“But I also think that the "we write meta-data synchronously, but then the actual data shows up at some random later time" is just crazy talk. That's simply insane. It guarantees that there will be huge windows of times where data simply will be lost if something bad happens.

-- Linus Torvalds on filesystems
B-trees are datastructures which implement the **map** abstract datatype (find, insert, delete etc.)
A B-tree is a form of search tree.

A search tree is st. the keys in any subtree are bounded by [...keys in parent...]

Given a key, you can easily locate a (key,value) in a leaf by following the appropriate pointers and this **allows efficient implementation of map ops** such as find
A B-tree is a **balanced** search tree with **min and max size constraints** on nodes (nodes can be partially full).
Compared to OCaml's standard maps?

**Functor Map.Make**

```ocaml
module Make:
  functor (Ord : OrderedType) -> S with type key = Ord.t
  Functor building an implementation of the map structure given a totally ordered type.
```

✘ B-trees are n-ary trees, and the **code is more complicated**
   they also have **greater space overhead**

✔ Tree balancing ops (rotating, splitting etc) can be costly
   B-trees try to **minimize this work**
   eg insert into non-full nodes without doing any rebalancing

✔ B-trees are also **tuned to block devices:**
   choose max node size st. **every node fits into a single on-disk block**
B-tree usage

B-trees are widely used in **databases**
   such as Oracle, SQLServer, PostgreSQL... all of them?
   to provide fast access to large indexed (or key/value) data

B-trees are also used in modern **filesystems**
   such as HFS, HFS+, NTFS, jfs2, ext4, reiser4 and btrfs
   to support features like snapshots etc

Quick calculation for an **int -> int** map:
   assuming: 64 bit (8 byte) ints, blk size 4096 bytes
   have >16M (k,v) bindings, >256 MB of data, in a tree of height 3

If we **cache the top two layers** (root and children, 1MB)
   at most 1 block read to locate any (k,v) binding
Implementation in OCaml

Core code developed in Isabelle/HOL to allow formal verification

Interesting aspects:
  novel design
    to allow certain features (see later), and for correctness
  small step, framestack-based operational semantics
    (for concurrency and atomicity modelling)
  state-passing style, monad for state and error

Code then extracted to OCaml and wrapped in an OCaml-friendly API
OCaml, (int->int) map example usage

(* create and init store, write some values, and close *)
let do_write () = (  
  Printf.printf "Executing %d writes...\n" max;  
  print_endline "Writing...";  
  (* create and initialize *)  
  let s = ref (from_file ~fn ~create: true ~init: true) in  
  (* get map operations *)  
  let map_ops = imperative_map_ops s in  
  (* write values *)  
  for x = 1 to max do  
    map_ops.insert (k x) (v x);  
  done;  
  close !s;  
())
Open existing store and delete some entries

(* open store, delete some values, and close *)
let do_delete () = (  
  print_endline "Deleting...";
  let s = ref (from_file ~fn ~create:false ~init:false) in
  let map_ops = imperative_map_ops s in
  for x=100 to 200 do
    map_ops.delete (k x);
  done;
  close !s;
)
let do_full_check () = (  
    print_endline "Full check...";
    let s = ref (from_file ~fn ~create:false ~init:false) in
    let map_ops = imperative_map_ops s in
    for x = 1 to max do
      if (100 <= x && x <= 200) then
        assert(map_ops.find (k x) = None)
      else
        assert(map_ops.find (k x) = Some(v x))
    done;
    close !s)
Quick demo
Quick demo

$ src $ time ./ii_example.native
Executing 10000 writes...
Writing...
Deleting...
Checking...
Full check...

real  0m0.941s
user  0m0.756s
sys   0m0.168s

Is this good? Nothing really to compare against, and see next slide.
Quick demo

$ src $ time ./ii_example.native
Executing 10000 writes...
Writing...
Deleting...
Checking...
Full check...

real 0m0.941s
user 0m0.756s
sys 0m0.168s

# expected size 16B * 10k = 160kB (+tree overhead)? why 79MB?
$ src $ ls -alh btree.store
-rw-r----- 1 tr61 tr61 79M Sep 1 19:26 btree.store
Persistent datastructures

Don't confuse **persistent datastructures** with **persistent storage**!

A **persistent datastructure** (eg OCaml's maps)
   allows access to previous versions when modified

This library provides a B-tree **persistent datastructure**
   backed by **persistent storage** (this scheme is similar to "copy-on-write")

Expected store size? \(160\text{kB} \times 10k = 1600 \text{ MB} = 1.6 \text{ GB}\)
   so why only 76MB?

In real use, **caching** would reduce the number of on-disk states
   an explicit API **sync** op would force cache flush and write a state to disk
   and **atomicity** via on-disk pointer swinging
Take away point

A fast (maybe), correct (hopefully) CoW B-tree library in OCaml suitable for storing and accessing large, indexed data.
The bigger picture: "Future filesystems"

Project funded by EPSRC and Microsoft Research (PhD student)

"Formal methods applied to filesystems"
(specification and implementation)

Main goal: to be able to write correct programs
that use the filesystem or other persistent storage eg block dev

Filesystem specification: see the paper on SibylFS, SOSP'15

Filesystem implementation: currently writing ImpFS
the B-tree library is a key component
Questions?
Extra slide: components

- block device (eg raw, or backed by a file, or a network connection etc)
- store (above blk dev; keeps track of free blocks)
- btree API, including "bindings" and "insert many"
- marshalling/on-disk layout, currently courtesy of Jane Street's binprot
- LRU caching (eg above blk dev, store, btree etc)

Other interesting points

- components are flexible: assemble your own stack with shared caches, or disjoint caches etc
- testing via wf assertion checking and exhaustive state space exploration
| End |