

# Designing an Index for ZooDB

Jonas Nick & Bogdan Vancea

June 2, 2014

# Outline

- 1 Introduction
- 2 Goals & Challenges
- 3 The new Index Implementation
- 4 Benchmarks



- ▶ an open source object database written in Java



- ▶ an open source object database written in Java
- ▶ based on JDO standard



- ▶ an open source object database written in Java
- ▶ based on JDO standard
- ▶ 4 times faster than competitor db4o



- ▶ an open source object database written in Java
- ▶ based on JDO standard
- ▶ 4 times faster than competitor db4o
- ▶ [zoodb.org](http://zoodb.org)

# Database Index

## Key-Value data structure

1. **fast** retrieval
2. **ordered** iteration
3. stored in a **file**

# Database Index

## Key-Value data structure

1. **fast** retrieval
2. **ordered** iteration
3. stored in a **file**

```
ZooJdoHelper.createIndex(pm, Person.class, "name",  
false);
```



# Database Index

## Key-Value data structure

1. **fast** retrieval
2. **ordered** iteration
3. stored in a **file**

```
ZooJdoHelper.createIndex(pm, Person.class, "name",  
false);
```

**Attribute Index**  
Value → Object-ID

# Database Index

## Key-Value data structure

1. **fast** retrieval
2. **ordered** iteration
3. stored in a **file**

```
ZooJdoHelper.createIndex(pm, Person.class, "name",  
false);
```

**Attribute Index**  
Value → Object-ID

**ObjectID Index**  
OID → Diskpos

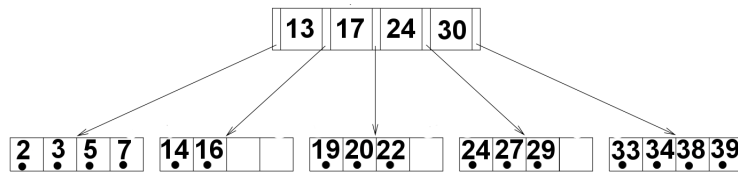
**Extension Index**  
Diskpos →  
0|follow Diskpos

# B+ Tree

---

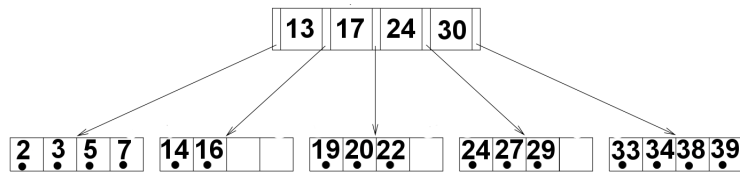
Images adapted from Database Management Systems by Ramakrishnan and Gehrke.

# B+ Tree



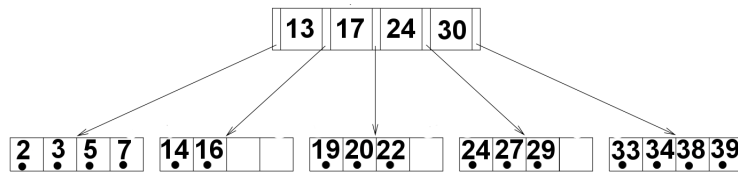
- ▶ Inner node contains keys and children pointer, leaf contains keys and values.

# B+ Tree



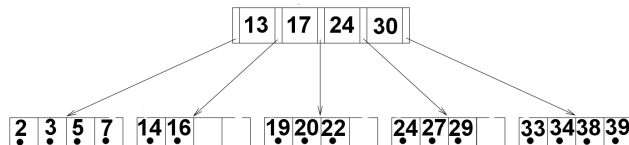
- ▶ Inner node contains keys and children pointer, leaf contains keys and values.
- ▶ **Node fills one disk page.**

# B+ Tree

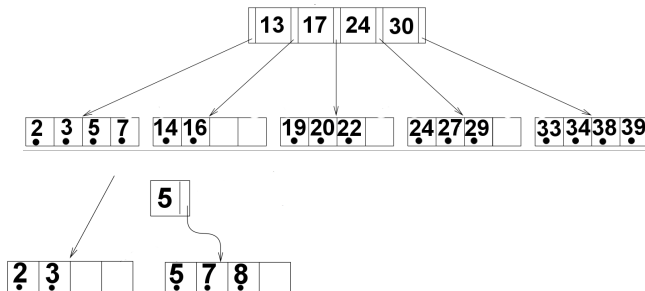


- ▶ Inner node contains keys and children pointer, leaf contains keys and values.
- ▶ **Node fills one disk page.**
- ▶ Node has maximum and minimum number of entries.

# Example: insert (8, v)

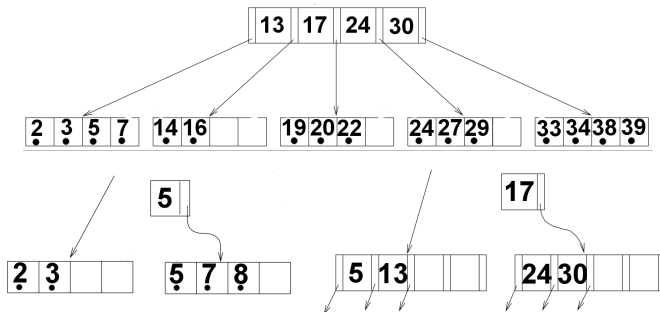


# Example: insert (8, v)

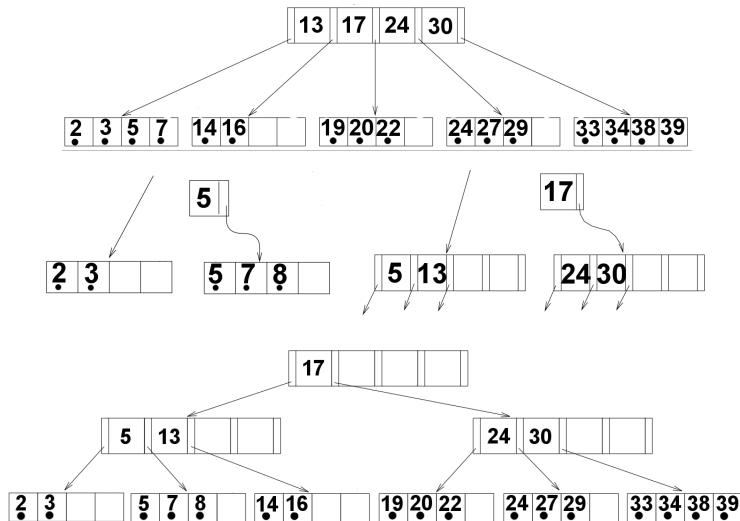




# Example: insert (8, v)

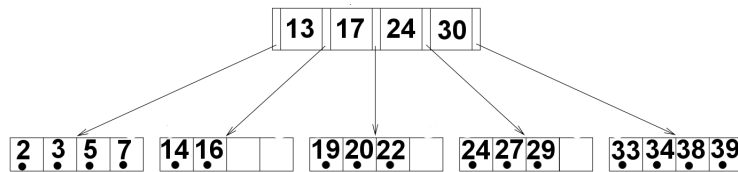


# Example: insert (8, v)



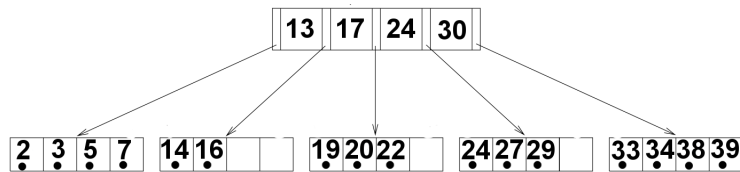
Images adapted from Database Management Systems by Ramakrishnan and Gehrke.

# B+ Tree



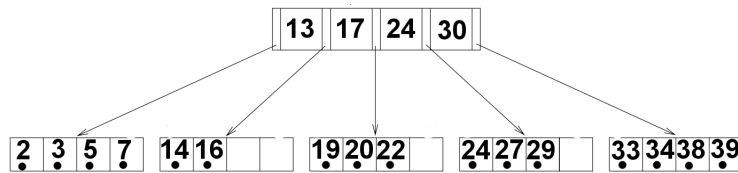
- ▶ Inner node contains keys and children pointer, leaf contain keys and values.
- ▶ **Node fills one disk page.**
- ▶ Node has maximum and minimum number of entries.

# B+ Tree



- ▶ Inner node contains keys and children pointer, leaf contain keys and values.
- ▶ **Node fills one disk page.**
- ▶ Node has maximum and minimum number of entries.
- ▶ Rebalancing
  - ▶ on insert: split
  - ▶ on delete: merge or redistribute

# B+ Tree



- ▶ Inner node contains keys and children pointer, leaf contain keys and values.
- ▶ **Node fills one disk page.**
- ▶ Node has maximum and minimum number of entries.
- ▶ Rebalancing
  - ▶ on insert: split
  - ▶ on delete: merge or redistribute
- ▶ Insert, remove, search are logarithmic.

# Goals

- ▶ faster B+ tree index

# Goals

- ▶ faster B+ tree index
- ▶ key unique and key-value unique
  - ▶ Ex. insert (1,1), (1,2)

# Goals

- ▶ faster B+ tree index
- ▶ key unique and key-value unique
  - ▶ Ex. insert (1,1), (1,2)
- ▶ range query iterators



# Goals

- ▶ faster B+ tree index
- ▶ key unique and key-value unique
  - ▶ Ex. insert (1,1), (1,2)
- ▶ range query iterators
- ▶ buffer manager to allow caching
  - ▶ fetches pages

# Goals

- ▶ faster B+ tree index
- ▶ key unique and key-value unique
  - ▶ Ex. insert (1,1), (1,2)
- ▶ range query iterators
- ▶ buffer manager to allow caching
  - ▶ fetches pages
- ▶ prefix sharing

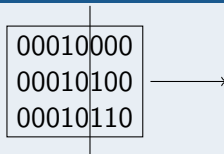
# Prefix Sharing

Exploit common prefix

00010000
00010100
00010110

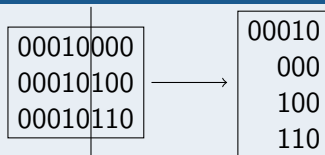
# Prefix Sharing

Exploit common prefix



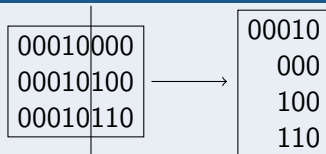
# Prefix Sharing

Exploit common prefix



# Prefix Sharing

## Exploit common prefix



- ▶ allows storing more entries in a node
- ▶ determines if node under- or overflows

# Challenges

- ▶ runtime dominated by disk access

# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes



# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes
  - ▶ rarely modify nodes

# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes
  - ▶ rarely modify nodes
- ▶ New features are costly.

# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes
  - ▶ rarely modify nodes
- ▶ New features are costly.
- ▶ Textbook algorithms need to be adapted.

# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes
  - ▶ rarely modify nodes
- ▶ New features are costly.
- ▶ Textbook algorithms need to be adapted.
  1. not optimized for our practical scenario

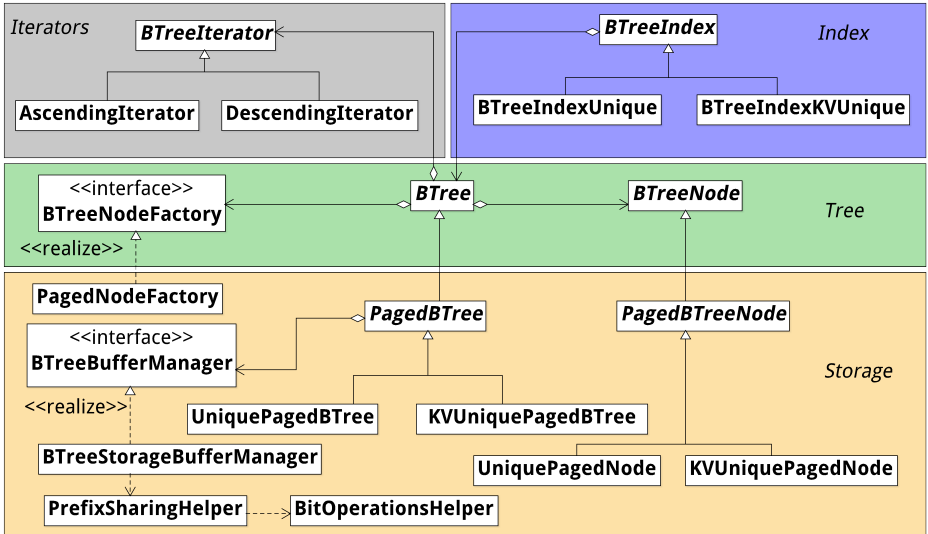
# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes
  - ▶ rarely modify nodes
- ▶ New features are costly.
- ▶ Textbook algorithms need to be adapted.
  1. not optimized for our practical scenario
  2. do not cover prefix sharing nor duplicates

# Challenges

- ▶ runtime dominated by disk access
  - ▶ prefer fewer nodes
  - ▶ rarely modify nodes
- ▶ New features are costly.
- ▶ Textbook algorithms need to be adapted.
  1. not optimized for our practical scenario
  2. do not cover prefix sharing nor duplicates
- ▶ low-level implementation optimizations

# Index Implementation



# Operations



# Operations

- ▶ Search - similar to normal B+ Tree

# Operations

- ▶ Search - similar to normal B+ Tree
- ▶ Insert overflow
  1. redistribute left ?
  2. split

# Operations

- ▶ Search - similar to normal B+ Tree
- ▶ Insert overflow
  1. redistribute left ?
  2. split
- ▶ Delete underflow
  1. merge with left/right ?
  2. split between left and right ?
  3. redistribute left/right

# Operations

- ▶ Search - similar to normal B+ Tree
- ▶ Insert overflow
  1. redistribute left ?
  2. split
- ▶ Delete underflow
  1. merge with left/right ?
  2. split between left and right ?
  3. redistribute left/right
- ▶ Write
  - ▶ only write dirty nodes
  - ▶ prefix encoding

# Operations

- ▶ Search - similar to normal B+ Tree
- ▶ Insert overflow
  1. redistribute left ?
  2. split
- ▶ Delete underflow
  1. merge with left/right ?
  2. split between left and right ?
  3. redistribute left/right
- ▶ Write
  - ▶ only write dirty nodes
  - ▶ prefix encoding
- ▶ insert/delete more costly, exactly how much?

# Microbenchmarks

- ▶ full in-memory, index only tests

# Microbenchmarks

- ▶ full in-memory, index only tests

## Duration - Old index is the baseline

Operation	No Prefix sharing	Prefix sharing
Search	1	0.9 - 1.1
Insert	1	1.6 - 2.8
Delete	1	1.45 - 2.9

## Size of B+ tree - Old index is the baseline

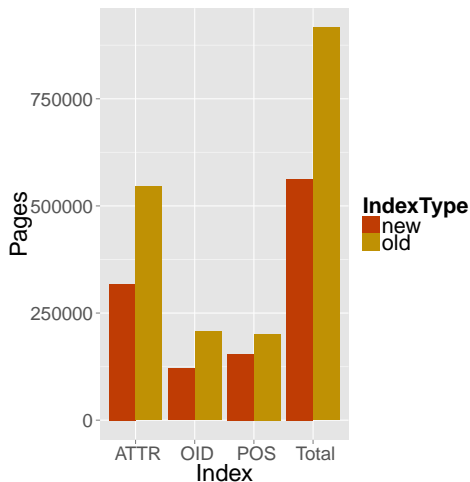
Operation	No Prefix sharing	Prefix sharing
Insert	1	0.5 - 1.1
Delete	1	0.5 - 0.75

# StackOverflow Data Import

- ▶ real-world workload
- ▶ StackOverflow data
  - ▶ 1.3 million users
  - ▶ 10.3 million posts
  - ▶ 13 million comments
  - ▶ 25 million votes
- ▶ 3 key unique attribute indexes
- ▶ 9 key-value unique attribute indexes



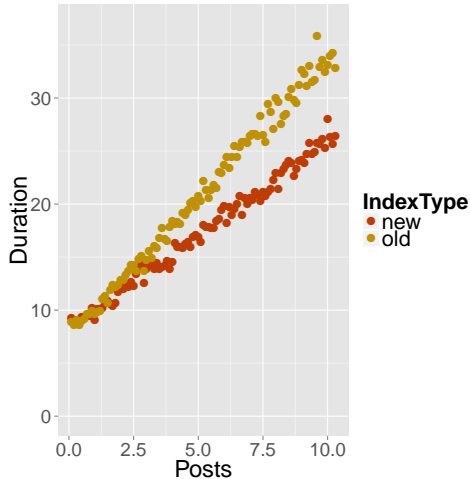
# StackOverflow Import - Index Sizes



- ▶ page size: 4KB
- ▶ database size: 31 GB

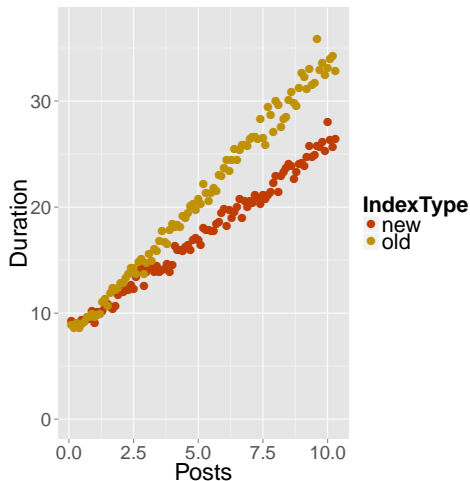
Index	Space saving (%)
Attribute	41.6
OID	41.5
POS	23.1
Total	38.5

# StackOverflow Import - Commit times



- ▶ import with new index 25% faster
- ▶ why?

# StackOverflow Import - Commit times



- ▶ import with new index 25% faster
- ▶ why?
- ▶ more entries in a node  
→ fewer dirty nodes
- ▶ data locality

# Summary

- ▶ prefix sharing: trade-off between speed and space
- ▶ works well in practice
- ▶ microbenchmarks
- ▶ implementation complexity.

# Q&A

- ▶ Thank you for your attention!
- ▶ Questions ?