nbtree:
An architectural perspective

PGCon 2019 — May 30, 2019

https://speakerdeck.com/peterg/nbtree-arch-pgcon

Peter Geoghegan

@petervgeoghegan

https://speakerdeck.com/peterg/nbtree-arch-pgcon

My perspective

- Good mental model important for working on the nbtree code.

  - Perhaps this talk will make that easier.

- Must approximate reality, while leaving out inessential details that hinder understanding.

- PostgreSQL 12 work will be discussed along the way.

Overview

1. **Big picture with B-Trees**

   What’s the point of this “high key” business, anyway?

2. **Seeing the forest for the trees**

   Reasoning about nbtree invariants when designing enhancements.

3. **A place for everything, and everything in its place**

   How reliably unique keys simplify many things.

4. **Future work**

   Outlook for future improvements.

Big picture with B-Trees

- Page splits add new pages.
- **Recursive** growth — page splits occur in leaf pages that fill with tuples pointing to table, and cascade upwards to maintain tree.
- Actually bush-like — very short, and very, very wide.
    - New levels added to tree at **logarithmic** intervals, during root page split.
- Just a few *localized* atomic operations that affect only a few pages at a time used for *everything*.

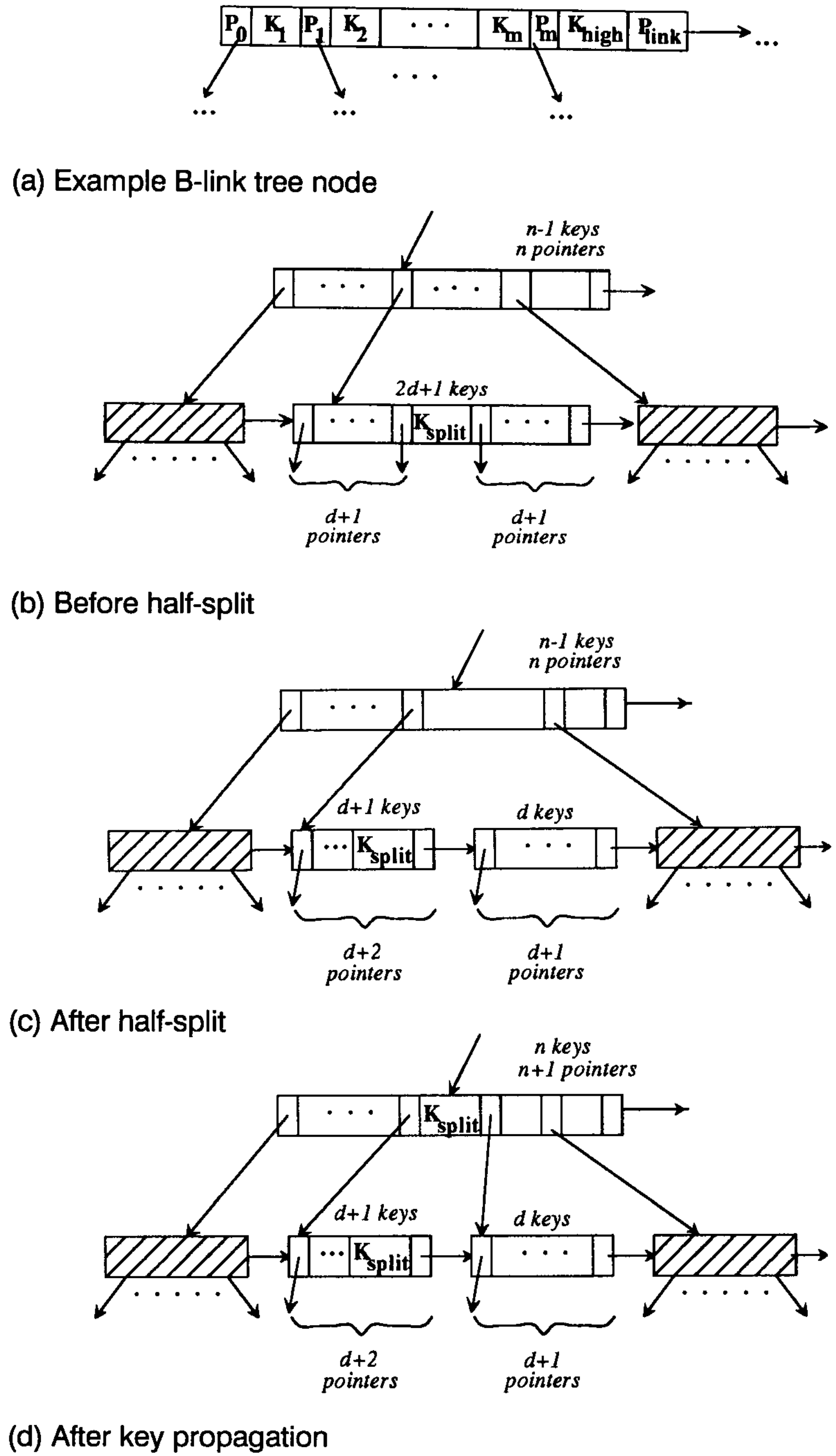
The key space

- Every page "owns" a range of values in the key space/key domain.
  - Starts out with a single root page (also a leaf), that owns the range "-∞" through to "+∞".
  - Splitting rightmost leaf page creates new leaf page that owns a range starting just after the final tuple in new left half, through to the sentinel "+∞".
- We *always* have **one** particular page that any possible new tuple *should* go on (at least on Postgres 12).

Protecting tree structure

- Locks used to protect **physical structure** as tree grows.
  - Must prevent the tree structure from becoming **inconsistent** (e.g., in a state that causes an index scan to skip over relevant data).
  - Various schemes used over past 40+ years.
- nbtree uses **Lehman & Yao** algorithm.
  - Have **right sibling pointer** and **high key**.
  - Sometimes called “B-Link Trees”.

Figure 3. A B-link tree page split



Pictured: Diagram from
"Performance of B+Tree Concurrency Control Algorithms"
by V. Srinivasan and Michael J. Carey

Moving right to recover

B-Link trees (Lehman and Yao B-Trees) take an **optimistic** approach, in contrast with earlier, **pessimistic** designs.

- Concurrent page splits might confuse searches that descend tree — can be dealt with a few ways.
- Earlier approaches involved “coupling” locks, **preventing** concurrent page splits altogether.
- Lehman and Yao’s algorithm **detects** and **recovers** from concurrent splits instead.

Recovering from a concurrent page split

- Lehman and Yao **divide** complicated page split into **two simpler** atomic steps.

  - Initial step creates new right sibling, and shares tuples amongst original (left) page and new right page.

  - Second step inserts new downlink for right page.

- Meanwhile, scans must **check high key** after descending on to a page — verifies that this is **still** the page covering the value of interest.

Overview

1. **Big picture with B-Trees**

   What’s the point of this “high key” business, anyway?

2. **Seeing the forest for the trees**

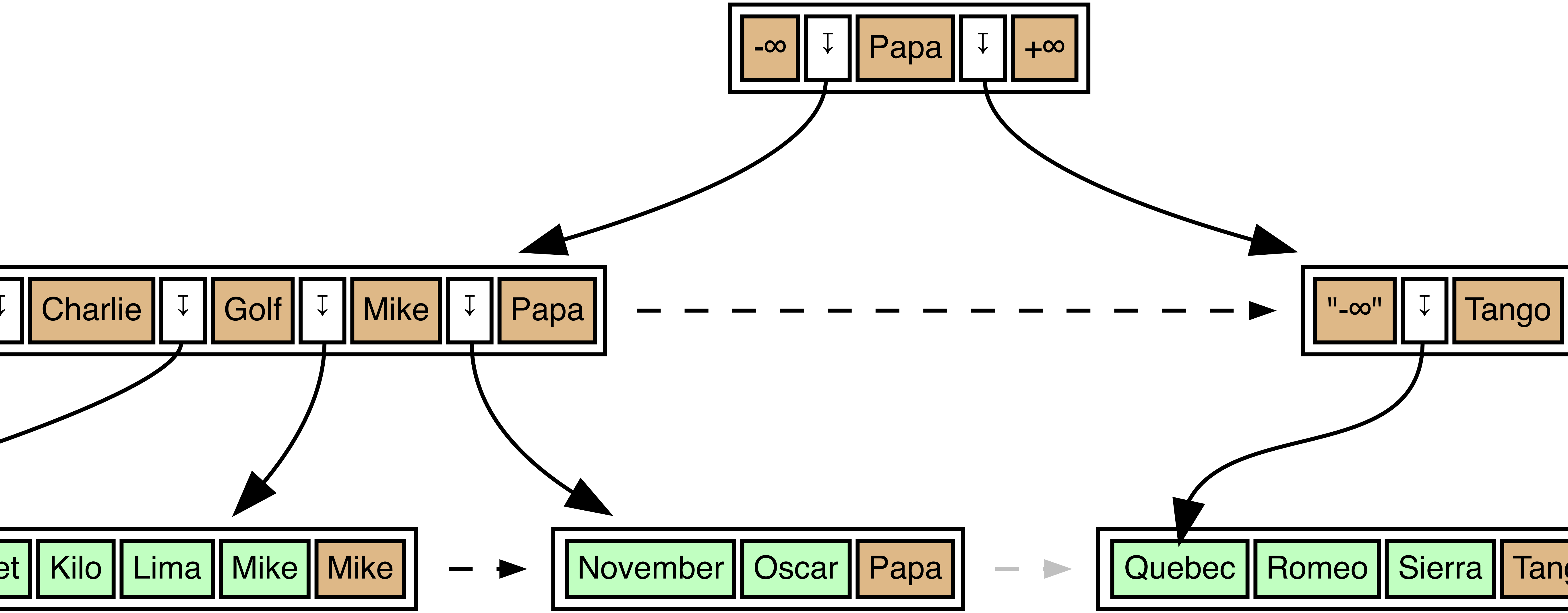
   Reasoning about nbtree invariants when designing enhancements.

3. **A place for everything, and everything in its place**

   How reliably unique keys simplify many things.

4. **Future work**

   Outlook for future improvements.



https://speakerdeck.com/peterg/nbtree-arch-pgcon

crunchy data

Seeing the forest for the trees

Lehman and Yao paper not a particularly good guide to nbtree.

- nbtree is concerned with distinctions that L&Y either ignore or couldn’t possibly anticipate.

  - Variable-sized keys.

  - Page model, `IndexTuple` struct format.

- Few *true* special cases, despite appearances to the contrary.

- Problem made worse by generally odd approach L&Y take.

"The locking model used in [LY81] assumed that an entire node could be read or written in one indivisible operation

…

Since the atomicity of node reads and writes is not a **reasonable assumption** in some environments (such as when the structure is in **primary memory**), and in order to make comparisons to other algorithms easier, we use a more general locking scheme similar to the one in [BS77]"

– Lanin & Sasha paper (LS86)

[**emphasis** added], from "2.2 Locks"

Terminology

Terminology makes things harder — **equivalent but not identical** representation lets nbtree **use IndexTuple struct for everything**. This is convenient for low-level page code, but can make high-level discussions confusing.

- Pivot tuples.
  - Contain separator keys and/or downlinks — **guide scans**.
  - Usually have both together, sometimes just separator (high key), other times just a downlink (“-∞” tuple).
- Non-pivot tuples.
  - Only on leaf level, cannot be truncated, always **point to table**.

Invariants

Carefully considering *how* to satisfy invariants can simplify the design of nbtree enhancements.

- **Relationship** between separator keys and real keys can be **fairly loose**.
  - Values in same domain as entries, but it's okay if they don't actually match any real entry (non-pivot key).
  - Separators are a good target for **prefix compression** (a generic optimization) — there is seldom any need to decompress, and a good whole-page prefix is **already available**.

Invariants (cont.)

Good B-Tree designs not only anticipate future work — they *simplify* it as a concomitant advantage.

- **Subtrees** can be isolated and reasoned about as independent units.
  - All subtrees own discrete range in the key space.
  - Page deletion relies on this to isolate subtree undergoing deletion (multi-level deletion).
  - Prefix compression of leaf page items would probably work based on similar principles — if only because compression based on *current* keys might **break page deletion**.

Overview

1. **Big picture with B-Trees**

   What's the point of this "high key" business, anyway?

2. **Seeing the forest for the trees**

   Reasoning about nbtree invariants when designing enhancements.

3. **A place for everything, and everything in its place**

   How reliably unique keys simplify many things.

4. **Future work**

   Outlook for future improvements.

A place for everything, and everything in its place

- **Uniqueness required** by Lehman and Yao.
- nbtree treats **heap TID** as tiebreaker column in v12. L&Y’s requirement now met, finally.
- TIDs are reliably unique, so now keys are themselves unique.
  - Needed for “retail index tuple deletion”.
  - **Surprisingly** helpful in other ways.

Heap TID as a tiebreaker

- For the most part, “heap TID column” is **not special**, at least internally.
- Inserts *must* specify heap TID.
- “Retail index tuple deletion” would have to work in the same way, since it’s necessary to unambiguously identify the same tuple when there are (logical) duplicates.

The false economy of “getting tired” when inserting duplicates

- **Old approach** had insertion place a duplicate *anywhere* it wanted to among leaf pages that have ever had duplicates.

  - Go through pages that store duplicates on the leaf level until some free space is located…

  - …or until we “get tired” — implementation unable to **spend too long** locating **theoretically available** free space.

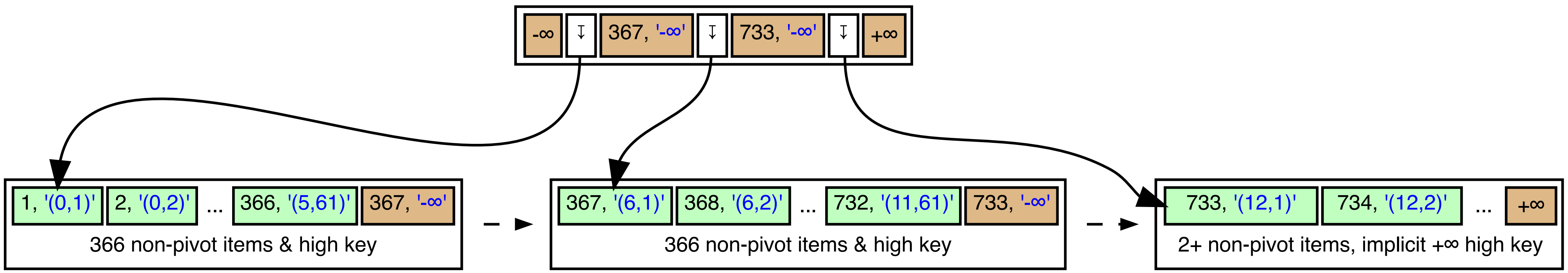
  - Getting tired occurs at random — **give up** and split page.

- Insertion won’t “get tired” with Postgres 12 indexes, which can make affected indexes **~16% smaller** in simple cases.

- Gitlab may have been affected [1].

[1] https://about.gitlab.com/handbook/engineering/infrastructure/blueprint/201901-postgres-bloat/

Realistic small Postgres 12 index (root page + 3 leaf pages)



https://speakerdeck.com/peterg/nbtree-arch-pgcon

Bigbird, Burt, Cookiemonster, Ernie, Snuffleopogus

In order to insert the key "Grouch" with its record, we must split this leaf into two as follows:

Bigbird, Burt, Cookiemonster	Ernie, Grouch, Snuffleopogus

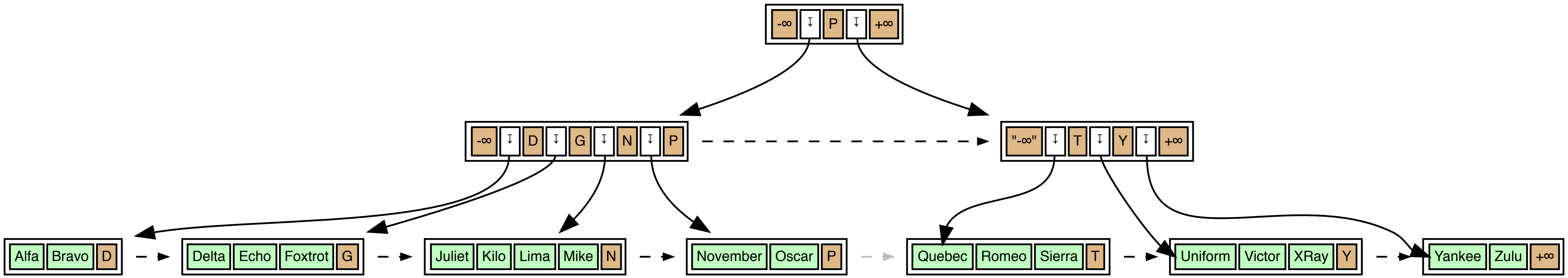
Instead of storing the key "Ernie" in the index, it obviously suffices to use one of the one-letter strings "D", "E" for the same purpose. In general we can select any string s with the property

$$\text{Cookiemonster} < s \leq \text{Ernie} \tag{1}$$

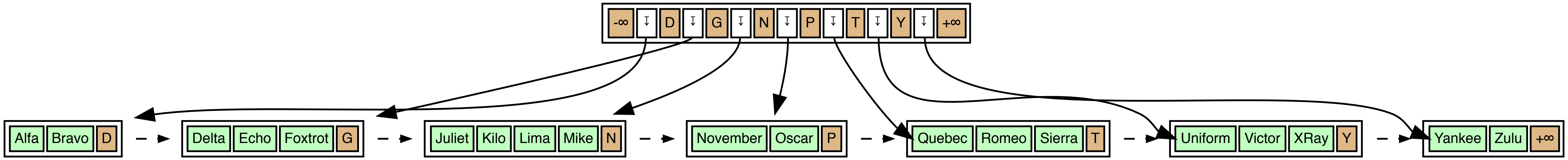
and store it in the index part to separate the two nodes. We call such a string s a *separator* (between Cookiemonster and Ernie). It seems prudent to choose one of the shortest separators.

Pictured: Diagram from "Prefix B-Trees" by Bayer and Unterauer, 1977

Classic suffix truncation applied to earlier example



Classic suffix truncation applied to earlier example



Choosing a split point

Leaf page splits primarily about equalizing free space on each side, to meet future needs.

- Also *only* place where new separator keys are made.
  - New **high key for left page** becomes **separator** before new downlink in parent for **right page**.
  - Internal page splits only use copies (truncating an already-truncated key would be wrong).
- Suffix truncation occurs when new separator created by leaf split.

Choosing a split point (cont.)

Algorithm can give some weight to suffix truncation, while continuing to make space utilization the first priority.

- Even very small adjustments can help suffix truncation a lot.

- Algorithm won't accept a *totally* lopsided split to make suffix truncation more effective.

a split point anywhere between the short arrows is acceptable, a single letter suffices. A single comparison of the two keys defining the range of acceptable split points can determine the shortest possible separator key. For example, in Figure 3.6, a comparison between "Johnson, Lucy" and "Smith, Eric" shows their first difference in the first letter, indicating that a separator key with a single letter suffices. Any letter

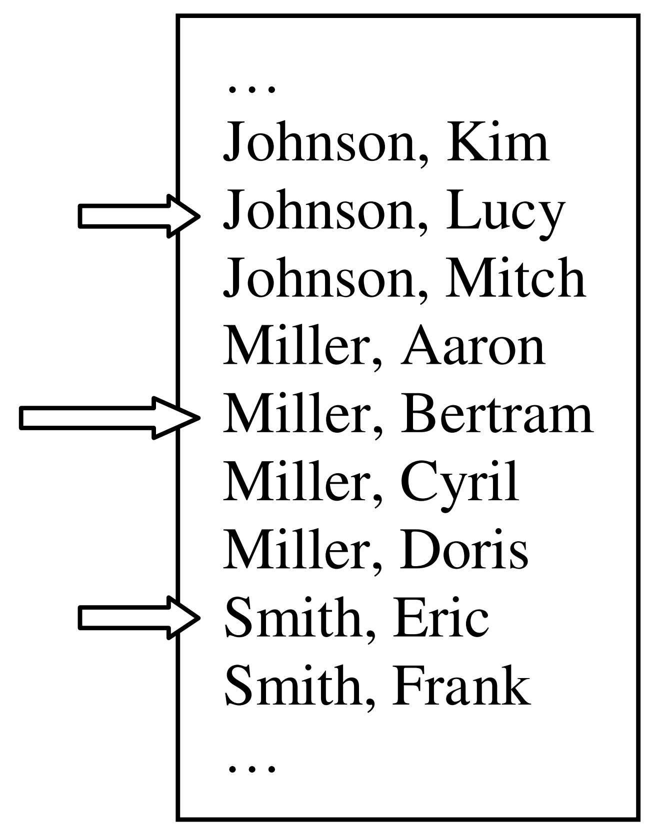


Fig. 3.6 Finding a separator key during a leaf split.

Pictured: Diagram from "Modern B-Tree Techniques" by Goetz Graefe — Prefix B-Trees chapter

Choosing a split point (cont.)

Algorithm in Postgres 12 takes a **holistic** view of the problem.

- May make slight adjustment with simple, common cases (e.g. pgbench indexes).
- But sometimes *radically* different to previous approach!
  - Behavior with duplicates is important with heap TID as a tiebreaker column.
  - A 50:50 page split is essentially a **guess**, and not necessarily a good one.
  - A 90:10 page split (rightmost split) is well known case where split point is based on **inferring** insertion patterns.

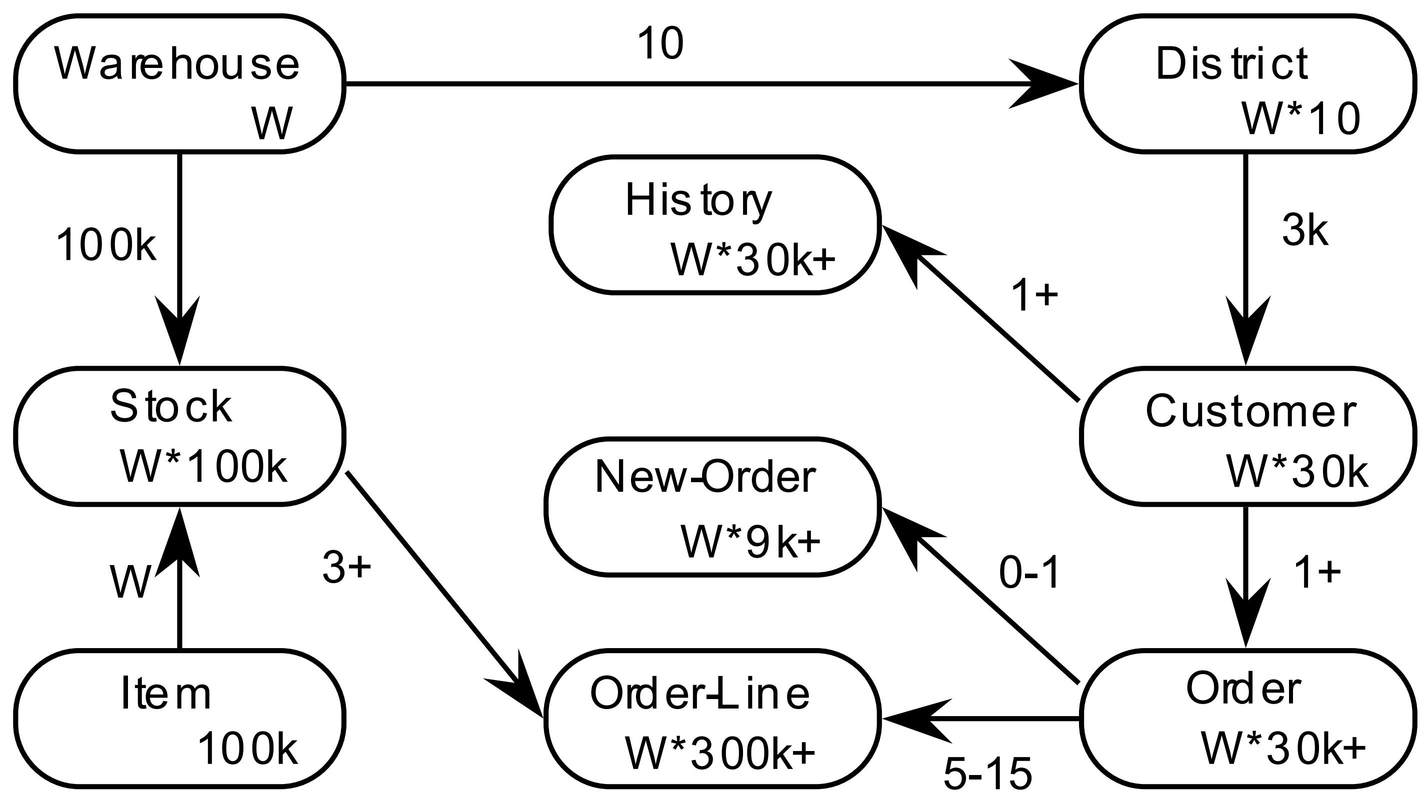
TPC-C indexes and “split after new tuple” optimization

Insertion pattern is very often *not* random

- Successive splits over short period of time that affect same area are *very* common.
- Multi-column indexes may have auto-incrementing identifiers grouped by an order number or similar.
- Industry standard TPC-C benchmark has lots of this. All indexes taken together are **~40% smaller** with Postgres 12.

1.2 Database Entities, Relationships, and Characteristics

1.2.1 The components of the TPC-C database are defined to consist of nine separate and individual tables. The relationships among these tables are defined in the entity-relationship diagram shown below and are subject to the rules specified in Clause 1.4.



TPC-C's order system is more or less a **circular buffer**, or **queue**

https://github.com/petergeoghegan/benchmarksql

Pictured: Diagram from TPC-C spec, Revision 5.11

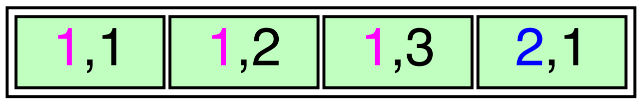
“Split after new tuple” example

1,1	1,2	1,3	2,1

Order numbers: 1, 2

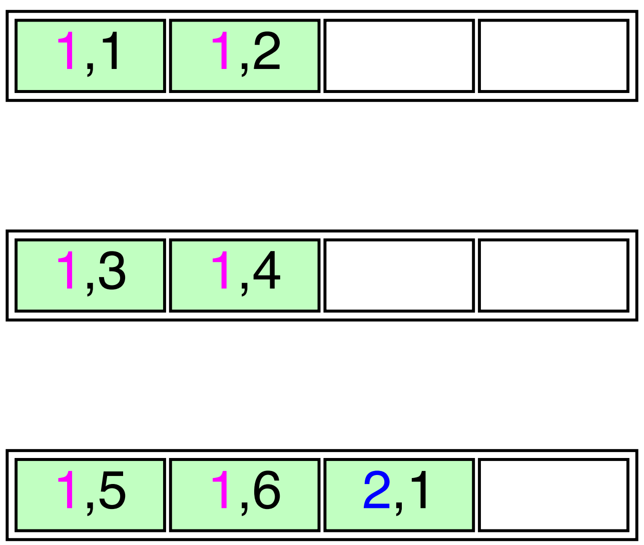
Line items: 1, 2, 3…

Initial state: one page, already **100% full**

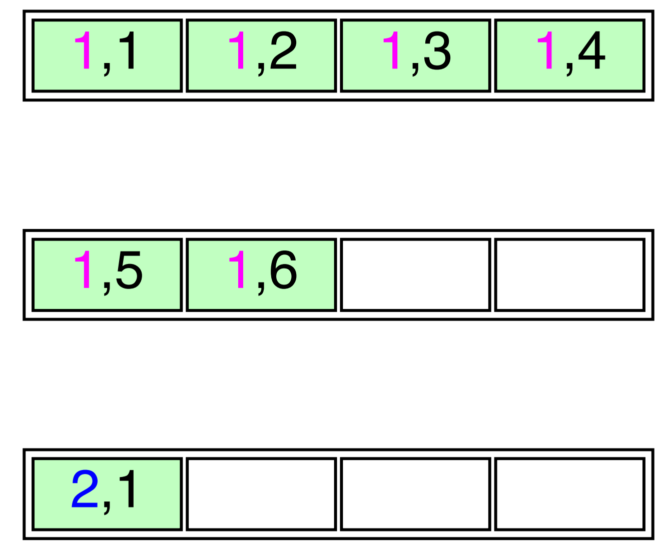


Insert 4, 5, 6…

50:50 page splits:



Optimized page splits:



(Last slide's state) → Insert 7, 8, 9…

50:50 page splits:

1,1	1,2		
1,3	1,4		
1,5	1,6		
1,7	1,8	1,9	2,1

Optimized page splits:

1,1	1,2	1,3	1,4
1,5	1,6	1,7	1,8
1,9			
2,1			

(Last slide's state) → Insert 10,11,12…

50:50 page splits:

1,1	1,2		
1,3	1,4		
1,5	1,6		
1,7	1,8		
1,9	1,10		
1,11	1,12	2,1	

Optimized page splits:

1,1	1,2	1,3	1,4
1,5	1,6	1,7	1,8
1,9	1,10	1,11	1,12
2,1			

Overview

1. **Big picture with B-Trees**

   What's the point of this "high key" business, anyway?

2. **Seeing the forest for the trees**

   Reasoning about nbtree invariants when designing enhancements.

3. **A place for everything, and everything in its place**

   How reliably unique keys simplify many things.

4. **Future work**

   Outlook for future improvements.

Future work

- Key normalization [1] — make separator keys into conditioned binary string that is simply `strcmp()`'d during index scans, regardless of "tuple shape".

  - Prefix compression.

  - "Classic" suffix truncation.

- Go even further — "abbreviated keys" in internal pages?

[1] https://wiki.postgresql.org/wiki/Key_normalization

CPU cache misses

- Binary searches incur cache misses during descent of tree — these can be minimized.

  - Abbreviated keys in line pointer array.

- These optimizations can be **natural adjuncts**.

  - Lehman & Yao don't care about how values are represented on the page.

  - **"Modern B-Tree techniques"** survey paper is a great reference.

Index tuple header with offsets

May need to accommodate table access methods with row identifiers that are not at all like TIDs.

- Tuple header offset makes it easy for that to be accessed quickly, but also accessed as just another attribute.

- Skip scans.

- [Dynamic] prefix truncation.

Conclusions

- It pays to consult multiple sources when working on nbtree codebase.

  - If only to confirm your original understanding.

  - **Terminology** causes problems — sometimes subtle distinctions matter a lot.

- **Visualizing** real indexes using tools like contrib/pageinspect can be very helpful.

Thanks!

https://speakerdeck.com/peterg/nbtree-arch-pgcon