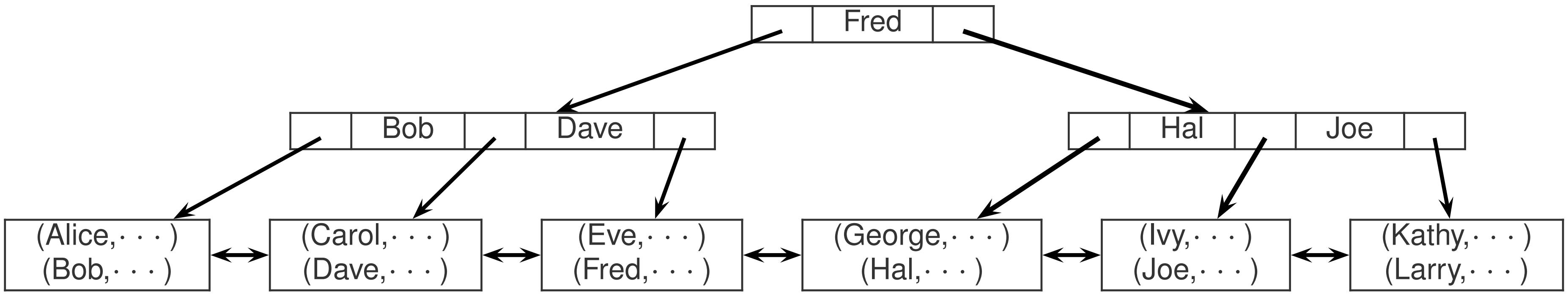
CS4224/CS5424 Lecture 4
Storage & Indexing

B$^+$-tree Index

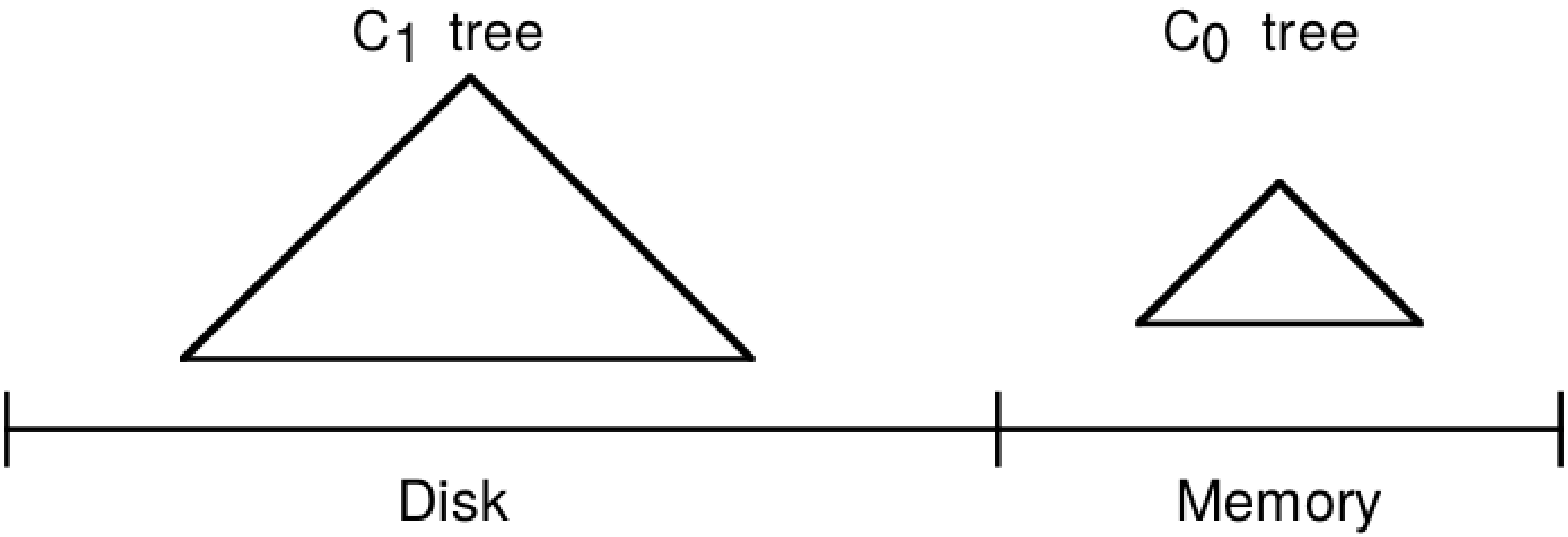


Fred
Bob
Dave
Hal
Joe
(Alice,· · ·)
(Bob,· · ·)
(Carol,· · ·)
(Dave,· · ·)
(Eve,· · ·)
(Fred,· · ·)
(George,· · ·)
(Hal,· · ·)
(Ivy,· · ·)
(Joe,· · ·)
(Kathy,· · ·)
(Larry,· · ·)

LSM Storage

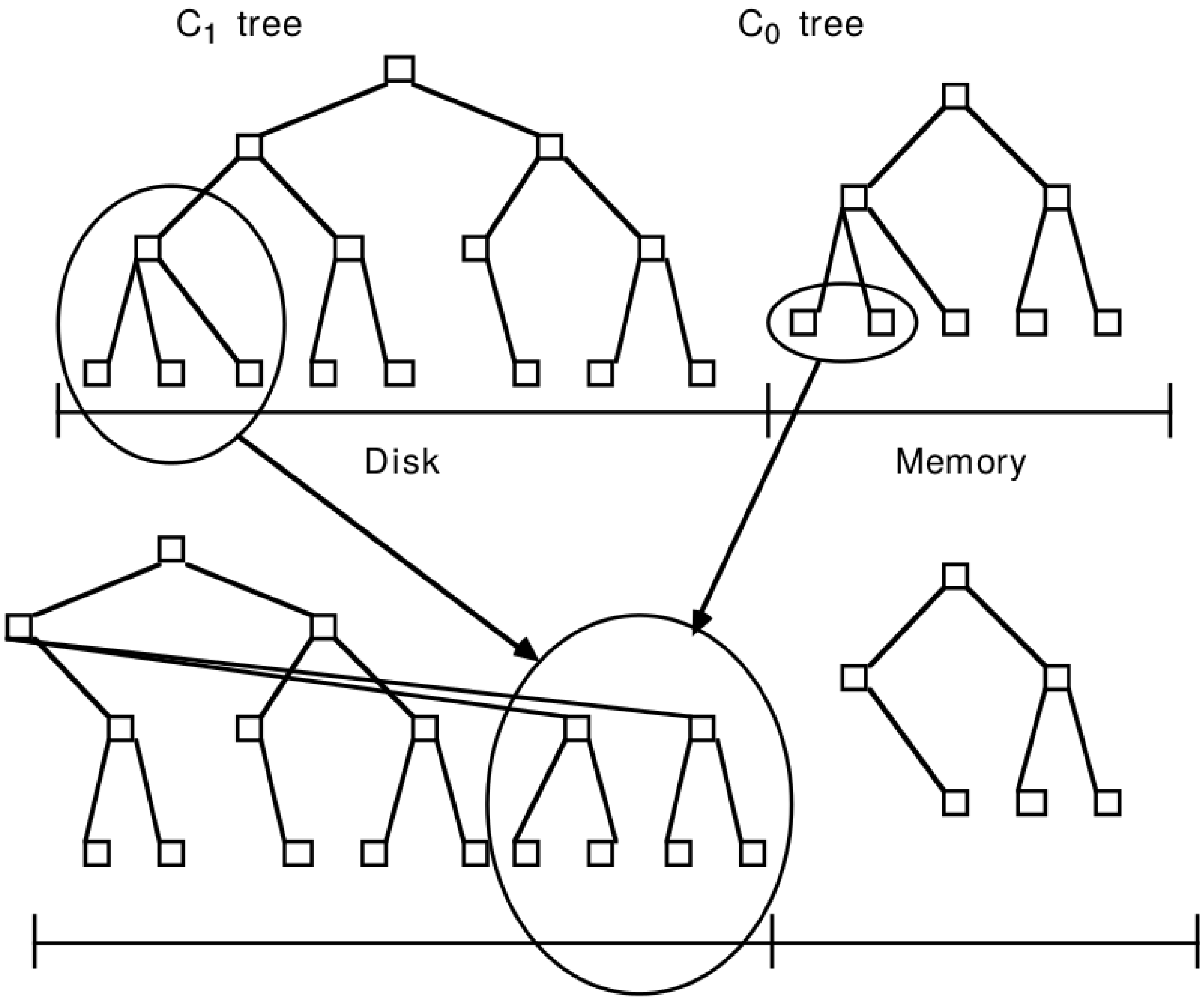
- LSM = **Log-Structured Merge**
- Inspired by LSM-Tree
  - P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, *The Log-Structured Merge-Tree (LSM-Tree)*, Acta Inf., 1996
- Improve write throughput by “converting” random I/O to sequential I/O
  - Append-only updates instead of in-place updates
- Used in BigTable, Cassandra, DynamoDB, HBase, LevelDB, MyRocks, RocksDB, SQLite4, Voldemort, WiredTiger, YugabyteDB, etc.

LSM-Tree



(O'Neil, Cheng, Gawlick, & O'Neil, 1996)

LSM-Tree (cont.)



(O'Neil, Cheng, Gawlick, & O'Neil, 1996)

LSM Storage

- LSM storage for a relation $R(K, V)$ consists of:
  - A main-memory structure MemTable
  - A set of disk-based structures SSTables
  - A commit log file
- **MemTable** = Memory Table
  - Contains the most recent updates organized in main-memory
  - MemTable is updated in-place
    - Deleted records aren't removed but marked with tombstones (denoted by $\perp$)
  - When size of MemTable reaches a certain threshold (e.g., 2MB), the records in MemTable are sorted and flushed to disk as a new SSTable
- A key may have multiple versions of values

SSTable (Sorted String Table)

- SSTables are immutable structures
- SSTable records are sorted by relation's key K
- Each SSTable is associated with a range of key values & a timestamp

Commit Log File

- A **commit log file** is used to ensure durability
- Each new update is appended to commit log & updated to MemTable

LSM Storage: Example

MemTable	SSTable 1	SSTable 2	SSTable 3
7, x	5, a	160, $\perp$	7, m
192, $\perp$	160, b	192, c	180, j
	180, d	300, a	230, n

timestamp(SSTable 1) $<$ timestamp(SSTable 2) $<$ timestamp(SSTable 3)

Range(SSTable 1) = $[5, 180]$
Range(SSTable 2) = $[160, 300]$
Range(SSTable 3) = $[7, 230]$

Compaction of SSTables

- Maintenance task to merge SSTable records
  - Improves read performance by defragmenting table records
  - Improves space utilization by eliminating tombstones & stale values
- Compaction Strategies
  - Size-tiered Compaction Strategy (STCS)
  - Leveled Compaction Strategy (LCS)
  - etc.

Compaction organizes SSTables into tiers

MemTable

MemTable
7, x
192, $\perp$

$S_{0,1}$	$S_{0,2}$
160, e	5, a
192, c	160, b
300, a	180, d

$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
15, a	8, m	19, u
70, $\perp$	12, $\perp$	192, v
180, b	230, n	200, w

$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
110, p	44, x	2, q	180, $\perp$
180, e	50, y	12, r	270, f
200, q	70, z	37, s	300, g

Size-Tiered Compaction Strategy (STCS)

- SSTables are organized into tiers with SSTables in each tier having approximately the same size
- Compaction is triggered at a tier L when the number of SSTables reaches a threshold (e.g., 4)
  - All SSTables in tier L are merged into a single SSTable that is stored in tier $L+1$
  - Tier L becomes empty after compaction

Size-Tiered Compaction: Example

Tier 0: $S_{0,1}$ $S_{0,2}$ $S_{0,3}$ $S_{0,4}$

Tier 1: $S_{1,1}$ $S_{1,2}$

Tier 0:

Tier 1: $S_{1,1}$ $S_{1,2}$ $S_{1,3}$

Example: Merging SSTables

$S_{0,1}$	$S_{0,2}$	$S_{0,3}$	$S_{0,4}$
2, q	11, x	50, p	7, e
13, r	50, y	180, $\perp$	50, f
180, s	250, z	200, q	109, g

$\longrightarrow$

$S_{1,3}$
2, q
7, e
11, x
13, r
50, f
109, g
180, $\perp$
200, q
250, z

Leveled Compaction Strategy (LCS)

MemTable

7, x
192, $\perp$

$S_{0,1}$	$S_{0,2}$
160, e	5, a
192, c	160, b
300, a	180, d

$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
8, m	50, a	190, u
12, $\perp$	70, $\perp$	192, v
23, n	180, $\perp$	200, w

$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
2, q	44, x	110, p	240, e
12, r	50, y	180, b	270, f
37, s	70, z	200, q	300, g

Leveled Compaction Strategy (LCS)

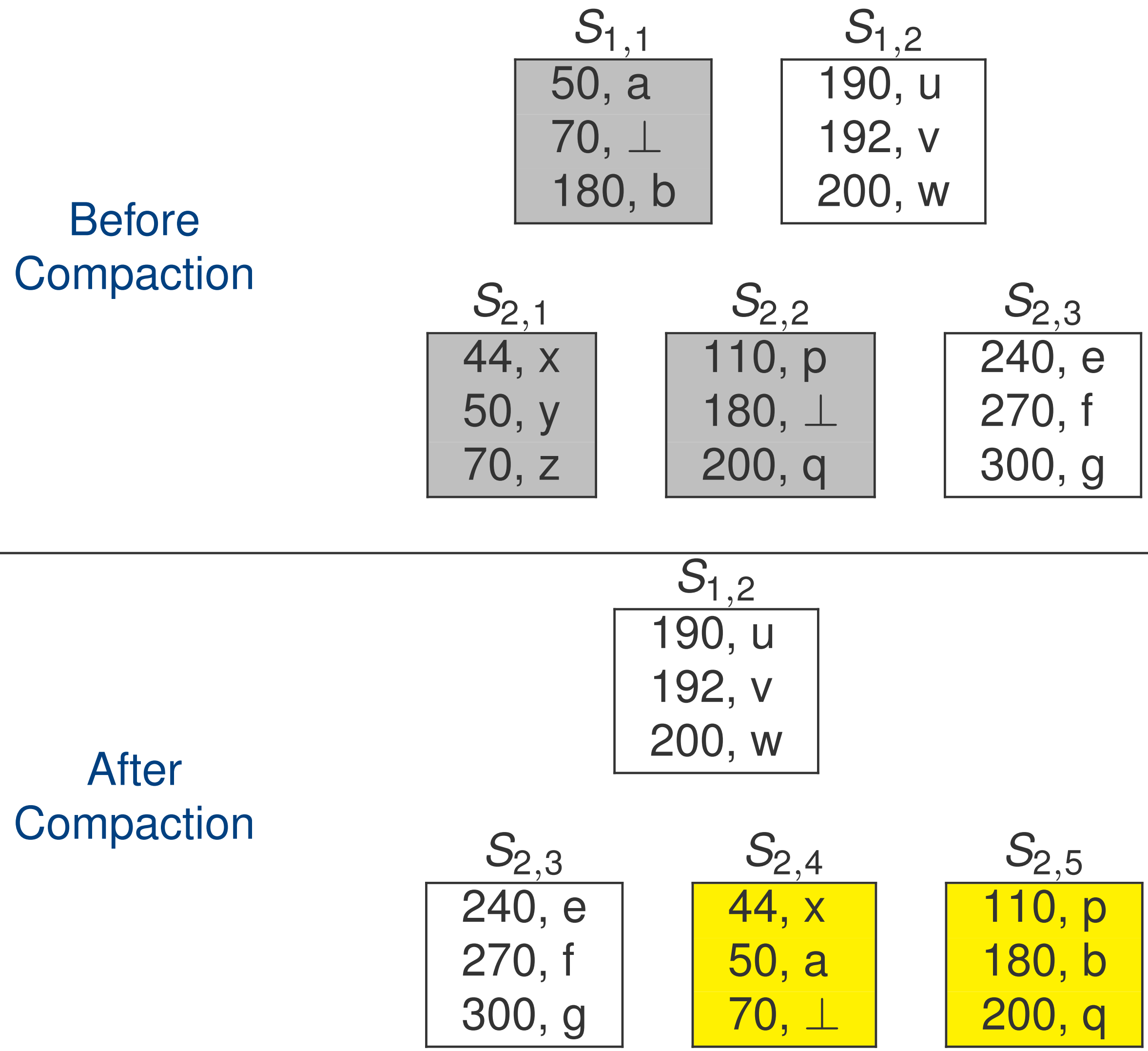
- SSTables are organized into a sequence of levels: level-0, level-1, etc.
- Two SSTables overlap if their key ranges overlap
- SSTables at level 0 may overlap
- For each level $L \geq 1$
  - Each SSTable has the same size (e.g., 2MB)
  - SSTables at the same level do not overlap
  - Each SSTable at level L overlaps with at most F SSTables at level L+1 (F = compaction factor)
- If a key appears in two SSTables at different levels i & j, $i < j$, the version at level i is more recent
- $S_{i,j}$ is more recently created than $S_{i,k}$ if $j > k$

Leveled Compaction of SSTables

- How to perform compaction at level L?
- **L** $\geq$ **1**:
  - Select a SSTable S at level L
    - Let v be the ending key of the last compaction at level L
    - S is the first level-L SSTable that starts after v if it exists; otherwise, S is the level-L SSTable with smallest start key value
  - Merge S with all overlapping SSTables at level $L + 1$
- **L** $=$ **0**:
  - Merge all SSTables at level 0 with all overlapping SSTables at level 1
- New SSTables are stored at level $L + 1$
- Old SSTables are removed

Example: Compaction of $S_{1,1}$

- Merges $S_{1,1}$ with $\{S_{2,1}, S_{2,2}\}$ to $\{S_{2,4}, S_{2,5}\}$



Example: Compaction at Level 0

- Merge all level-0 SSTables with overlapping level-1 SSTables
- **Example**:

Before Compaction

$\text{Range}(S_{0,1}) = [20, 400]$	$\text{Range}(S_{1,1}) = [2, 201]$
$\text{Range}(S_{0,2}) = [12, 601]$	$\text{Range}(S_{1,2}) = [250, 419]$
$\text{Range}(S_{0,3}) = [5, 507]$	$\text{Range}(S_{1,3}) = [520, 680]$
$\text{Range}(S_{0,4}) = [40, 101]$	$\text{Range}(S_{1,4}) = [708, 1001]$
	$\text{Range}(S_{1,5}) = [1040, 1560]$

After Compaction

$\text{Range}(S_{1,4}) = [708, 1001]$	$\text{Range}(S_{1,6}) = [2, 185]$
$\text{Range}(S_{1,5}) = [1040, 1560]$	$\text{Range}(S_{1,7}) = [199, 240]$
	$\text{Range}(S_{1,8}) = [247, 376]$
	$\text{Range}(S_{1,9}) = [387, 520]$
	$\text{Range}(S_{1,10}) = [543, 680]$

When to trigger leveled compaction?

- Based on size threshold for SSTables
- *Size*(L) = total size (in MB) of all level-L SSTables
- Level 0: Compact when the number of level-0 STTables reaches a threshold (e.g., 8)
- Level L, $L \geq 1$: Compact when $Size(L) > F^L$ MB
  - F = 10 in LevelDB
- Each level stores F times as much data as previous level
  - $Size(L) \leq F^L$ MB, $L \geq 1$

Searching LSM Storage

MemTable

7, x
192, $\perp$

$S_{0,1}$	$S_{0,2}$
160, e	5, a
192, c	160, b
300, a	180, d

$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
8, m	50, a	190, u
12, $\perp$	70, $\perp$	192, v
23, n	180, $\perp$	200, w

$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
2, q	44, x	110, p	240, e
12, r	50, y	180, b	270, f
37, s	70, z	200, q	300, g

Search Example 1: search key = 7

MemTable

7, x
192, ⊥

$S_{0,1}$	$S_{0,2}$
160, e	5, a
192, c	160, b
300, a	180, d

$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
8, m	50, a	190, u
12, ⊥	70, ⊥	192, v
23, n	180, ⊥	200, w

$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$
2, q	44, x	110, p	240, e
12, r	50, y	180, b	270, f
37, s	70, z	200, q	300, g

Search Example 2: search key = 160

MemTable

7, x
192, $\perp$

$S_{0,1}$

160, e
192, c
300, a

$S_{0,2}$

5, a
160, b
180, d

$S_{1,1}$

8, m
12, $\perp$
23, n

$S_{1,2}$

50, a
70, $\perp$
180, $\perp$

$S_{1,3}$

190, u
192, v
200, w

$S_{2,1}$

2, q
12, r
37, s

$S_{2,2}$

44, x
50, y
70, z

$S_{2,3}$

110, p
180, b
200, q

$S_{2,4}$

240, e
270, f
300, g

Search Example 3: search key = 200

MemTable

- 7, x
- 192, $\perp$

$S_{0,1}$

- 160, e
- 192, c
- 300, a

$S_{0,2}$

- 5, a
- 160, b
- 180, d

$S_{1,1}$

- 8, m
- 12, $\perp$
- 23, n

$S_{1,2}$

- 50, a
- 70, $\perp$
- 180, $\perp$

$S_{1,3}$

- 190, u
- 192, v
- 200, w

$S_{2,1}$

- 2, q
- 12, r
- 37, s

$S_{2,2}$

- 44, x
- 50, y
- 70, z

$S_{2,3}$

- 110, p
- 180, b
- 200, q

$S_{2,4}$

- 240, e
- 270, f
- 300, g

LCS: Search Algorithm

EqualitySearch (k)
Input: search key k
Output: value of k if found; otherwise, *null*
if (k is found in MemTable) then return k's value
let $S_{0,1}, \cdots, S_{0,n}$ be the sequence of level-0 SSTables
  where $S_{0,i+1}$ is more recent than $S_{0,i}$
for i = n downto 1 do
  if ($k \in Range(S_{0,i})$) then
    Search $S_{0,i}$ for k; if found then return k's value
let m be the maximum number of levels of SSTables
for L = 1 to m do
  let $S_{L,1}, S_{L,2}, \cdots$ be the sequence of level-L SSTables
  if there exists i such that $k \in Range(S_{L,i})$ then
    Search $S_{L,i}$ for k; if found then return k's value
return *null*

Optimizing SSTable Search

- Each SSTable is stored as a file consisting of a sequence of data blocks

Block 1	Block 2	· · · · · · ·	Block n-1	Block n

- How to optimize SSTable search?
  - Given a SSTable S and search key k, which block in S could contain k?
  - Given a block B and search key k, does B contain k?

Optimization 1: Sparse Index

- Assume each SSTable is 2MB consisting of 512 4KB blocks
- **Problem**: How to quickly locate SSTable block for a given search key?
- **Solution**: Build a sparse index for each SSTable
  - Sparse index: $(k_1, k_2, \cdots, k_{512})$
  - Each k_i = the first key value in the i^{th} block of SSTable
- **Example**: Consider the following sparse index for a SSTable:

k_1	k_2	k_3	k_4	$\cdots$	k_{512}
5	26	79	204	$\cdots$	8790

To look for key 90 in this SSTable, search the third block

Optimization 2: Bloom Filter

- **Problem**: How to quickly determine whether a search key exists in a SSTable block?
- **Solution**: Build a bloom filter for each block
- Bloom filter = Space-efficient randomized data structure for representing a set to support membership queries
  - B. H. Bloom, *Space/Time Trade-offs in Hash Coding with Allowable Errors*, CACM, 13(7), 422-426, 1970
- Represent a set $S = \{x_1, x_2, \cdots, x_n\}$ using a m-bit array, $B[1...m]$
  - k independent hash functions: $h_1, h_2, \cdots, h_k$
  - $h_i : S \rightarrow \{1, 2, \cdots, m\}$

Optimization 2: Bloom Filter (cont.)

```
CreateBloomFilter (S, m, h_1, ..., h_k)
01.  Initialize B[i] = 0 for i = 1 to m
02.  for x ∈ S do
03.      for i = 1 to k do
04.          j = h_i(x)
05.          set B[j] = 1
06.  return B
```

How to use bloom filter to determine if $x \in S$?

- If there exists $i \in [1, k]$ such that $h_i(x) = j$ and $B[j] = 0$, then $x \notin S$
- Otherwise, x could be in S
  - x is called a false positive if x is actually not in S

Optimization 2: Bloom Filter (cont.)

- Build a bloom filter B for $S = \{\text{Curly, Larry, Moe}\}$ with m=16 & k=3

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
Curly	13	1	4
Larry	5	10	2
Moe	8	2	11

	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
B	0	0	0	1	0	1	1	0	1	0	0	1	1	0	1	1

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
Alice	4	6	13
Bob	1	10	8

- Based on B, is Alice $\in S$?
- Based on B, is Bob $\in S$?

Scatter-gather Queries

- Consider a relation R(A,B,C) that is hash partitioned using attribute A
- Consider two queries on R
  - Q1: select * from R where A = 10 and B $>$ 20
  - Q2: select * from R where B $>$ 20
- Q2 is an example of a scatter-gather query
  - Need to access every partition to process query!

Indexing

Customers

cust#	cname	city
1	Alice	Singapore
2	Bob	Jarkata
3	Carol	Bangkok
4	Dave	Jarkata
5	Eve	Singapore
6	Fred	Penang
7	George	Hanoi
8	Hal	Bangkok
9	Ivy	Singapore
10	Joe	Penang
11	Kathy	Singapore
12	Larry	Jarkata

Index on Customers.city

Bangkok	3, 8
Hanoi	7
Jarkata	2, 4, 12
Penang	6, 10
Singapore	1, 5, 9, 11

How to Index Partitioned Data?

Customers$_1$

cust#	cname	city
3	Carol	Bangkok
6	Fred	Penang
9	Ivy	Singapore
12	Larry	Jarkata

Customers$_2$

cust#	cname	city
1	Alice	Singapore
4	Dave	Jarkata
7	George	Hanoi
10	Joe	Penang

Customers$_3$

cust#	cname	city
2	Bob	Jarkata
5	Eve	Singapore
8	Hal	Bangkok
11	Kathy	Singapore

Approach 1: Local Indexing

Customers$_1$

cust#	cname	city
3	Carol	Bangkok
6	Fred	Penang
9	Ivy	Singapore
12	Larry	Jarkata

Index I_1 on Customers$_1$.city

Bangkok	3
Jarkata	12
Penang	6
Singapore	9

Customers$_2$

cust#	cname	city
1	Alice	Singapore
4	Dave	Jarkata
7	George	Hanoi
10	Joe	Penang

Index I_2 on Customers$_2$.city

Hanoi	7
Jakarta	4
Penang	10
Singapore	1

Customers$_3$

cust#	cname	city
2	Bob	Jarkata
5	Eve	Singapore
8	Hal	Bangkok
11	Kathy	Singapore

Index I_3 on Customers$_3$.city

Bangkok	8
Jakarta	2
Singapore	5, 11

Approach 2: Global Indexing

city	Hash(city)
Bangkok	3
Hanoi	3
Jakarta	1
Penang	2
Singapore	1

Index on Customers.city

Bangkok	3, 8
Hanoi	7
Jarkata	2, 4, 12
Penang	6, 10
Singapore	1, 5, 9, 11

Index I_1

Jakarta	2, 4, 12
Singapore	1, 5, 9, 11

Index I_2

Penang	6, 10

Index I_3

Bangkok	3, 8
Hanoi	7

Approach 2: Global Indexing (cont.)

Customers$_1$

cust#	cname	city
3	Carol	Bangkok
6	Fred	Penang
9	Ivy	Singapore
12	Larry	Jarkata

Index I_1

Jakarta	2, 4, 12
Singapore	1, 5, 9, 11

Customers$_2$

cust#	cname	city
1	Alice	Singapore
4	Dave	Jarkata
7	George	Hanoi
10	Joe	Penang

Index I_2

Penang	6, 10

Customers$_3$

cust#	cname	city
2	Bob	Jarkata
5	Eve	Singapore
8	Hal	Bangkok
11	Kathy	Singapore

Index I_3

Bangkok	3, 8
Hanoi	7

Local vs Global Indexing

Partitioned Data

Customers$_1$

cust#	cname	city
3	Carol	Bangkok
6	Fred	Penang
9	Ivy	Singapore
12	Larry	Jarkata

Customers$_2$

cust#	cname	city
1	Alice	Singapore
4	Dave	Jarkata
7	George	Hanoi
10	Joe	Penang

Customers$_3$

cust#	cname	city
2	Bob	Jarkata
5	Eve	Singapore
8	Hal	Bangkok
11	Kathy	Singapore

Local Index

Index I_1 on Customers$_1$.city

Bangkok	3
Jarkata	12
Penang	6
Singapore	9

Index I_2 on Customers$_2$.city

Hanoi	7
Jakarta	4
Penang	10
Singapore	1

Index I_3 on Customers$_3$.city

Bangkok	8
Jakarta	2
Singapore	5, 11

Global Index

Index I_1

Jakarta	2, 4, 12
Singapore	1, 5, 9, 11

Index I_2

Penang	6, 10

Index I_3

Bangkok	3, 8
Hanoi	7

DynamoDB: Data Model

- A table is a collection of data
    - Each table contains zero or more items
- An item is a group of attributes that is uniquely identifiable among all of the other items
    - Each item is composed of one or more attributes
- Each item in a table has a unique identifier, or primary key
    - Other than the primary key, each table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand
    - Each item can have its own distinct attributes
- Two types of primary key
    - Simple primary key = (partition key)
    - Composite primary key = (partition key, sort key)

DynamoDB: Data Model (cont.)

- Each table is partitioned by hashing on the partition key
- Items with the same partition key are stored stored together in sorted order by the sort key value

Secondary Indexes

- Base table = table being indexed
- Index key = partition key & (possibly) sort key
- Each index entry contains base table's primary key value & optionally projected attribute values
- Two types of secondary indexes
  - Global Secondary Index (GSI)
  - Local Secondary Index (LSI)

Global vs Local Secondary index

Global Index	Local Index
• Index key can be simple or composite • Partition key could be different from base table's	• Index key must be composite • Partition key must be the same as base table's

Example

- Base table: Customers (cust#, cname, email, city)
  - Partition key = cust#
- LSI with schema (cust#, city, email)
  - Partition key = cust#
  - Sort key = city
  - Projected attribute = email
- GSI with schema (city, cust#, email)
  - Partition key = city
  - Base table's primary key = cust#
  - Projected attribute = email

References

- S. Ghemawat, J. Dean, LevelDB implementation

  https://github.com/google/leveldb/blob/master/doc/impl.md

- Cassandra Database Internals: How is data maintained?

  https://docs.datastax.com/en/dse/6.8/dse-arch/datastax_enterprise/dbInternals/dbIntHowDataMaintain.html

- Core components of Amazon DynamoDB

  https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html

- Improving data access with secondary indexes

  https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html