University of Zurich^UZH

# Inserting Keys into the Robust Content-and-Structure (RCAS) Index

Kevin Wellenzohn, Luka Popovic, Michael H. Böhlen, Sven Helmer

University of Zurich

# Overview

Background:

- RCAS is an index for semi-structured hierarchical data [Wellenzohn, Böhlen, Helmer; **VLDB'20**]

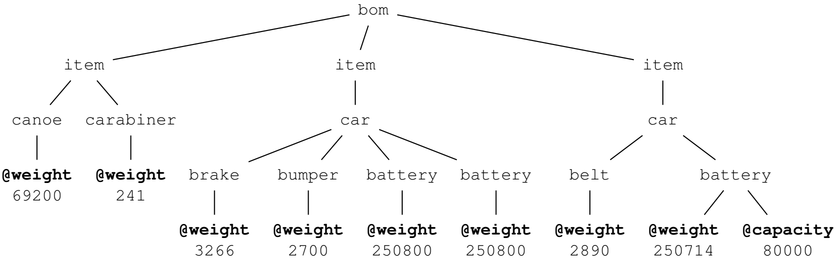
Problem:

- The RCAS index is **static** and cannot be updated easily
- Data is constantly generated and an index must keep up

In this paper/presentation:

- We make the RCAS index **dynamic**
- We focus on **inserting keys** (deletion is analogous)

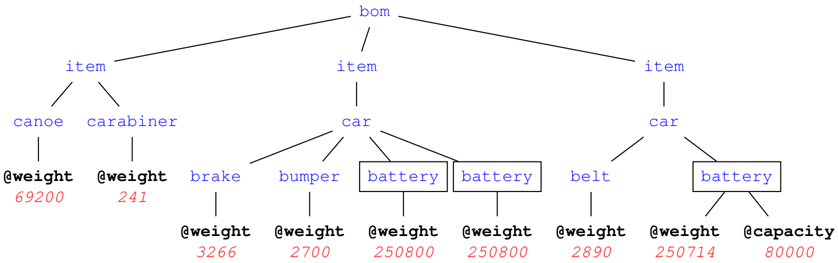
# Running Example – Bill of Materials (BOM)



Bill of Materials (BOM)

- Describes the hierarchical assembly of parts to products
- Nodes can have **attributes**, e.g., `@weight`, `@capacity`, ...

# Running Example – Bill of Materials (BOM)



bom

item item item

canoe carabiner car car

@weight 69200

@weight 241

brake bumper battery battery belt battery

@weight 3266

@weight 2700

@weight 250800

@weight 250800

@weight 2890

@weight 250714

@capacity 80000

**Content-and-Structure** (CAS) Queries

- Path predicate: e.g., all car parts . . . `/bom/item/car//`
- Value predicate: . . . that weigh at least 50kg **@weight** ≥ *50000*

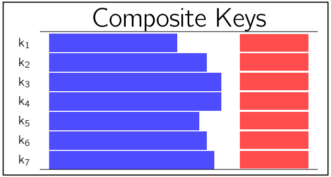
# Two-Dimensional Keys

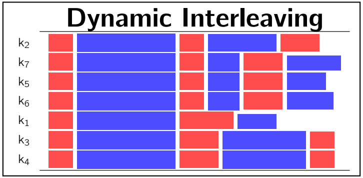
| | Path $P$ | Value $V$ (32bit integer) |
|---|---|---|
| k₁ | `/bom/item/canoe$` | 69200 (*00 01 0E 50*) |
| k₂ | `/bom/item/carabiner$` | 241 (*00 00 00 F1*) |
| k₃ | `/bom/item/car/battery$` | 250714 (*00 03 D3 5A*) |
| k₄ | `/bom/item/car/battery$` | 250800 (*00 03 D3 B0*) |
| k₅ | `/bom/item/car/belt$` | 2890 (*00 00 0B 4A*) |
| k₆ | `/bom/item/car/brake$` | 3266 (*00 00 0C C2*) |
| k₇ | `/bom/item/car/bumper$` | 2700 (*00 00 0A 8C*) |

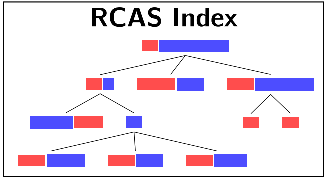
A **composite key** $k$ is two-dimensional:

- Path dimension $P$
- Value dimension $V$

# Background







Previous work [VLDB'20]:

- **Dynamic Interleaving** interleaves paths and values of composite keys
- **Robust Content-and-Structure (RCAS) Index**: trie-based index that stores dynamically-interleaved keys

# Dynamic Interleaving – In a Nutshell

- Paths and values are interleaved in a **fair and balanced way**
- No dimension (paths or values) is prioritized
- Makes the interleaving **robust**

# Dynamic Interleaving at Discriminative Bytes

The **discriminative byte** dsc$(K, D)$ of a set of keys $K$ in dimension $D$ is the first byte after the longest common prefix in dimension $D$.

| | Path $P$ | Value $V$ |
|---|---|---|
| $k_1$ | /bom/item/ca**n**oe$ | 00 **01** 0E 50 |
| $k_2$ | /bom/item/ca**r**abiner$ | 00 **00** 00 F1 |
| $k_3$ | /bom/item/ca**r**/battery$ | 00 **03** D3 5A |
| $k_4$ | /bom/item/ca**r**/battery$ | 00 **03** D3 B0 |
| $k_5$ | /bom/item/ca**r**/belt$ | 00 **00** 0B 4A |
| $k_6$ | /bom/item/ca**r**/brake$ | 00 **00** 0C C2 |
| $k_7$ | /bom/item/ca**r**/bumper$ | 00 **00** 0A 8C |
| | 1 3 5 7 9 11 13 15 17 19 21 | 1 2 3 4 |

| Keys $K$ | dsc$(K, P)$ | dsc$(K, V)$ |
|---|---|---|
| $\{k_1, \ldots, k_7\}$ | 13 | 2 |
| $\{k_5, k_6, k_7\}$ | 16 | 3 |

# Dynamic Interleavings of Keys $k_1, \ldots, k_7$

| | Dynamic Interleaving |
|---|---|
| $k_2$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{00}, \mathtt{r}, P), (\mathtt{\mathbf{a}biner\$}, \mathit{00\,F1}, \perp))$ |
| $k_7$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{00}, \mathtt{r}, P), (\mathtt{\mathbf{/}b}, \epsilon, V), (\mathbf{0A}\,\mathit{8C}, \mathtt{umper\$}, \perp))$ |
| $k_5$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{00}, \mathtt{r}, P), (\mathtt{\mathbf{/}b}, \epsilon, V), (\mathbf{0B}\,\mathit{4A}, \mathtt{elt\$}, \perp))$ |
| $k_6$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{00}, \mathtt{r}, P), (\mathtt{\mathbf{/}b}, \epsilon, V), (\mathbf{0C}\,\mathit{C2}, \mathtt{rake\$}, \perp))$ |
| $k_1$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{01}\,\mathit{0E\,50}, \mathtt{noe\$}, \perp))$ |
| $k_3$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{03}\,\mathit{D3}, \mathtt{r/battery\$}, V), (\mathbf{5A}, \epsilon, \perp))$ |
| $k_4$ | $((\mathit{00}, \mathtt{/bom/item/ca}, V), (\mathbf{03}\,\mathit{D3}, \mathtt{r/battery\$}, V), (\mathbf{B0}, \epsilon, \perp))$ |

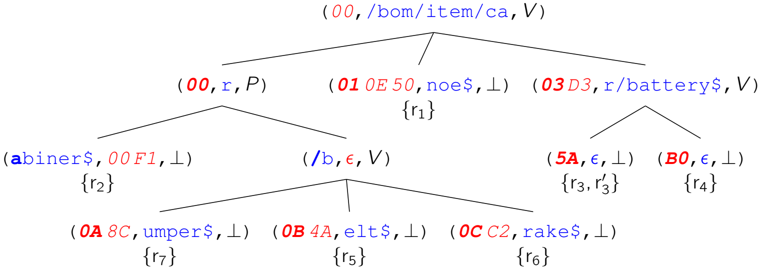
- **Dynamic interleaving** alternatingly interleaves paths and values at their discriminative bytes
- No dimension is prioritized; this makes the interleaving **robust**

# Dynamic Interleavings of Keys $k_1, \ldots, k_7$

| | Dynamic Interleaving | | | |
|---|---|---|---|---|
| $k_2$ | ((00, /bom/item/ca, $V$), | (00, r, $P$), | (abiner$, 00F1, $\perp$)) | |
| $k_7$ | ((00, /bom/item/ca, $V$), | (00, r, $P$), | (/b, $\epsilon$, $V$), | (0A 8C, umper$, $\perp$)) |
| $k_5$ | ((00, /bom/item/ca, $V$), | (00, r, $P$), | (/b, $\epsilon$, $V$), | (0B 4A, elt$, $\perp$)) |
| $k_6$ | ((00, /bom/item/ca, $V$), | (00, r, $P$), | (/b, $\epsilon$, $V$), | (0C C2, rake$, $\perp$)) |
| $k_1$ | ((00, /bom/item/ca, $V$), | (01 0E 50, noe$, $\perp$)) | | |
| $k_3$ | ((00, /bom/item/ca, $V$), | (03 D3, r/battery$, $V$), | (5A, $\epsilon$, $\perp$)) | |
| $k_4$ | ((00, /bom/item/ca, $V$), | (03 D3, r/battery$, $V$), | (B0, $\epsilon$, $\perp$)) | |

**RCAS Index**: Collapse each box (i.e., common prefix) into a node

# Robust Content-and-Structure (RCAS) Index



(00, /bom/item/ca, $V$)

(00, r, $P$)

(01 0E 50, noe\$, $\perp$) $\{r_1\}$

(03 D3, r/battery\$, $V$)

(abiner\$, 00 F1, $\perp$) $\{r_2\}$

(/b, $\epsilon$, $V$)

(5A, $\epsilon$, $\perp$) $\{r_3, r_3'\}$

(B0, $\epsilon$, $\perp$) $\{r_4\}$

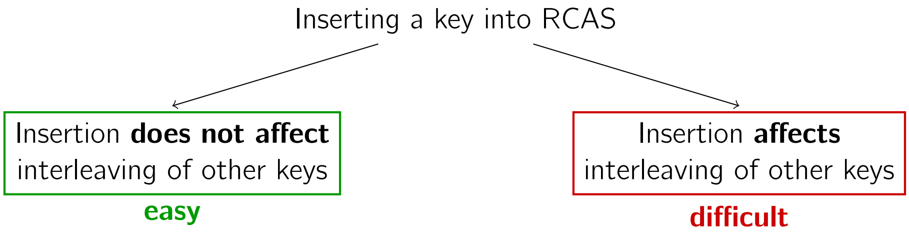
(0A 8C, umper\$, $\perp$) $\{r_7\}$

(0B 4A, elt\$, $\perp$) $\{r_5\}$

(0C C2, rake\$, $\perp$) $\{r_6\}$

- RCAS is an **in-memory** and **trie-based** index
- A root-to-leaf path describes a composite key
- Leaves contain references *r* to nodes in the DB

# Problem – Updating RCAS



Inserting a key into RCAS

Insertion **does not affect** interleaving of other keys

**easy**

Insertion **affects** interleaving of other keys

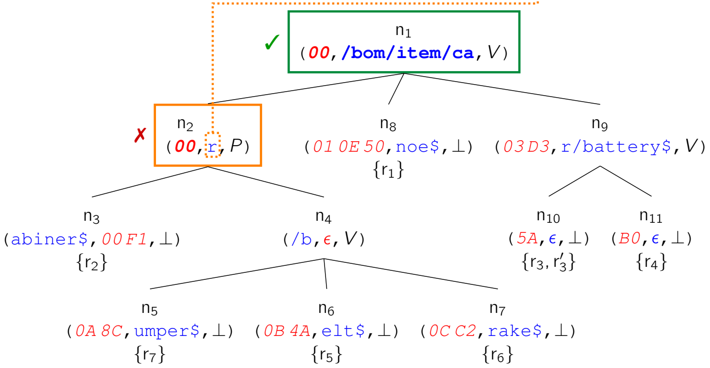
**difficult**

Inserting a key can change the positions of the discriminative bytes

- This can affect the dynamic interleaving of existing keys in RCAS
- In the **worst case**, inserting one key affects the dynamic interleaving of **all** keys!

# Problem – Example

Let's insert the key $k_8 =$ (/bom/item/cassette$, 00 00 AB 12, $r_8$)



✓ $n_1$ (00, /bom/item/ca, V)

✗ $n_2$ (00, r, P)

$n_8$ (01 0E 50, noe$, ⊥) {$r_1$}

$n_9$ (03 D3, r/battery$, V)

$n_3$ (abiner$, 00 F1, ⊥) {$r_2$}

$n_4$ (/b, ε, V)

$n_{10}$ (5A, ε, ⊥) {$r_3, r_3'$}

$n_{11}$ (B0, ε, ⊥) {$r_4$}

$n_5$ (0A 8C, umper$, ⊥) {$r_7$}

$n_6$ (0B 4A, elt$, ⊥) {$r_5$}

$n_7$ (0C C2, rake$, ⊥) {$r_6$}

- There's a mismatch in node $n_2$
- Letter r in $n_2$ was a common prefix, now becomes a disc. byte
- This affects the interleaving of all keys rooted in $n_2$

# Solution – Restructuring RCAS

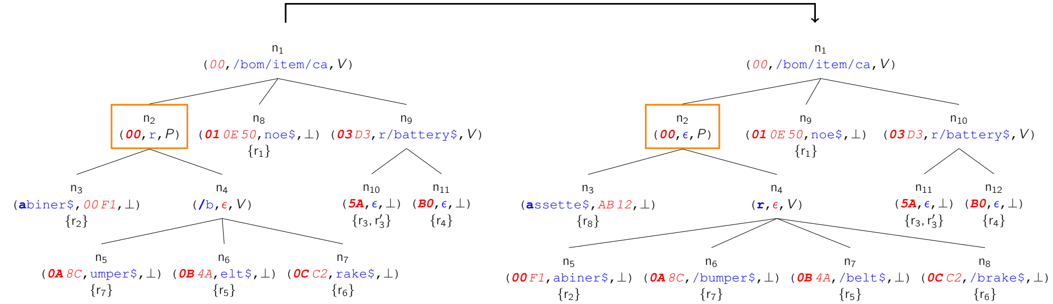
We propose two methods to restructure RCAS:

- **Strict restructuring**
  - ✓ Maintains dynamic interleaving
  - ✗ Efficient
- **Lazy restructuring**
  - ✗ Maintains dynamic interleaving
  - ✓ Efficient

# Solution 1 – Strict Restructuring

Strict restructuring recomputes the dynamic interleaving of **all affected keys**.

Insert $k_8$ = (`/bom/item/cassette$`, *00 00 AB 12*, $r_8$)



$n_1$ (00, /bom/item/ca, $V$)

$n_2$ (**00**, r, $P$)

$n_8$ (**01** 0E 50, noe$, ⊥) {$r_1$}

$n_9$ (**03** D3, r/battery$, $V$)

$n_3$ (**a**biner$, 00 F1, ⊥) {$r_2$}

$n_4$ (**/**b, $\epsilon$, $V$)

$n_{10}$ (**5A**, $\epsilon$, ⊥) {$r_3, r_3'$}

$n_{11}$ (**B0**, $\epsilon$, ⊥) {$r_4$}

$n_5$ (**0A** 8C, umper$, ⊥) {$r_7$}

$n_6$ (**0B** 4A, elt$, ⊥) {$r_5$}

$n_7$ (**0C** C2, rake$, ⊥) {$r_6$}

$n_1$ (00, /bom/item/ca, $V$)

$n_2$ (**00**, $\epsilon$, $P$)

$n_9$ (**01** 0E 50, noe$, ⊥) {$r_1$}

$n_{10}$ (**03** D3, r/battery$, $V$)

$n_3$ (**a**ssette$, AB 12, ⊥) {$r_8$}

$n_4$ (**r**, $\epsilon$, $V$)

$n_{11}$ (**5A**, $\epsilon$, ⊥) {$r_3, r_3'$}

$n_{12}$ (**B0**, $\epsilon$, ⊥) {$r_4$}

$n_5$ (**00** F1, abiner$, ⊥) {$r_2$}

$n_6$ (**0A** 8C, /bumper$, ⊥) {$r_7$}

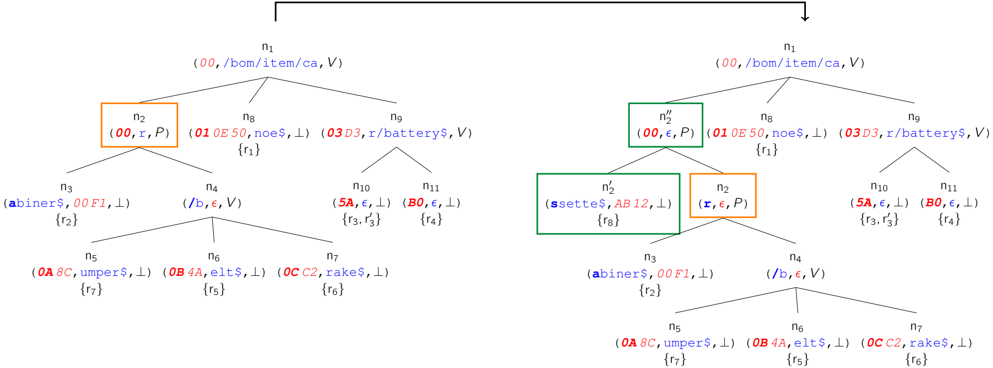
$n_7$ (**0B** 4A, /belt$, ⊥) {$r_5$}

$n_8$ (**0C** C2, /brake$, ⊥) {$r_6$}

# Solution 2 – Lazy Restructuring

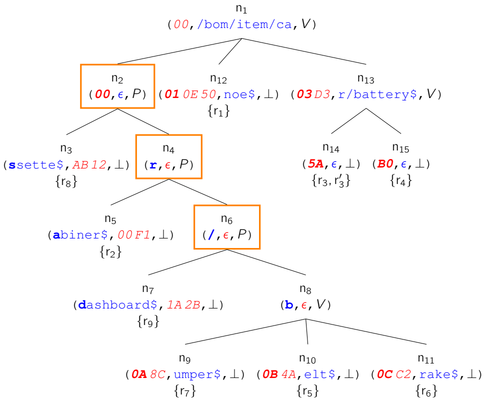
Lazy restructuring **adds exactly two nodes** to resolve the mismatch.

Insert $k_8$ = (/bom/item/cassette$, *00 00 AB 12*, $r_8$)



# Solution 2 – Lazy Restructuring

With lazy restructuring the **index can deteriorate over time**



- Nodes $n_2$, $n_4$, $n_6$ partition the data in the path dimension
- **Violates alternating interleaving** at discriminative path and value bytes

# Solutions – Recap

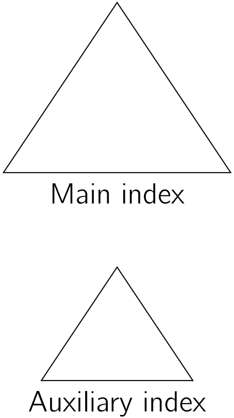
- **Strict restructuring**
  - ✓ Maintains dynamic interleaving
  - ✗ Efficient: cost is dominated by the **size of the subtree that must be restructured**

- **Lazy restructuring**
  - ✗ Maintains dynamic interleaving
  - ✓ Efficient: cost is dominated by the **height of RCAS**

# Auxiliary Index

We use **two** RCAS indexes

- **Main index**
  - Created from initial data
  - Used for easy insertion cases
  - ✓ Maintains dynamic interleaving
  - ✓ Efficient insertions
- **Auxiliary index**
  - Used for difficult insertions that require restructuring
  - We can use strict or lazy restructuring
  - Occasionally merged back into main index



# Experiments – Setup

**Goal**:

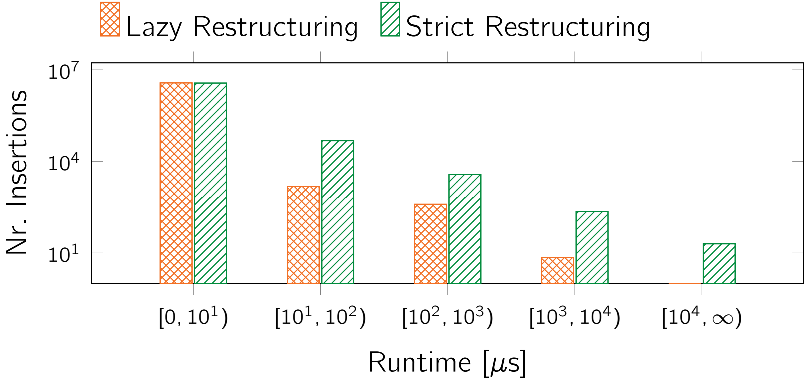
- Evaluate the **insertion performance** with strict and lazy restructuring
- Evaluate the **query performance** of the resulting indexes

**Dataset**: ServerFarm

- Paths: The paths of all files on 100 servers
- Values: The sizes of those files
- 9.3M keys

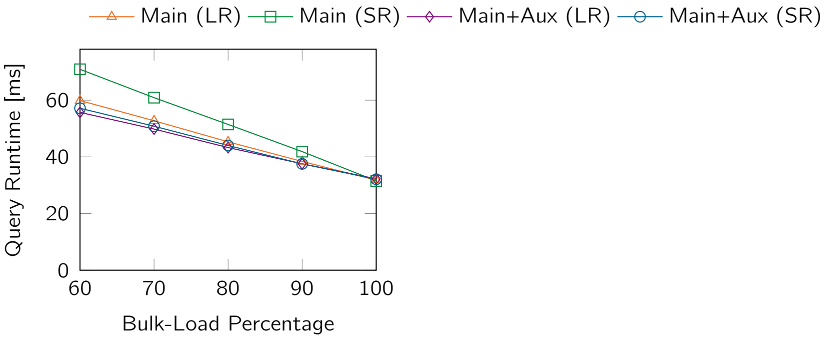
# Experiments – Insertion Performance

We insert 3.7 million keys using lazy or strict restructuring



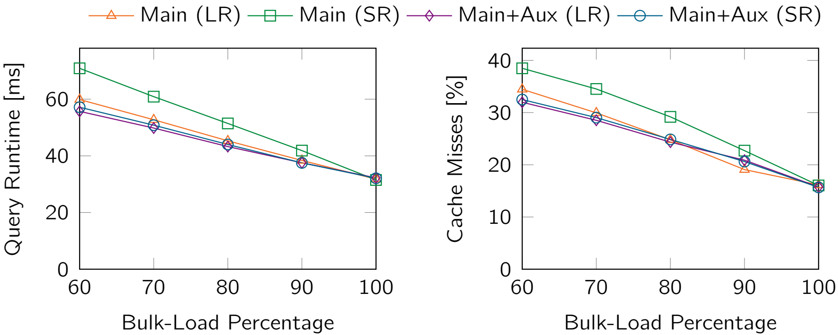
# Experiments – Query Performance

We report the average runtime of six queries from [VLDB'20]



# Experiments – Query Performance

We report the average runtime of six queries from [VLDB'20]



# Summary

Problem:

- Inserting a key into RCAS can affect the dynamic interleaving of other keys

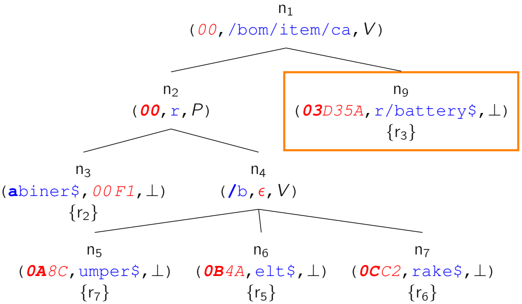
Contributions:

- **Strict Restructuring** optimizes for query performance
- **Lazy Restructuring** optimizes for insertion performance
- **Auxiliary Index** strikes a balance

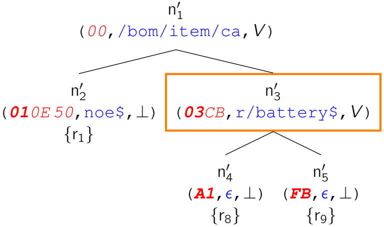
Thanks!

# Auxiliary Index – Merging

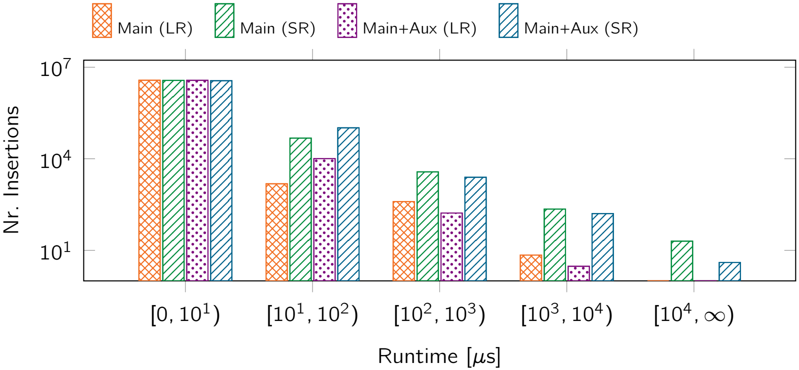
Main Index



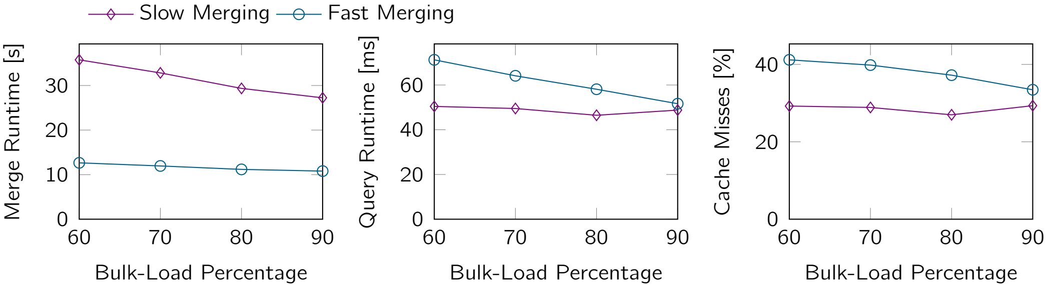
Auxiliary Index



# Experiments – Insertion Performance



# Experiments – Auxiliary Index



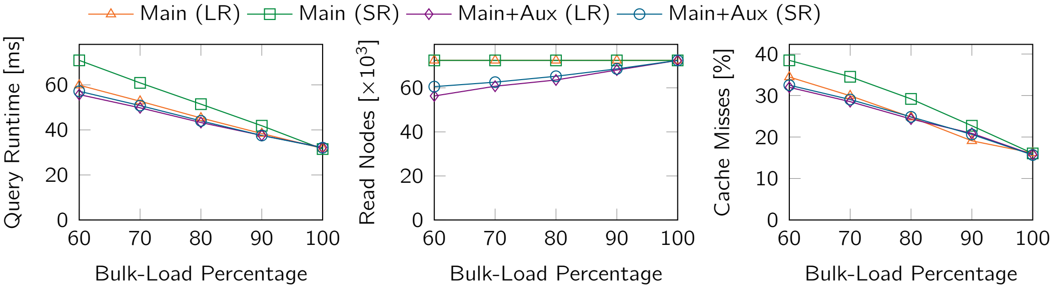
Merge Performance

Query Performance

Query Cache Misses

# Experiments – Query Performance

We report the average runtime of six queries from [VLDB'20]


Main (LR)
Main (SR)
Main+Aux (LR)
Main+Aux (SR)
Query Runtime [ms]
Bulk-Load Percentage
Read Nodes [$\times 10^3$]
Bulk-Load Percentage
Cache Misses [%]
Bulk-Load Percentage