstract

We prove essentially optimal fine-grained lower bounds on the gap between a data structure and a partially retroactive version of the same data structure. Precisely, assuming any one of three standard conjectures, we describe a problem that has a data structure where operations run in $O(T(n, m))$ time per operation, but any partially retroactive version of that data structure requires $T(n, m) \cdot m^{1-o(1)}$ worst-case time per operation, where n is the size of the data structure at any time and m is the number of operations. Any data structure with operations running in $O(T(n, m))$ time per operation can be converted (via the “rollback method”) into a partially retroactive data structure running in $O(T(n, m) \cdot m)$ time per operation, so our lower bound is tight up to an $m^{o(1)}$ factor common in fine-grained complexity.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Retroactivity, time travel, rollback, fine-grained complexity

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.32

Acknowledgements This work was initiated during open problem solving in the MIT class on Advanced Data Structures (6.851) in Spring 2021. We thank the other participants of that class — in particular, Joshua Ani, Josh Brunner, and Naveen Venkat — for related discussions and providing an inspiring atmosphere. We also thank Michael Coulombe for helpful pointers regarding time travel.

1 Introduction

A trope in popular science fiction is the idea of traveling back in time, making a change to the world, then traveling forward in time (usually to the original time) to observe the effected changes. This style of time travel (as opposed to stable time loops or multiverses) was one of the inspirations for retroactive data structures. Such sequences of events occur in many pieces of media including in the movies *Back To The Future* (1985), *Timecop* (1994), *12 Monkeys* (1995), *Star Trek: First Contact* (1996), *The Butterfly Effect* (2004), *Looper* (2012), and *Avengers: Endgame* (2019); TV shows *Doctor Who* (1963/2005–), *Heroes* (2006–), *Marvel’s Agents of S.H.I.E.L.D.* (2013–), *DC’s Legends of Tomorrow* (2016–), and *The Umbrella Academy* (2019–); anime/manga *Doraemon* (1969/1973–) and *Steins;Gate* (2009/2011–); short story *A Sound of Thunder* (Ray Bradbury, 1952); books *The Hitchhiker’s Guide to the Galaxy* (Douglas Adams, 1979) and *Pastwatch: The Redemption of Christopher Columbus* (Orson Scott Card, 1996); and video games *Chrono Trigger* (1995) and *The Legend of Zelda: Ocarina of Time* (1998).

Is this kind of “jump back, change, jump forward” or “Back To The Future” time travel realistic? It is inconsistent with known models of physics, but it is nonetheless interesting to prove inconsistency with other assumptions about the physical world. Here we give



© Lily Chung, Erik D. Demaine, Dylan Hendrickson, and Jayson Lynch;
licensed under Creative Commons License CC-BY 4.0

33rd International Symposium on Algorithms and Computation (ISAAC 2022).

Editors: Sang Won Bae and Heejin Park; Article No. 32; pp. 32:1–32:13

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 evidence of such inconsistency under a *fine-grained physical Church–Turing thesis*:
 46 the universe’s physics while traveling forward in time are implemented by a computer with
 47 similar power (up to polylogarithmic or $n^{o(1)}$ factors) to standard theoretical computers
 48 such as the word RAM, possibly randomized or quantum, or possibly a parallel system of
 49 such computers. (The word RAM is the standard in data structures and most of theoretical
 50 computer science, and it can efficiently simulate e.g. an $O(1)$ -head d -dimensional Turing
 51 machine for any d , making it a good choice of universal model, at least for lower bounds.)
 52 Under this assumption, is time travel *computationally* realistic?

53 We can turn such questions about computational realism into problems about data
 54 structures, by thinking of the state of the universe as a data structure. For example,
 55 jumping back in time (for viewing only) is essentially asking for a *partially persistent* universe
 56 computation, while jumping back in time and making a change to branch a new universe is
 57 essentially asking for a *fully persistent* universe computation, both of which are efficiently
 58 possible for essentially any universe computation (with logarithmic overhead in general, and
 59 constant overhead assuming bounded in-degree which may be reasonable given geometric
 60 constraints) [7]. Returning to “Back to the Future” time travel, making a (blind) change
 61 in the past and instantly seeing the cascaded effects on the present is essentially asking
 62 for a *partially retroactive* universe computation, while observing and changing the past
 63 and immediately updating the future is essentially asking for a *fully retroactive* universe
 64 computation.¹

65 1.1 Our Results

66 In this paper, we give new evidence that data structures cannot be made efficiently retroactive,
 67 even partially retroactive, which complements a prior separation between partial and full
 68 retroactivity [4]. Specifically, under any of three standard assumptions in fine-grained
 69 complexity, we prove near-optimality of the trivial “rollback” method for retroactive changes
 70 to the past [5, Theorem 1]: rewind time by undoing operations to when the change should
 71 occur, make the desired change, and then replay the previously undone operations (by
 72 maintaining the current timeline of changes and their inverses). In the worst case (and
 73 assuming no amortization in the input data structure), this method occurs a multiplicative
 74 overhead of $\Theta(m)$ where m is the number of operations in the timeline. We prove an
 75 essentially matching worst-case lower bound of $\Omega(m^{1-o(1)})$ multiplicative overhead under any
 76 one of the following assumptions, which are all common in fine-grained complexity [14, 4]
 77 and were the same assumptions made by the prior retroactive separation results [4]:

- 78 1. CircuitSAT on an n -input $2^{o(n)}$ -gate circuit requires $2^{n-o(n)}$ time.
 79 This assumption is weaker than the standard Strong Exponential Time Hypothesis (SETH)
 80 [13] which makes a similar statement about formulas (which cannot re-use computations
 81 like circuits can).
- 82 2. $(\min, +)$ matrix–vector product with an $n \times n$ integer matrix and an online sequence of
 83 n integer vectors of length n requires $n^{3-o(1)}$ time.
 84 This assumption is weaker than the standard APSP assumption that All-Pairs Shortest
 85 Paths in an n -vertex graph with integer weights (equivalent to *offline* $(\min, +)$ matrix–
 86 vector product) requires $n^{3-o(1)}$ time [12].
- 87 3. 3SUM (given n integers, do any three sum to zero?) requires $n^{2-o(1)}$ time [9].

¹ For this reason, an early unused name for retroactive data structures was “time-travel data structures”,
 as evidenced by the keyword list of [5].

88 This assumption is arguably the beginning of fine-grained complexity.
 89 Our lower bounds hold even if the retroactive data structure is amortized, and apply to
 90 randomized algorithms if the assumptions do.

91 Applied to time travel, our results show that making a change in the past and then
 92 jumping forward to the present should take roughly as much computation as the universe
 93 needed to originally advance through that much time. In other words, adding “Back To The
 94 Future” time travel to the universe requires the universe computation to slow down by a
 95 factor nearly linear in the lifetime of the universe, which seems computationally unrealistic.
 96 Here we assume the fine-grained physical Church–Turing Thesis described above, and ignore
 97 possible speedups from quantum computation (which remains an area for future research).
 98 If the universe is a parallel system of computers, then our worst-case instance consists of a
 99 proportional number of instances of our worst-case data structures. Our result can be seen
 100 as justification for the model of time travel implemented in the video game *Achron* (2011),
 101 where changing the past propagates a wave of consequential changes to the future, but the
 102 wave moves “slowly” (a constant factor faster than normal forward time travel).

103 1.2 Related Work

104 The original paper on retroactive data structures [5] proves an $\Omega(m)$ worst-case lower bound
 105 on the multiplicative overhead required for partial retroactivity, where m is the number of
 106 operations in the timeline [5, Theorem 2]. Asymptotically, this result is an improvement
 107 over our result, matching the $O(m)$ upper bound up to a constant factor instead of an
 108 $m^{o(1)}$ factor. The difference is in the model: the existing $\Omega(m)$ lower bound holds in the
 109 history-dependent algebraic-computation-tree model, the integer RAM, and generalized
 110 real RAM. (The reduction is from online polynomial evaluation, which provably requires
 111 linear time in these models.) But the result is not known to hold on the word RAM, for
 112 example, where the best known unconditional lower bound on retroactivity is $\Omega(\sqrt{m/\log m})$
 113 [5, Theorem 3].

114 By contrast, our results hold wherever the fine-grained assumptions above hold, which so
 115 far seems to include all “reasonable” models of computation. For example, the fastest known
 116 algorithm for 3SUM on a word RAM runs in $O\left(n^2\left(\frac{\lg \lg n}{\lg n}\right)^2\right)$ time [2]. This bound is roughly
 117 a logarithmic factor better than the fastest known algorithm for 3SUM on a real RAM, which
 118 runs in $O\left(n^2 \frac{\lg \lg n}{\lg n}\right)$ time [8], an improvement on the recent subquadratic breakthrough [11].
 119 But all of these algorithms are quadratic up to polylogarithmic factors.

120 On the other hand, in the decision tree model of computation, 3SUM can be solved in
 121 $O(n^{3/2}\sqrt{\log n})$ time [11], breaking the 3SUM conjecture. However, this model of computation
 122 is generally considered *unreasonable* because it allows the choice of algorithm to depend on the
 123 problem size n . Further, like most such decision-tree results, the proof is not “algorithmic”: the
 124 best known algorithm to compute which algorithm to run for a given n uses exponential time.

125 Our work is closely based on another paper proving a separation between partial and
 126 full retroactivity [4]. In particular, our fine-grained complexity assumptions are the same
 127 as theirs. Under these assumptions, Chen et al. [4] proved that the worst-case separation
 128 between a partially retroactive and a fully retroactive data structure for the same underlying
 129 problem is a multiplicative factor of $\Omega(m^{1/2-o(1)})$, which is essentially tight against the
 130 known upper bound of $O(\sqrt{m})$ [5].

131 This paper is not the first to consider applications of computational complexity to time
 132 travel. Aaronson, Bavarian, and Giusteri [1] show that consistent timelines from closed
 133 timelike curves (another common model for time travel in popular science fiction) require
 134 solving PSPACE-complete problems. This result suggests that that this model of time

135 travel, despite being plausible in existing models of physics, is not actually computationally
 136 feasible. Our result can be viewed as a complementary negative result for the “Back To The
 137 Future” model of time travel. (However, our result shows only a conjectured linear separation,
 138 whereas [1] shows a conjectured exponential separation.) Both negative results only apply
 139 to “long-distance” time travel; making changes in the recent past remains computationally
 140 feasible, leaving ample room for science fiction. Both results also leave open the possibility
 141 that a time traveler jumping forward simply does not experience time (their subjective time
 142 is frozen) while the universe rolls time forward.

143 1.3 Techniques

144 Our approach builds on previous work by Chen et al. [4], which proves an $\tilde{\Omega}(\sqrt{m})$ gap
 145 between partial and full retroactivity under the same fine-grained complexity assumptions,
 146 where m is the number of operations in the data structure’s timeline. This result implies an
 147 $\tilde{\Omega}(\sqrt{m})$ worst-case separation between a basic (nonretroactive) data structure and a fully
 148 retroactive version of that data structure. Our work improves on this separation in two ways:
 149 it provides a separation between a basic data structure and a *partially* retroactive version of
 150 that data structure, and it improves the separation from $\tilde{\Omega}(\sqrt{m})$ to $\tilde{\Omega}(m)$.

151 To achieve these improvements, we adapt Chen et al.’s constructions using the following
 152 techniques:

- 153 1. We define the “lazy” version of an arbitrary abstract data type (data structure interface),
 154 which allows us to move all queries to the end while still preserving their answers, making
 155 partial and full retroactivity essentially equivalent. We prove a general result (Theorem 8)
 156 which lets us translate gaps for full retroactivity into gaps for partial retroactivity.
- 157 2. We use fewer operations than Chen et al. by combining a long sequence of simple operations
 158 into one more complicated operation. In particular, Chen et al.’s data structures use
 159 one operation to set a single element of a list, while ours set the entire list in a single
 160 operation. At the cost of operations being more expensive, this reduces the number of
 161 operations performed. As a result, the same gap is larger when considered as a function
 162 of the number of operations.

163 Using Technique 1 alone, applying Theorem 8 to Chen et al.’s results, we obtain an $\tilde{\Omega}(\sqrt{m})$
 164 gap between nonretroactivity and partial retroactivity, which is interesting but not optimal.
 165 Using Technique 2 without Technique 1 does not achieve anything new, because Chen et al.
 166 use a large number of nonconsecutive query operations. To significantly reduce the number of
 167 operations, we need to batch the query operations into a single more complicated operation,
 168 which our lazy transform makes possible by moving them all to the end.

169 1.4 Outline

170 In Section 2, we provide the definitions needed for this paper, including fully and partially
 171 retroactive data structures, and the three fine-grained complexity assumptions.

172 In Section 3, we define lazy data structures, and use them to prove a result that converts
 173 gaps for full retroactivity into gaps for partial retroactivity. Applying this to Chen et al.’s
 174 work gives a $\Omega(\sqrt{m})$ gap between a base data structure and the partial retroactive version,
 175 under each of the three fine-grained complexity assumptions.

176 In the remaining sections, we prove a stronger $\Omega(m)$ slowdown for partial retroactivity
 177 under each of the same assumptions. Each proof is based on a construction due to Chen et
 178 al. [4], with Lazy applied. We also apply the second idea described above—merging consecutive
 179 operations into a single more complicated operation—which gives the improvement from $\Omega(\sqrt{m})$

180 to $\Omega(m)$. We are also able to make a few simplifications: first, we do not always need the full
 181 power of inserting queries and calling EVALUATE, and can use an ADT which can only query
 182 some simpler summary of past states. Second, Chen et al.'s lower bound from CircuitSAT is
 183 more complicated than necessary.

184 We prove this $\Omega(m)$ separation under Conjecture 1 about CircuitSAT in Section 4, under
 185 Conjecture 2 about Online (min, +) Product in Section 5, and finally under Conjecture 3
 186 about 3SUM in Section 6.

187 2 Definitions

188 In this section, we introduce a simple formalism for data structures and their problem/interface
 189 specifications (ADTs) that lets us to define general transformations on those specifications.
 190 This enables formal definitions of “a retroactive version of a data structure” as well as our
 191 Lazy transformation, which in turn help us state our results precisely.

192 ► **Definition 1 (ADT).** An *abstract data type (ADT)* \mathcal{A} consists of
 193 ■ A set \mathcal{U} of *update* operations, where each $u \in \mathcal{U}$ takes an argument in $\text{domain}(u)$ but
 194 does not return a value. (Instead, each update influences the results of future queries.)
 195 ■ A set \mathcal{Q} of *query* operations, which each $q \in \mathcal{Q}$ takes an argument in $\text{domain}(q)$ and
 196 returns a value in $\text{codomain}(q)$. (Queries cannot affect the result of future queries.)
 197 ■ A partial function \mathcal{A} specifying the result of a query operation given the sequence of
 198 prior calls to update operations. That is, given a sequence $\hat{u} = [u_1(x_1), \dots, u_k(x_k)]$ of
 199 calls to update operations and a call to a query operation $q(x)$, the ADT may specify the
 200 result $\mathcal{A}(\hat{u}, q(x))$ of $q(x)$ after calling precisely the operations in \hat{u} . Note that the result
 201 of $q(x)$ can depend on prior update operations, but not on prior query operations. Some
 202 behavior may be left *undefined* (meaning that any result is valid).²

203 For instance, the DICTIONARY ADT has two update operations INSERT and DELETE, and
 204 one query operation CONTAINS, all of which take an integer as an argument. The desired
 205 behavior is that CONTAINS(x) returns True if INSERT(x) has been called more recently than
 206 DELETE(x), and False otherwise. This ADT represents a set which starts empty, and can
 207 have elements inserted and deleted.

208 ► **Definition 2 (Data structure, implementation).** A *data structure* consists of some internal
 209 storage (e.g., a pointer machine or word RAM), and some algorithms acting on this internal
 210 storage, starting from a specified initial state. A data structure *implements* an ADT \mathcal{A} if
 211 the algorithms are labelled with the operations of \mathcal{A} , and have the correct behavior: suppose
 212 we run a sequence of operation calls on the data structure, ending in a query $q(x)$. Let \hat{u}
 213 be the sequence of these operations that are updates. Then the final query must return
 214 $\mathcal{A}(\hat{u}, q(x))$, if this value is defined.

215 An ADT may have many different implementations; for example, the DICTIONARY ADT
 216 can be implemented by hash tables or balanced search trees. The usual goal of data structure
 217 design is to invent faster implementations for a given ADT.

² For convenience in future ADT definitions, we allow a data structure to output anything when behavior is not defined, but in all of our results it is easy to test for undefined behavior, and thus one could modify the ADTs we define to have no undefined behavior by defining a “default” return value, without affecting our results.

218 **2.1 Retroactivity**

219 Now we can define partial and full retroactivity as transformations on ADTs.

220 ► **Definition 3** (Partially retroactive). Let \mathcal{A} be an ADT. We define an ADT called *partially retroactive* \mathcal{A} , written $\text{Partial-Retro}(\mathcal{A})$. It has the following operations:

- 222 ■ For each update operation u of \mathcal{A} , an update operation INSERT_u with domain $\mathbb{N} \times \text{domain}(u)$.
- 223 $\text{domain}(u)$.
- 224 ■ An update operation DELETE with domain \mathbb{N} .
- 225 ■ The same query operations as \mathcal{A} .

226 To describe the expected behavior, we imagine that a data structure maintains a sequence
 227 \hat{u} of calls to update operations of \mathcal{A} (the “timeline”). Then $\text{INSERT}_u(k, x)$ means we insert
 228 $u(x)$ at position k in this list, and $\text{DELETE}(k)$ means we remove the call at position k . A call
 229 to query operation $q(x)$ in $\text{Partial-Retro}(\mathcal{A})$ should return $\mathcal{A}(\hat{u}, q(x))$ if that value is defined;
 230 that is, we evaluate $q(x)$ at the end of the sequence of updates, which is thought of as the
 231 “present”.

232 ► **Definition 4** (Fully retroactive). Let \mathcal{A} be an ADT. We define an ADT called *fully retroactive* \mathcal{A} , written $\text{Full-Retro}(\mathcal{A})$. It has the following operations:

- 234 ■ For each update operation u of \mathcal{A} , an update operation INSERT_u with domain $\mathbb{N} \times \text{domain}(u)$.
- 235 $\text{domain}(u)$.
- 236 ■ An update operation DELETE with domain \mathbb{N} .
- 237 ■ For each query operation q of \mathcal{A} , a query operation QUERY_q with domain $\mathbb{N} \times \text{domain}(q)$
 238 and the same codomain as q .

239 As with partially retroactive \mathcal{A} , we imagine that a data structure maintains a sequence \hat{u}
 240 of calls to update operations of \mathcal{A} . The meanings of INSERT_u and DELETE are the same
 241 as above. A call $\text{QUERY}_q(k, x)$ in $\text{Full-Retro}(\mathcal{A})$ should return $\mathcal{A}(\hat{u}_{1..k}, q(x))$ if that value is
 242 defined, where $\hat{u}_{1..k}$ is the first k elements of \hat{u} ; that is, it should give the result of calling
 243 $q(x)$ after time k in the timeline.

244 ► **Lemma 5** ([5, Theorem 1]). *Suppose \mathcal{A} has an implementation in which each operation takes*
 245 *$O(T)$ time. Then $\text{Partial-Retro}(\mathcal{A})$ and $\text{Full-Retro}(\mathcal{A})$ have implementations in which each*
 246 *operation takes $O(mT)$ time, where m denotes the number of calls to update operations in*
 247 *the timeline. Furthermore, the $\text{Partial-Retro}(\mathcal{A})$ implementation can support query operations*
 248 *in the same time bound as \mathcal{A} .*

249 **Proof.** Both partial and full retroactivity can be achieved by the simple rollback method in
 250 which all operations and the changes they make to the data structure’s internal storage get
 251 recorded in a stack. These operations can then be “rolled back”, undoing their effect; the new
 252 update operation applied at the appropriate location; and then all rolled back operations
 253 re-applied. ◀

254 **2.2 Complexity Assumptions**

255 Our results rely on the same computational complexity assumptions used by Chen et al. [4]
 256 to show their gap between partial and full retroactivity. These assumptions are implied by
 257 (i.e. are weaker than) the three main conjectures used in fine-grained complexity — SETH,
 258 the APSP Conjecture, and the 3SUM Conjecture [14, 4] — respectively.

259 ► **Conjecture 1.** Time $2^{n-o(n)}$ is required to solve $\text{SIZE}(2^{o(n)})$ *CircuitSAT*: given an
 260 n -input circuit of size $2^{o(n)}$, decide whether it is satisfiable.

261 Boolean circuits are more general structures than Boolean formulas, making this conjecture
 262 weaker, and thus more believable than, the foundational Strong Exponential Time Hypothesis
 263 (SETH) [13].

► **Conjecture 2.** Time $n^{3-o(1)}$ is required to solve **Online (min, +) Product**: given an $n \times n$ integer matrix A , and n vectors v_1, \dots, v_n which are revealed one by one, compute each (min, +) product $A \diamond v_i$. The next vector v_{i+1} is revealed after outputting $A \diamond v_i$. The (min, +) product $A \diamond v$ is an n -component vector whose j th component is defined as

$$(A \diamond v)_j = \min_{k=1}^n (A_{j,k} + v_k).$$

264 The offline (and thus easier) version of Online (min, +) Product [12] is equivalent to the
 265 well-known all-pairs shortest path problem, which is commonly assumed hard in fine-grained
 266 complexity. Online (min, +) Product is also a generalization of Online Boolean Matrix–Vector
 267 Product, another problem commonly used in fine-grained complexity and not currently known
 268 to be equivalent to APSP [12].

269 ► **Conjecture 3.** Time $n^{2-o(1)}$ is required to solve **3SUM**: given three size- n sets A, B , and
 270 C of integers, decide whether there exists $(a, b, c) \in A \times B \times C$ such that $a + b + c = 0$.

271 The 3SUM conjecture was one of the first and remains one of the main conjectures in
 272 fine-grained complexity [9]. The version of 3SUM stated in Conjecture 3 is actually called
 273 3SUM' in [9], while Section 1.1 states the original 3SUM problem. But there is a simple
 274 linear-time reduction between 3SUM and 3SUM' [9, Theorem 3.1], so these two versions
 275 of the 3SUM conjecture are equivalent. We use the version stated in Conjecture 3 in our
 276 construction.

277 **3 Transforming Partially Retroactive Transformations into Fully** 278 **Retroactive Transformations**

279 In this section, we develop the first technique mentioned in Section 1.3. First we define
 280 a “lazy” version of an abstract data type (Definition 1) \mathcal{A} . Roughly speaking, $\text{Lazy}(\mathcal{A})$
 281 makes the queries of \mathcal{A} record their value but not return anything, and adds a new query
 282 operation EVALUATE to extract the recorded result of a query. This transformation allows us
 283 to simulate full retroactivity using partial retroactivity: insert a “query” that is actually an
 284 update operation at some point in the past, and then call EVALUATE in the present to read
 285 its result. This is the idea behind the main result of this section, Theorem 8.

286 ► **Definition 6 (Lazy).** Let \mathcal{A} be an ADT. We define an ADT called *lazy* \mathcal{A} , written $\text{Lazy}(\mathcal{A})$.
 287 It has the following operations:

- 288 ■ The same update operations as \mathcal{A} .
- 289 ■ For each query operation q of \mathcal{A} , an *update* operation q^\dagger with the same domain as q .
- 290 ■ A query operation EVALUATE with domain \mathbb{N} and codomain $\bigsqcup_{\text{query } q \text{ of } \mathcal{A}} \text{codomain}(q)$.

291 Suppose \hat{u}^\dagger is the sequence of calls to update operations of $\text{Lazy}(\mathcal{A})$, and suppose the k th
 292 call u_k^\dagger is a modified query $q^\dagger(x)$. Then $\text{EVALUATE}(k)$ is supposed to return what $q(x)$ would
 293 return in \mathcal{A} in the corresponding sequence of calls. Formally, let $\hat{u}_{1\dots k}$ be the sequence of
 294 update calls not of the form $q^\dagger(x)$ from the first k elements of \hat{u}^\dagger , so $\hat{u}_{1\dots k}$ is a sequence of
 295 calls to update operations of \mathcal{A} . Then $\text{Lazy}(\mathcal{A})(\hat{u}^\dagger, \text{EVALUATE}(k)) = \mathcal{A}(\hat{u}_{1\dots k}, q(x))$, provided
 296 this is defined. If the k th call u_k^\dagger is not of the form $q^\dagger(x)$, then the result of $\text{EVALUATE}(k)$ is
 297 undefined.

298 ► **Lemma 7.** *Suppose a list of m items can be maintained subject to looking up the i th item,
299 and appending or removing a new item at the end, in $O(\ell(m))$ time per operation. (For
300 word-RAM machines, $\ell(m) = 1$; for pointer machines, $\ell(m) = \log m$.)*

301 *Suppose \mathcal{A} has an implementation in which each operation f takes $O(T_f)$ time. Then
302 $\text{Lazy}(\mathcal{A})$ has an implementation in which EVALUATE takes $O(\ell(m))$ time, and each other
303 operation f or f^\dagger takes $O(T_f + \ell(m))$ time, where m is the total number of calls to operations.*

304 **Proof.** To achieve $\text{Lazy}(\mathcal{A})$ with the desired running times, we create an auxiliary list to
305 store results of queries. For a modified query call $q^\dagger(x)$ occurring as the i th update call, we
306 store the corresponding result of $q(x)$ in position i . Then $\text{EVALUATE}(i)$ can simply look up
307 the value stored at position i . The $O(\ell(m))$ overhead comes from maintaining and accessing
308 this list. ◀

309 Composing Lemmas 5 and 7, we obtain an implementation of $\text{Partial-Retro}(\text{Lazy}(\mathcal{A}))$ in
310 which EVALUATE takes time $O(\ell(m))$, and each other operation takes $O(m(T + \ell(m)))$ time.
311 We now show how a partially retroactive version of $\text{Lazy}(\mathcal{A})$ is able to efficiently simulate
312 a fully retroactive version of \mathcal{A} . The key idea is that the lazy version of a data structure
313 has effectively converted the queries into update operations which are then allowed to be
314 inserted at different times in the partially retroactive model.

315 ► **Theorem 8.** *If $\text{Partial-Retro}(\text{Lazy}(\mathcal{A}))$ has an implementation in which each operation
316 takes amortized time $O(T)$, then so does $\text{Full-Retro}(\mathcal{A})$.*

317 **Proof.** We will use the implementation \mathcal{D} of $\text{Partial-Retro}(\text{Lazy}(\mathcal{A}))$ to implement INSERT_u ,
318 DELETE , and QUERY_q with constant overhead. The operations available to \mathcal{D} are INSERT_u ,
319 $\text{INSERT}_{q^\dagger}$, DELETE , and EVALUATE . The collision of notation will not be a problem because
320 the functions that share a name are handled by calling their namesake.

321 Calls to INSERT_u and DELETE can simply be run on \mathcal{D} . On $\text{QUERY}_q(k, x)$,

- 322 1. Call $\text{INSERT}_{q^\dagger}(k + 1, x)$.
- 323 2. Call $\text{EVALUATE}(k + 1)$.
- 324 3. Call $\text{DELETE}(k + 1)$.
- 325 4. Output the result from step 2.

326 This new data structure uses \mathcal{D} to maintain a list \hat{u}^\dagger of calls to update operations of $\text{Lazy}(\mathcal{A})$,
327 but we make sure to immediately remove any calls to q^\dagger . Thus it equivalently maintains the
328 list \hat{u} of update calls to \mathcal{A} .

329 When we call $\text{QUERY}_q(k, x)$, we insert the operation $q^\dagger(x)$ into \hat{u}^\dagger at position $k + 1$, and
330 then use EVALUATE to extract the result of the corresponding $q(x)$ in \hat{u} . More formally, the
331 result is $\text{Lazy}(\mathcal{A})(\hat{u}^\dagger, \text{EVALUATE}(k + 1))$ (where \hat{u}^\dagger is taken in the middle of the call, so it
332 includes the q^\dagger), which by the definition of Lazy is $\mathcal{A}(\hat{u}_{1\dots k}, q(x))$, as desired. ◀

333 This result allows us to transform a separation between \mathcal{A} and $\text{Full-Retro}(\text{ADT})$ into a
334 separation between $\text{Lazy}(\mathcal{A})$ and $\text{Partial-Retro}(\text{Lazy}(\mathcal{A}))$. In particular, suppose \mathcal{A} has an $O(f)$
335 implementation but $\text{Full-Retro}(\mathcal{A})$ requires $\Omega(g)$. Then by Lemma 7, $\text{Lazy}(\mathcal{A})$ is $O(f + \ell(m))$,
336 and by the contrapositive of Theorem 8, $\text{Partial-Retro}(\text{Lazy}(\mathcal{A}))$ requires $\Omega(g)$. That is, a
337 slowdown for fully retroactive \mathcal{A} implies a slowdown for partially retroactive $\text{Lazy}(\mathcal{A})$. If
338 $\ell(m) = 1$ such as for the word-RAM model, this slowdown is as large as the original.

339 Assuming any one of Conjectures 1–3, Chen et al. [4] define an ADT \mathcal{A} with an $O(f)$
340 implementation and prove a lower bound of $O(m^{1/2-o(1)}f)$ on implementing $\text{Full-Retro}(\mathcal{A})$.
341 Assuming $\ell(m) = m^{o(1)}$, the generic result of Theorem 8 implies that $\text{Lazy}(\mathcal{A})$ has an
342 $O(f)$ -time implementation but $\text{Partial-Retro}(\text{Lazy}(\mathcal{A}))$ requires $\Omega(m^{1/2-o(1)}f)$, giving an
343 $\Omega(m^{1/2-o(1)})$ separation between an ADT and its partially retroactive version under each of

344 the three assumptions. In Sections 4, 5, and 6, we improve the separation to $m^{1-o(1)}$ when
 345 assuming Conjecture 1, 2, and 3 respectively.

346 **4 Lower Bound from SIZE($2^{o(n)}$) CircuitSAT**

347 In this section, we prove a conditional gap for partial retroactivity using the ADT *Circuit*
 348 *Counter*, which has the following operations and behavior:

- 349 ■ INITIALIZE(C): given a description of a circuit C of size $r = 2^{o(n)}$ which takes n inputs,
 350 remember C . This can only be called once, and must be the first operation (otherwise
 351 the behavior is undefined). We assume $r > n$ for convenience.
- 352 ■ SET(x): given an n -bit string x , set the *current string* to x .
- 353 ■ INCREMENT(): increment the current string as a binary number.
- 354 ■ QUERY(): output True if *any* past value of the current string satisfies C , and False
 355 otherwise.

356 ► **Lemma 9.** *There is an implementation of Circuit Counter in which each operation takes*
 357 *$2^{o(n)}$ time.*

358 **Proof.** The implementation will maintain what the current response to QUERY would be;
 359 initially this is False. The rest is straightforward:

- 360 ■ INITIALIZE(C) simply records C , taking time $O(r) = 2^{o(n)}$.
- 361 ■ SET(x) sets the current string to x , and evaluates C on it. If C is satisfied, set the
 362 response to True. The runtime is dominated by evaluating C , which takes time $O(r)$.
- 363 ■ INCREMENT() increments the current string, evaluates C on it, and sets the response to
 364 True if C is satisfied. This also takes time $O(r)$.
- 365 ■ QUERY() returns the current response, in time $O(1)$. ◀

366 ► **Theorem 10.** *There is a sequence of $\Theta(2^{n/2})$ operation calls for Partial-Retro(Circuit Counter)*
 367 *such that, assuming Conjecture 1, any implementation requires $2^{n-o(n)}$ total time, for an*
 368 *amortized lower bound of $n^{n/2-o(n)}$ time per operation.*

369 **Proof.** We will use a partially retroactive Circuit Counter to solve SIZE($2^{o(n)}$) CircuitSAT.
 370 Given a circuit C of size $r = 2^{o(n)}$, we run the following sequence of operations:

- 371 ■ INSERT_{INITIALIZE}(1, C)
- 372 ■ $2^{n/2}$ copies of INSERT_{INCREMENT}(2)
- 373 ■ For each binary string y of length $n/2$:
 374 ■ INSERT_{SET}(2, $y0^{n/2}$)
 375 ■ QUERY()
 376 ■ DELETE(2)

377 After the first three steps, the sequence of calls to the Circuit Counter is INITIALIZE(C) and
 378 then $2^{n/2}$ copies of INCREMENT(). For each y , we insert SET($y0^{n/2}$) (i.e. y followed by $n/2$
 379 zero bits) right after the INITIALIZE, and query at the end. Since the INCREMENTS count
 380 through all strings starting with y , the QUERY returns True if and only if any such string
 381 satisfies C . We then clean up the operation inserted before the next loop.

382 After all $2^{n/2}$ loops, we have tested every possible input to C : thus C is satisfiable if and
 383 only if any call to QUERY returned True. Conjecture 1 says that this whole algorithm must
 384 take $2^{n-o(n)}$ time. ◀

385 The implementation from Lemma 9 uses $O(2^{o(n)})$ time per operation and we have
 386 $m = \Theta(2^{n/2})$ operations, so the gap here is $2^{n/2-o(n)}/2^{o(n)} = 2^{n/2-o(n)} = m^{1-o(1)}$.

387 **5** Lower Bound from Online (min, +) Product

388 In this section, we prove a conditional gap for partial retroactivity using the ADT (min, +)
389 *Multiplier*, which has the following operations and behavior:

- 390 ■ SET-A(v): given an n -component row vector v , save it internally as a .
- 391 ■ SET-B(v): given an n -component column vector v , save it internally as b .
- 392 ■ QUERY(): output the list of $a \diamond b$ for all historical values of the pair (a, b) . Here $a \diamond b$ is
393 the (min, +) product $\min_i(a_i + b_i)$.

394 ► **Lemma 11.** *There is an implementation of (min, +) Multiplier in which SET-A and SET-B*
395 *take $O(n)$ time, and QUERY takes $O(m)$ time, where m is the number of calls to update*
396 *operations.*

397 **Proof.** This is straightforward; we maintain the list of what QUERY should output:

- 398 ■ SET-A(v) records v as a , computes $a \diamond b$, and appends it to the list. This takes time $O(n)$.
- 399 ■ SET-B(v) records v as b , computes $a \diamond b$, and appends it to the list. This also takes time
400 $O(n)$.
- 401 ■ QUERY() returns the list, which takes time $O(m)$ because it has length $O(m)$. ◀

402 ► **Theorem 12.** *There is a sequence of $\Theta(n)$ operation calls for Partial-Retro((min, +) Multiplier)*
403 *such that, assuming Conjecture 2, any implementation requires $n^{3-o(1)}$ total time, for an*
404 *amortized lower bound of $n^{2-o(1)}$ time per operation.*

405 **Proof.** We will use a partially retroactive (min, +) Multiplier to solve Online (min, +)
406 Product. Given the $n \times n$ matrix A and the n vectors v_i one by one, we run the following
407 sequence of operations:

- 408 ■ For each row A_j of A :
 - 409 ■ INSERT_{SET-A}(j, A_j)
- 410 ■ For each v_i :
 - 411 ■ INSERT_{SET-B}($1, v_i$)
 - 412 ■ QUERY(), and output the result
 - 413 ■ DELETE(1)

414 The first loop just sets up a sequence of n SET-A operations for the rows of A . The second
415 loop inserts SET-B(v_i) at the beginning. Now the historical values of (a, b) are precisely
416 (A_j, v_i) for each row A_j , so the result of the QUERY is $A \diamond v_i$. Finally it removes the SET-B
417 to prepare for the next iteration.

418 Conjecture 2 says that this whole algorithm must take $n^{3-o(1)}$ time. ◀

419 The sequence of operations in Theorem 12 has $m = \Theta(n)$ operations, in which case all
420 operation running times from Lemma 11 are $O(n)$. So the gap is $n^{1-o(1)} = m^{1-o(1)}$.

421 **6** Lower Bound from 3SUM

422 In this section, we prove a conditional gap for partial retroactivity using the ADT *3-Summer*,
423 which has the following operations and behavior:

- 424 ■ SET-A(L): given a length- \sqrt{n} list L , save it internally as A .
- 425 ■ SET-B(L): given a length- \sqrt{n} list L , save it internally as B .
- 426 ■ SET-C(L): given a length- n list L , save it internally as C .
- 427 ■ QUERY(): return True if at any time, there has been a triple $(a, b, c) \in A \times B \times C$ satisfying
428 $a + b + c = 0$. We call such a triple *good*.

429 Note that the internal lists A and B have length \sqrt{n} while C has length n .

430 ► **Lemma 13.** *There is an implementation of 3-Summer supporting each SET operation in*
431 *$O(n)$ time and QUERY in $O(1)$ time.*

432 **Proof.** This is a bit more complicated than the implementations from the previous two
433 sections. As usual, we maintain the value that QUERY should return. The key fact is that we
434 can test whether there is *currently* a good triple in time $O(n)$: make a hash table containing
435 the values of C , then iterate through all n values of $(a, b) \in A \times B$, and check whether $-a - b$
436 is in the hash table. The rest is straightforward:

- 437 ■ SET-A(L) records L as A , checks if there's a good triple, and sets the query value to True
438 if so. The runtime is dominated by checking for a good triple, which takes time $O(n)$.
- 439 ■ SET-B(L) records L as B , and then proceeds just like SET-A.
- 440 ■ SET-C(L) records L as C , and checks for a good triple. Both steps take $O(n)$ time.
- 441 ■ QUERY() returns the current query value, in time $o(1)$. ◀

442 ► **Theorem 14.** *There is a sequence of $\Theta(\sqrt{n})$ operation calls for Partial-Retro(3-Summer)*
443 *such that, assuming Conjecture 3, any implementation requires $n^{2-o(1)}$ total time, for an*
444 *amortized lower bound of $n^{3/2-o(1)}$ time per operation.*

445 **Proof.** We will use a partially retroactive 3-Summer to solve 3SUM. Given the lists A , B ,
446 and C of length n , we first divide A into \sqrt{n} lists $A_1, \dots, A_{\sqrt{n}}$ of length \sqrt{n} , and similarly for
447 B .³ Now we run the following sequence of operations on the partially retroactive 3-Summer:

- 448 ■ INSERT_{SET-C}(C)
- 449 ■ For j from 1 to \sqrt{n} :
 - 450 ■ INSERT_{SET-B}($j + 1, B_j$)
- 451 ■ For i from 1 to \sqrt{n} :
 - 452 ■ INSERT_{SET-A}($2, A_i$)
 - 453 ■ QUERY()
 - 454 ■ DELETE(2)

455 The first two steps set up operations to initialize C , and then set the internal list for B to
456 each section of B in order. Then, for each section A_i of A , we insert SET-A(A_i) right after
457 the SET-C. Now the historical values of (A, B) are (A_i, B_j) varying over j , so the QUERY
458 returns True if and only if there is a good triple $(a, b, c) \in A_i \times B \times C$. Finally, we remove
459 the SET-A to prepare for the next iteration.

460 Through the second loop, we check each section of A for a good pair. Ultimately, at least
461 one of the calls to QUERY returns True if and only if there is a good pair $(a, b, c) \in A \times B \times C$.
462 Conjecture 3 says that this whole algorithm must take $n^{2-o(1)}$ time. ◀

463 The gap between the retroactive lower bound in Theorem 14 and the nonretroactive
464 upper bound in Lemma 13 is $n^{1/2-o(1)}$. Because the sequence of operations in Theorem 14
465 uses $m = \Theta(\sqrt{n})$ operations, this gap is again $m^{1-o(1)}$.

466 7 Conclusion

467 This paper gives compelling evidence that there is no general way to make certain data
468 structures retroactive with significantly better running time than the naive rollback method.
469 This worst-case impossibility motivates the question of which problems *can* be made

³ For simplicity we're assuming n is a perfect square; otherwise one can take ceilings as appropriate.

470 retroactive with low overhead (say, a polylogarithmic factor), such as deques [5], priority
 471 queues [5, 6], predecessor [10], and data structures whose updates all commute[5].

472 On the other side, the L -evaluation framework of Chen et al. [4] appears to be a fairly
 473 general method for taking a problem with a computational lower bound and using it to
 474 generate a gap between partial and full retroactivity. Perhaps the methods in that paper and
 475 this one can be generalized to understand in a broader sense what makes data structures
 476 resistant to being made retroactive. This perspective may also be useful in the other direction,
 477 granting some intuition about what makes algorithmic problems computationally difficult.

478 Our applications to the computational cost of time travel fail to account for the physical
 479 world’s ability to perform quantum computation. It would be interesting to extend our
 480 results to obtain similar separations for quantum models of computation, where in particular
 481 search can be sped up by Grover’s algorithm. See [3] for quantum analogs to some of our
 482 fine-grained assumptions.

483 ——— References ———

- 484 **1** Scott Aaronson, Mohammad Bavarian, and Giulio G. Giusteri. Computability theory of
 485 closed timelike curves. Technical Report TR16-146, Electronic Colloquium on Computational
 486 Complexity, 2016. URL: <https://eccc.weizmann.ac.il/report/2016/146>.
- 487 **2** Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3SUM.
 488 *Algorithmica*, 50(4):584–596, 2007.
- 489 **3** Harry Buhrman, Subhasree Patro, and Florian Speelman. A framework of quantum Strong
 490 Exponential-Time Hypotheses. In Markus Bläser and Benjamin Monmege, editors, *Proceedings*
 491 *of the 38th International Symposium on Theoretical Aspects of Computer Science*, volume 187
 492 of *LIPICs*, pages 19:1–19:19, 2021. doi:10.4230/LIPICs.STACS.2021.19.
- 493 **4** Lijie Chen, Erik D. Demaine, Yuzhou Gu, Virginia Vassilevska Williams, Yinzhan Xu, and
 494 Yuancheng Yu. Nearly optimal separation between partially and fully retroactive data
 495 structures. In *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm*
 496 *Theory*, pages 33:1–33:12, Malmö, Sweden, June 2018.
- 497 **5** Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM*
 498 *Transactions on Algorithms*, 3(2): Article 13, May 2007.
- 499 **6** Erik D. Demaine, Tim Kaler, Quanquan Liu, Aaron Sidford, and Adam Yedidia.
 500 Polylogarithmic fully retroactive priority queues via hierarchical checkpointing. In *Proceedings*
 501 *of the 14th International Symposium on Algorithms and Data Structures*, pages 263–275,
 502 Victoria, Canada, August 2015.
- 503 **7** James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data
 504 structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
- 505 **8** Ari Freund. Improved subquadratic 3SUM. *Algorithmica*, 77(2):440–458, 2017. doi:10.1007/
 506 s00453-015-0079-6.
- 507 **9** Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational
 508 geometry. *Computational Geometry: Theory and Applications*, 5(3):165–185, 1995.
- 509 **10** Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar
 510 subdivisions. *ACM Transactions on Algorithms*, 5(3): Article 28, July 2009. doi:10.1145/
 511 1541885.1541889.
- 512 **11** Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *Journal of the*
 513 *ACM*, 65(4):22:1–22:25, 2018. doi:10.1145/3185378.
- 514 **12** Monika Henzinger, Sebastian Krinninger, Danupon Na Nongkai, and Thatchaphol Saranurak.
 515 Unifying and strengthening hardness for dynamic problems via the online matrix-vector
 516 multiplication conjecture. In *Proceedings of the 47th Annual Symposium on the Theory of*
 517 *Computing*, pages 21–30, Portland, OR, June 2015.
- 518 **13** Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly
 519 exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

- 520 14 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity.
521 In *Proceedings of the International Congress of Mathematicians*, pages 3447–3487, Rio de
522 Janeiro, Brazil, 2018. World Scientific.