Partially Persistent Trees in Python

Abhishek Purushothama & Nicholas Elsasser

Agenda

- Motivation
- Foundation
- Implementation
- Performance
- Hash Consing for AMR
- Do we need persistence for Hash Consing?

Motivation

Hash Consing[1]

Hash Consing is “a well-known technique to share data that are structurally equivalent.”

Hash Consing[1]

Hash Consing is “a well-known technique to share data that are structurally equivalent.”

In data structures, *isomorphic* structures and sub-structures can be identified (popularly hashing) and a single copy can be reused, providing space efficiency. It has been mainly used in *implementation* of functional programming languages.

Hash Consing[1]

Hash Consing is “a well-known technique to share data that are structurally equivalent.”

In data structures, *isomorphic* structures and sub-structures can be identified (popularly hashing) and a single copy can be reused, providing space efficiency. It has been mainly used in *implementation* of functional programming languages.

Immutability is a requirement for applying this technique and hence *persistence* in data structures provide an efficient implementation for using Hash Consing.

Maximal sharing refers to a broader method for sharing data and computation.

Symbolic representations

It *may* be possible to extend this technique to symbolic representations in other areas such as computational linguistics.

An example of such symbolic representations is Abstract Meaning Representation(AMR)[2].

“He drives carelessly.” is represented as AMR,

```
(d / drive-01
   :ARG0 (h / he)
   :manner (c / care-04
              :polarity -))
```

Structurally speaking

Structurally, AMRs are *directed acyclic graph with labeled nodes and edges*.

Foundation

Partial persistence[3][4]

- Ephemeral
- Fat Node
- Path Copying
- Hybrid

Implementing Partially Persistent Trees in Python

Implementation - Node Persistence

- Persistence detail is handled by node
  - Generalize information and pointer fields w/ hash table

```python
@dataclass
class PersistentNode:
    key: int
    value: Any
    field: Dict[int, Any]
    version: int
```

Implementation - Node Persistence

- Persistence detail is handled by node
  - Generalize information and pointer fields w/ hash table
  - `update` – manage persistence locally, can return copy to updated node

```
@dataclass
class PersistentNode:
    key: int
    value: Any
    field: Dict[int, Any]
    version: int

def update(self, field_key: int, version: int) -> Optional[PersistentNode]:
```

Implementation - Node Persistence

- Persistence detail is handled by node
  - Generalize information and pointer fields w/ hash table
  - `update` – manage persistence locally, can return copy to updated node
  - `get` – handles versioned query for specific persistence implementation

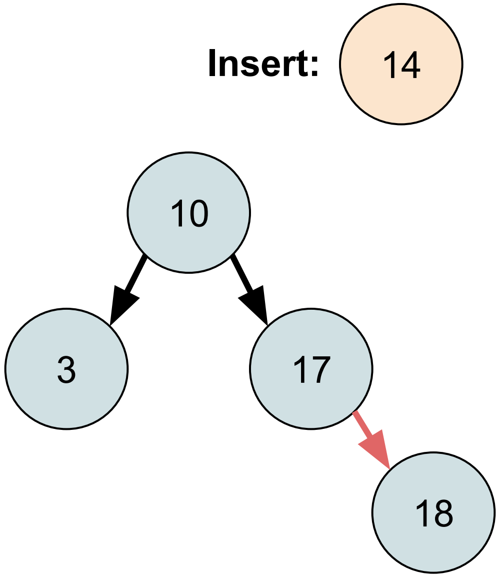
```
@dataclass
class PersistentNode:
    key: int
    value: Any
    field: Dict[int, Any]
    version: int

def update(self, field_key: int, version: int) -> Optional[PersistentNode]:

def get(self, field_key: int, version: int) -> Any:
```

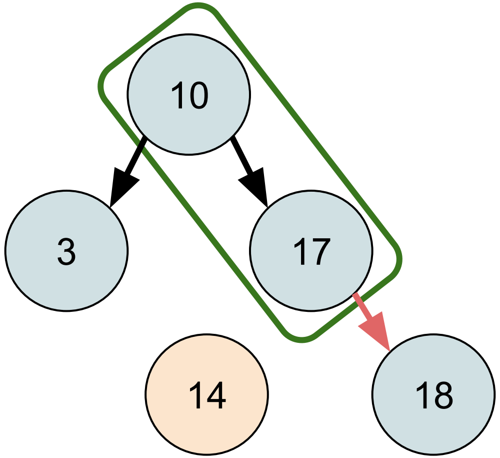
Implementation – Path Persistence

- Maintaining consistent path



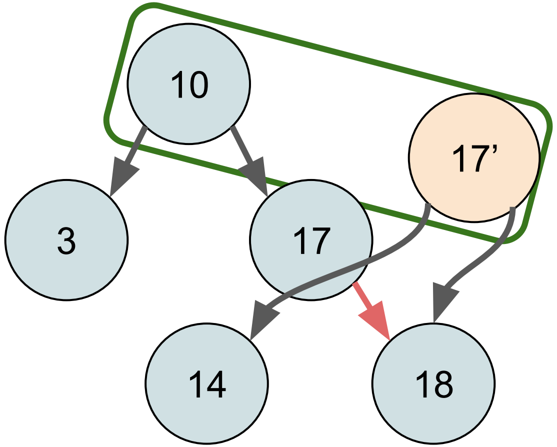
Implementation – Path Persistence

- Maintaining consistent path
  - Store access path v.s. Maintain parent references



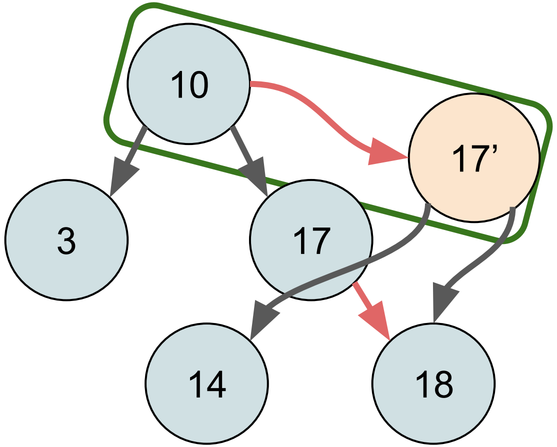
Implementation – Path Persistence

- Maintaining consistent path
  - Store access path v.s. Maintain parent references
  - When a node-copy occurs, update parent
  - Access path at target node stays consistent with updates



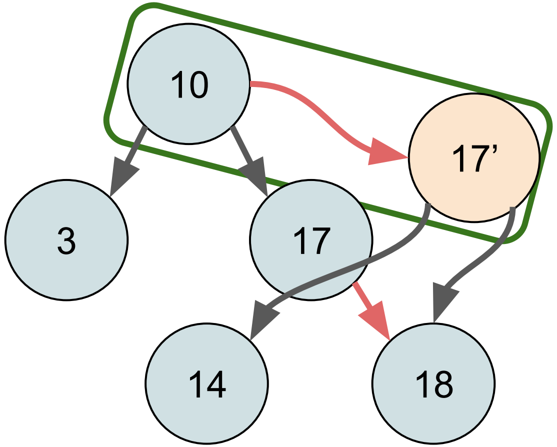
Implementation – Path Persistence

- Maintaining consistent path
  - Store access path v.s. Maintain parent references
  - When a node-copy occurs, update parent
  - Access path at target node stays consistent with updates
  - Continue until no copy or reach root
  - Path update invoked by tree author as necessary



Implementation – Path Persistence

- Maintaining consistent path
  - Store access path v.s. Maintain parent references
  - When a node-copy occurs, update parent
  - Access path at target node stays consistent with updates
  - Continue until no copy or reach root
  - Path update invoked by tree author as necessary



- Future work – make path updates implicit in node updates

Persistent Implementations

- Implement 4 generic tree node backends
  - Ephemeral – mutate fields in-place
  - Fat – each field maintains an ordered mutable list of versions
  - Copy – invoke a copy with updated field on each update
  - Hybrid – allow one update before invoking a copy

Persistent Implementations

- Implement 4 generic tree node backends
  - Ephemeral – mutate fields in-place
  - Fat – each field maintains an ordered mutable list of versions
  - Copy – invoke a copy with updated field on each update
  - Hybrid – allow one update before invoking a copy

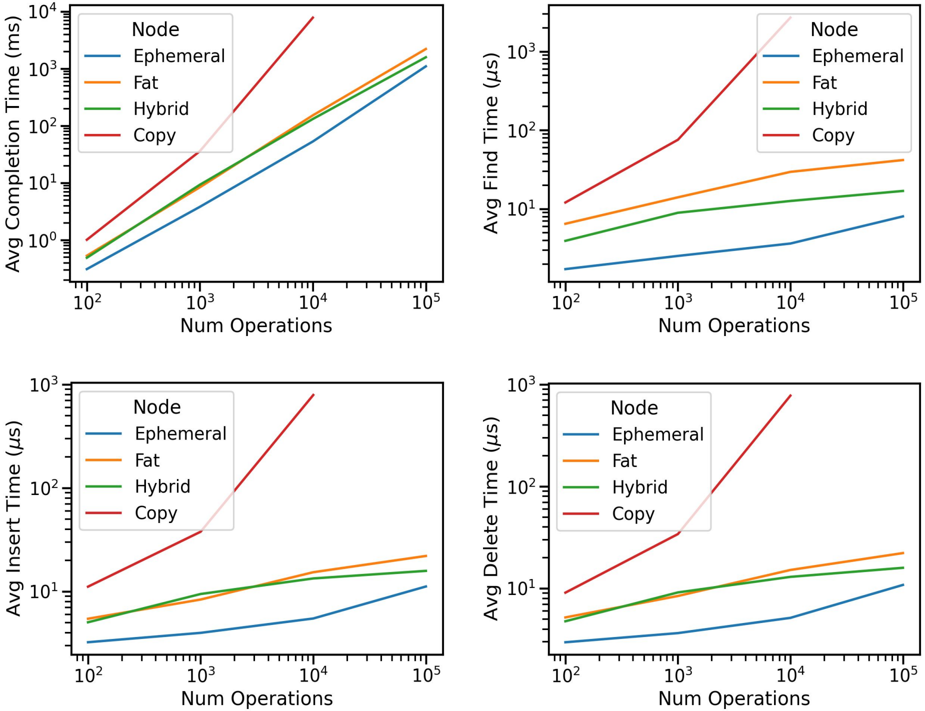
- One tree implementation can use all backends:
  - Unbalanced BST
  - Red-Black Tree
  - AMR Tree w/ maximal sharing

Performance

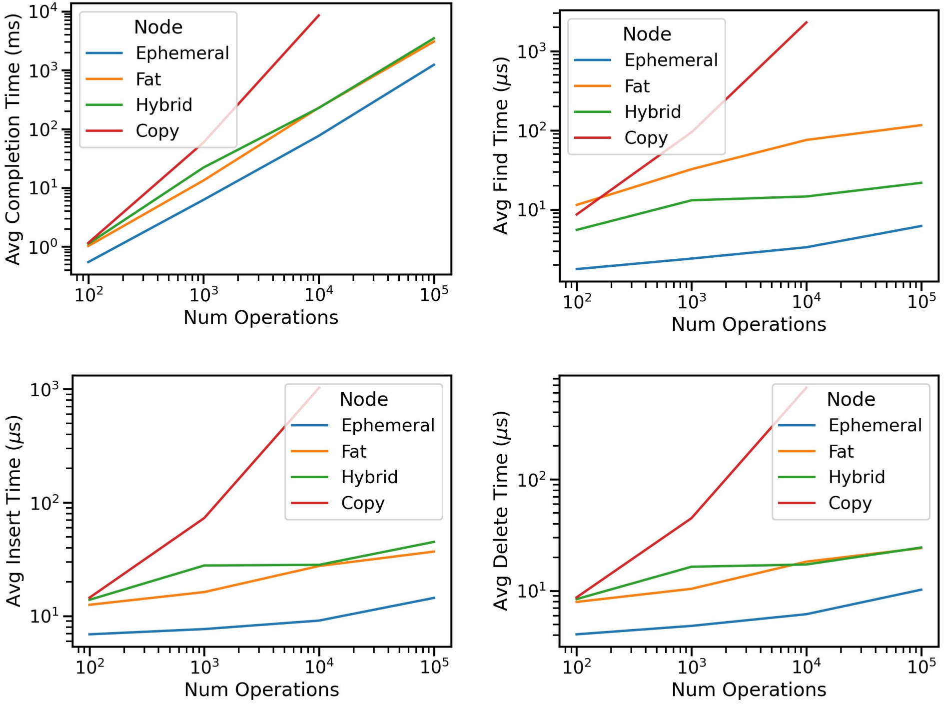
- Compare performance of node backends
  - Unbalanced BST and Red-Black tree support
    - `insert(self, key: int, value: Any)`
    - `delete(self, key: int) -> Any`
    - `find(self, key: int, version: int) -> Any`

- 100 trials
- Random sequence of 4*N* `insert`/`delete` and **keys** drawn from {0…*N*-1}
- On final version, execute `find` on 4*N* **keys** and **versions** drawn from {0…*N*-1} and {0…4N-1} respectively

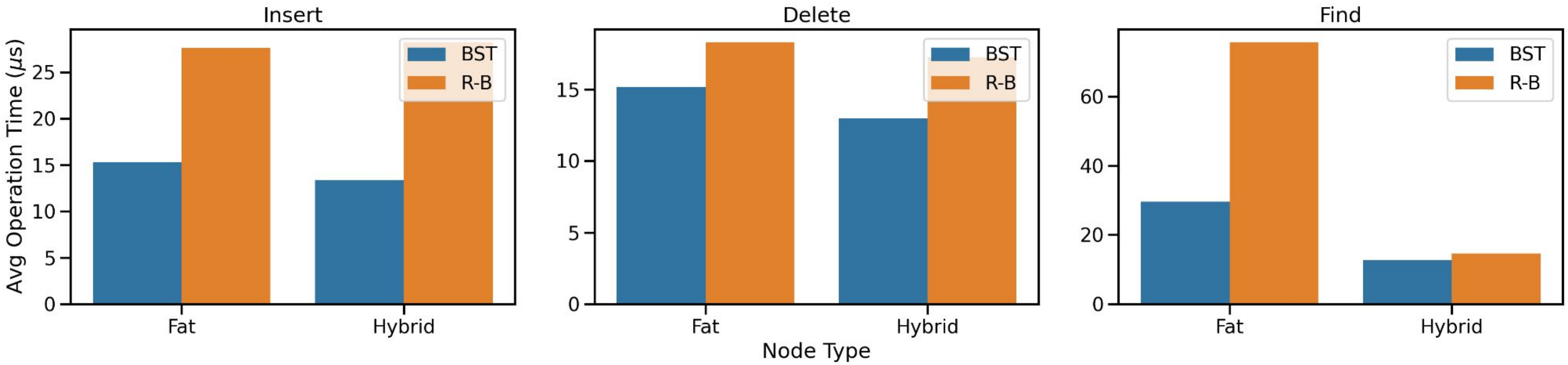
Unbalanced BST Performance



Red-Black Tree Performance



Backend v.s. Use-Case

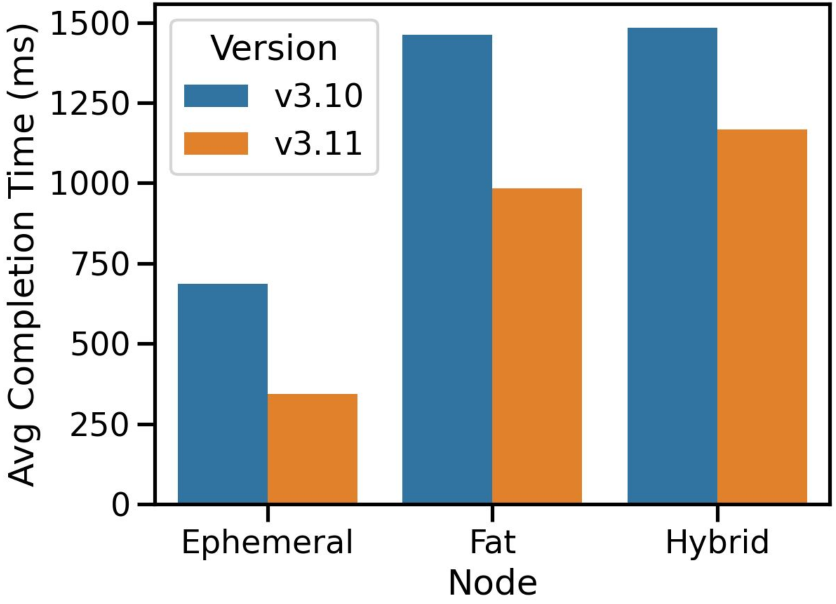


Python 3.11 v.s. 3.10

- Python 3.11 recently released
- Boasts faster interpreter
- Lift on our implementation?

Python 3.11 v.s. 3.10

- Python 3.11 recently released
- Boasts faster interpreter
- Lift on our implementation?



Hash Consing for AMR

The main library for working with AMR is *penman* package[5] in python.

It is relatively adhoc in its implementation, but it does hold to the AMR specifications.

Isomorphism is real in AMR too.

Isomorphism is real in AMR too.

Not all text is completely new.

Isomorphism is real in AMR too.

Not all text is completely new.

Not all meaning in text is completely new.

Isomorphism is real in AMR too.

Not all text is completely new.

Not all meaning in text is completely new.

Not all abstract meaning representations are completely new.

Isomorphism is real in AMR too.

In a analysis of dev split, consensus set (from AMR 3.0[6]) of 100 AMRs, made from Wall Street Journal and Weblog, there were ~190 instance of subtree(including leaf) duplication out of ~1500 (sub)trees.

Isomorphism is real in AMR too.

In a analysis of dev split, consensus set (from AMR 3.0[6]) of 100 AMRs, made from Wall Street Journal and Weblog, there were ~190 instance of subtree(including leaf) duplication out of ~1500 (sub)trees.

The meaning representation sub-structure for “country of South Korea” was present 4 times

```
(c / country
     :wiki "South_Korea"
     :name (s / name
          :op1 "South"
          :op2 "Korea"))
```

Implementation

We also implemented a sample bottom-up AMR serializer using hash-based maximal sharing.

Do we really need persistence to do ‘Hash Consing’?

Do we really need persistence to do ‘Hash Consing’?

Hash Consing requires

Do we really need persistence to do ‘Hash Consing’?

Hash Consing requires

- Immutability

Do we really need persistence to do ‘Hash Consing’?

Hash Consing requires

- Immutability
- Functional? “Consing”

Do we really need persistence to do ‘Hash Consing’?

Hash Consing requires

- Immutability
- Functional? “Consing”
    - Inductive definition

Do we really need persistence to do ‘Hash Consing’?

Hash Consing requires

- Immutability
- Functional? “Cons”ing
    - Inductive definition
    - Penman plan (grammar)

Do we really need persistence to do ‘Maximal sharing’?

Maybe not

Hash Consing vs Maximal Sharing

Just semantics?

Do we *really* need Hash Consing (or Maximal Sharing) for AMRs?

The total set of AMRs is about ~200,000 right now.

It’s practical impact may feel pennies in a game of thousand pounds.

“Take care of the pennies and the pounds will take care of themselves”

(attributed to Victorian era Lord Chesterfield and William Lowndes.)

Next step

1. Formal definition of Maximal Sharing for AMR
2. Implementation of Maximal Sharing de-serialization for AMR
3. Formal definition of Functional AMR
4. Implementation of Functional-ish AMR
5. Implementation of Functional-ish AMR de-serialization with Hash Consing

Acknowledgement

Thanks to Dr, Martha Palmer and CLEAR lab for providing access to AMR 3.0 release.

Thanks to Linguistic Data Consortium for the work on AMR 3.0 release.

Thanks

Questions

References

1. Goubault, J. (1994). Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. Journées Francophones des Langages Applicatifs (JFLA'93), pages 222–238.
2. Irene Langkilde and Kevin Knight. 1998. Generation that Exploits Corpus-Based Statistical Knowledge. In 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1, pages 704–710, Montreal, Quebec, Canada. Association for Computational Linguistics.
3. J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. 1986. Making data structures persistent. In Proceedings of the eighteenth annual ACM symposium on Theory of computing (STOC '86). Association for Computing Machinery, New York, NY, USA, 109–121. https://doi.org/10.1145/12130.12142
4. Okasaki, Chris. 1998. *Purely Functional Data Structures / Chris Okasaki*. Cambridge University Press.
5. Michael Wayne Goodman. 2020. Penman: An Open-Source Library and Tool for AMR Graphs. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 312–319, Online. Association for Computational Linguistics.
6. Knight, Kevin , et al. Abstract Meaning Representation (AMR) Annotation Release 3.0 LDC2020T02. Web Download. Philadelphia: Linguistic Data Consortium, 2020.