Distributed Data Management
Storage and Retrieval

Thorsten Papenbrock
F-2.04, Campus II
Hasso Plattner Institut
Introduction
Layering Data Models

1. Conceptual layer
   - Data structures, objects, modules, ...
     - Application code

2. Logical layer
   - Relational tables, JSON, XML, graphs, ...
     - Database management system (DBMS) or storage engine

3. Representation layer
   - Bytes in memory, on disk, on network, ...
     - Database management system (DBMS) or storage engine

4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, ...
     - Operating system and hardware drivers

our focus now
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets
Design a distributed DBMS for **fast storage and retrieval of huge and evolving datasets**

Most data models resemble key-value data.

- Let’s pretend that everything is key-value data!
Fast Storage DBMS
A Tiny Database

- Basic database tasks: (a) write given data, (b) read specific data
- A tiny key-value store in two Bash functions:

```bash
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}
db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

- It works:

```bash
$ db_set 1234 '{"name":"Berlin","type":"city"}'
$ db_get 1234
'{"name":"Berlin","type":"city"}''
```

Concatenate the first two parameters by "," and write/append them to the file named "database".

Find all lines starting with first parameter, remove first parameter from lines, and select the last line.

Why?
Assume the following input-sequence:

- $ db_set 1234 '{"name":"Berlin","type":"city"}'
- $ db_set 42 '{"name":"Germany","type":"country"}'
- $ db_set 42 '{"name":"Germany","type":"country","capital":"Berlin"}'

The according “database”-file (= CSV-file):

- $ cat database
  1234,"name":"Berlin","type":"city"
  42,"name":"Germany","type":"country"
  42,"name":"Germany","type":"country","capital":"Berlin"

“database” is a **Log** file:
- Append only, no removal of old values
  - Only the last entry for each key is valid.
- Fast writes ( $O(1)$ ) but slow reads ( $O(n)$ with $n$ records in the log )
- To speed-up reads: Indexes!
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets
Index

- An additional data structure that helps to locate data by some search criterion, i.e., the key
  - Key = one or more identifying attributes
- Basically a key-value store, where values can be actual data or pointers to relational records, documents, graph nodes/edges, ...
- Improves data retrieval operations
  - Usually $O(n)$ to $O(\log(n))$ or $O(1)$
- Costs additional writes to index structure and storage space
  - Use indexes carefully (not too many)!
- Different index implementations (data structures) have different strengths
  - Choose the right index for your queries (workload)!
Definition

- A hash index is a hash map (dictionary) that maps keys to the addresses (in memory, on disk, on the network, …) of their values/records.

- The hash map uses a hash function to calculate mapping of keys and positions and is usually kept in memory.

Uses

- key-value stores, multilayered indexes, data distribution (load balancing, sharding, …)

Strength

- Point queries: An index look-up delivers a value’s position in $O(1)$.

Weaknesses

- Range queries require to look up each key individually.

- Hash map must fit into main memory; hash maps on disk perform poorly.
Fast Retrieval DBMS
Hash Index – Example

In-memory hash map

Log-structured file on disk (each box is one byte)

key | byte offset
---|---
1234 | 0
42 | 37

Distributed Data Management
Storage and Retrieval

ThorstenPapenbrock
Slide 11
Design a **distributed DBMS**
for fast storage and retrieval
of huge and evolving datasets
### Distributed DBMS

**Remote Pointers**

#### Storage and Retrieval

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>172.168.0.1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>172.168.0.1</td>
<td>37</td>
</tr>
<tr>
<td>534</td>
<td>172.168.0.3</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>172.168.0.6</td>
<td>0</td>
</tr>
<tr>
<td>7245</td>
<td>172.168.0.6</td>
<td>52</td>
</tr>
</tbody>
</table>

*E.g., one file per node*

---

**Distributed Data Management**

*ThorstenPapenbrock*

Slide 13
Key-to-node assignment strategies:

a) Random
   - Great for load balancing and efficient for point queries

b) Fixed key ranges
   - Great for compression and efficient for range queries

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>172.168.0.1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>172.168.0.1</td>
<td>37</td>
</tr>
<tr>
<td>534</td>
<td>172.168.0.3</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>172.168.0.6</td>
<td>0</td>
</tr>
<tr>
<td>7245</td>
<td>172.168.0.6</td>
<td>52</td>
</tr>
</tbody>
</table>
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets.
Controlling file growth

- Indexed data, i.e., log is insert-only (for good write performance).
  - Frequent updates make files unnecessarily large.
  - Example: a store that maps products to stock-counts
    - Each purchase increments a stock-count → new record!
    - Each sale decrements a stock-count → new record!
    - But: Collection of products is almost constant...

- Solution: Consolidate/compact the log regularly freeing up disk space.
  - How do we do this on a running system?
    - Segmentation!
### Segmentation

- Break log into segments of fixed size.
- Each segment ...
  - stores a range of keys.
  - can be subject for distribution!
  - has two representations:
    - **Compacted**
      - Static (= does not allow writes)
      - Purged (= only most recent value for each key)
    - **Current**
      - Dynamic (= allows appending writes)
      - Unchecked (= same key might appear multiple times)
Whenever a segment reaches max size:

1. Close the segment and **redirect** writes to a fresh current segment file.
2. **Compact** the closed segment file:
   1. Create a new compacted segment file.
   2. Read the closed segment file backwards.
   3. If a key is read for the first time:
      - Write the entry (key + value) into the new compacted segment file.
3. **Merge** the old compacted segment file into the new compacted segment file:
   1. Read the old compacted segment file backwards.
   2. If a key is not present in the new compacted segment file:
      - Write the entry (key + value) into the new compacted segment file.
4. **Delete** the closed segment file and the old compacted segment file.
Huge and Evolving Datasets

Segmentation

- Compact:

  - Writes:
    - milk:13
    - water:146
    - coffee:27
    - tee:37
    - water:145
    - water:144
    - water:143
    - water:142
    - coffee:24
    - tea:35
    - soda:141
    - beer:241
    - milk:12

  - Reads:
    - milk:13
    - water:146
    - coffee:27
    - tee:37
    - water:145
    - water:144
    - water:143
    - water:142
    - coffee:24
    - tea:35
    - soda:141
    - beer:241
    - milk:12

Distributed Data Management

Storage and Retrieval

Thorsten Papenbrock
Slide 19
Huge and Evolving Datasets

Segmentation

- Compact:
- Merge:

<table>
<thead>
<tr>
<th>Writes</th>
<th>Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk:13</td>
<td>beer:241</td>
</tr>
<tr>
<td>water:146</td>
<td>soda:141</td>
</tr>
<tr>
<td>coffee:27</td>
<td>tee:37</td>
</tr>
<tr>
<td>water:145</td>
<td>coffee:27</td>
</tr>
<tr>
<td>water:144</td>
<td>water:146</td>
</tr>
<tr>
<td>water:143</td>
<td>milk:13</td>
</tr>
<tr>
<td>tee:36</td>
<td>coke:13</td>
</tr>
<tr>
<td>coke:25</td>
<td>coke:26</td>
</tr>
</tbody>
</table>

Distributed Data Management
Storage and Retrieval

Thorsten Pappenbrock
Slide 20
Huge and Evolving Datasets

Segmentation

- Compact:
- Merge:

<table>
<thead>
<tr>
<th>Writes</th>
<th>Reads</th>
</tr>
</thead>
</table>
| milk:13  
water:146
coffee:27
tee:37
water:145
water:144
water:143
tee:36
coffee:26
coffee:25 | beer:241
soda:141
tee:37
coffee:27
water:146
milk:13 |

Distributed Data Management
Storage and Retrieval

Thorsten Papenbrock
Slide 21
### Huge and Evolving Datasets

**Segmentation**

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>172.168.0.1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>172.168.0.1</td>
<td>37</td>
</tr>
<tr>
<td>534</td>
<td>172.168.0.3</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>172.168.0.6</td>
<td>0</td>
</tr>
<tr>
<td>7245</td>
<td>172.168.0.6</td>
<td>52</td>
</tr>
</tbody>
</table>

Every log is now split into segmentation files.
Huge and Evolving Datasets
Segmentation

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>172.168.0.1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>172.168.0.1</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>534</td>
<td>172.168.0.3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>172.168.0.6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7245</td>
<td>172.168.0.6</td>
<td>2</td>
<td>52</td>
</tr>
</tbody>
</table>

Distributed Data Management
Storage and Retrieval

Multiple files per node for parallel writes on each node!

hash function
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets

If the data is really large, then this dense index will also be too large and updating it too expensive!
### Huge and Evolving Datasets

#### Segmentation

- **Storage and Retrieval**
- **Distributed Data Management**

---

**Thorsten Papenbrock**

---

### Table: Hash Function Example

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>172.168.0.1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>172.168.0.1</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>534</td>
<td>172.168.0.3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>172.168.0.6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7245</td>
<td>172.168.0.6</td>
<td>2</td>
<td>52</td>
</tr>
</tbody>
</table>

---

**Distributed Data Management**

Storage and Retrieval

ThorstenPapenbrock

Slide 25
Huge and Evolving Datasets

Segmentation

Index = key-value store

Segmentation!

Distributed Data Management

Storage and Retrieval

Thorsten Papenbrock
Slide 26
Wait!

- If the index becomes too large and we start partitioning our index, we are back to our initial problem!
  - What is different now with segmented index data?

- Advantages:
  - Index entries are much smaller than data entries (faster compact and merge).
  - Index entries are of fixed length (will enable binary search).
  - Index could use key range partitioning while the data still uses random partitioning.
Huge and Evolving Datasets
Segmentation

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>172.168.0.1</td>
<td>1</td>
<td>243</td>
</tr>
<tr>
<td>262</td>
<td>172.168.0.1</td>
<td>1</td>
<td>134</td>
</tr>
<tr>
<td>123</td>
<td>172.168.0.1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>221</td>
<td>172.168.0.1</td>
<td>2</td>
<td>37</td>
</tr>
<tr>
<td>234</td>
<td>172.168.0.3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

We just lost our $O(1)$ look-up time and are back to $O(n)$ reads ...

Distributed Data Management
Storage and Retrieval
ThorstenPapenbrock
Slide 28
Design a distributed DBMS for **fast storage and retrieval** of huge and evolving datasets
### Huge and Evolving Datasets

#### Segmentation

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>172.168.0.1</td>
<td>1</td>
<td>243</td>
</tr>
<tr>
<td>262</td>
<td>172.168.0.1</td>
<td>1</td>
<td>134</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>172.168.0.1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>221</td>
<td>172.168.0.1</td>
<td>2</td>
<td>37</td>
</tr>
<tr>
<td>234</td>
<td>172.168.0.3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>314</td>
<td>172.168.0.3</td>
<td>1</td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>434</td>
<td>172.168.0.3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>593</td>
<td>172.168.0.6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>712</td>
<td>172.168.0.6</td>
<td>3</td>
<td>325</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>662</td>
<td>172.168.0.6</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>724</td>
<td>172.168.0.6</td>
<td>2</td>
<td>52</td>
</tr>
</tbody>
</table>

---

**Distributed Data Management**

Storage and Retrieval

Thorsten Papenbrock
Slide 30
Fast Retrieval DBMS
How do we find a certain key efficiently?

a) A dense index?
   - All key-value pairs in the segment files are indexed.
   - Direct look ups but index size equal to segment file size

b) A sparse index?
   - First key-value pair in each segment file is indexed.
   - Small index but lookup still in $O(n/p)$ with $p$ segment files

c) A sparse index + sorting?
   - First key-value pair in each segment file is indexed and segment files are sorted.
   - If a query key is not directly indexed:
     - find the next smaller key in the index (binary search)
     - find the segment file of the next smaller key (look up)
     - search for the query key in the block (binary search)
   - Small index and lookups in $O(\log(n))$
Fast Retrieval DBMS
Architecture (so far)

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hash-index**
on first key

**Sorted Segments**
with dense pointers

**Data Segments**
with some partitioning and
data of arbitrary length

Distributed Data Management
Storage and Retrieval

ThorstenPapenbrock
Slide 32
Fast Retrieval DBMS
Sorted Segments

- Current approach:
  1. All key-value pairs are sorted by their key.
     - Only pairs with larger keys can be appended:
       - If a new key cannot be appended, trigger compact+merge with compacted segment and start a new current segment!
  2. Each key appears only once.
     - No pair with an existing key can be appended:
       - If a key already exists, trigger compact+merge with compacted segment and start a new current segment!
  3. Key-value pairs have same length.
     - Find a key via binary search in the sorted segment (and its compacted sorted segment).
     - \( O(2 \times \log(n)) \) read performance now (with binary search), but we lost our \( O(1) \) write performance!
Compact + Merge

- Given two (or more) sorted segment files, their merge is calculated in linear time similarly to the merge-sort algorithm:

  1. Create an empty compacted segment file.
  2. Read all sorted segment files simultaneously.
  3. Until all files are read entirely: Copy the smallest key with its value into the compacted segment and read the file’s next key-value pair.

  - If keys are equal: Copy only the most recent key-value pair and advance both pointers.

- More efficient than merging general segment files, but still too slow for random inserts of key-value pairs!
Compact + Merge

- Given two (or more) sorted segment files, their merge is calculated in linear time similarly to the merge-sort algorithm:
Fast Retrieval DBMS

Architecture (so far)

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Hash-index on first key

Sorted Segments with dense pointers

Data Segments with some partitioning and data of arbitrary length

Let’s make this a bit worse to find an even better solution ;-)

Distributed Data Management
Storage and Retrieval

ThorstenPapenbrock
Slide 36
Sorted String Tables (SSTables)

- SSTables are special segment files with two properties:
  - Sorted (by their keys)
  - Immutable (hence, no appending writes)
- Assume variable length data, i.e., no binary search!
- First introduced by Google (in BigTable and Google File System GFS).
- Divergent interpretations of this concept exist.

Structure:

```
SSTable
<table>
<thead>
<tr>
<th>block1</th>
<th>block2</th>
<th>block3</th>
<th>block4</th>
<th>block5</th>
<th>block6</th>
<th>block7</th>
<th>block8</th>
<th>block9</th>
<th>block10</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:xa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69:6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>71:7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76:8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91:9</td>
</tr>
</tbody>
</table>
```
Sorted String Tables (SSTables)

- **Blocks**
  - Typically 64 KB
  - Are read in one disk seek operation
  - Store key-value data of any length (key look-up = sequential scan in memory)

- **Block index**
  - Stored in the last block of an SSTable
  - Indexes the first key of each block
  - Supports binary search

- **Structure:**

<table>
<thead>
<tr>
<th>SSTable</th>
<th>block1</th>
<th>block2</th>
<th>block3</th>
<th>block4</th>
<th>block5</th>
<th>block6</th>
<th>block7</th>
<th>block8</th>
<th>block9</th>
<th>block10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12:xa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9:9</td>
</tr>
</tbody>
</table>
Sorted String Tables (SSTables)

- Example:

Keep this index block in memory or read it on-demand in one additional I/O.

Compressible, because blocks are immutable and read in one I/O!
Fast Retrieval DBMS Architecture (so far)

Because every insert now triggers a compact + merge, we need only one segment file.

Hash-index
on first key

SSTables
with dense pointers

Data Segments
with some partitioning and data of arbitrary length

Distributed Data Management
Storage and Retrieval
Because SSTables can store values of arbitrary length, we can turn the data segments into SSTables and skip this index level.

---

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Hash-index**
on first key

**SSTables**
with dense pointers

**Data Segments**
with some partitioning and data of arbitrary length
Because SSTables can store values of arbitrary length, we can turn the data segments into SSTables and skip this index level.

Note: This turns the index into a clustered index and, hence, works only for primary indexes. For secondary indexes, the SSTable values are still pointers (IP+file+offset) to arbitrarily partitioned records.

<table>
<thead>
<tr>
<th>key</th>
<th>IP</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Hash-index**

on first key

**SSTables**

with range partitioning and data of arbitrary length

**Distributed Data Management**

Storage and Retrieval

ThorstenPapenbrock

Slide 42
Hash-indexes are good for **point queries**, but can we do better for **range queries**?

- **Hash-index** on first key

- We solve the write issue later ;-)
Definition

- A self-balancing, tree-based data structure, that stores values sorted by key and allows read/write access in logarithmic time.
- A generalization of a binary search tree as nodes can have more than two children:

Structure

- **Blocks:**
  - Nodes in the tree that contain key-value pairs and pointers to other blocks
  - Correspond to physical, fixed sized disk blocks/pages that are addressed and read as single units
- **Pointers:**
  - Edges in the tree that connect blocks in a tree structure
  - Correspond to physical block/page addresses

---

Fast Retrieval DBMS
B-Tree

Constraints

- Balanced:
  - Same distance from root-node to all leaf-nodes
    - Depth of the tree is in $O(\log(n))$ (= key-look-up complexity)
    - Insert/delete procedures ensure balance

- Block-Content:
  - A block contains $n$ keys and $n+1$ pointers in alternating order
  - Pointers left to a key point to blocks containing smaller keys
  - Pointers right to a key point to blocks containing larger/equal keys
    - All values in the tree are sorted by their keys!

- Block-Size:
  - Typically 4096 Byte per block; 4 Byte per key; 8 Byte per pointer
    - $4n + 8(n+1) \leq 4096 \Rightarrow n = 340$
Fast Retrieval DBMS

B-Tree

Constraints

- **Root Node:**
  - Points to underlying nodes (and values)
  - At least 2 pointers used

- **Inner Node:**
  - Points to underlying nodes (and values)
  - At least \(\lceil(n + 1)/2\rceil\) pointers used

- **Leaf Node:**
  - Points to right neighbor leaf and values
  - At least 1 neighbor pointer (if present) and \(\lceil(n + 1)/2\rceil\) value pointers used

Uses

- Any data store: most widely used index structure for DBMSs
- Sorted, dense, and sparse indexes

---

**B-Tree vs. B+-Tree**

B-Trees store keys and values in both internal and leaf nodes;
B+-Trees store values only in leaf nodes.

- The following examples show B+-Trees
Example

- **Key**
- **Block**
- **Pointer**
- **free space**

Leafs are linked!

- **Range query**: find start of the range through the tree, then scan leafs.
Fast Retrieval DBMS

B-Trees

Neighbor pointer

Grow scenario

Look up scenario

Distributed Data Management
Storage and Retrieval
Thorsten Papenbrock
Slide 48
Split and Merge operations guarantee that the B-Tree is always balanced and the blocks are filled sufficiently.
Fast Retrieval DBMS
Architecture
(so far)

B⁺-tree
on first key of each SSTable

Recall:
Each SSTable comes with its small one-block mini-index!

Distributed Data Management
Storage and Retrieval

Thorsten Papanbrock
Slide 50
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets.

Recall: Every insert still triggers a compact + merge!
Fast Storage and Retrieval

LSM-Trees

Definition

- **Log-Structured Merge-Trees (LSM Trees)** are multilayered search trees for key-value log-data that use different data structures, each of which optimized for its underlying storage medium.

Example

- First layer ($C_0$ Tree): index structure that...
  1) efficiently takes new key-value pairs in any order.
  2) outputs all contained key-value pairs in sorted order.
    - B-trees, skip-lists, red-black trees, AVL trees, ...

- Second layer ($C_1$ Tree): index structure that...
  1) is able to merge with sorted key-value pair lists.
  2) effectively compacts/compresses contained key-value pairs.
    - SSTables (+ some index structure, e.g. B-tree or block index)

Intuition

- Sorted trees are fast in-memory indexes but they outgrow main memory.
- SSTables are indexable and compact but don’t support random inserts.
  - **Insert**: Add new key-value pairs to $C_0$ Tree; frequently merge trees down the hierarchy ($C_0 \rightarrow C_1 \rightarrow C_2 \ldots$) to free memory.
  - **Read**: Search the key chronologically in every layer ($C_0 \rightarrow C_1 \rightarrow C_2 \ldots$) until the first, i.e., most recent value is found.

Merge is **required** if a block is full!
Fast Storage and Retrieval Architecture

C₀ Tree: In-memory B⁺-tree

C₁ Tree: On-disk SSTables (and B⁺-tree)
Fast Storage and Retrieval Architecture

C₀ Tree: In-memory B⁺-tree

C₁ Tree: On-disk SSTables (and B⁺-tree)

If SSTable become full, add more SSTables.
➢ C₁ tree needs to grow!

If B-block range overlaps with multiple SSTable ranges.
Fast Storage and Retrieval Architecture

**C<sub>0</sub> Tree: In-memory B<sup>+</sup>-tree**

**C<sub>1</sub> Tree: On-disk SSTables (and B<sup>+</sup>-tree)**

**C<sub>2</sub> Tree: On-disk SSTables (and B<sup>+</sup>-tree)**

Adding more layers keeps size of middle layers smaller.

- Only last layer needs to grow!
Fast Storage and Retrieval Architecture

C₀ Tree: In-memory B⁺-tree

C₁ Tree: On-disk SSTables (and B⁺-tree)

C₂ Tree: On-disk SSTables (and B⁺-tree)

Local look-up failed!

Found it!
Fast Storage and Retrieval Architecture

$C_0$ Tree: **In-memory** $B^+$-tree

$C_1$ Tree: **On-disk** SSTables (and $B^+$-tree)

$C_2$ Tree: **On-tape** SSTables (and $B^+$-tree)

Can use trees for distribution and to move older data to slower storage.
Fast Storage and Retrieval

LSM-Tree Example: $B^+$-Tree & SSTables

Example

- Insert everything into the $B^+$-tree first.
- Depth of the tree is fix.
Insert everything into the B⁺-tree first.
Depth of the tree is fixed.
If leaf is full:
1. Split without increasing depth over max?
2. Merge leaf into C₁’s SSTables.

Merge:
- Find SSTable that would take the first key of the leaf.
- Start merging that SSTable with the leaf.
- If current leaf key >= start key of next SSTable:
  - Continue merge with that SSTable.

For this example: Assume all inner nodes are full and no redistribution possible.
Fast Storage and Retrieval

**LSM-Tree Example: B⁺-Tree & SSTables**

**Example**

- Insert everything into the B⁺-tree first.
- Depth of the tree is fixed.
- If leaf is full:
  1. Split without increasing depth over max?
  2. Merge leaf into C₁'s SSTables.
- Merge:
  - Find SSTable that would take the first key of the leaf.
  - Start merging that SSTable with the leaf.
  - If current leaf key ≥ start key of next SSTable:
    - Continue merge with that SSTable.
  - If some SSTable gets full:
    - Merge that SSTable down the hierarchy.
  - If no further level exists:
    - Split the SSTable.
Insert everything into the B+-tree first.
Depth of the tree is fixed.
If leaf is full:
1. Split without increasing depth over max?
2. Merge leaf into C1’s SSTables.
Merge:
- Find SSTable that would take the first key of the leaf.
- Start merging that SSTable with the leaf.
- If current leaf key \( \geq \) start key of next SSTable:
  - Continue merge with that SSTable.
- If some SSTable gets full:
  - Merge that SSTable down the hierarchy.
  - If no further level exists:
    - Split the SSTable.

Don’t forget to balance your B+-tree!
Insert everything into the B+-tree first.
- Depth of the tree is fix.
- If leaf is full:
  1. Split without increasing depth over max?
  2. Merge leaf into C1's SSTables.

Merge:
- Find SSTable that would take the first key of the leaf.
- Start merging that SSTable with the leaf.
- If current leaf key >= start key of next SSTable:
  - Continue merge with that SSTable.
- If some SSTable gets full:
  - Merge that SSTable down the hierarchy.
  - If no further level exists:
    - Split the SSTable.
Fast Storage and Retrieval Architecture

C₀ Tree: In-memory B⁺-tree

C₁ Tree: On-disk SSTables (and B⁺-tree)

C₂ Tree: On-disk SSTables (and B⁺-tree)

Local look-up failed!

Looking up non-existing keys is super expensive!

Not found!
Fast Storage and Retrieval Architecture

C₀ Tree: In-memory B⁺-tree

C₁ Tree: On-disk SSTables (and B⁺-tree)

C₂ Tree: On-disk SSTables (and B⁺-tree)
A **Bloom filter** is a probabilistic data structure that answers set containment questions in constant time and with constant memory consumption.

- “Does element X appear in the set?”
- Answer “no” is guaranteed to be correct.
- Answer “yes” has a certain probability to be wrong (hence, “maybe”).
  - But then the concrete look-up will just fail.
  - Very nice property that allows the use of Bloom filters in exact systems.
- **Structure**
  - **Bitset** of fixed size (typically a long array)
  - One (or more) **hash functions**

Insert

Hash functions: $h_1(), h_2(), h_3()$

- A hash function hashes the key to one bit in the bitset.
- The Bloom filter implementation can use one or multiple functions.
- Trade-off: More functions reduce the probability of hash collisions but they also exhaust the bitset faster, which produces more collisions later.
Fast Storage and Retrieval

Bloom filter

Insert

Given the number of hash functions, the number of expected items, and a target false positives rate, the minimum size of the bitset can be calculated [1].


Bitset

- Fixed array of bits.
- Increasing the array size decreases the probability of hash collisions especially when multiple hash functions are used.

Presentation Title

Speaker, Job
Description, Date if needed
Chart 69
Fast Storage and Retrieval
Bloom filter

Query

1 0 0 1 1 0 1 0 0 1 1 1 0 1 0 0

30

h₁() h₂() h₃()

maybe

Presentation Title
Speaker, Job
Description, Date if needed
Chart 70
Fast Storage and Retrieval

Bloom filter

Query

no
Fast Storage and Retrieval

Bloom filter

Do you have 'key1'?  
No  

Do you have 'key2'?  
Yes: here is key2  

Do you have 'key3'?  
No  

Filter:  
Yes  

False Positive  

Filter:  
Yes  

unnecessary disk access  

Yes: here is key2

Storage:  
Yes

Storage:  
No


Presentation Title

Speaker, Job
Description, Date if needed
Chart 72
Optimizations

Querying non-existent values is expensive (check all layers)

- Catch most of these queries with a Bloomfilter

- Size-tiered compaction
  - Merge newer, smaller SSTable layers successively into older, larger SSTable layers

- Catch most of these queries with a Bloomfilter

- Merge newer, smaller SSTable layers successively into older, larger SSTable layers

- More merging → fewer runs

- Look up cost vs. update cost

- Bloom filters

- Fence pointers

- Memory

- Storage

- LSM-tree
Design a distributed DBMS for fast storage and retrieval of huge and evolving datasets.
Excursus

Alternative Index Types

Clustered Index with Data (see LSM-Trees)
- Stores indexed data or parts of it within the index (plus/instead of pointers to data)
- Example: An index on attribute delivery_status allows to count pending deliveries without data access.
  - Improves the performance of certain queries.
  - Might reduce write performance and require additional storage.
  - Redundant values (in data and index) complicate data consistency.

Multi-Column Index

a) Concatenated index: Merge keys into one key.

b) Multi-dimensional index: Split multi-dim. key domain into multi-dim. shapes.
   - Example: An index on two-dim. geo locations (longitude, latitude) to answer intersection, containment, and nearest neighbor queries.
   - Most common implementation: R-Trees
**Excursus**

**R-Tree**

- A variation of a B-Tree that uses a hierarchy of rectangles as keys
- Also: balanced and block-sized nodes
- Indexed points …
  - are clustered into leaf nodes.
  - might occur in multiple clusters.
- Insertion:
  - into appropriate clusters
  - split cluster if too large
  - find smallest cluster extension \textit{via heuristic} if no cluster fits directly
Excursus
Alternative Index Types

Fuzzy Index

- Index on terms/keys that allows for value misspellings, synonyms, variations, ...
- Idea: sparse, sorted index (e.g. SSTable or B-Tree) with similarity look up
- Example: An index on attribute firstname where names might be misspelled.
  1. Look up most similar key.
  2. Scan the (sorted!) neighborhood of that key’s value for similar values.

Distributed Data Management
Storage and Retrieval
ThorstenPapenbrock
Slide 77
1. Given these two **segment files** from 16/11/2018 and 17/11/2018, calculate their compacted merge.

2. Given these two **SSTables** from 16/11/2018 and 17/11/2018, calculate their compacted merge.

### Storage and Retrieval

**Check yourself**

<table>
<thead>
<tr>
<th></th>
<th>16/11/2018</th>
<th>17/11/2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambition</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>area</td>
<td>71</td>
<td>14</td>
</tr>
<tr>
<td>argument</td>
<td>59</td>
<td>27</td>
</tr>
<tr>
<td>assumption</td>
<td>87</td>
<td>78</td>
</tr>
<tr>
<td>atmosphere</td>
<td>40</td>
<td>56</td>
</tr>
<tr>
<td>attitude</td>
<td>53</td>
<td>79</td>
</tr>
</tbody>
</table>

File here grows in this direction.
Chapter 3. Storage and Retrieval