Index

Miao Qiao
The University of Auckland

THE UNIVERSITY OF
AUCKLAND
Te Whare Wananga o Tamaki Makaurau
NEW ZEALAND

Write Optimized Indices

- Performance of B+-trees can be poor for write-intensive workloads
  - One I/O per leaf, assuming all internal nodes are in memory
  - With magnetic disks, < 100 inserts per second per disk
  - With flash memory, one page overwrite per insert
- Two approaches to reducing cost of writes
  - **Log-structured merge tree**
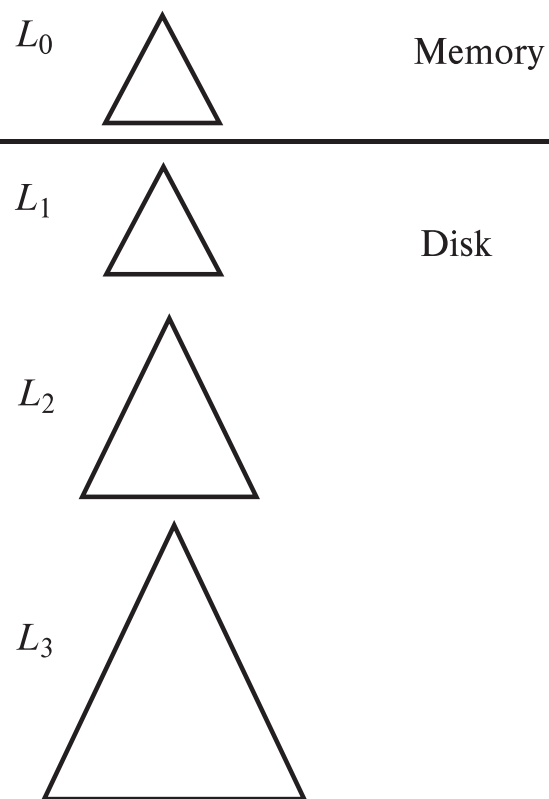 - **Buffer tree**

DATABASE STORAGE ENGINES

B-TREE
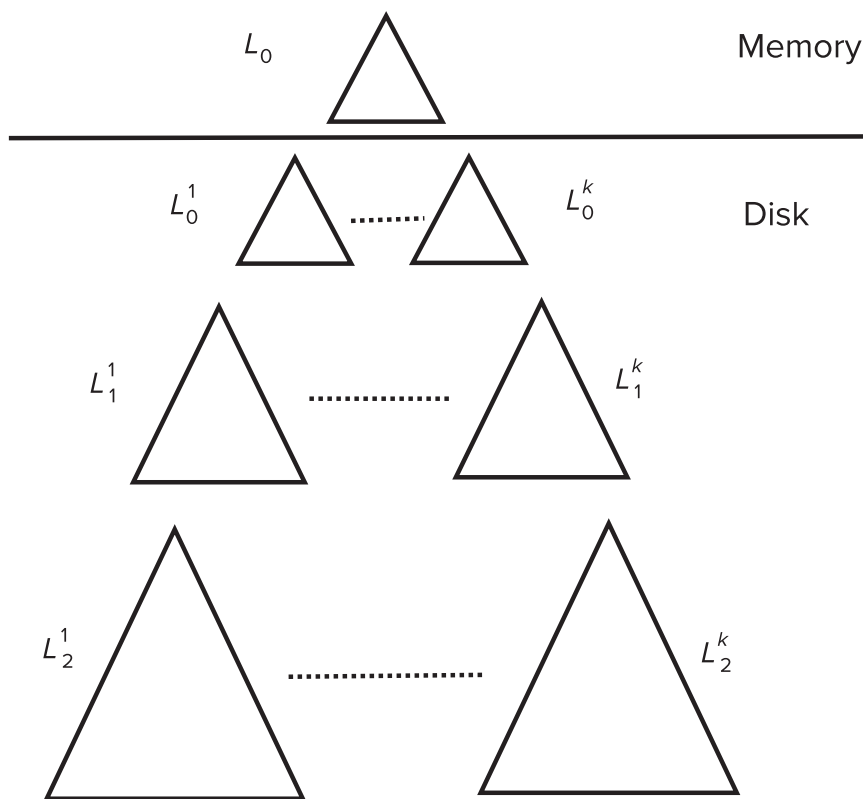
LSM TREE

Log Structured Merge (LSM) Tree

- Rolling Merge

Stepped Merge

L_0 Memory

L_1 Disk

L_2

L_3

L_0 Memory

L_0^1 .... L_0^k Disk
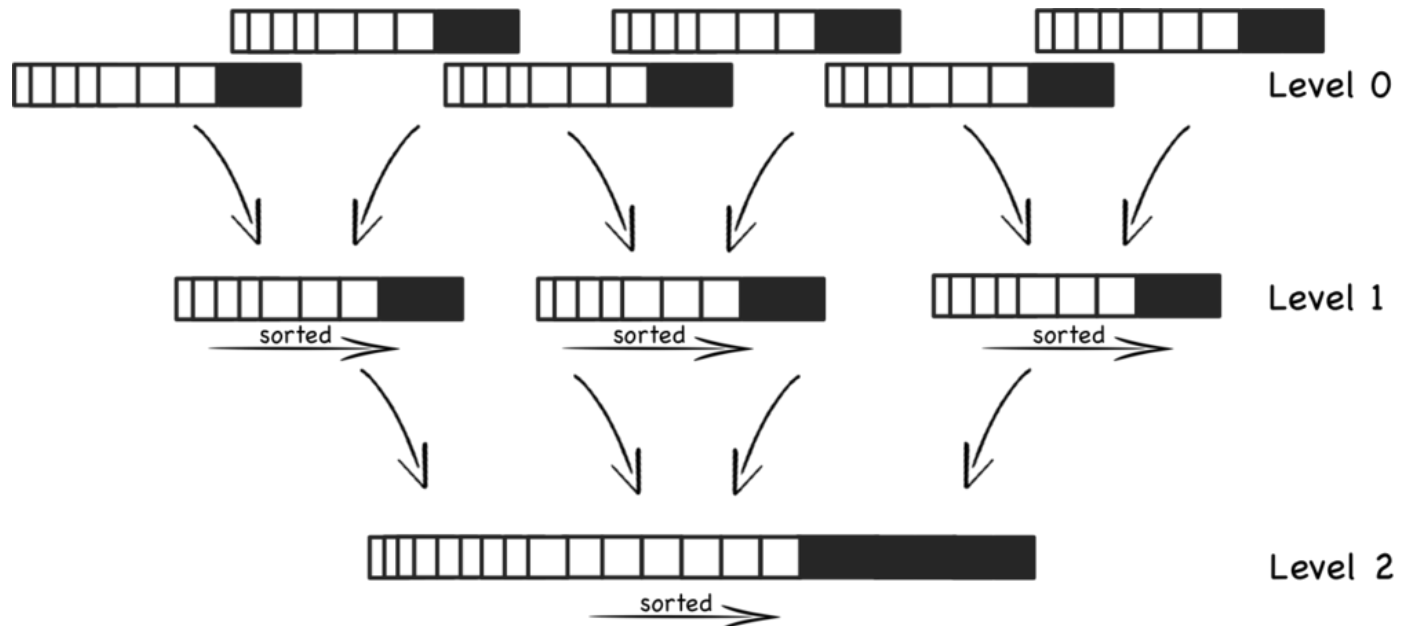
L_1^1 .... L_1^k

L_2^1 .... L_2^k

Log Structured Merge (LSM) Tree

- Stepped Merge

Compaction continues creating fewer, larger and larger files

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Log Structured Merge (LSM) Tree

- Consider only inserts/queries for now

- Records inserted first into in-memory tree (L_0 tree)

- When in-memory tree is full, records moved to disk (L_1 tree)

  - B^+-tree constructed using bottom-up build by merging existing L_1 tree with records from L_0 tree

- When L_1 tree exceeds some threshold, merge into L_2 tree

  - And so on for more levels
  - Size threshold for L_{i+1} tree is k times size threshold for L_i tree

L_0 Memory

L_1 Disk

L_2

L_3
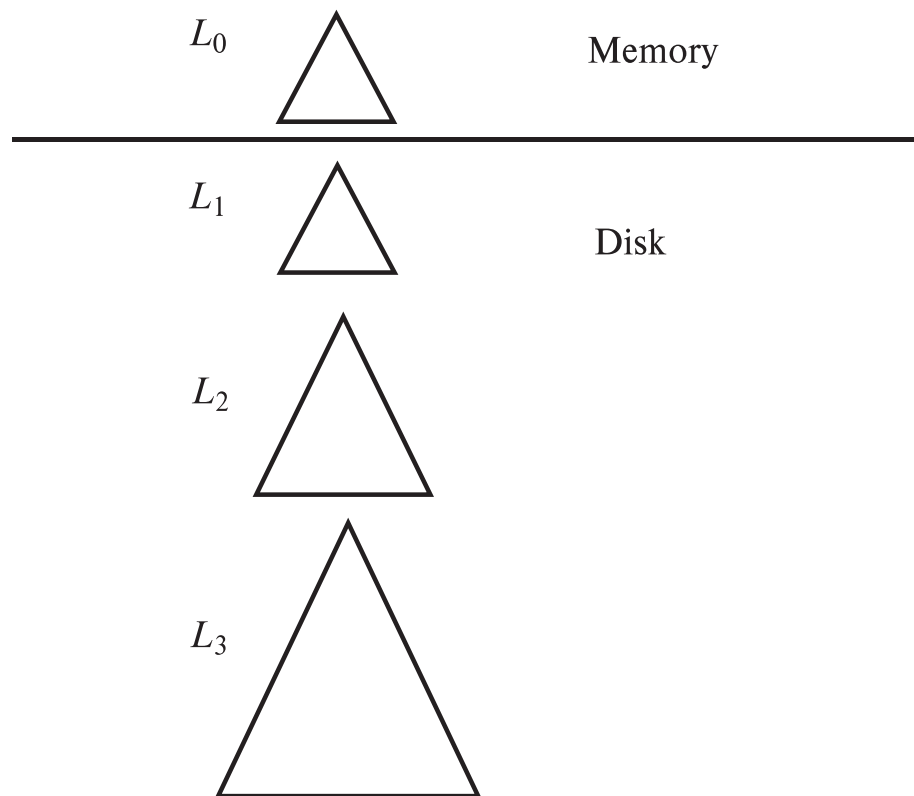
LSM Trees (Cont.)

- Deletion handled by adding special "delete" entries
    - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
    - When trees are merged, if we find a delete entry matching an original entry, both are dropped.
- Update handled using insert + delete
- LSM trees were introduced for disk-based indices
    - But useful to minimize erases with flash-based indices
    - The stepped-merge variant of LSM trees is used in many BigData storage systems
        - Google BigTable, Apache Cassandra, MongoDB
        - And more recently in SQLite4, LevelDB, and MyRocks storage engine of MySQL

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

LSM Tree (Cont.)

- Benefits of LSM approach

  - Inserts are done using only sequential I/O operations

  - Leaves are full, avoiding space wastage

  - Reduced number of I/O operations per record inserted as compared to normal B^+-tree (up to some size)

- Drawback of LSM approach

  - Queries have to search multiple trees

  - Entire content of each level copied multiple times

Optimizations of LSM

- **Rolling merge**
- LSM/Stepped Merge often implemented on a partitioned relation
  - Each partition size set to some max, split if over-sized
  - Spread partitions over multiple machines

Stepped Merge Index

- Stepped-merge index: variant of LSM tree with *k* trees at each level on disk

    - When all *k* indices exist at a level, merge them into one index of next level.

    - Reduces write cost compared to LSM tree

- But queries are even more expensive since many trees need to be queries

- Optimization for point lookups

    - Compute Bloom filter for each tree and store in-memory

    - Query a tree only if Bloom filter returns a positive result

L_0 Memory

L_0^1 .... L_0^k Disk

L_1^1 .... L_1^k

L_2^1 .... L_2^k
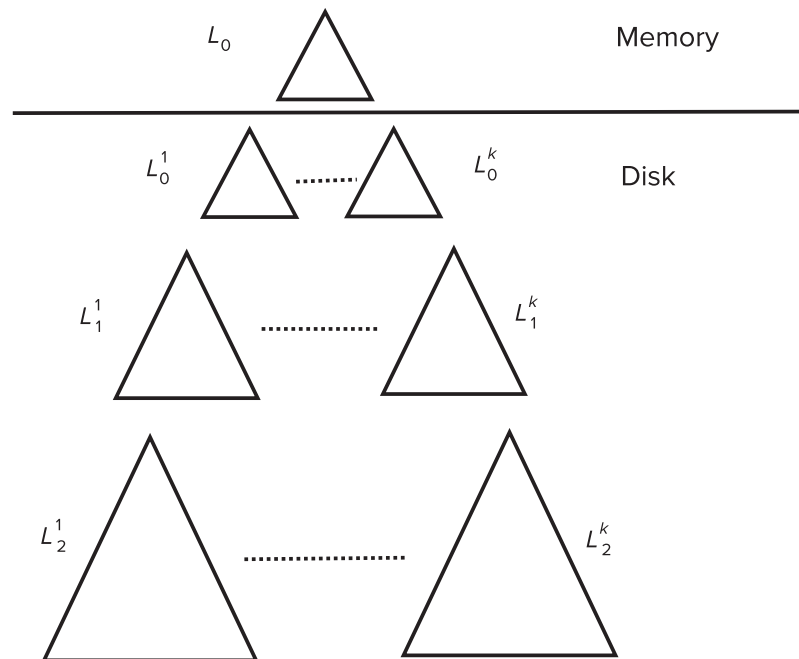
Bloom Filters

Probabilistic data structure (bitmap) that answers set membership queries.

→ False negatives will never occur.

→ False positives can sometimes occur.

→ See Bloom Filter Calculator.

Insert(x):

→ Use k hash functions to set bits in the filter to 1.

Lookup(x):

→ Check whether the bits are 1 for each hash function.

Insert 'RZA'

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Insert 'RZA'

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

$hash_1('RZA') = 2222 \% 8 = 6$

$hash_2('RZA') = 4444 \% 8 = 4$

Insert 'RZA'

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	**1**	0	**1**	0

$$hash_1('RZA') = 2222 \ \%8 = 6$$

$$hash_2('RZA') = 4444 \ \%8 = 4$$

Insert 'RZA'

Insert 'GZA'

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	**1**	0	**1**	0

$hash_1('GZA') = 5555 \%8 = 3$

$hash_2('GZA') = 7777 \%8 = 1$

Insert 'RZA'

Insert 'GZA'

Bloom Filter

0	1	2	3	4	5	6	7
0	**1**	0	**1**	**1**	0	**1**	0

$hash_1('GZA') = 5555 \% 8 = 3$

$hash_2('GZA') = 7777 \% 8 = 1$

BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA'

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('RZA') = 2222 \ \%8 = 6$

$hash_2('RZA') = 4444 \ \%8 = 4$

BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' →*TRUE*

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('RZA') = 2222 \ \%8 = 6$

$hash_2('RZA') = 4444 \ \%8 = 4$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup'Raekwon'

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon'

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$$hash_1('Raekwon') = 3333 \% 8 = 5$$

$$hash_2('Raekwon') = 8899 \% 8 = 3$$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon'

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('Raekwon') = 3333 \% 8 = 5$

$hash_2('Raekwon') = 8899 \% 8 = 3$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Bloom Filter

0	1	2	3	4	5	6	7
0	**1**	0	**1**	**1**	0	**1**	0

$hash_1('Raekwon') = 3333 \% 8 = 5$

$hash_2('Raekwon') = 8899 \% 8 = 3$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Bloom Filter

0	1	2	3	4	5	6	7
0	**1**	0	**1**	**1**	0	**1**	0

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Lookup'ODB'

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('ODB') = 6699 \% 8 = 3$

$hash_2('ODB') = 9966 \% 8 = 6$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Lookup'ODB'

Bloom Filter

0	1	2	3	4	5	6	7
0	**1**	0	**1**	**1**	0	**1**	0

$hash_1('ODB') = 6699 \ \%8 = \quad 3$

$hash_2('ODB') = 9966 \ \%8 = \quad 6$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Lookup'ODB' → *TRUE*

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$$hash_1('ODB') = 6699 \,\%8 = \quad 3$$

$$hash_2('ODB') = 9966 \,\%8 = \quad 6$$

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Lookup'ODB' →*TRUE*

Bloom filter calculator: **https://hur.st/bloomfilter/**

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1(\text{'ODB'}) = 6699 \,\%8 = \quad 3$

$hash_2(\text{'ODB'}) = 9966 \,\%8 = \quad 6$

Bloom Filters

- A **bloom filter** is a probabilistic data structure used to check membership of a value in a set

  - May return true (with low probability) even if an element is not present

  - But never returns false if an element is present

  - Used to filter out irrelevant sets

- Key data structure is a single bitmap

  - For a set with n elements, typical bitmap size is 10n

- Uses multiple independent hash functions

- With a single hash function h() with range=number of bits in bitmap:

  - For each element s in set S compute $h(s)$ and set bit $h(s)$

  - To query an element v compute $h(v)$, and check if bit $h(v)$ is set

- Problem with single hash function: significant chance of false positive due to hash collision

  - 10% chance with 10n bits

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Bloom Filters (Cont.)

- Key idea of Bloom filter: reduce false positives by use multiple hash functions $h_i()$ for $i = 1..k$

  - For each element s in set S *for each i* compute $h_i(s)$ and set bit $h_i(s)$

  - To query an element v *for each i* compute $h_i(v)$, and check if bit $h_i(v)$ is set

    - If bit $h_i(v)$ is set for every i then report v as present in set

    - Else report v as absent

  - With 10n bits, and $k = 7$, false positive rate reduces to 1% instead of 10% with $k = 1$

Buffer Tree

- Alternative to LSM tree
- Key idea: each internal node of B$^+$-tree has a buffer to store inserts
  - Inserts are moved to lower levels when buffer is full
  - With a large buffer, many records are moved to lower level each time
  - Per record I/O decreases correspondingly
- Benefits
  - Less overhead on queries
  - Can be used with any tree index structure
  - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree

Internal node
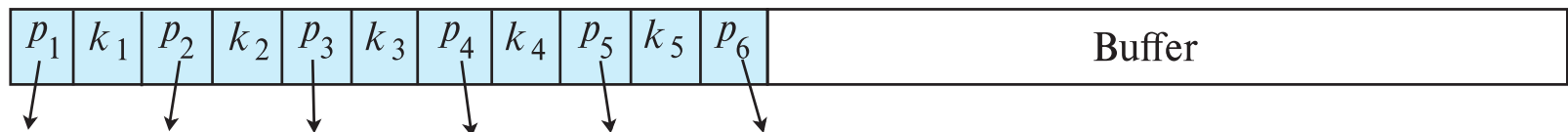
| p_1 | k_1 | p_2 | k_2 | p_3 | k_3 | p_4 | k_4 | p_5 | k_5 | p_6 | Buffer |

Trie Index

Use a digital representation of keys to examine prefixes one-by-one.
→ aka *Digital Search Tree*, *Prefix Tree*.

Shape depends on keys and lengths.
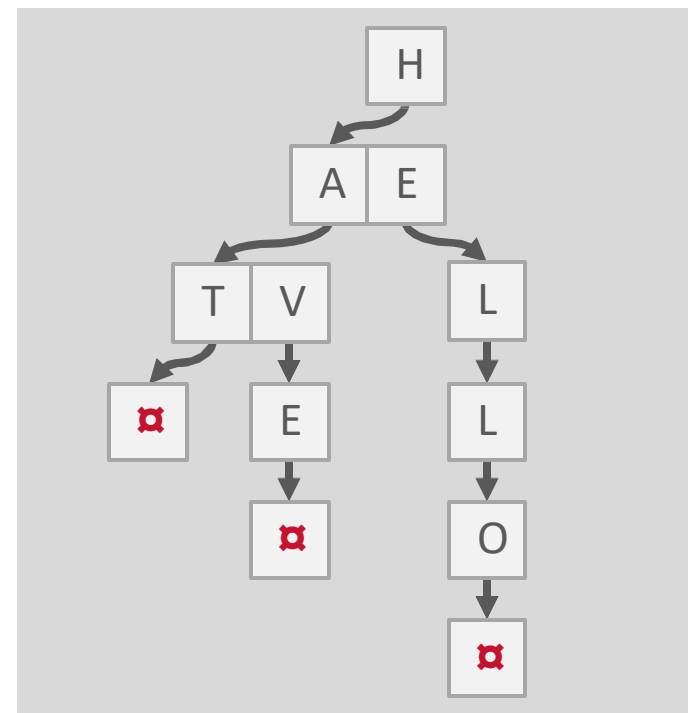→ Does <u>not</u> depend on existing keys or insertion order.
→ Does <u>not</u> require rebalancing operations.

All operations have **O(k)** complexity where **k** is the length of the key.
→ Path to a leaf node represents a key.
→ Keys are stored implicitly and can be reconstructed from paths.

Keys: HELLO, HAT, HAVE

Trie Index

Use a digital representation of keys to examine prefixes one-by-one.

→ aka *Digital Search Tree*, *Prefix Tree*.

Shape depends on keys and lengths.

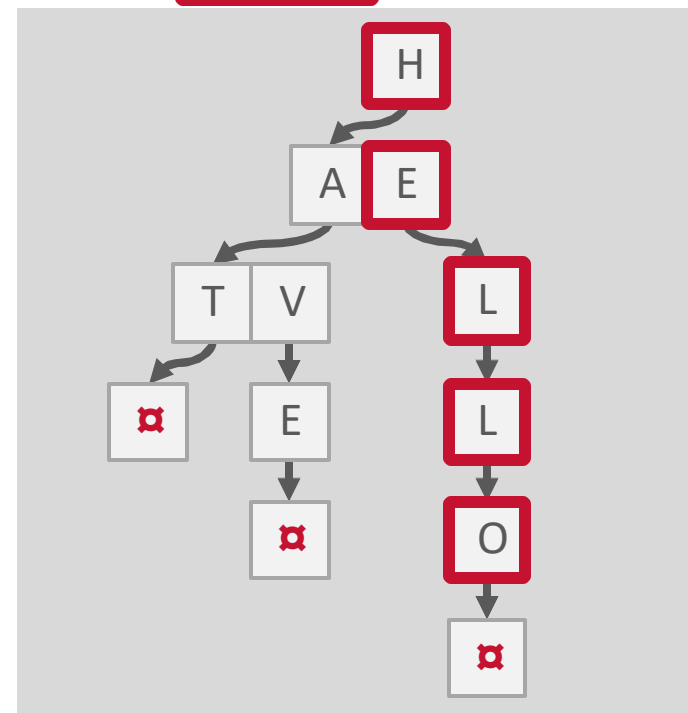→ Does <u>not</u> depend on existing keys or insertion order.

→ Does <u>not</u> require rebalancing operations.

All operations have **O(k)** complexity where **k** is the length of the key.

→ Path to a leaf node represents a key.

→ Keys are stored implicitly and can be reconstructed from paths.

Keys: HELLO, HAT, HAVE

Vector Indexes

Specialized data structures to perform nearest-neighbor searches on **embeddings**.
→ An embedding is an array of floating point numbers.
→ May also need to filter data before / after vector searches.
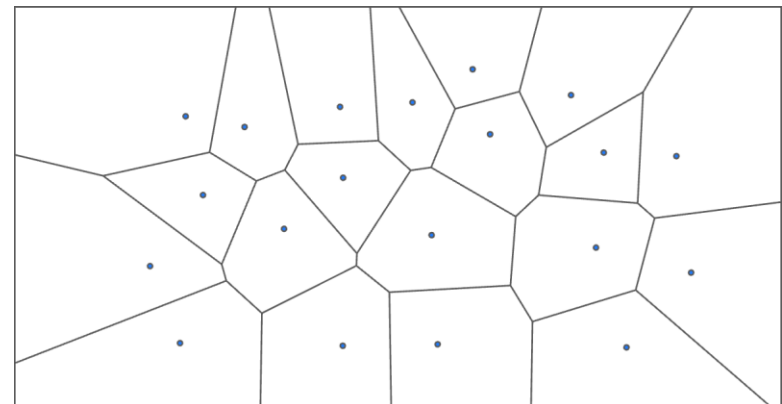
Vector Indexes: Inverted File

Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

→ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat

Source: Chi Zhang

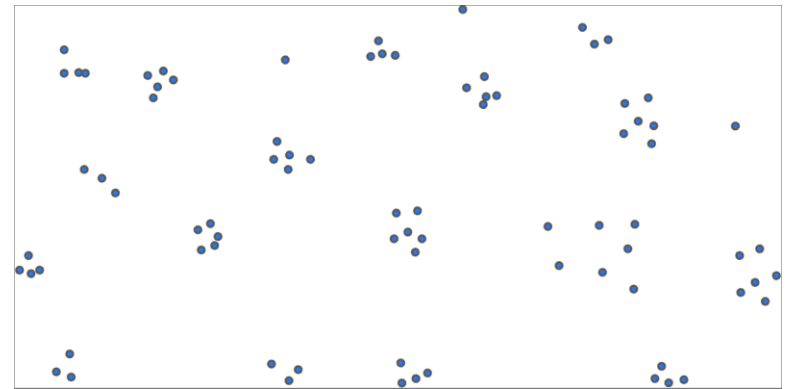# Vector Indexes: Inverted File

Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.
$\rightarrow$ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat

Source: Chi Zhang
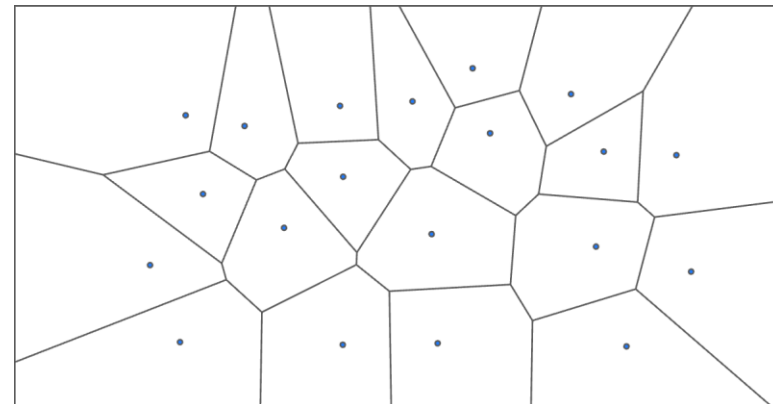
Vector Indexes: Inverted File

Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

$\rightarrow$ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat
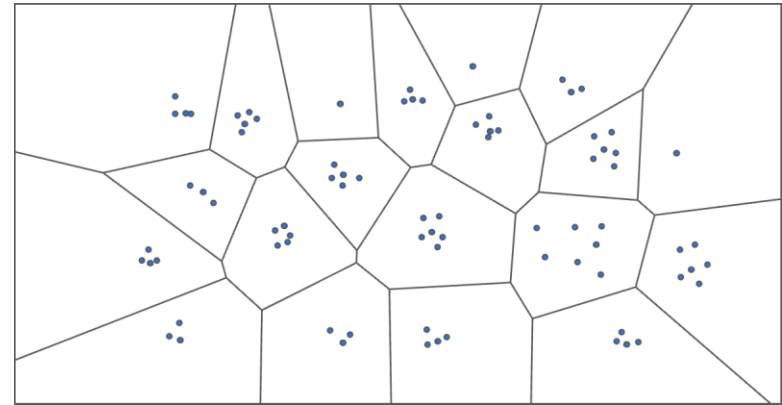
Vector Indexes: Inverted File

Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

→ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat

Vector Indexes: HNSW Graph

Build a graph where each node represents a vector and it has edges to its **_n_** nearest neighbors.
→ Can use multiple levels of graphs (HNSW)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Example: Faiss, hnswlib

Search Vector

Entry Point

Vector Indexes: HNSW Graph

Build a graph where each node represents a vector and it has edges to its **n** nearest neighbors.
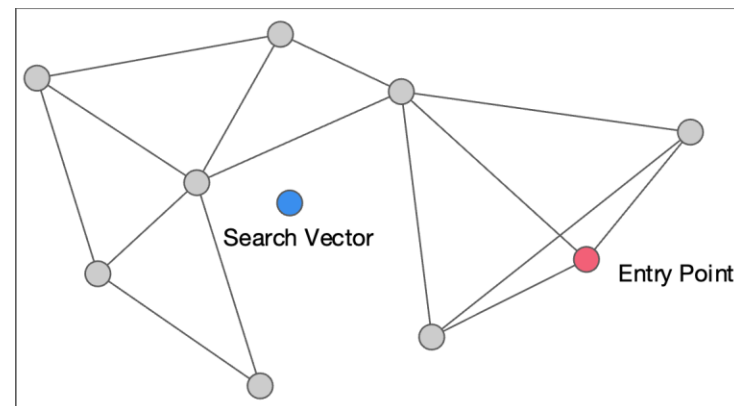→ Can use multiple levels of graphs (HNSW)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.
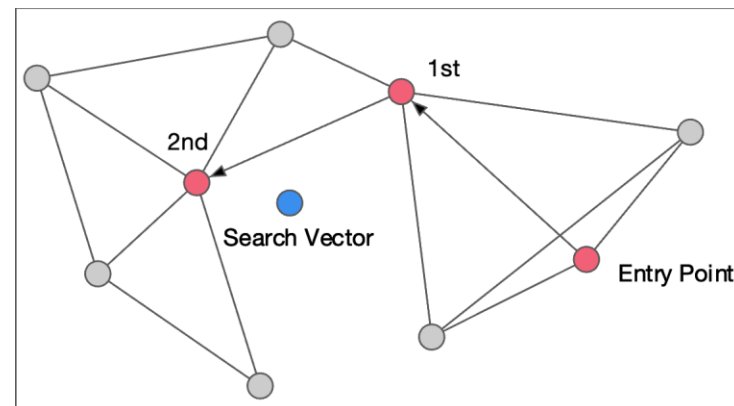
Example: Faiss, hnswlib

FIN

Any questions?

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.

bucket 0

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 1

15151	Mozart	Music	40000

bucket 5

76766	Crick	Biology	72000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 7

Example of Hash Index

bucket 0

76766	

bucket 1

45565	
76543	

bucket 2

22222	

bucket 3

10101	

bucket 4

bucket 5

15151		58583	
33456		98345	

bucket 6

83821	

bucket 7

12121	
32343	

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

hash index on *instructor,* on attribute *ID*
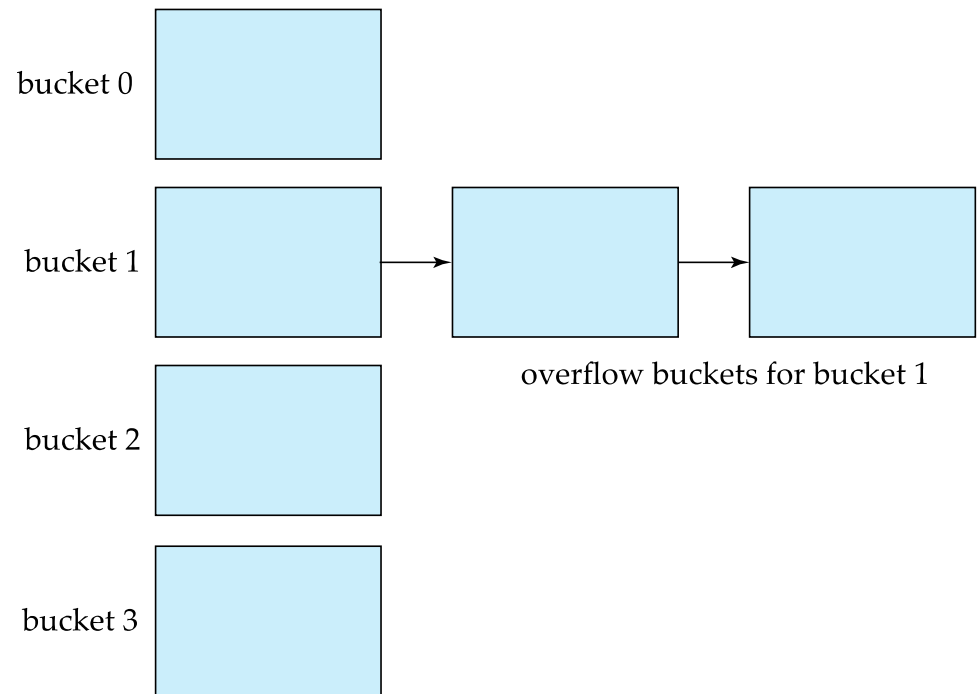
Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).

  - we obtain the bucket of an entry from its search-key value using a **hash function**

- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B.

- Hash function is used to locate entries for access, insertion as well as deletion.

- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.

- In a **hash index**, buckets store entries with pointers to records

- In a **hash file-organization** buckets store records

Handling of Bucket Overflows

- Bucket overflow can occur because of

  - Insufficient buckets

  - Skew in distribution of records. This can occur due to two reasons:

    - multiple records have same search-key value

    - chosen hash function produces non-uniform distribution of key values

- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.

- Above scheme is called **closed addressing (**also called **closed hashing or open hashing** depending on the book you use**)**

  - An alternative, called **open addressing (**also called **open hashing** or **closed hashing** depending on the book you use) which does not use over-flow buckets, is not suitable for database applications.

bucket 0

bucket 1

bucket 2

bucket 3

overflow buckets for bucket 1

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.

  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.

  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).

  - If database shrinks, again space will be wasted.

- One solution: periodic re-organization of the file with a new hash function

  - Expensive, disrupts normal operations

- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Periodic rehashing

  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,

    - create new hash table of size (say) 2 times the size of the previous hash table

    - Rehash all entries to new table

- Linear Hashing

  - Do rehashing in an incremental manner

- Extendable Hashing

  - Tailored to disk based hashing, with buckets shared by multiple hash values

  - Doubling of # of entries in hash table, without doubling # of buckets

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Extendable Hashing

- **Extendable hashing** – one form of dynamic hashing
    - Hash function generates values over a large range — typically b-bit integers, with $b = 32$.
    - At any time use only a prefix of the hash function to index into a table of bucket addresses.
    - Let the length of the prefix be i bits, $0 \leq i \leq 32$.
        - Bucket address table size = 2^i. Initially $i = 0$
        - Value of i grows and shrinks as the size of the database grows and shrinks.
    - Multiple entries in the bucket address table may point to a bucket (why?)
    - Thus, actual number of buckets is $< 2^i$
        - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure
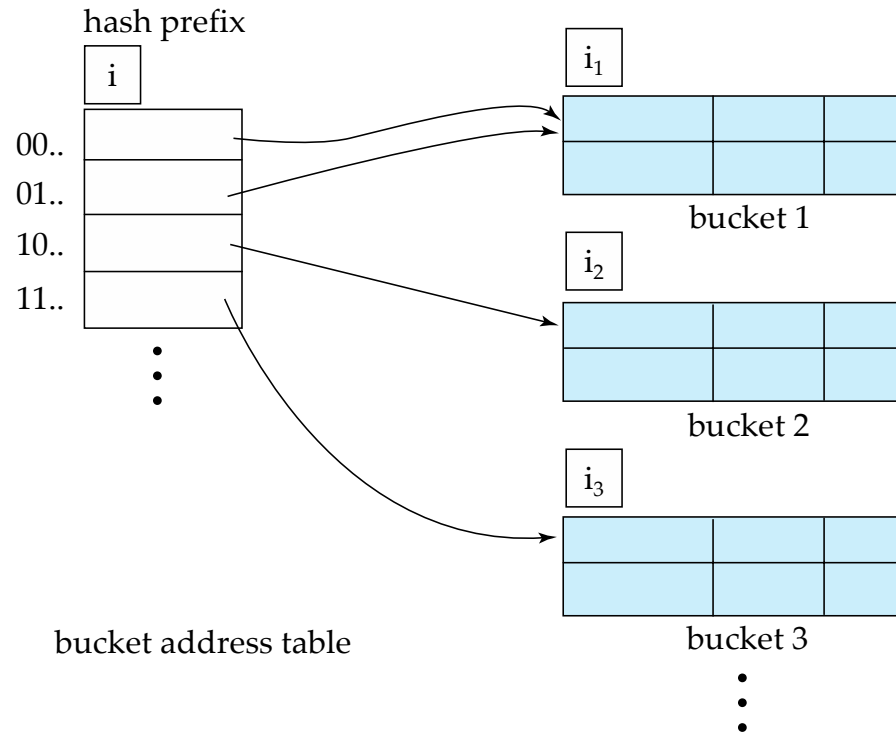
In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket j stores a value i_j

  - *A*ll the entries that point to the same bucket have the same values on the first i_j bits.

- To locate the bucket containing search-key K_j:

  1. Compute $h(K_j) = X$

  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket

- To insert a record with search-key value K_j

  - follow same procedure as look-up and locate the bucket, say j.

  - If there is room in the bucket j insert record in the bucket.

  - Else the bucket must be split and insertion re-attempted (next slide.)

    - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure (Cont.)

To split a bucket j when inserting record with search-key value K_j:

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

Deletion in Extendable Hash Structure

- To delete a key value,

    - locate it in its bucket and remove it.

    - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).

    - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)

    - Decreasing bucket address table size is also possible

        - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example (Cont.)

- Initial hash structure; bucket size = 2

hash prefix

0

bucket address table

bucket 1
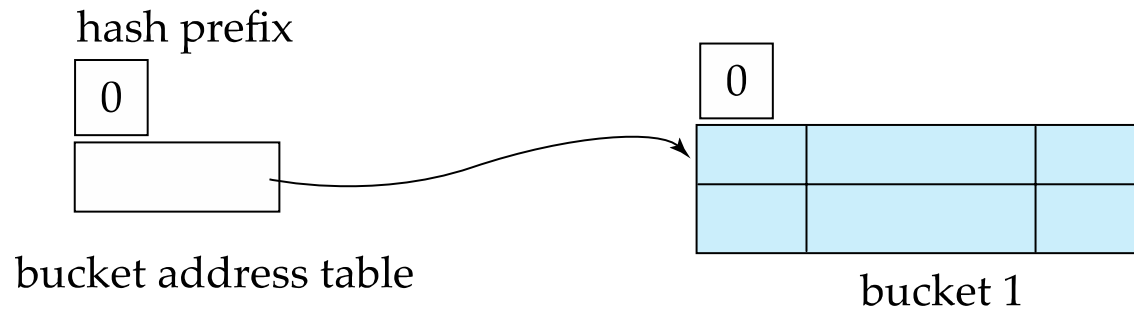
Example (Cont.)

- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records

hash prefix

1

bucket address table

1

15151	Mozart	Music	40000

1

10101	Srinivasan	Comp. Sci.	90000
12121	Wu	Finance	90000

dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example (Cont.)

- Hash structure after insertion of Einstein record
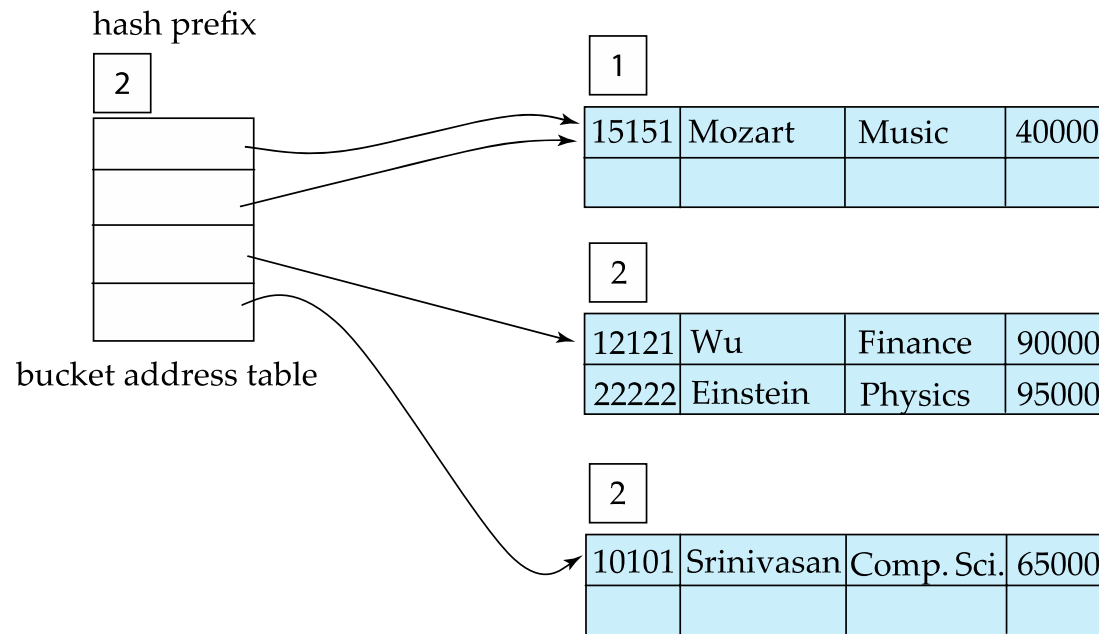
hash prefix

| 2 |

bucket address table

| 1 |

| 15151 | Mozart | Music | 40000 |
| | | | |

| 2 |

| 12121 | Wu | Finance | 90000 |
| 22222 | Einstein | Physics | 95000 |

| 2 |

| 10101 | Srinivasan | Comp. Sci. | 65000 |
| | | | |

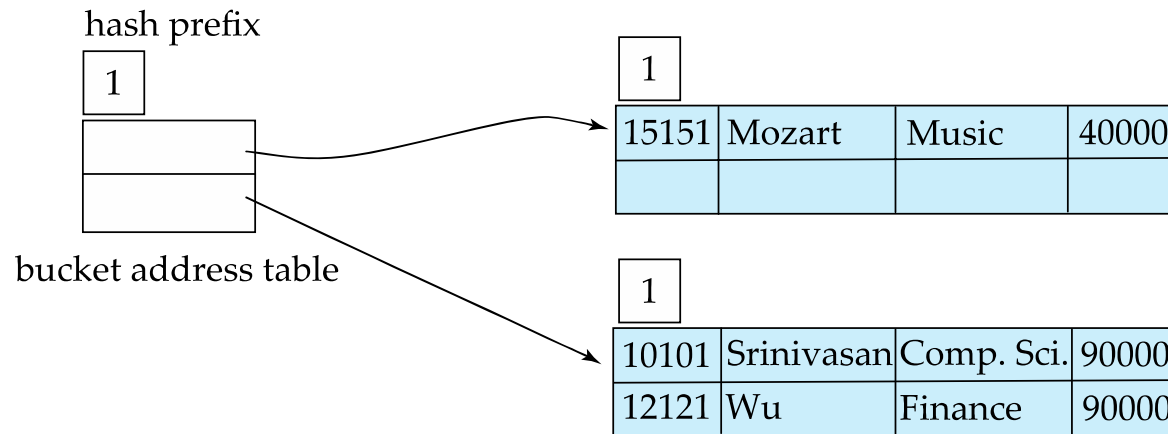| dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Example (Cont.)

- Hash structure after insertion of Gold and El Said records

hash prefix

3

bucket address table

1			
15151	Mozart	Music	40000

3			
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

3			
12121	Wu	Finance	90000

2			
10101	Srinivasan	Comp. Sci.	65000
32343	El Said	History	60000

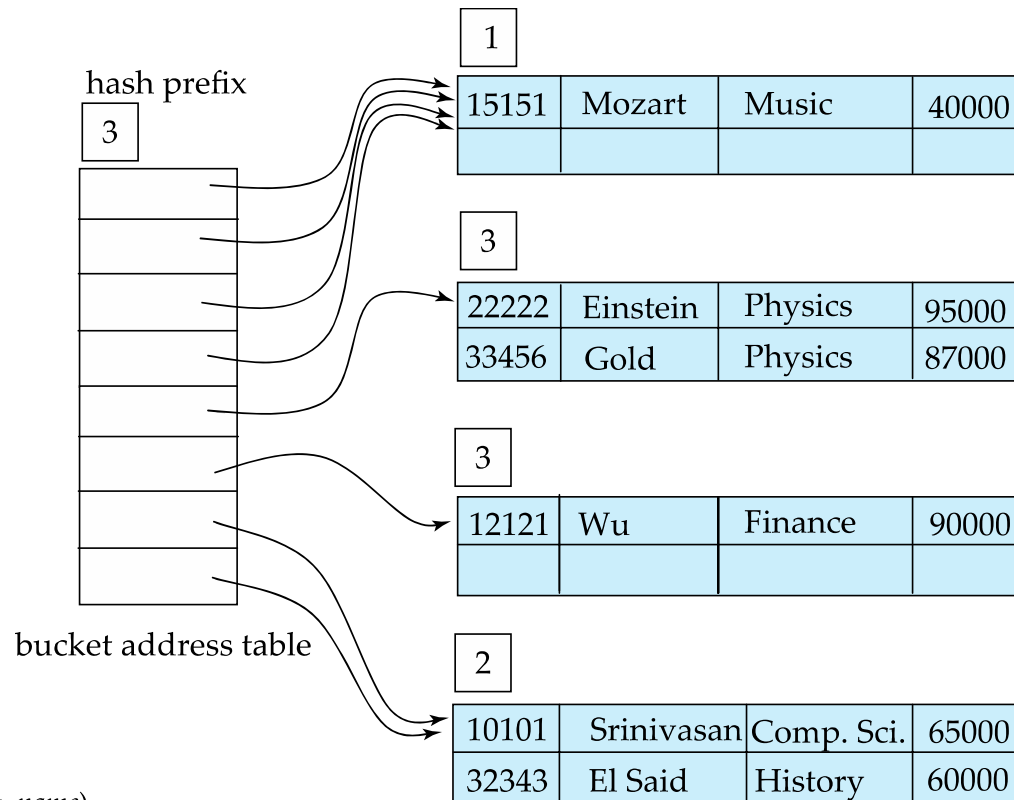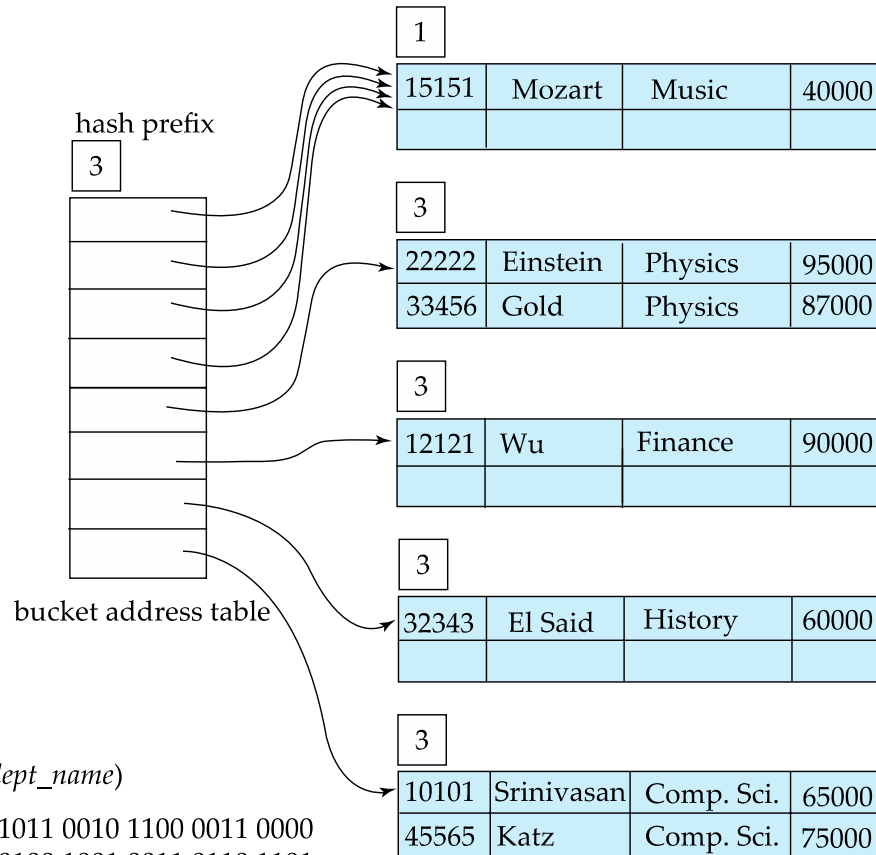| *dept_name*	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF
COMPUTER SCIENCE

Example (Cont.)

- Hash structure after insertion of Katz record

1			
15151	Mozart	Music	40000

hash prefix
3

3			
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

3			
12121	Wu	Finance	90000

bucket address table

3			
32343	El Said	History	60000

3			
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000

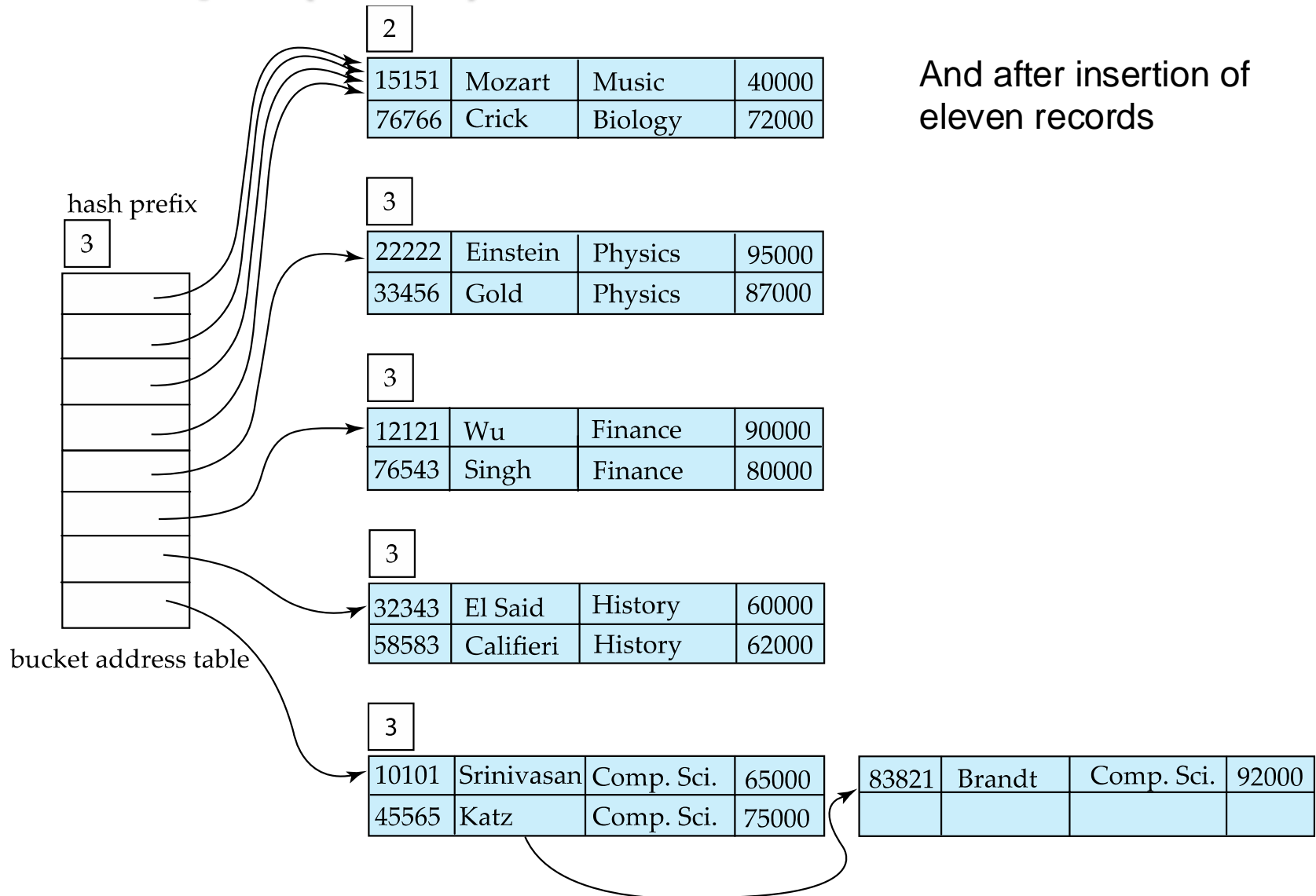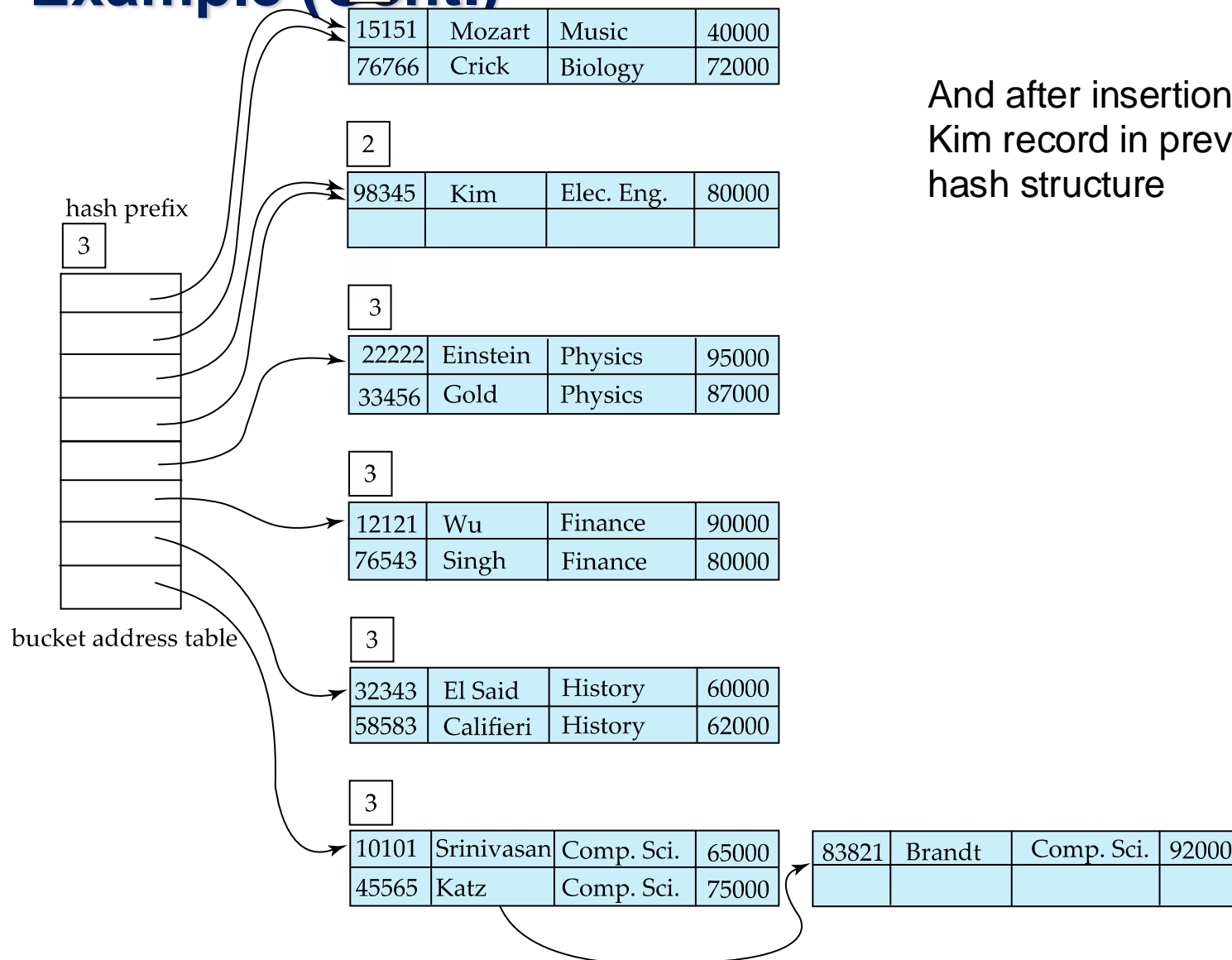| dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example (Cont.)

And after insertion of eleven records

Example (Cont.)

15151	Mozart	Music	40000
76766	Crick	Biology	72000

2

98345	Kim	Elec. Eng.	80000

hash prefix

3

bucket address table

3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000

3

12121	Wu	Finance	90000
76543	Singh	Finance	80000

3

32343	El Said	History	60000
58583	Califieri	History	62000

3

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000

83821	Brandt	Comp. Sci.	92000

And after insertion of Kim record in previous hash structure

Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B$^+$-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation
- **Linear hashing** is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows