PersiFS$_2$: Structures for Efficient File System-Scale Partial Persistence

Austin Clements Dan Ports

Thursday, May 12, 2005

What is PersiFS?

- A persistent file system
- A persistent file system

What is PersiFS?

- A persistent file system (in the systems sense)
  - We'll call this durable
- A persistent file system

What is PersiFS?

- A persistent file system (in the systems sense)
  - We'll call this durable
- A persistent file system (in the data structures sense)

Persistence (in the data structures sense)

Definition

Partially persistent data structures allow queries on any previous version, but only allow modifications to the current version. Each modification produces a new version.

Fully persistent data structures allow modifications to previous versions. The history of the structure forms a tree.

Persistent File Systems

Definition

A *persistent file system* allows access to past versions of the file system.

Persistent File Systems

Definition

A *persistent file system* allows access to past versions of the file system.

Examples

- Version control systems like CVS, Subversion, etc.
- Snapshot and backup systems like AFS's `OldFiles`

PersiFS

PersiFS goes a few steps further

PersiFS

PersiFS goes a few steps further

- Continuously versioned so every modification is saved
- Real file system interface

PersiFS

PersiFS goes a few steps further

- Continuously versioned so every modification is saved
- Real file system interface

How do we do this efficiently, both time and space-wise?

A File System Data Structure

- Needs to support:
  - READ(*file*, *timestamp*, *offset*) $\rightarrow$ *substring*
  - MODIFY(*file*, *offset*, *new-substring*)
- Very large data sets — must be space-efficient
- Need fast access to both current and past revisions

What was PersiFS$_1$?

An implementation of PersiFS using silly, simple data structures from the systems world.

File System Structures

- Chunking
  - Divides data into content-sensitive chunks for efficient storage of modifications

File System Structures

- Chunking
  - Divides data into content-sensitive chunks for efficient storage of modifications
- Superblob
  - Stores chunks in a big append-only vector

File System Structures

- Chunking
  - Divides data into content-sensitive chunks for efficient storage of modifications
- Superblob
  - Stores chunks in a big append-only vector
- Metadata log
  - Stores sequence of file metadata changes over time (including pointers to file contents)

Content-sensitive Chunking

- Use a sliding Rabin fingerprint, $f(A)$
- When $f(A) \equiv 42 \pmod{2^{13}}$, draw a chunk boundary

...the way to hear the Rabbit say to itself, ‘Oh dear! Oh dea

- Modifications (even insertions) have only local effects on chunk contents

Metadata Log

Time	File	Modification
11:56	908	Chunks are now 56, 57, 94, 59
11:57	539	Chunks are now 80, 95
12:00	908	Chunks are now 56, 57, 96, 59

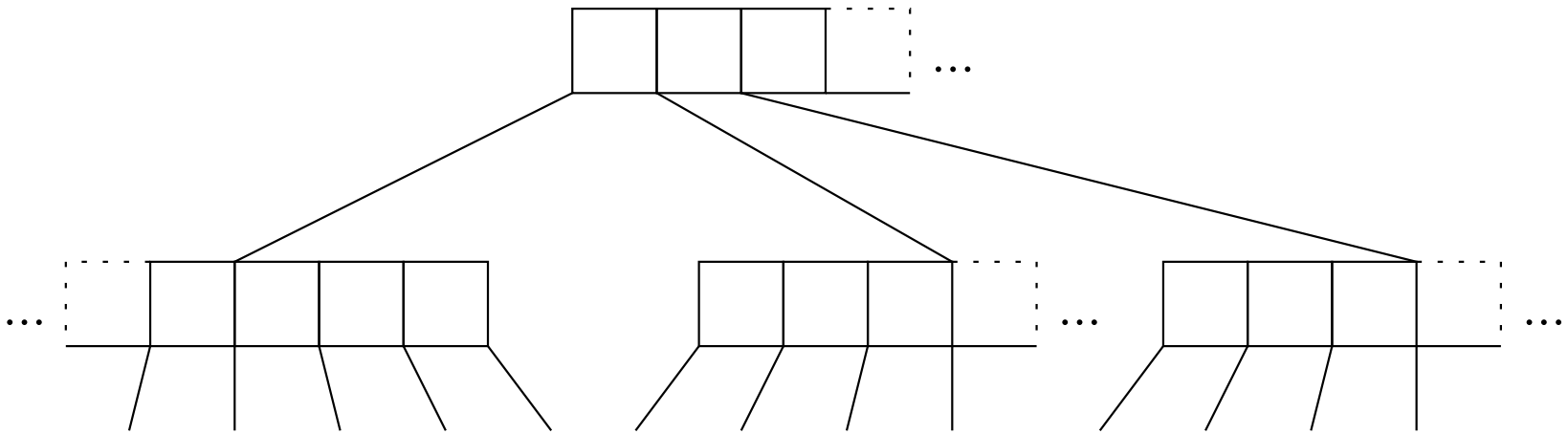
- $O(n)$ replay (and thus read) time
- Must periodically store large snapshots for reasonable replay
- $O(1)$ write time and space

What is PersiFS$_2$?

- The superblob can be improved
- The metadata log can be replaced

Model

- External memory model
- Need partial persistence
- Start with a B$^+$-tree



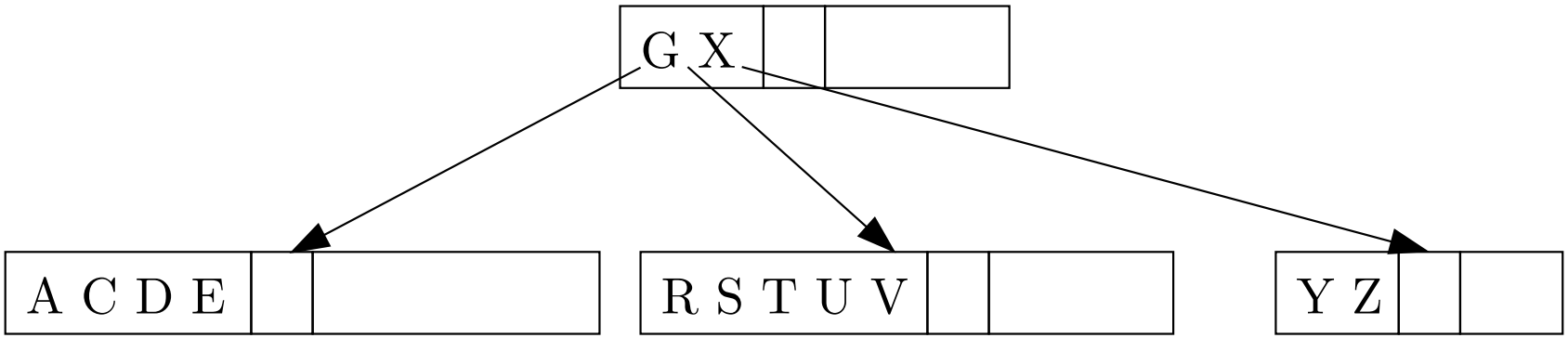
Chunk Fusion

- Utilizes regular B$^+$-tree to store fingerprint-to-address mapping
- Chunks with identical content can be fused and only stored once in the super blob
- $O(\log_{B+1} n)$ memory transfers for write
- $O(1)$ for read (unaffected by fusion)
- Potentially massive space savings at very little potential space cost

A Persistent B$^+$-tree

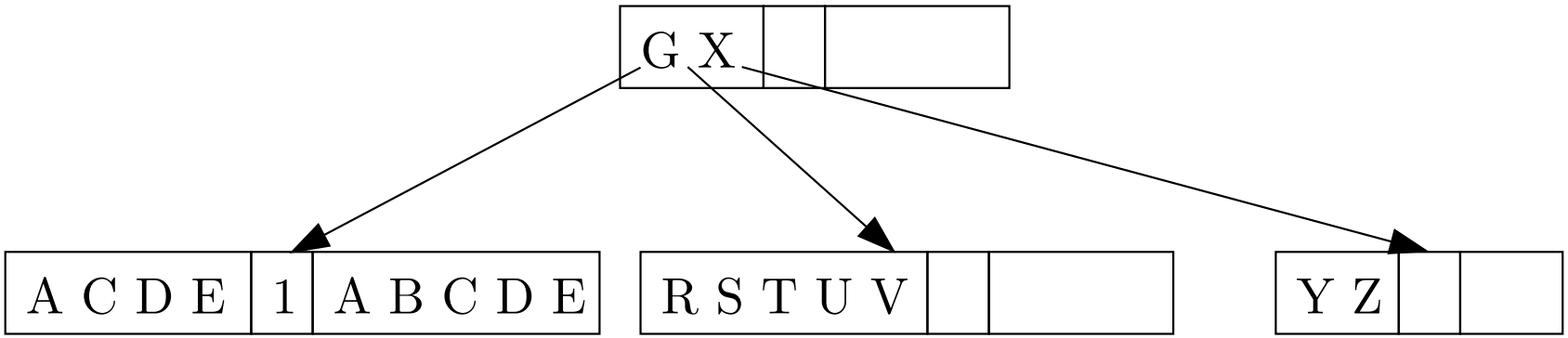
- INSERT(*key*, *value*)
- SEARCH(*key*, *timestamp*) $\rightarrow \langle key, value \rangle$
  - Exact key match or predecessor query
- DELETE(*key*)
- COMMIT() $\rightarrow$ *timestamp*
  - Allows multiple modifications grouped under a single timestamp
  - Grouping conceals *unnecessary* states (for efficiency), and *inconsistent* states (for correctness)

Persistifying a B$^+$-tree



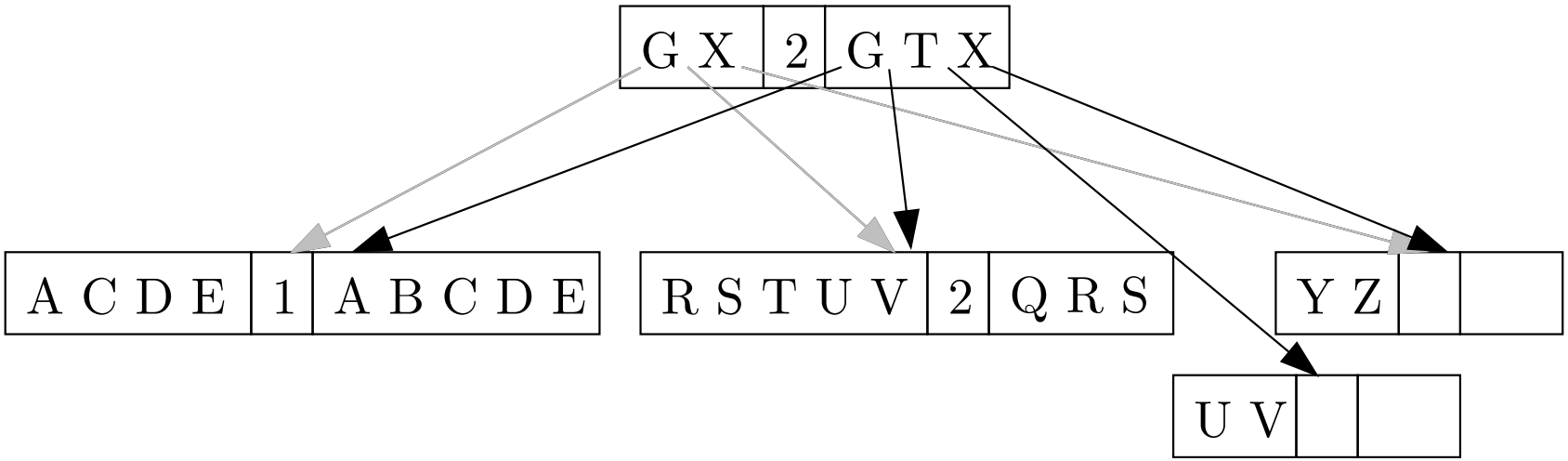
- Similar to "modification box" approach by Sleator and Tarjan
- Nodes may store a second, modified copy with some version
- If mod box is full, create a new node and fix parent's link

Persistifying a B$^+$-tree



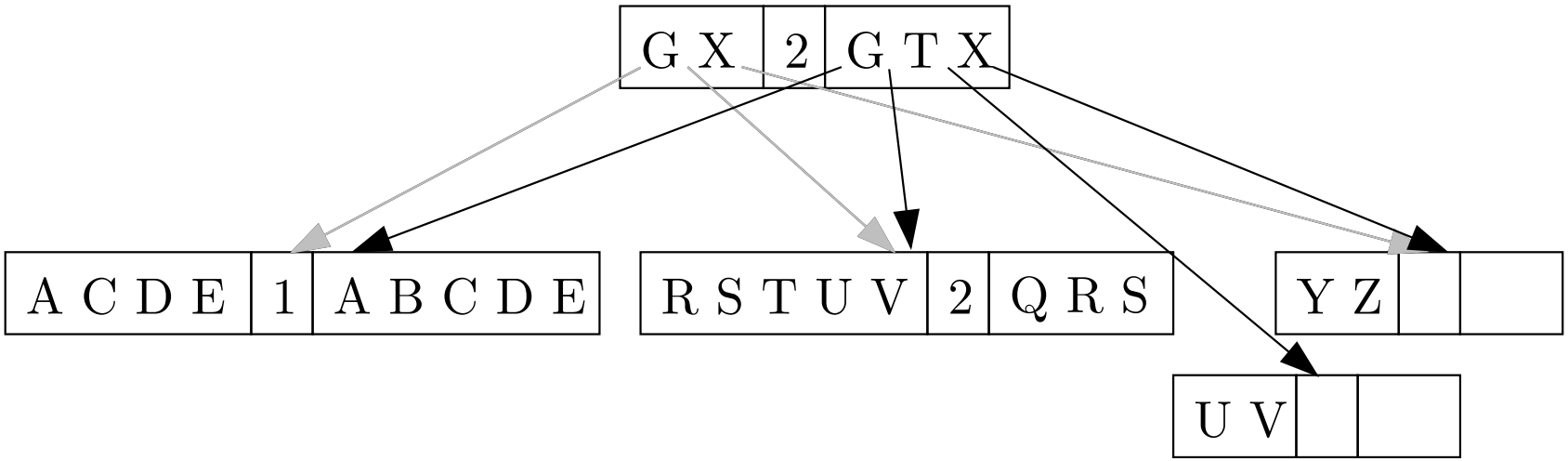
- Similar to "modification box" approach by Sleator and Tarjan
- Nodes may store a second, modified copy with some version
- If mod box is full, create a new node and fix parent's link

Persistifying a B$^+$-tree



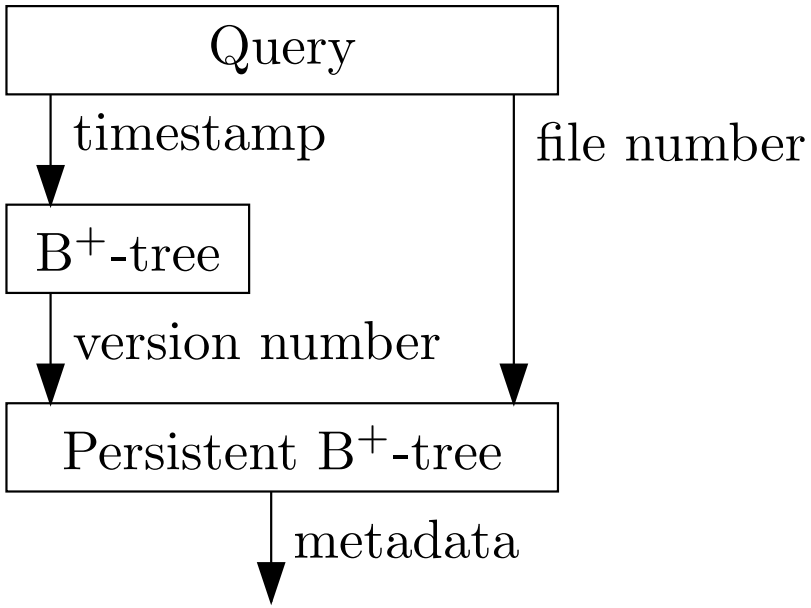
- Similar to "modification box" approach by Sleator and Tarjan
- Nodes may store a second, modified copy with some version
- If mod box is full, create a new node and fix parent's link

Persistifying a B$^+$-tree



- $O(\log_{B+1} n)$ memory transfers for read and write
- $O(1)$ additional space per modification

Replacing the Metadata Log



Results

- Chunk fusion is a clear win
  - Potentially large space savings with minimal cost
- Metadata log vs. arborescent metadata map: less clear
  - Depends on filesystem usage patterns
  - e.g. metadata log snapshot frequency vs. usage

Questions?