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, 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.

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 evidence of such inconsistency under a ***fine-grained physical Church–Turing thesis***:

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

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

1.1   Our Results

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

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

Our lower bounds hold even if the retroactive data structure is amortized, and apply to randomized algorithms if the assumptions do.

---

[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].

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

1.2 Related Work

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

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

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

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

This paper is not the first to consider applications of computational complexity to time travel. Aaronson, Bavarian, and Giusteri [1] show that consistent timelines from closed timelike curves (another common model for time travel in popular science fiction) require solving PSPACE-complete problems. This result suggests that that this model of time travel, despite being plausible in existing models of physics, is not actually computationally feasible. Our result can be viewed as a complementary negative result for the "Back To The

Future" model of time travel. (However, our result shows only a conjectured linear separation, whereas [1] shows a conjectured exponential separation.) Both negative results only apply to "long-distance" time travel; making changes in the recent past remains computationally feasible, leaving ample room for science fiction. Both results also leave open the possibility that a time traveler jumping forward simply does not experience time (their subjective time is frozen) while the universe rolls time forward.

1.3 Techniques

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

To achieve these improvements, we adapt Chen et al.'s constructions using the following techniques:
1. We define the "lazy" version of an arbitrary abstract data type (data structure interface), which allows us to move all queries to the end while still preserving their answers, making partial and full retroactivity essentially equivalent. We prove a general result (Theorem 8) which lets us translate gaps for full retroactivity into gaps for partial retroactivity.
2. We use fewer operations than Chen et al. by combining a long sequence of simple operations into one more complicated operation. In particular, Chen et al.'s data structures use one operation to set a single element of a list, while ours set the entire list in a single operation. At the cost of operations being more expensive, this reduces the number of operations performed. As a result, the same gap is larger when considered as a function of the number of operations.

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

1.4 Outline

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

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

In the remaining sections, we prove a stronger $\Omega(m)$ slowdown for partial retroactivity under each of the same assumptions. Each proof is based on a construction due to Chen et al. [4], with Lazy applied. We also apply the second idea described above–merging consecutive operations into a single more complicated operation–which gives the improvement from $\Omega(\sqrt{m})$ to $\Omega(m)$. We are also able to make a few simplifications: first, we do not always need the full power of inserting queries and calling EVALUATE, and can use an ADT which can only query some simpler summary of past states. Second, Chen et al.'s lower bound from CircuitSAT is more complicated than necessary.

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

2 Definitions

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

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

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

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

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

2.1 Retroactivity

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

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

---

[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.

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

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

▶ **Definition 4** (Fully retroactive). Let $\mathcal{A}$ be an ADT. We define an ADT called ***fully retroactive $\mathcal{A}$***, written Full-Retro($\mathcal{A}$). It has the following operations:
- For each update operation u of $\mathcal{A}$, an update operation INSERT_u with domain $\mathbb{N} \times \text{domain}(u)$.
- An update operation DELETE with domain $\mathbb{N}$.
- For each query operation q of $\mathcal{A}$, a query operation QUERY_q with domain $\mathbb{N} \times \text{domain}(q)$ and the same codomain as q.

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

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

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

2.2 Complexity Assumptions

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

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

Boolean circuits are more general structures than Boolean formulas, making this conjecture weaker, and thus more believable than, the foundational Strong Exponential Time Hypothesis (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 $v_1, \ldots, v_n$ 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 jth component is defined as

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

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

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

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

3 Transforming Partially Retroactive Transformations into Fully Retroactive Transformations

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

▶ **Definition 6** (Lazy). Let $\mathcal{A}$ be an ADT. We define an ADT called ***lazy* $\mathcal{A}$**, written $\mathsf{Lazy}(\mathcal{A})$. It has the following operations:
- The same update operations as $\mathcal{A}$.
- For each query operation q of $\mathcal{A}$, an *update* operation $q^\dagger$ with the same domain as q.
- A query operation EVALUATE with domain $\mathbb{N}$ and codomain $\bigsqcup_{\text{query } q \text{ of } \mathcal{A}}$ codomain(q).

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

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

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

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

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

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

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

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

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

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

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

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

Assuming any one of Conjectures 1–3, Chen et al. [4] define an ADT $\mathcal{A}$ with an $O(f)$ implementation and prove a lower bound of $O(m^{1/2-o(1)}f)$ on implementing Full-Retro($\mathcal{A}$). Assuming $\ell(m) = m^{o(1)}$, the generic result of Theorem 8 implies that Lazy($\mathcal{A}$) has an $O(f)$-time implementation but Partial-Retro(Lazy($\mathcal{A}$)) requires $\Omega(m^{1/2-o(1)}f)$, giving an $\Omega(m^{1/2-o(1)})$ separation between an ADT and its partially retroactive version under each of the three assumptions. In Sections 4, 5, and 6, we improve the separation to $m^{1-o(1)}$ when assuming Conjecture 1, 2, and 3 respectively.

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

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

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

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

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

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

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

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

- INSERT$_{\text{INITIALIZE}}(1, C)$
- $2^{n/2}$ copies of INSERT$_{\text{INCREMENT}}(2)$
- For each binary string y of length $n/2$:
  - INSERT$_{\text{SET}}(2, y0^{n/2})$
  - QUERY()
  - DELETE(2)

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

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

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

5   Lower Bound from Online $(\min, +)$ Product

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

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

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

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

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

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

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

- For each row A_j of A:
    - INSERT$_{\text{SET-A}}(j, A_j)$
- For each v_i:
    - INSERT$_{\text{SET-B}}(1, v_i)$
    - QUERY(), and output the result
    - DELETE(1)

The first loop just sets up a sequence of n SET-A operations for the rows of A. The second loop inserts SET-B(v_i) at the beginning. Now the historical values of (a, b) are precisely (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 to prepare for the next iteration.

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

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

6   Lower Bound from 3SUM

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

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

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

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

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

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

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

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

- INSERT$_{\text{SET-C}}(C)$
- For j from 1 to $\sqrt{n}$:
  - INSERT$_{\text{SET-B}}(j + 1, B_j)$
- For i from 1 to $\sqrt{n}$:
  - INSERT$_{\text{SET-A}}(2, A_i)$
  - QUERY$()$
  - DELETE(2)

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

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

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

7   Conclusion

This paper gives compelling evidence that there is no general way to make certain data structures retroactive with significantly better running time than the naive rollback method. This worst-case impossibility motivates the question of which problems *can* be made retroactive with low overhead (say, a polylogarithmic factor), such as deques [5], priority queues [5, 6], predecessor [10], and data structures whose updates all commute[5].

---

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

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

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

References

1 Scott Aaronson, Mohammad Bavarian, and Giulio G. Giusteri. Computability theory of closed timelike curves. Technical Report TR16-146, Electronic Colloquium on Computational Complexity, 2016. URL: `https://eccc.weizmann.ac.il/report/2016/146`, `arXiv:TR16-146`.

2 Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2007.

3 Harry Buhrman, Subhasree Patro, and Florian Speelman. A framework of quantum Strong Exponential-Time Hypotheses. In Markus Bläser and Benjamin Monmege, editors, *Proceedings of the 38th International Symposium on Theoretical Aspects of Computer Science*, volume 187 of *LIPIcs*, pages 19:1–19:19, 2021. `doi:10.4230/LIPIcs.STACS.2021.19`.

4 Lijie Chen, Erik D. Demaine, Yuzhou Gu, Virginia Vassilevska Williams, Yinzhan Xu, and Yuancheng Yu. Nearly optimal separation between partially and fully retroactive data structures. In *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm Theory*, pages 33:1–33:12, Malmö, Sweden, June 2018.

5 Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Transactions on Algorithms*, 3(2): Article 13, May 2007.

6 Erik D. Demaine, Tim Kaler, Quanquan Liu, Aaron Sidf ord, and Adam Yedidia. Polylogarithmic fully retroactive priority queues via hierarchical checkpointing. In *Proceedings of the 14th International Symposium on Algorithms and Data Structures*, pages 263–275, Victoria, Canada, August 2015.

7 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

8 Ari Freund. Improved subquadratic 3SUM. *Algorithmica*, 77(2):440–458, 2017. `doi:10.1007/s00453-015-0079-6`.

9 Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry: Theory and Applications*, 5(3):165–185, 1995.

10 Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3): Article 28, July 2009. `doi:10.1145/1541885.1541889`.

11 Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *Journal of the ACM*, 65(4):22:1–22:25, 2018. `doi:10.1145/3185378`.

12 Monika Henzinger, Sebastian Krinninger, Danupon Na Nongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the 47th Annual Symposium on the Theory of Computing*, pages 21–30, Portland, OR, June 2015.

13 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

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