DB101 – Foster b-trees

Goetz Graefe – Madison, Wis.

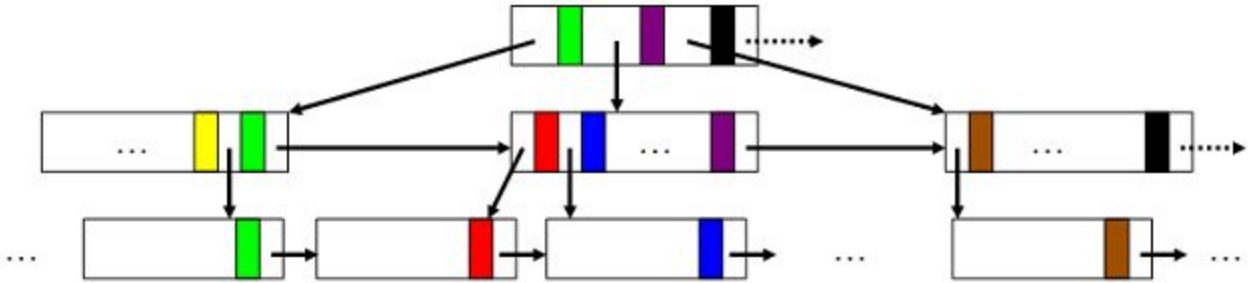
Overview

- Context:
  - transactional guarantees for database b-trees;
  - data in leaf nodes, branch keys in branch nodes;
  - low-level & high-level concurrency control (latching & locking);
  - high concurrency and contention; and
  - write-optimized page moves + in-buffer pointer swizzling.
- Goals:
  - minimal concurrency control needs for the data structure;
  - efficient migration of nodes to new storage locations; and
  - support for continuous and comprehensive self-testing.

Results

- B-tree without sibling pointers
  - Local temporary overflow nodes
  - Move key-pointer pair from “foster parent” to permanent parent
- Efficient pointer swizzling, page moves, node splits, node deletions
  - Move key-pointer pair from permanent parent to foster parent
- The one-two-three of the foster tree:
  - one pointer per node;
  - two fence keys per node (and two latches per system transaction);
  - three system transactions per structure change; and
  - four system transactions to change the b-tree height.

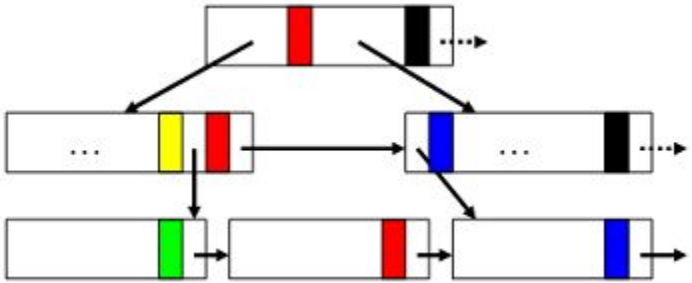
B^{link}-trees



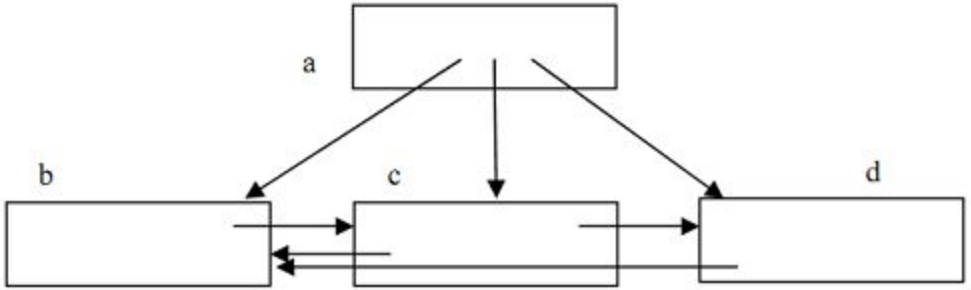
B^{link}-tree: intermediate state

Two latches each to …

1. allocate & format an overflow node
2. load balancing
3. post key-pointer pair in the parent

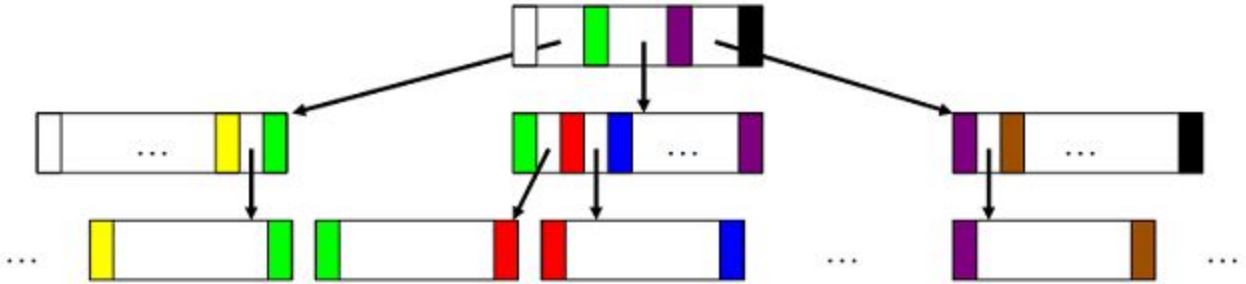


Inconsistencies in b-trees



Fence keys

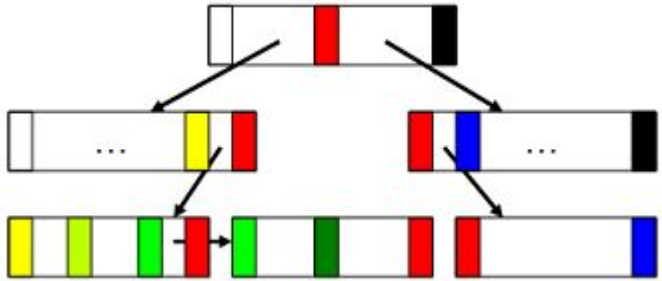
Write-optimized b-trees with fence keys

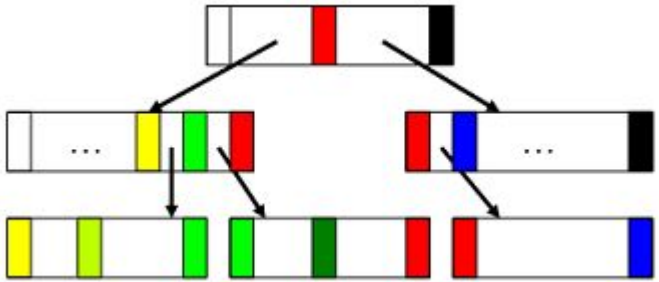


Foster b-tree: intermediate & final state

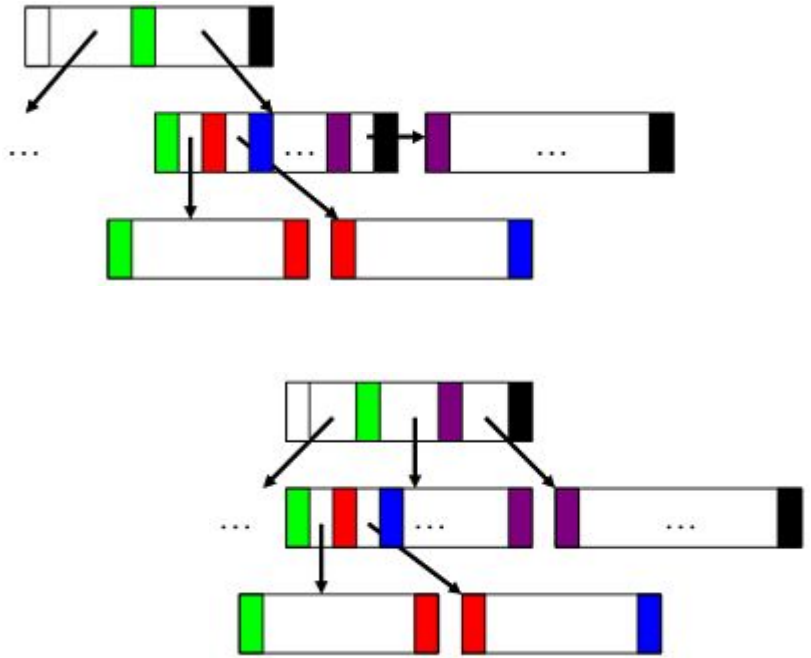
Two latches each to …

1. allocate & format overflow node
2. load balancing
3. move key-pointer pair from foster parent to permanent parent: "adoption"

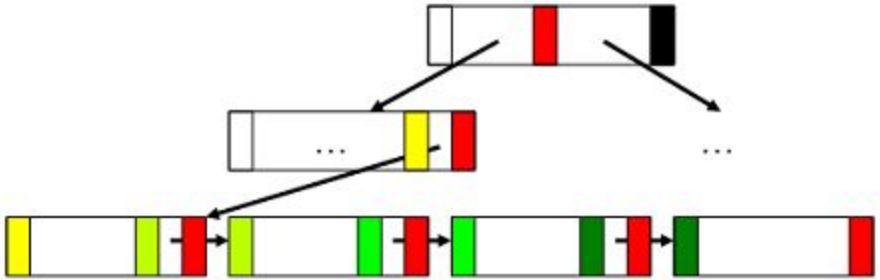




Branch node with foster child & adoption



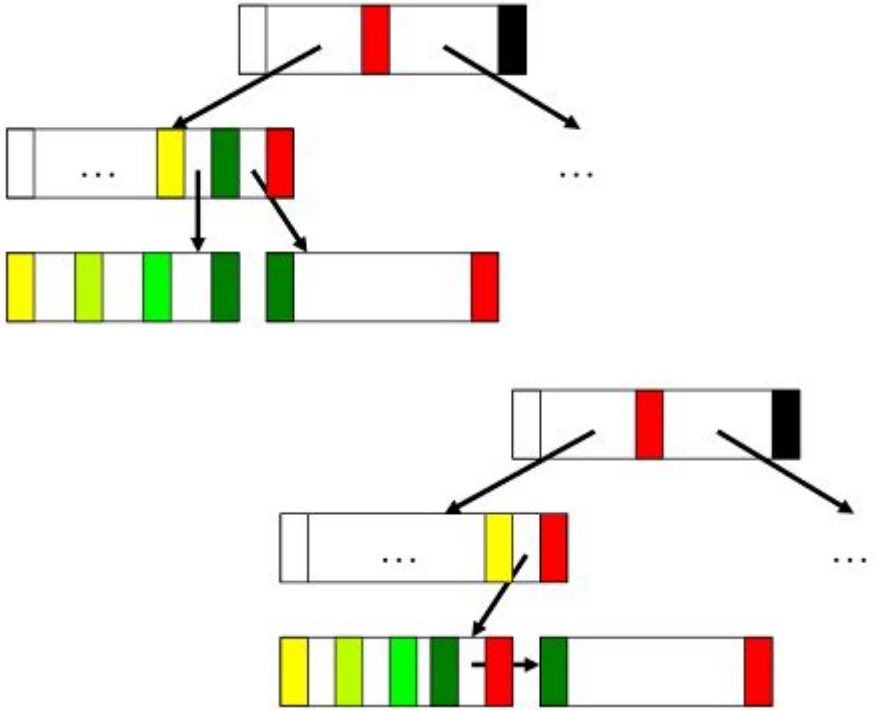
A chain of foster children



Load balancing

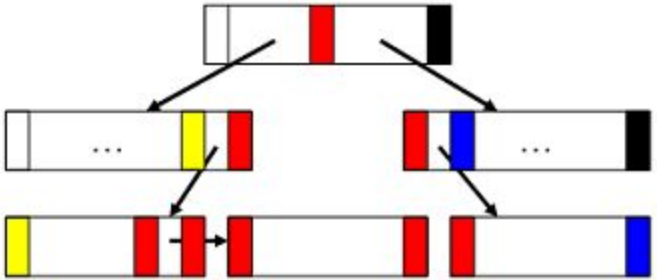
Two latches each to …

1. Move key-pointer pair to foster parent
2. Load balancing
3. Adoption



After page allocation, before load balancing

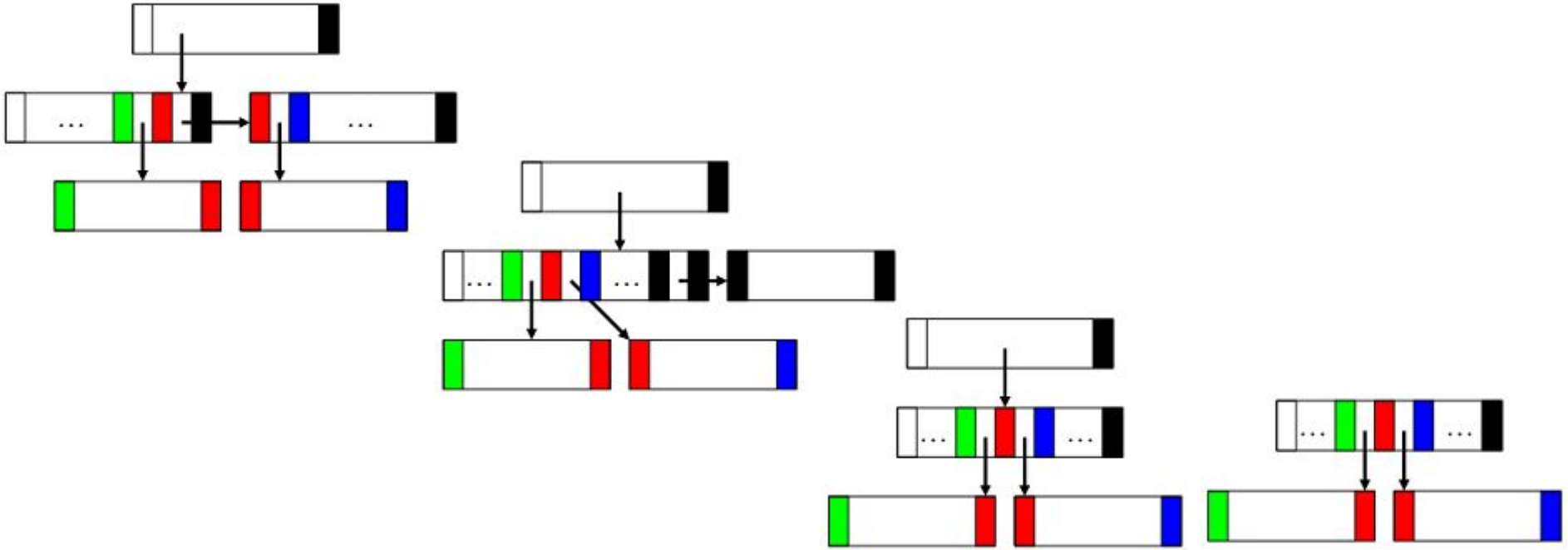
Allocation ⇒ format empty



Node split and page allocation in detail

1. A root-to-leaf search on behalf of an insertion finds a node to be full and marks it for subsequent splitting. Any subsequent operation might split the node when convenient, e.g., with respect to latch acquisition.
2. A new node is allocated, formatted with an empty key range, and becomes a foster child of the overflowing node. The foster key is the high fence key.
3. Load balancing between the overflowing node and the new node, with appropriate adjustment of the foster key and the foster child's low fence key.
4. Adoption of the new node by the permanent parent, i.e., moving foster key and pointer from the foster parent to the permanent parent.
5. Reorganization (compression by additional prefix truncation exploiting the tightened key range) in the old, formerly overflowing node.
6. Reorganization (compression by additional prefix truncation using a key range smaller than in the old, formerly overflowing node) in the newly allocated node.

Node removal, page release



Summary: foster b-trees

- Minimal concurrency control needs for the data structure
- Efficient migration of nodes to new storage locations
- Support for continuous and comprehensive self-testing

- A single (incoming) pointer per node at all times
- Moving key-pointer pairs between parent and child
- Fence keys + “foster keys”