Index

Miao Qiao
The University of Auckland

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

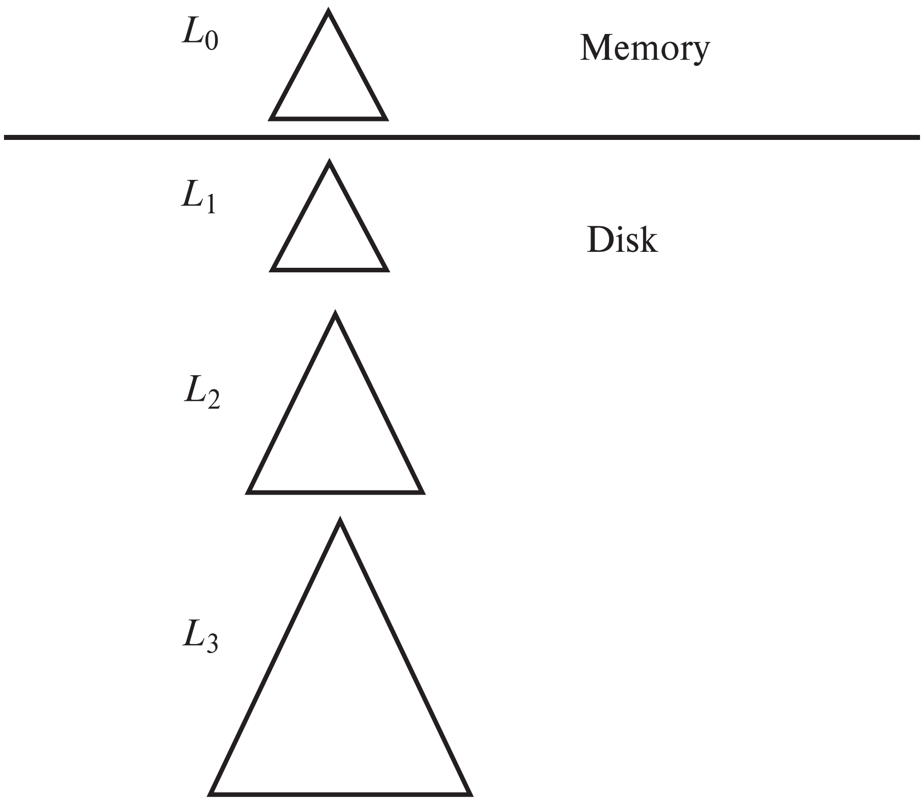
ORACLE
Microsoft SQL Server
IBM DB2
PostgreSQL
MySQL
mongoDB
Couchbase

LSM TREE

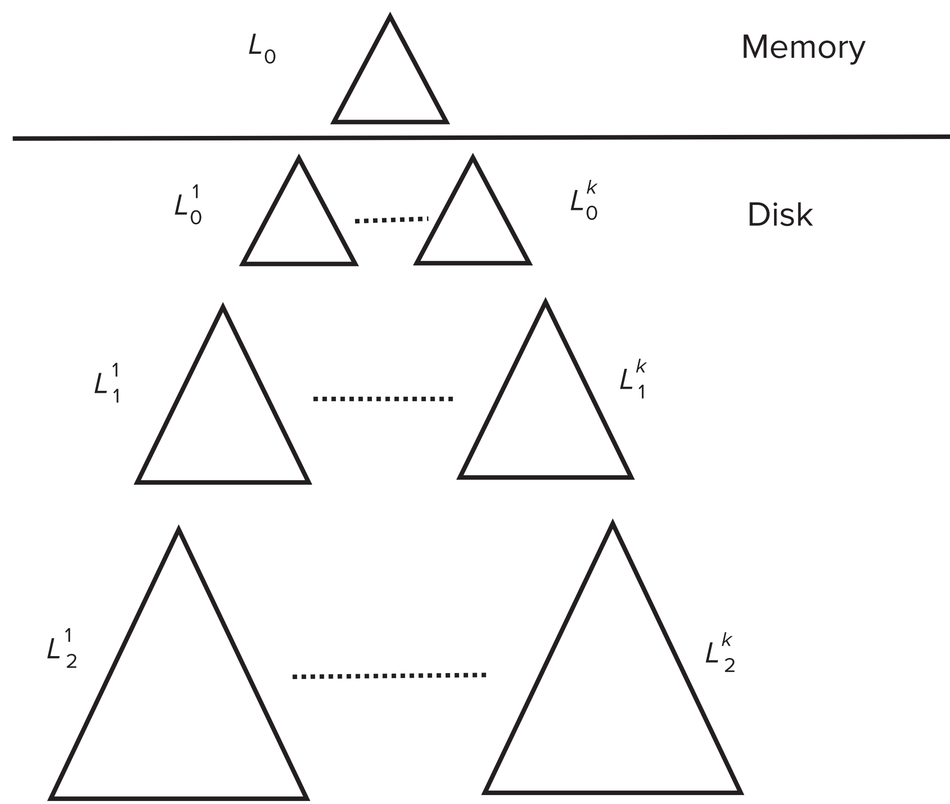
cassandra
APACHE HBASE
Google Cloud Bigtable
elasticsearch
influxdb
LEVELDB
RocksDB
yugabyteDB

Log Structured Merge (LSM) Tree

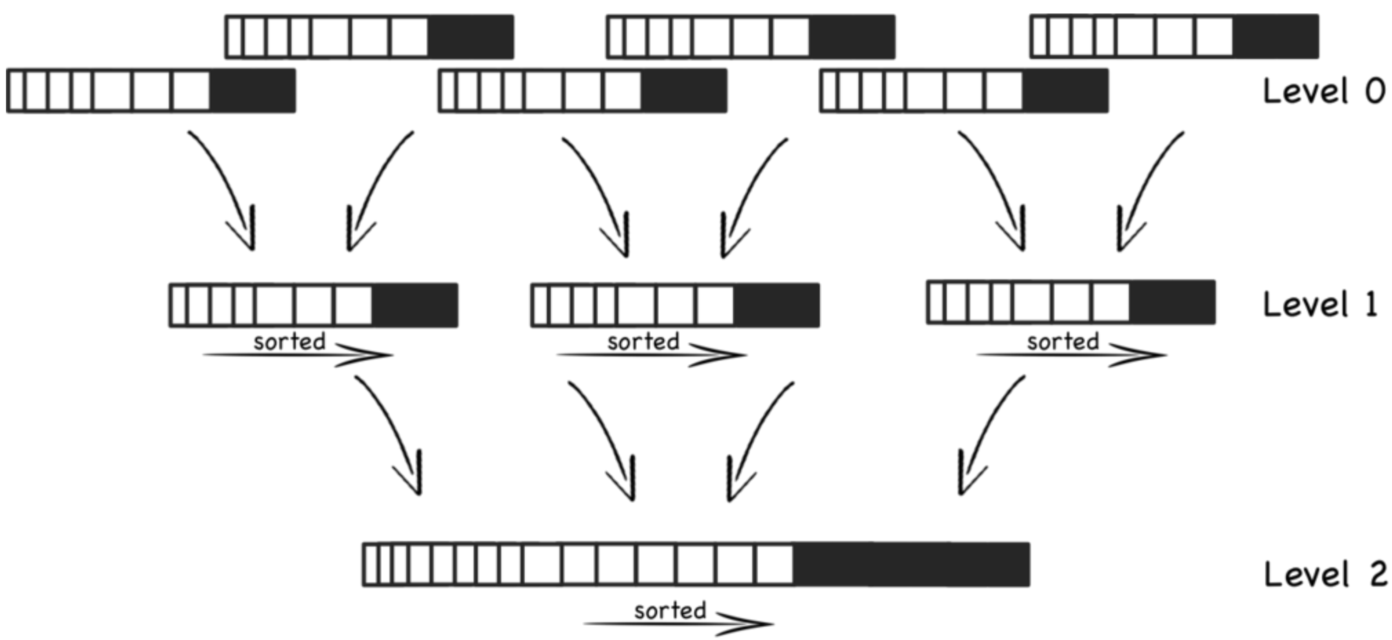
- Rolling Merge



Stepped Merge



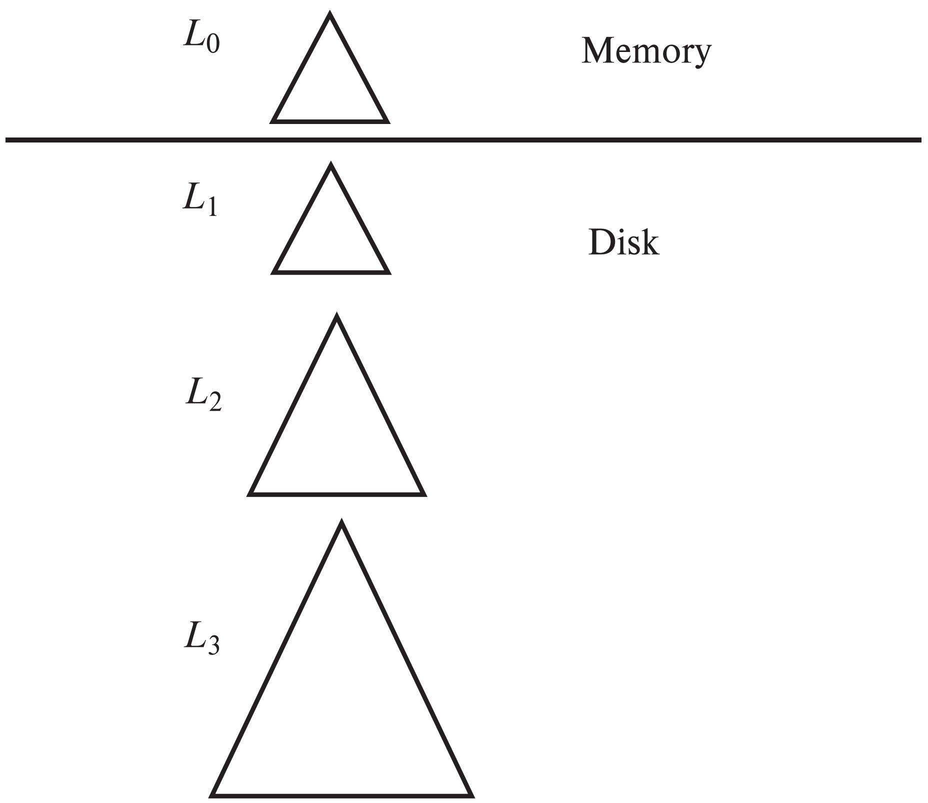
Log Structured Merge (LSM) Tree



Compaction continues creating fewer, larger and larger files

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



LSM Trees (Cont.)

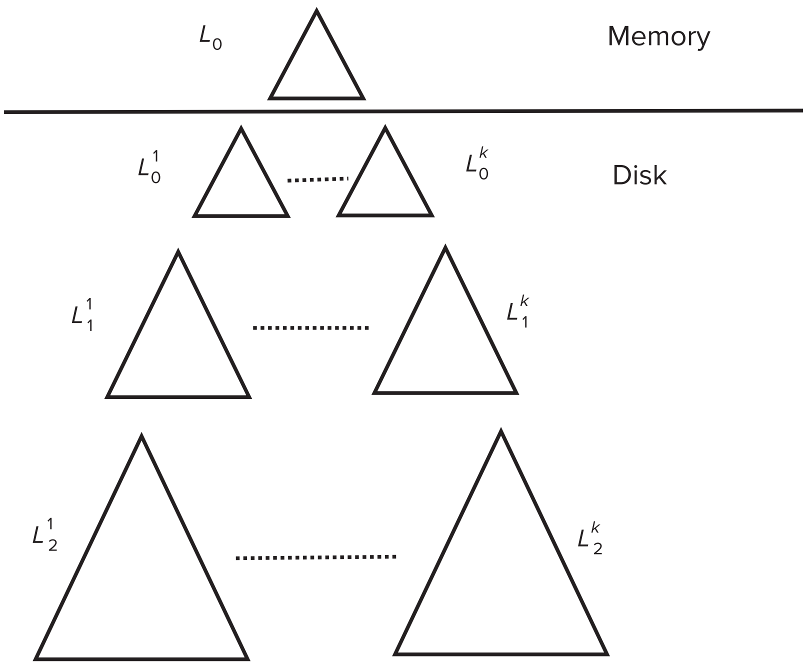
- Deletion handled by adding special "delete" entries
  - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
  - When trees are merged, if we find a delete entry matching an original entry, both are dropped.
- Update handled 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

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

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



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'

Bloom Filter

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

Insert 'RZA'

Bloom Filter

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

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

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

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'
```
$\rightarrow$ *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

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

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

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

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

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

`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

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

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

`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

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

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

`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

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

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

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

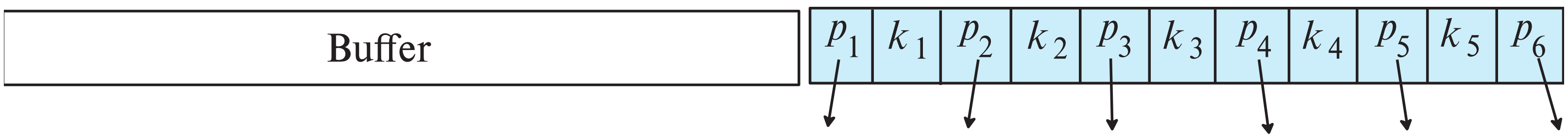
Bloom Filters

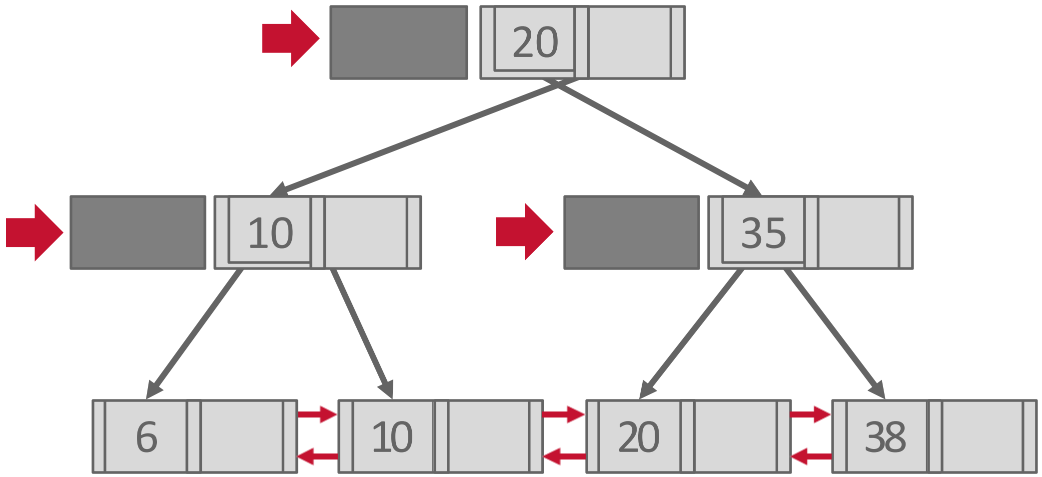
- A **bloom filter** is a probabilistic data structure used to check membership of a value in a set
  - May return true (with low probability) even if an element is not present
  - But never returns false if an element is present
  - Used to filter out irrelevant sets
- Key data structure is a single 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

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





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

Write Optimized Indices

- What trade-offs do write-optimized indices pose as compared to B+-tree indices?
- The stepped merge variant of the LSM tree allows multiple trees per level. What are the tradeoffs in having more trees per level?

Outline

- Write Optimized Indices
- **Hashing**
  - **Extendible Hashing**
- String Index
- Spatial Index

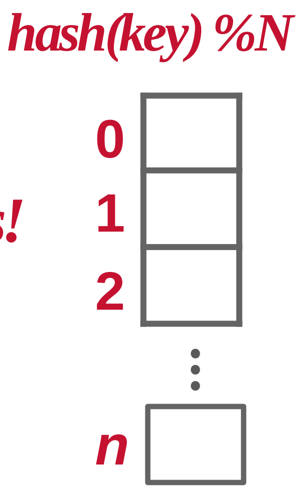
Hashing

- A **hash table** implements an unordered associative array that maps keys to values.
- It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.
- Space Complexity: **O(n)**
- Time Complexity:

  → Average: **O(1)**

  → Worst: **O(n)**

Databases care about constants!



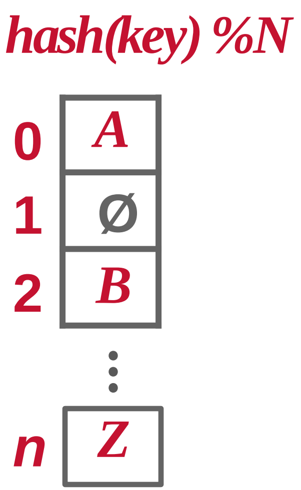
Hash Functions

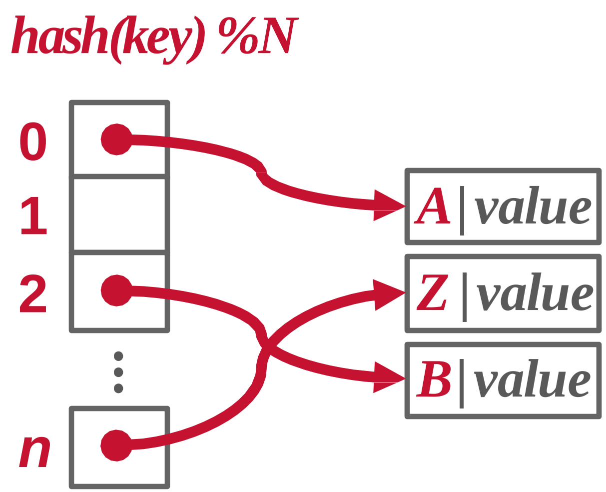
- **For any input key, return an integer representation of that key.**
- **Aim: Fast and low collision rate**.
  - **CRC-64 (1975)**
    - Used in networking for error detection.
  - **MurmurHash (2008)**
    - Designed as a fast, general-purpose hash function.
  - **Google CityHash (2011)**
    - Designed to be faster for short keys (<64 bytes).
  - **Facebook XXHash (2012)**
    - From the creator of zstd compression.
  - **Google FarmHash (2014)**
    - Newer version of CityHash with better collision rates.

Hash Table

- Assumptions
  - Number of elements is known ahead of time and fixed.
  - Each key is unique.

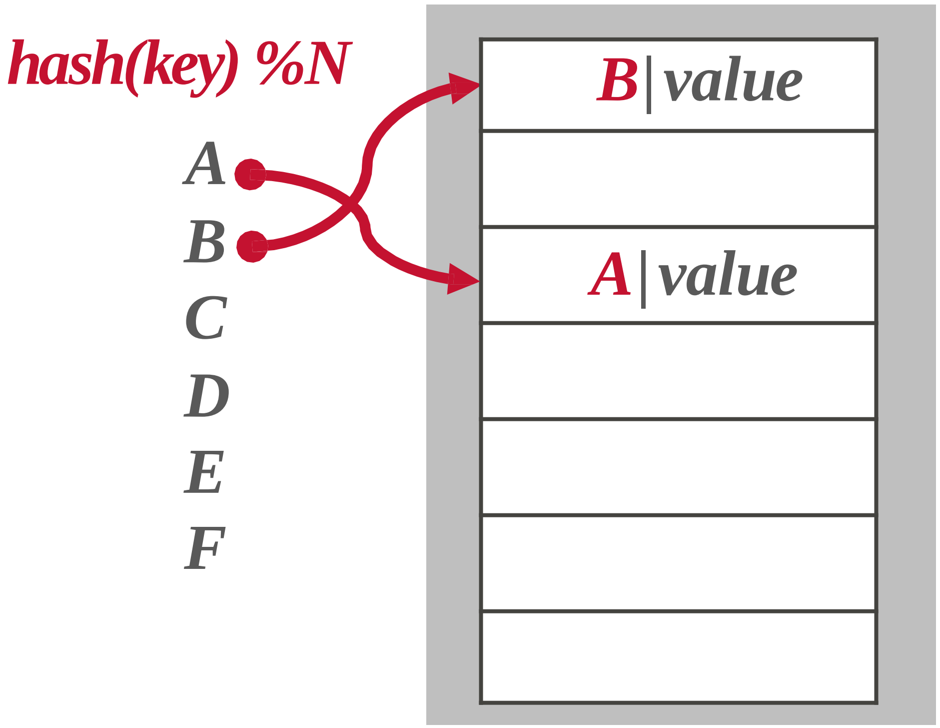
.



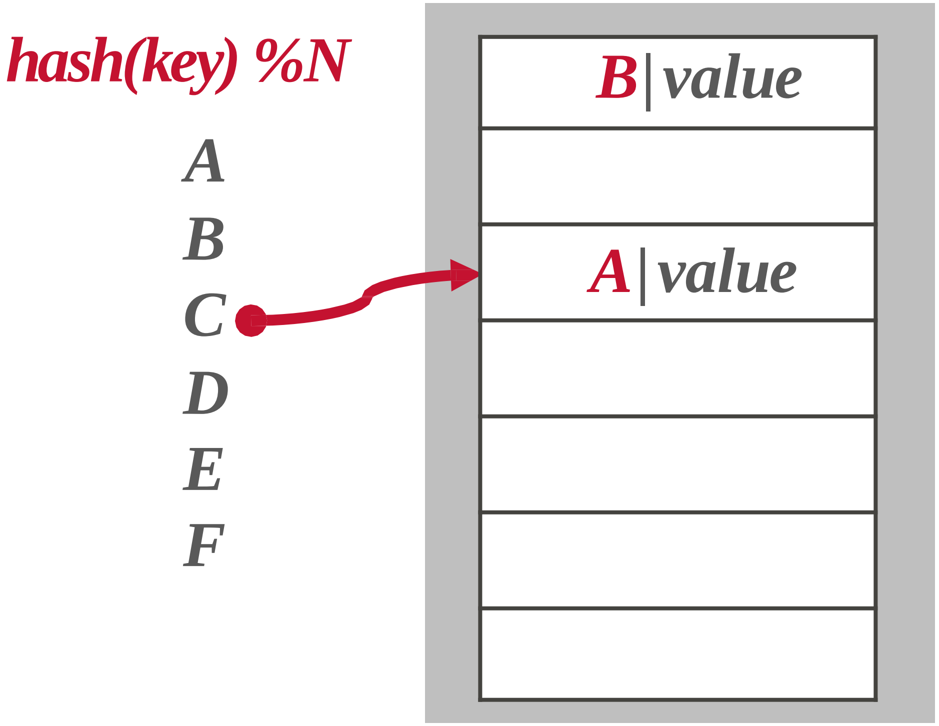


THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

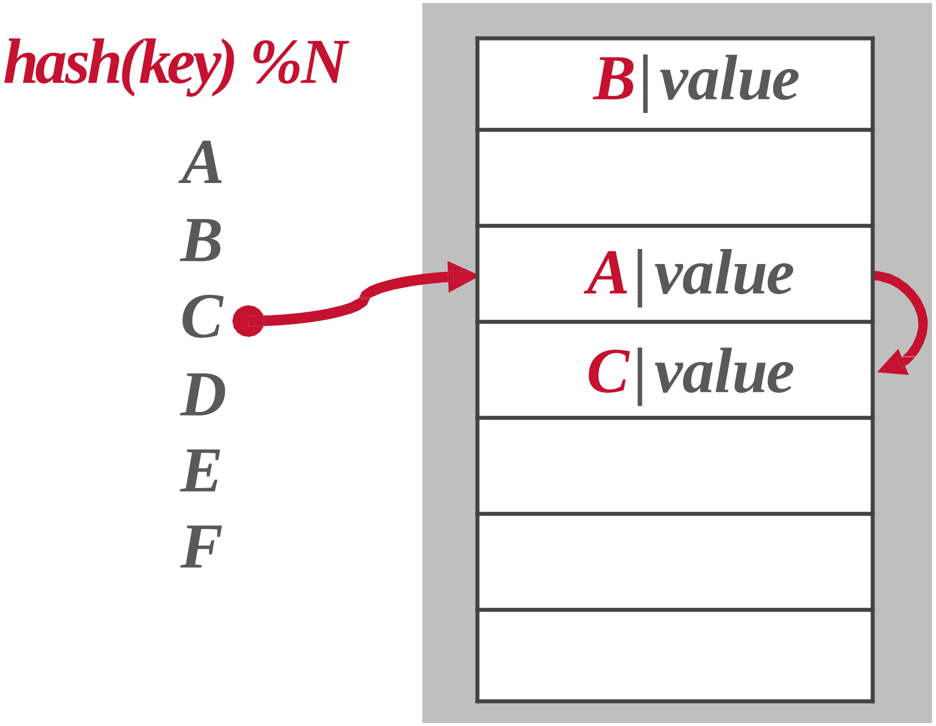
Resolve Collisions: Linear Probe



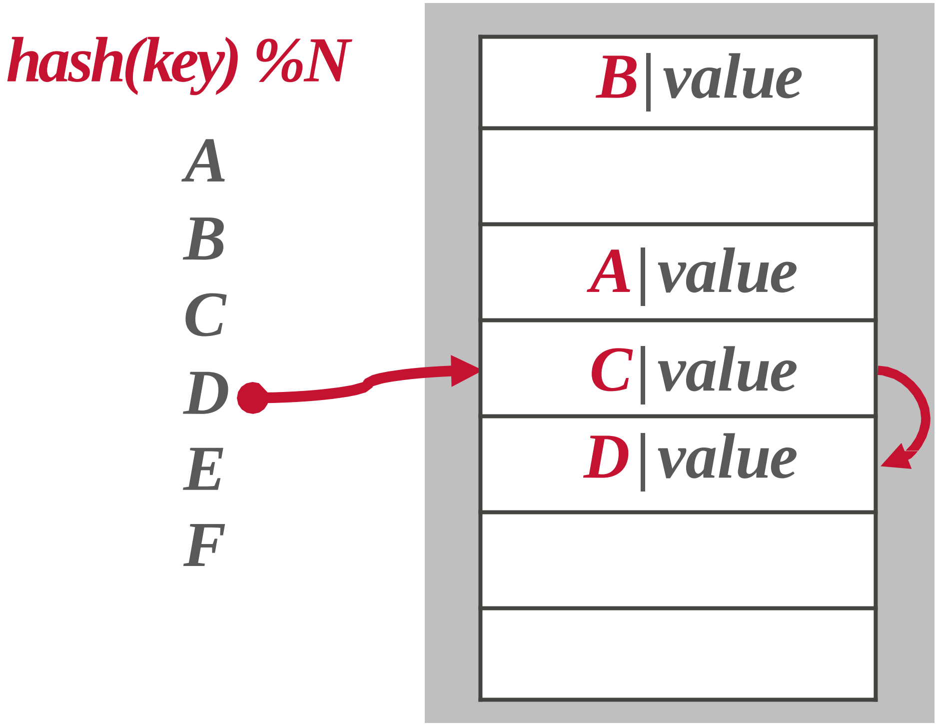
Resolve Collisions: Linear Probe



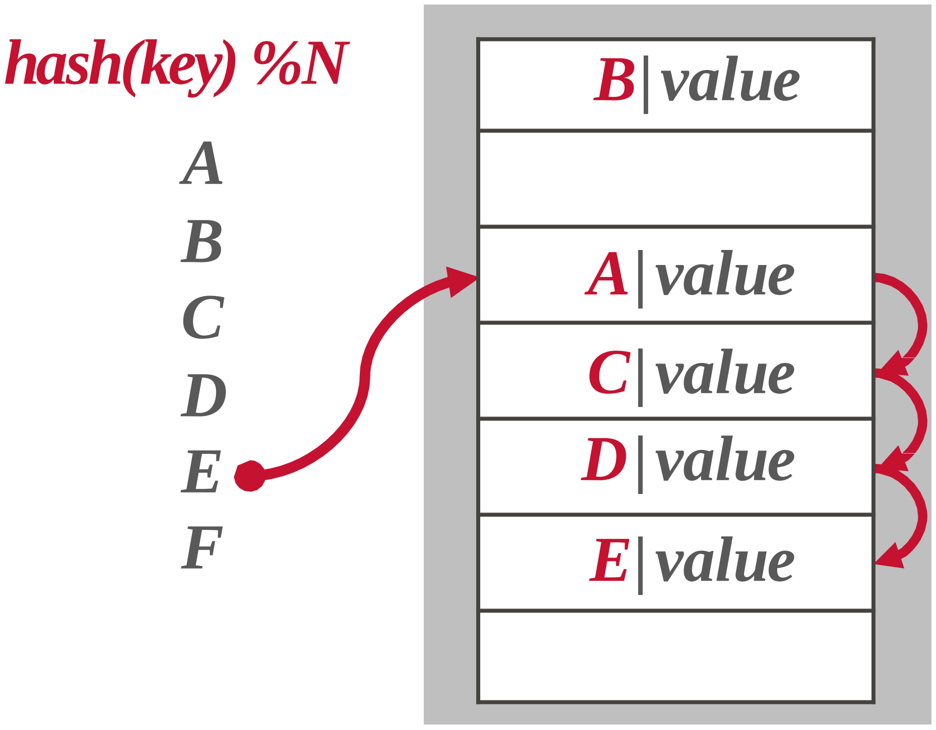
Resolve Collisions: Linear Probe



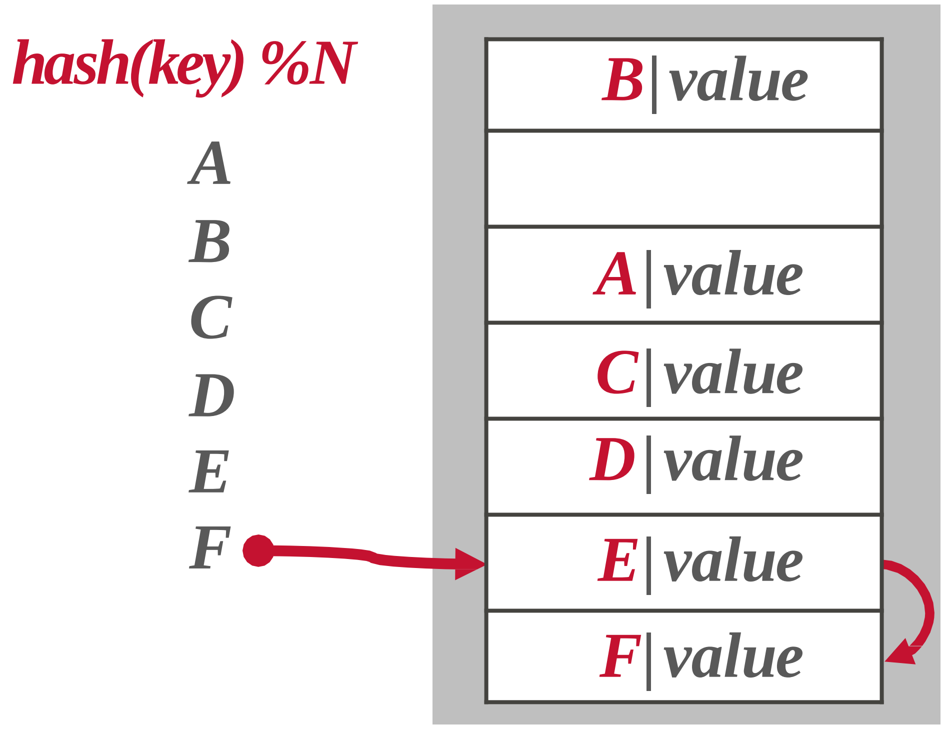
Resolve Collisions: Linear Probe



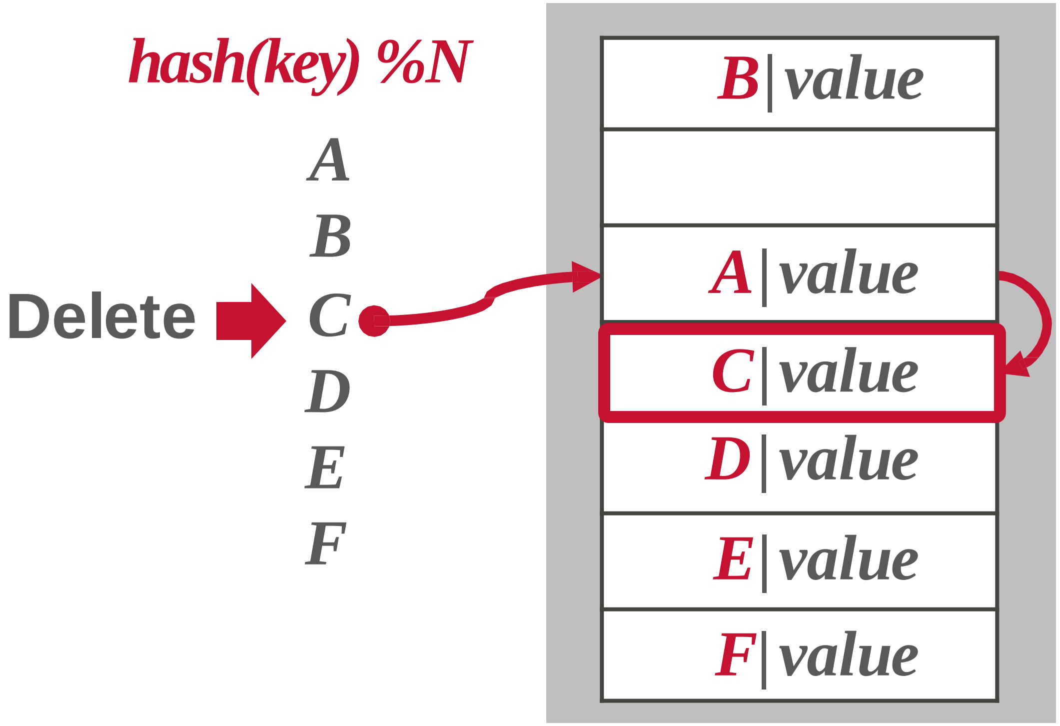
Resolve Collisions: Linear Probe



Resolve Collisions: Linear Probe

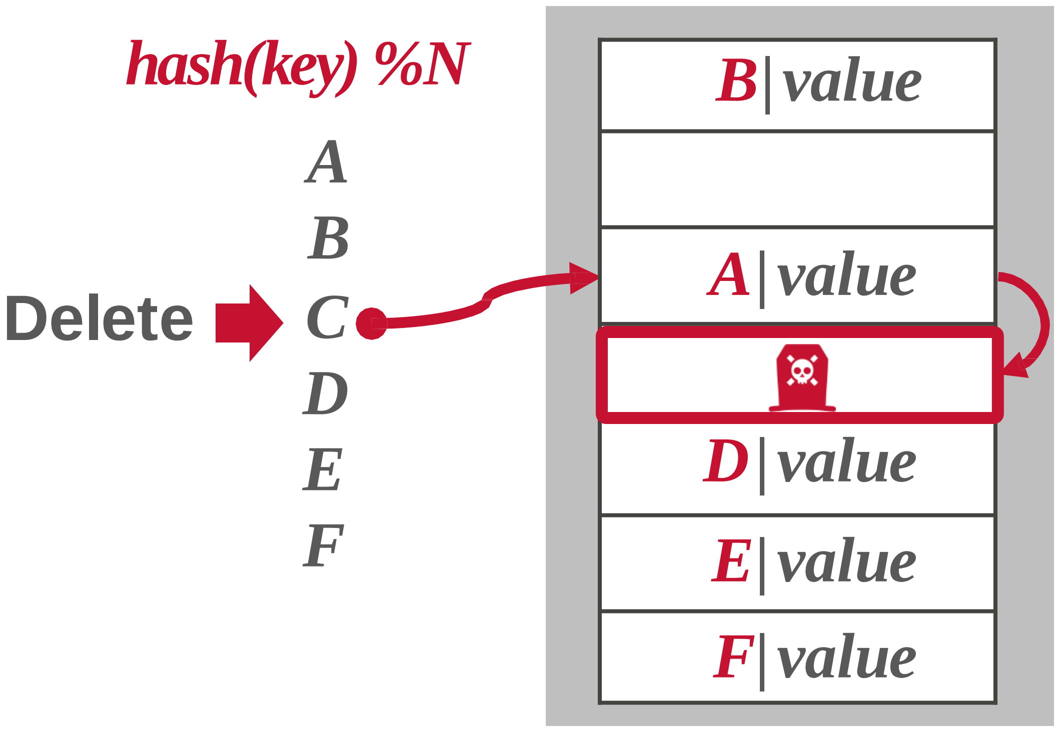


Resolve Collisions: Linear Probe

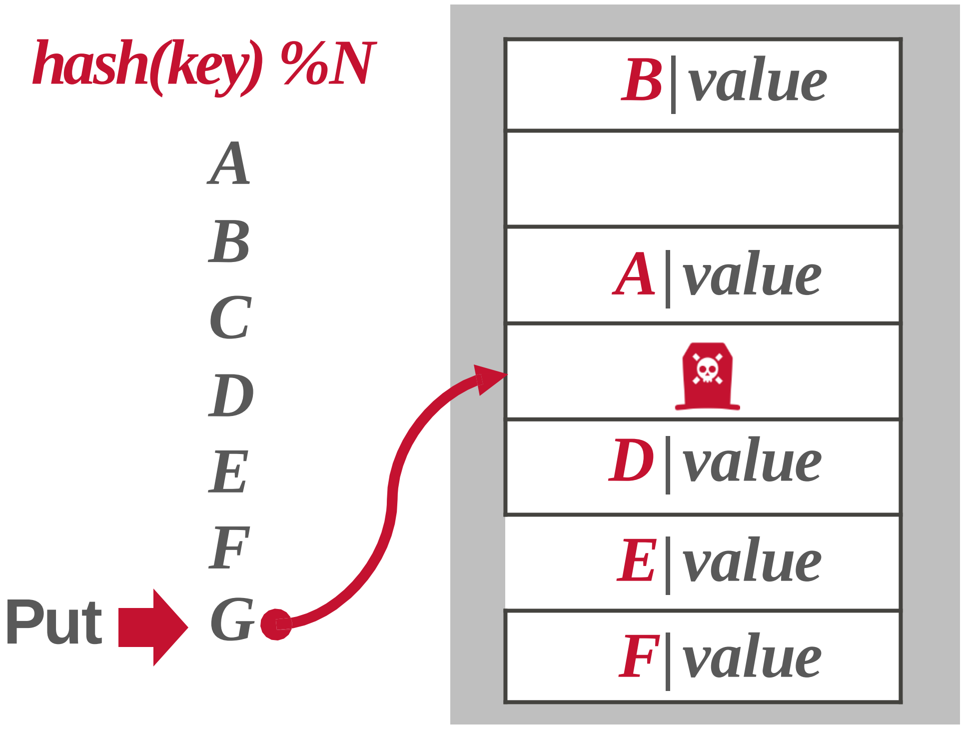


THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

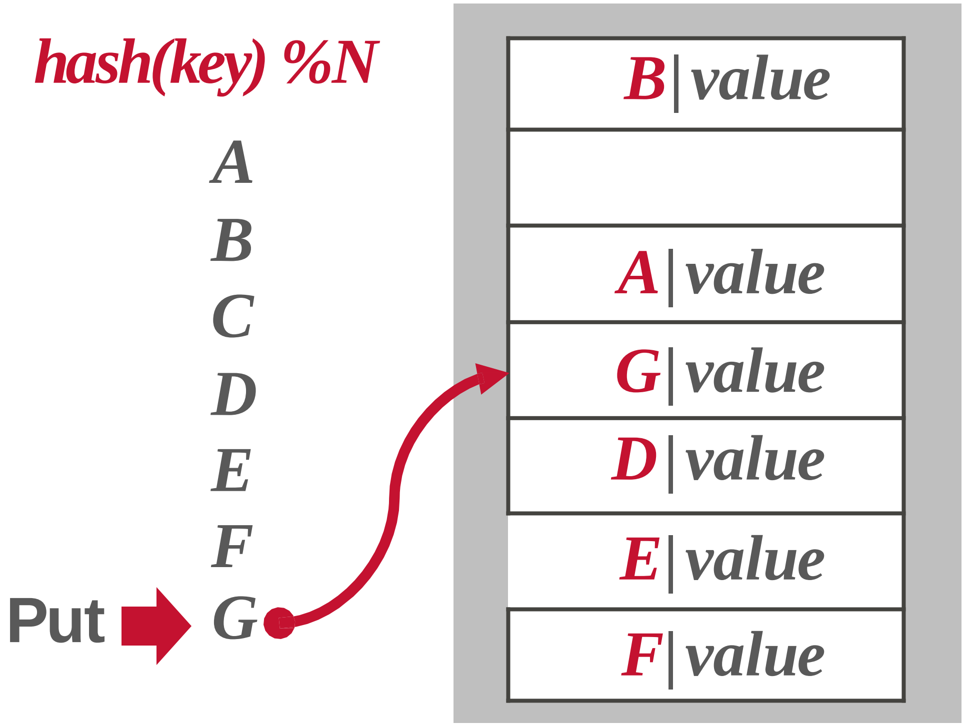
Resolve Collisions: Linear Probe



Resolve Collisions: Linear Probe



Resolve Collisions: Linear Probe

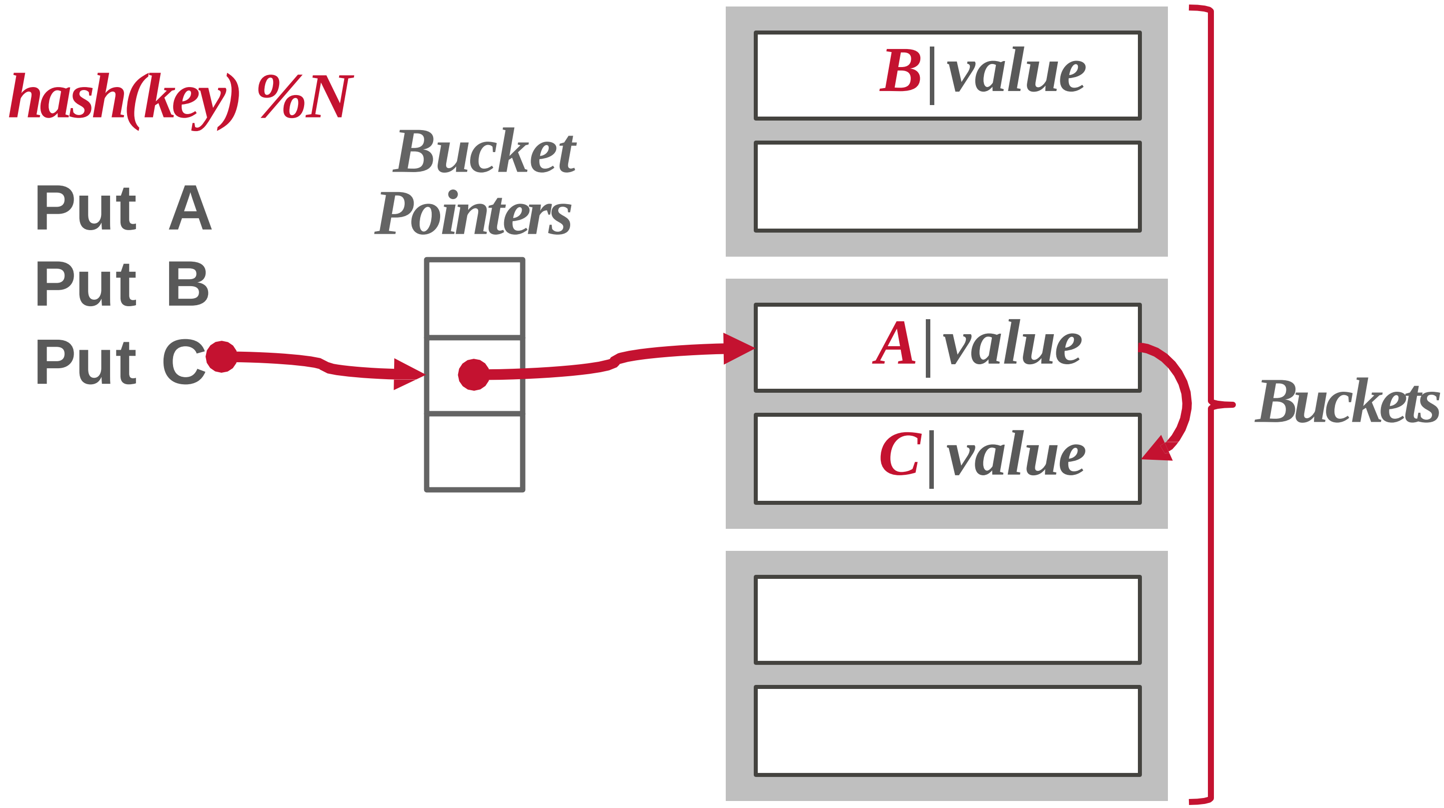


Resolve Collisions: Linear Probe

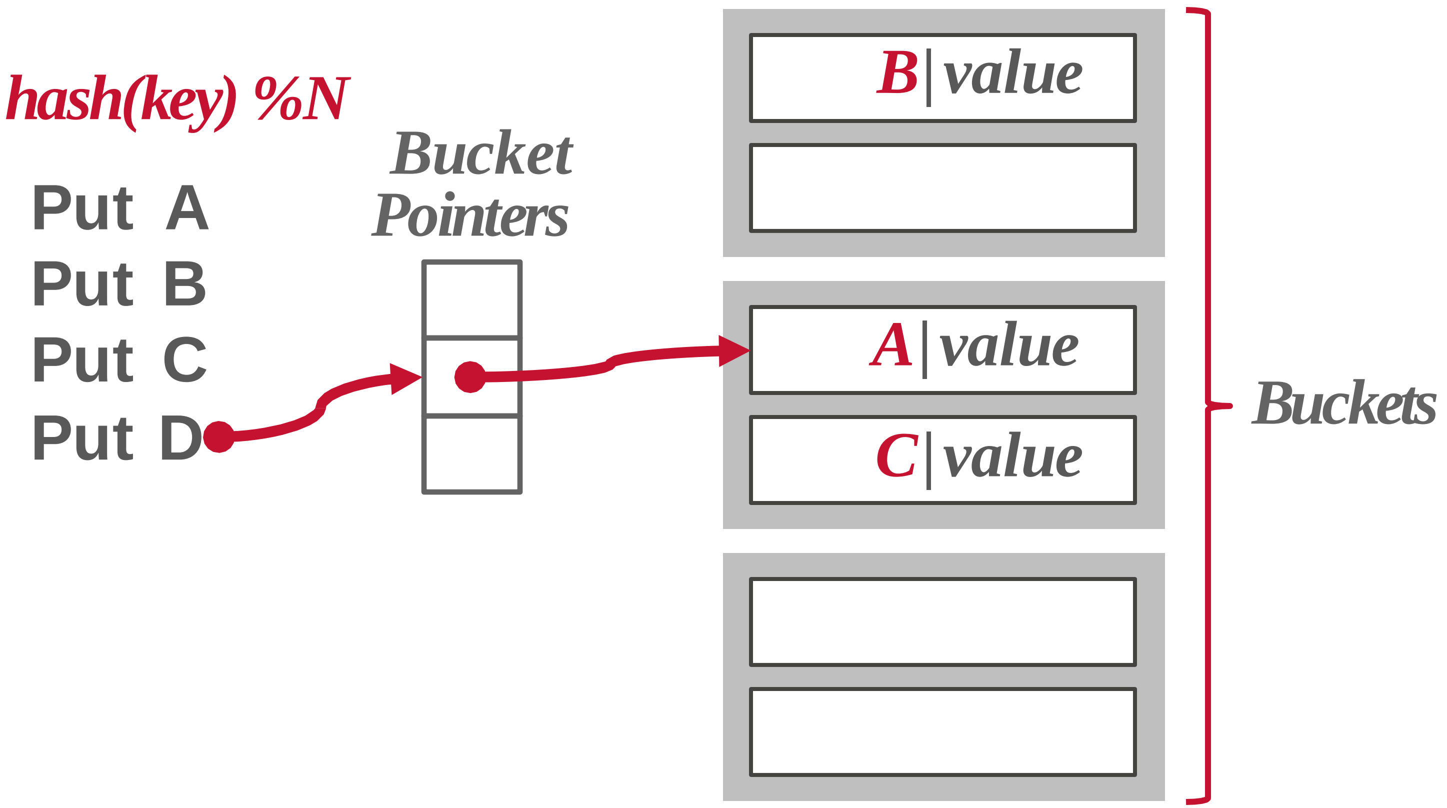
- Single giant table of **fixed-length slots**.
- **Resolve collisions** by linearly **searching for the next free slot** in the table.
  - To determine whether an element is present, hash to a location in the table and scan for it.
  - Store keys in table to know when to stop scanning.
  - Insertions and deletions are generalizations of lookups.

- The table's **load factor** determines when it is becoming too full and should be resized.
  - Allocate a new table twice as large and rehash entries.

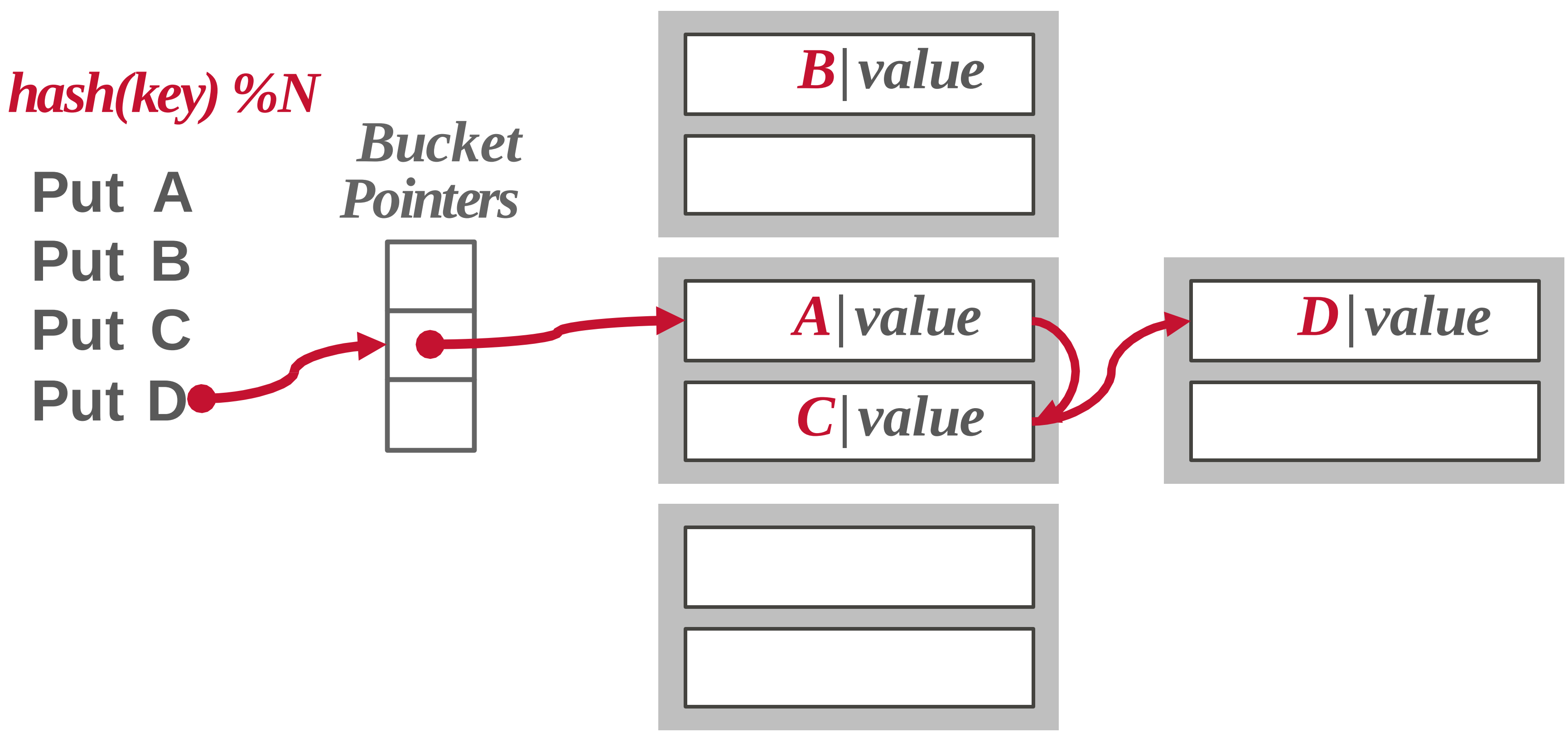
Resolve Collisions: Chaining



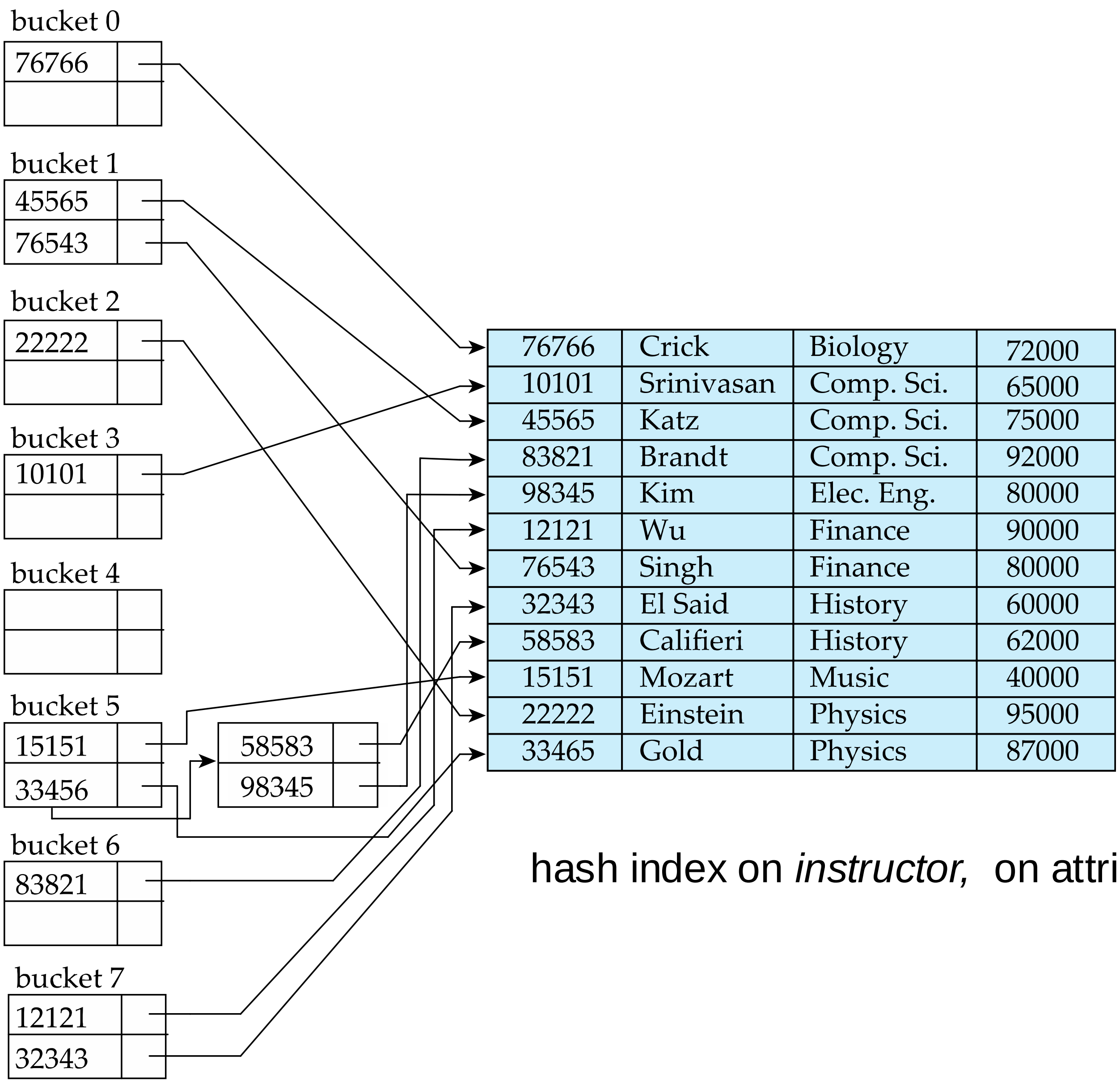
Resolve Collisions: Chaining



Resolve Collisions



Resolve Collisions



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*

Resolve Collisions

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

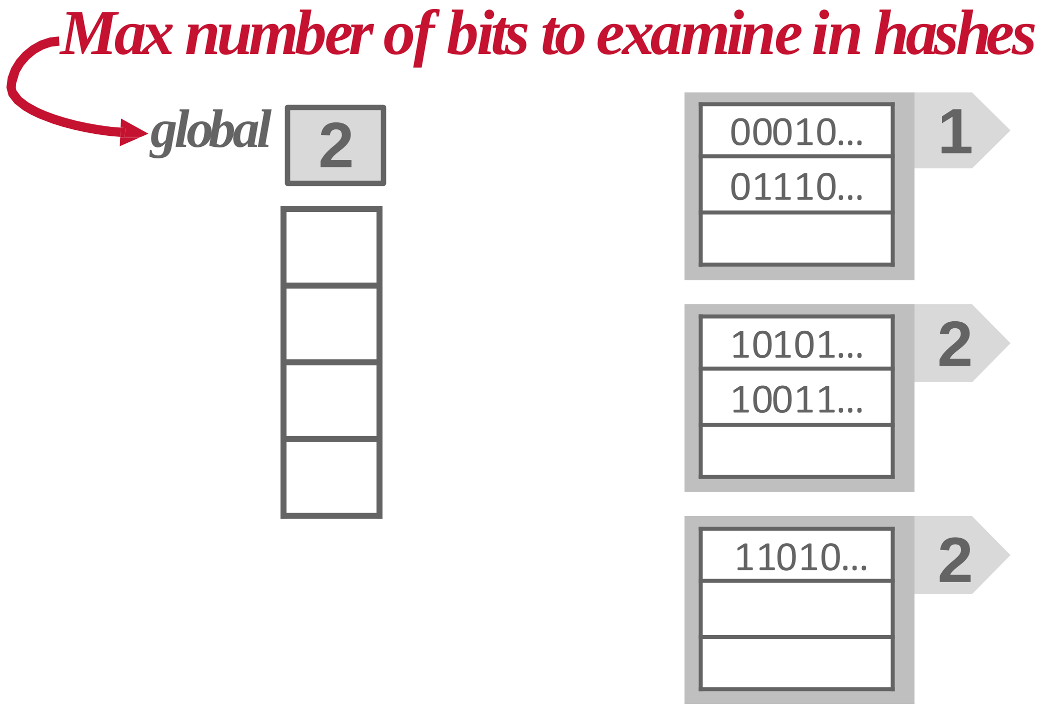
Deficiencies of Static Hashing

- 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.

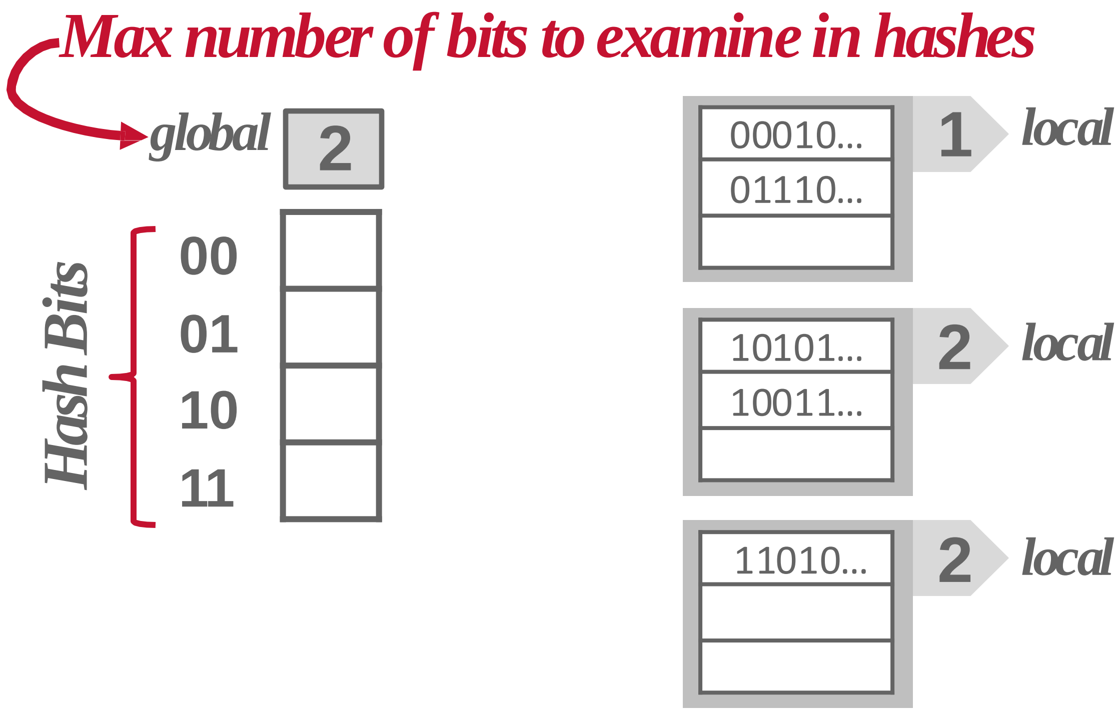
Dynamic Hashing

- 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
- Extendible Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets

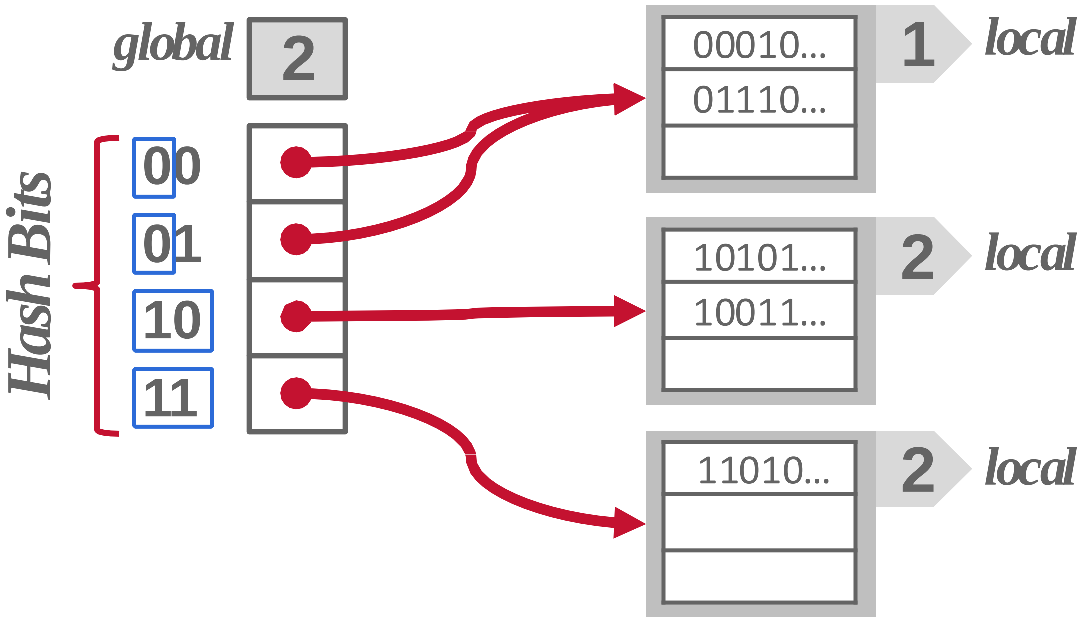
Extendible Hashing



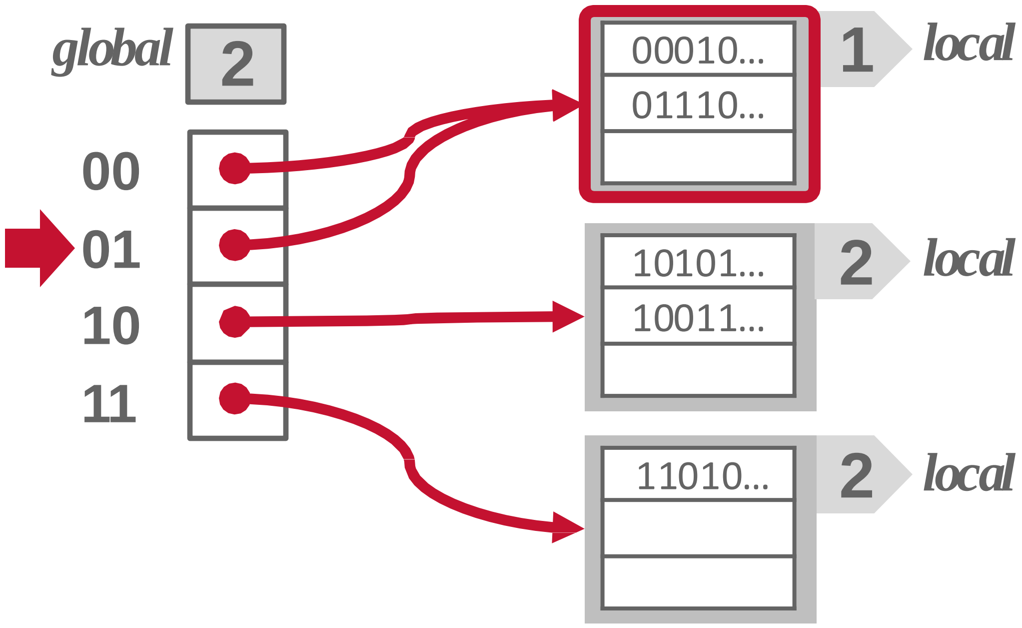
Extendible Hashing



Extendible Hashing



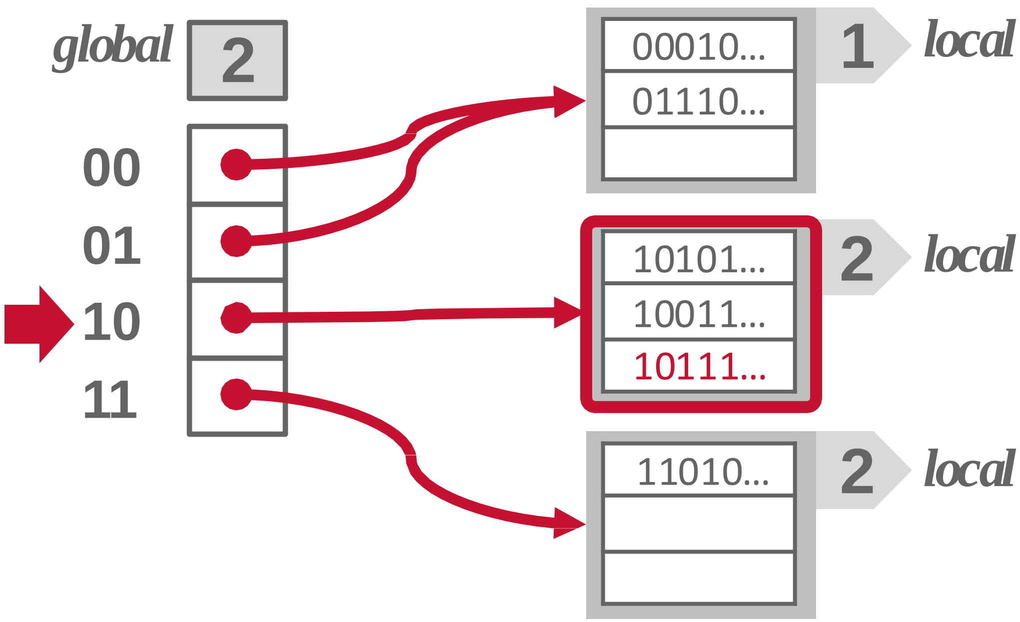
Extendible Hashing



Get A

hash(A) = $\boxed{01}$110...

Extendible Hashing



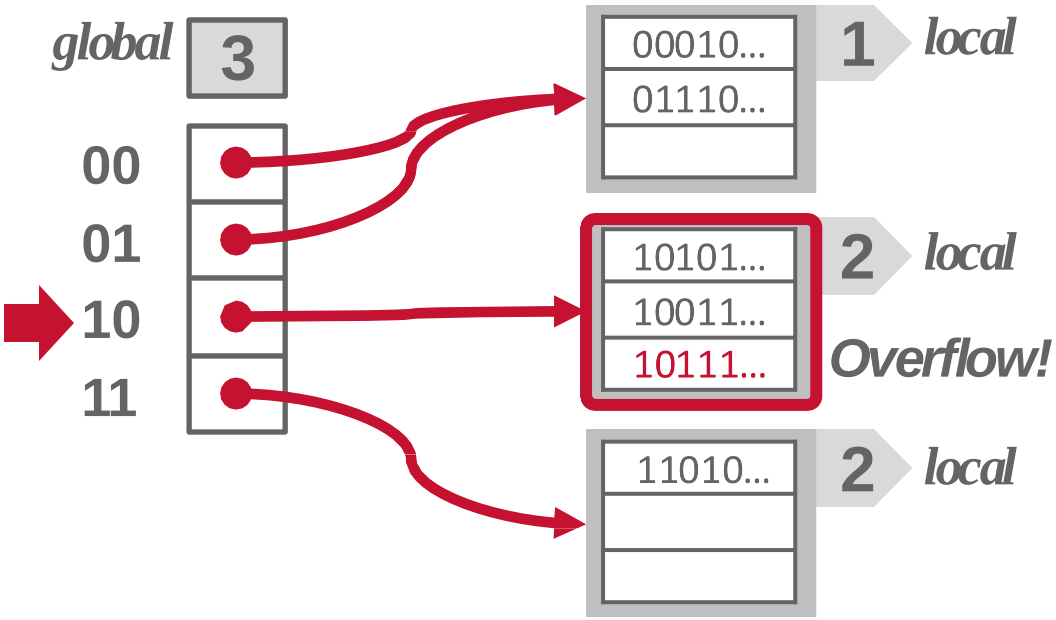
Get A

hash(A) = 01110...

Put B

hash(B) = $\boxed{10}$111...

Extendible Hashing



Get A

hash(A) = 01110...

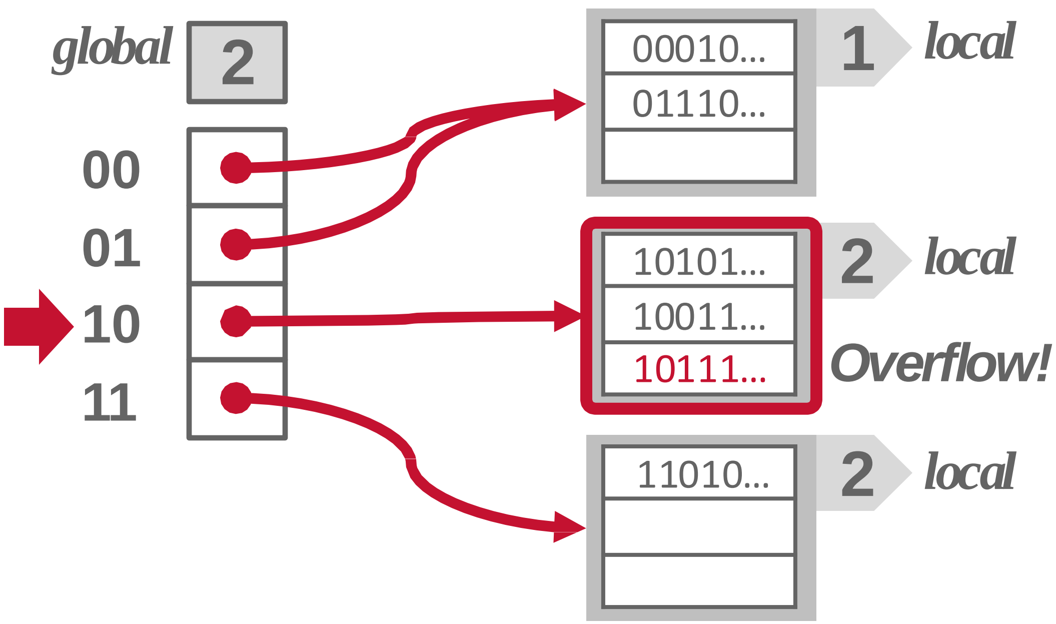
Put B

hash(B) = 10111...

Put C

hash(C) = $\boxed{101}$00...

Extendible Hashing



Get A

hash(A) = 01110...

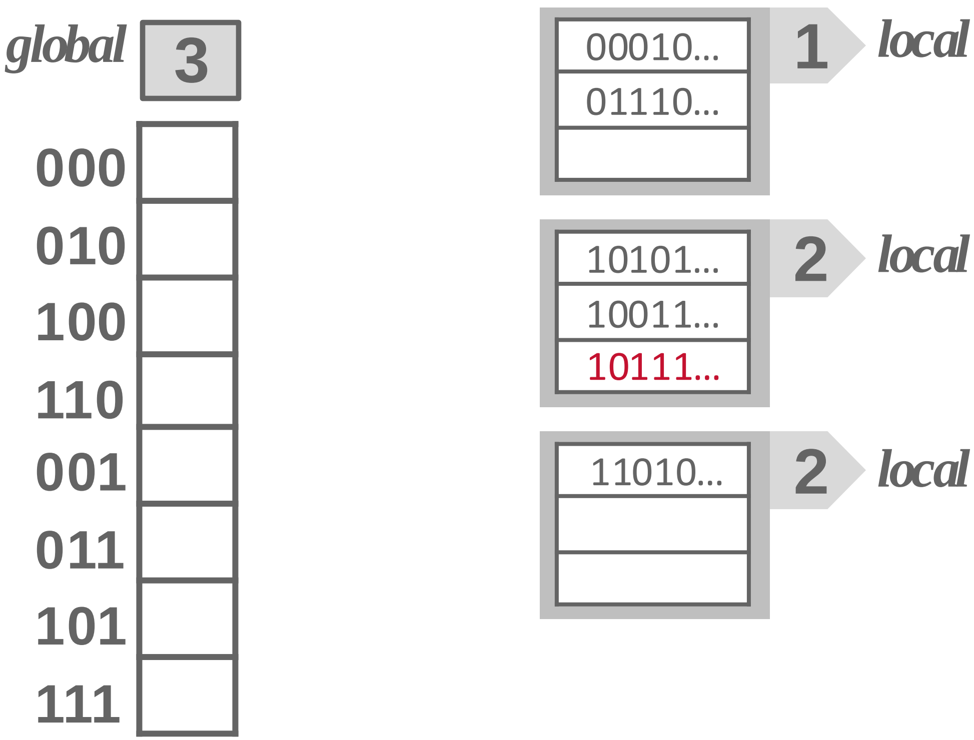
Put B

hash(B) = 10111...

Put C

hash(C) = $\boxed{101}00\ldots$

Extendible Hashing



Get A

hash(A) = 01110…

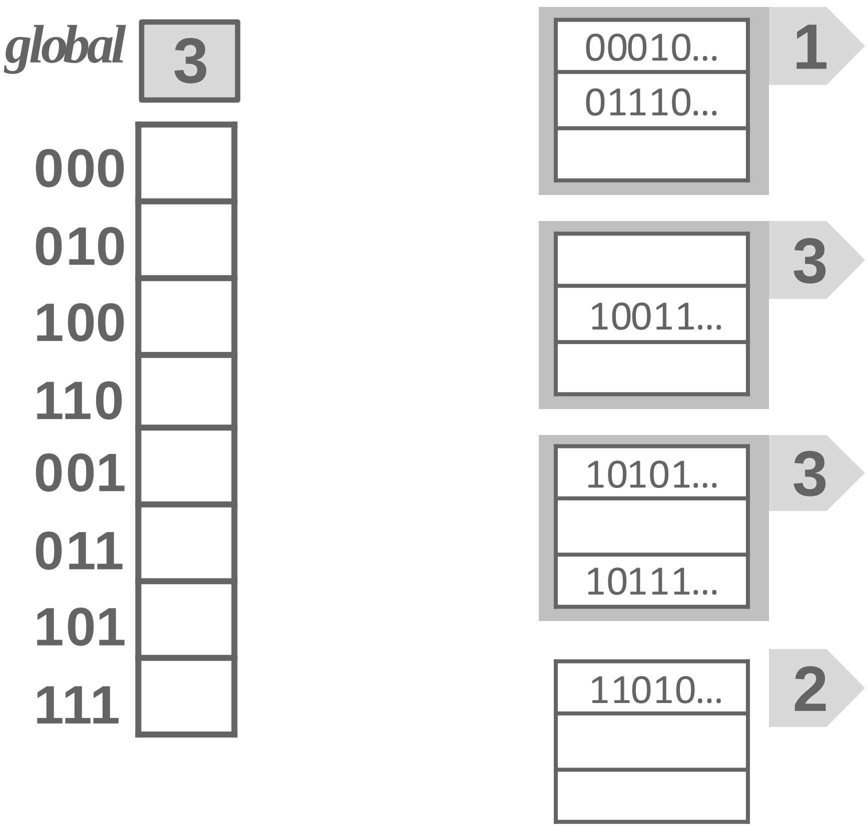
Put B

hash(B) = 10111…

Put C

hash(C) = $\boxed{101}$00…

Extendible Hashing



Get A

hash(A) = 01110...

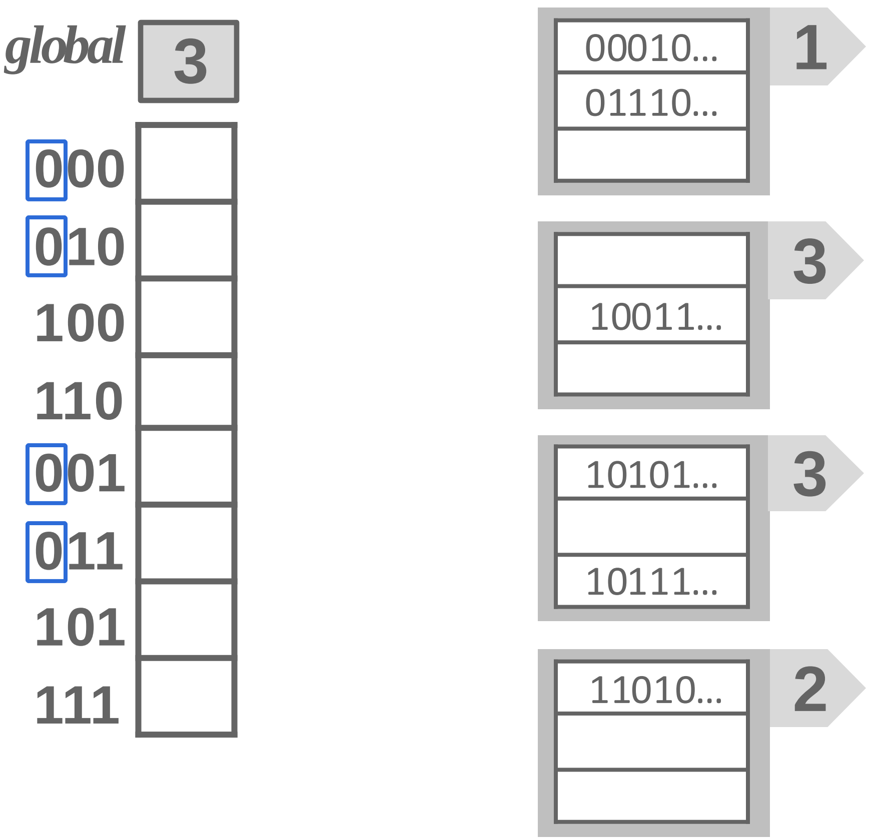
Put B

hash(B) = 10111...

Put C

hash(C) = 10100...

Extendible Hashing



Get A

hash(A) = 01110...

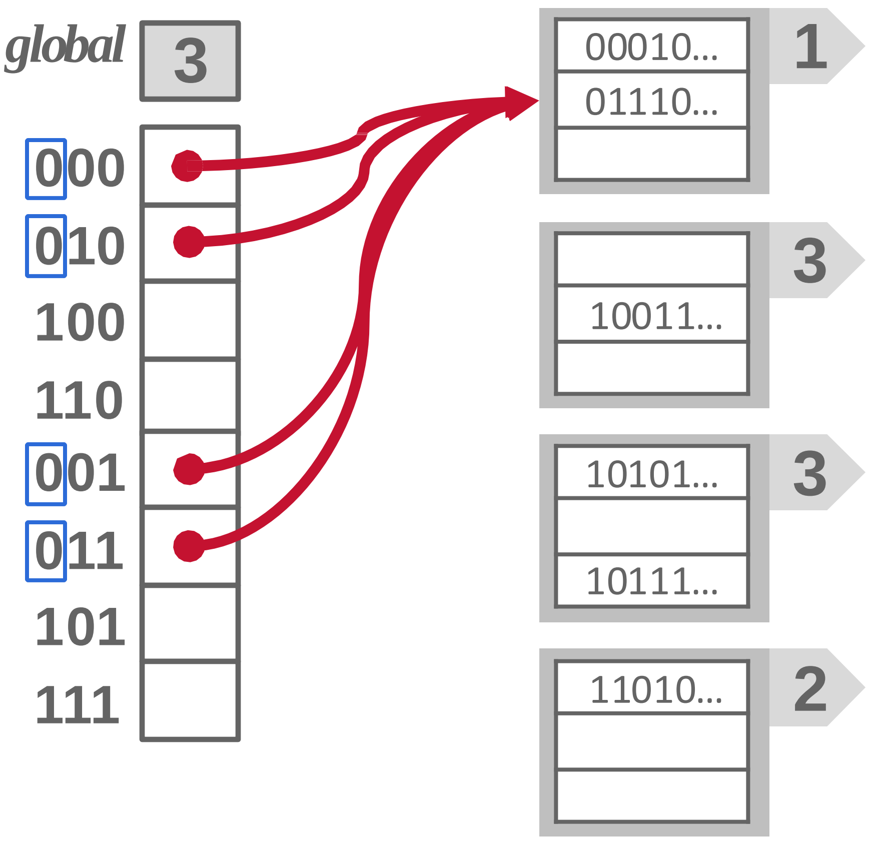
Put B

hash(B) = 10111...

Put C

hash(C) = 10100...

Extendible Hashing



Get A

hash(A) = 01110…

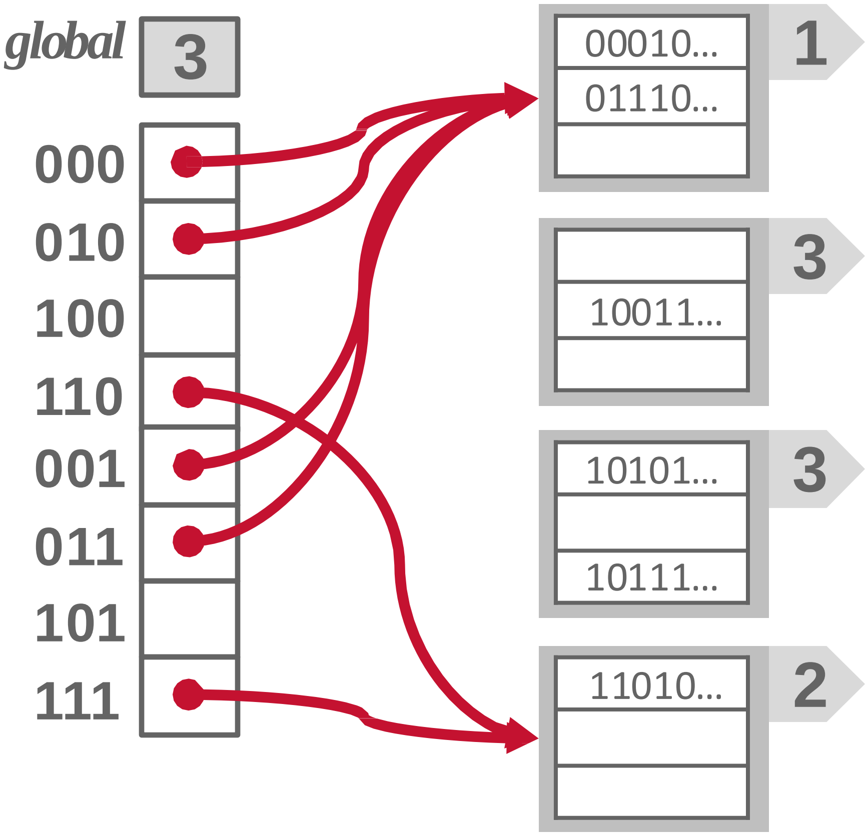
Put B

hash(B) = 10111…

Put C

hash(C) = 10100…

Extendible Hashing

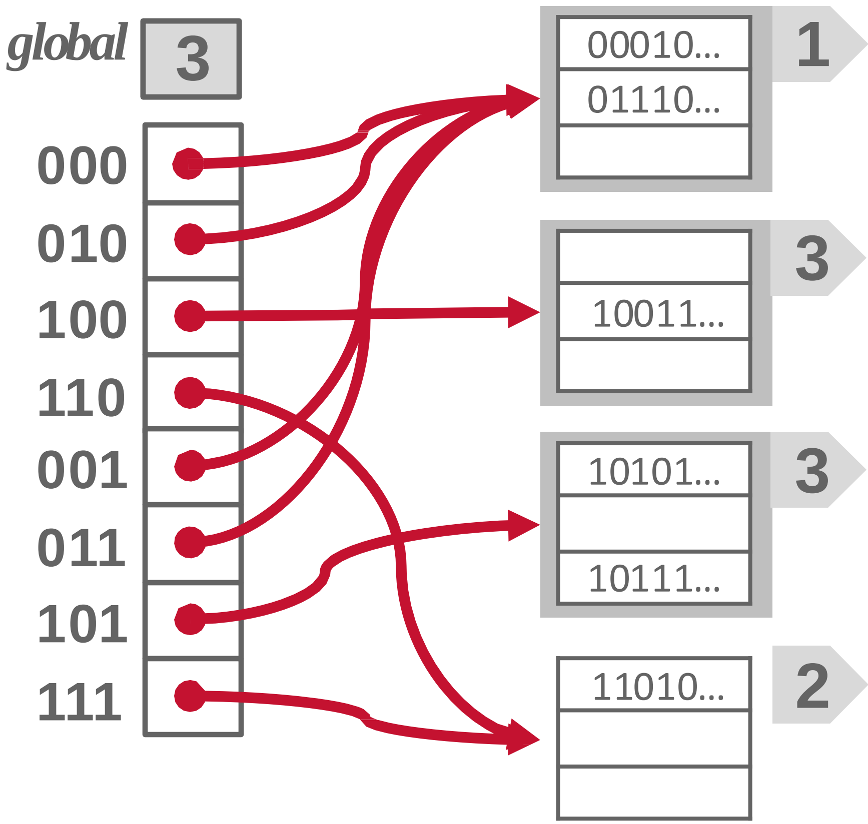


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

Extendible Hashing



Get A

hash(A) = 01110...

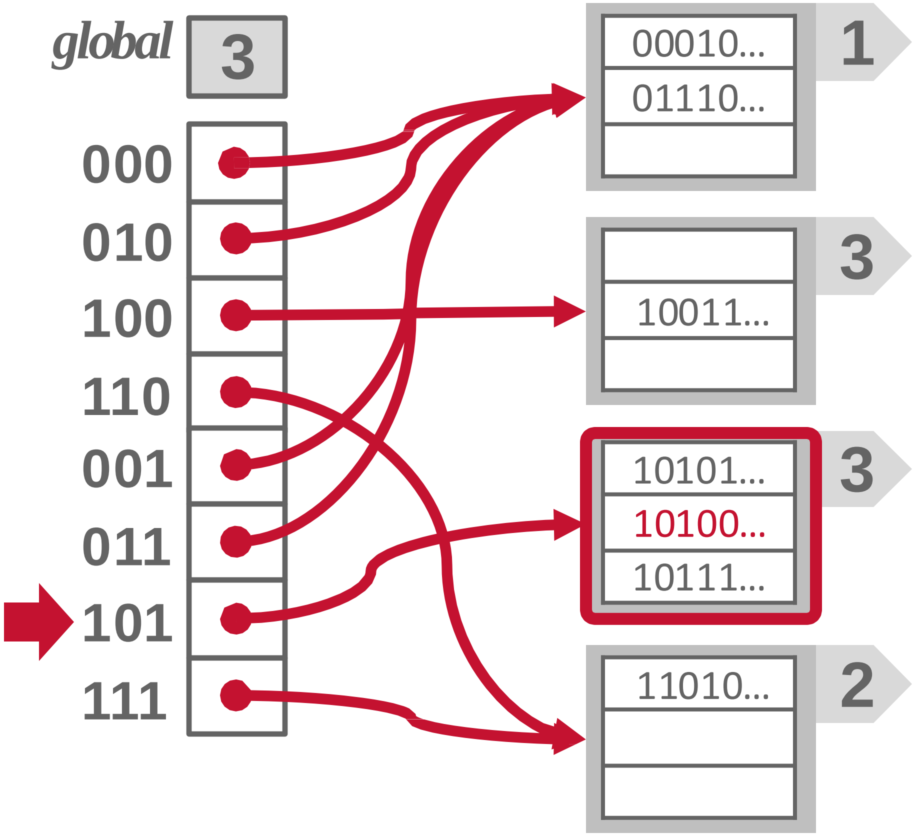
Put B

hash(B) = 10111...

Put C

hash(C) = 10100...

Extendible Hashing



Get A

hash(A) = 01110…

Put B

hash(B) = 10111…

Put C

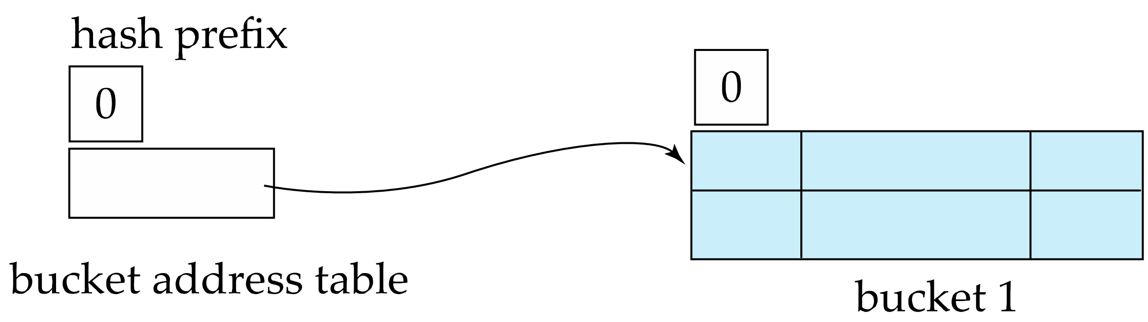
hash(C) = $\boxed{101}$00…

Use of Extendible Hashing: 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

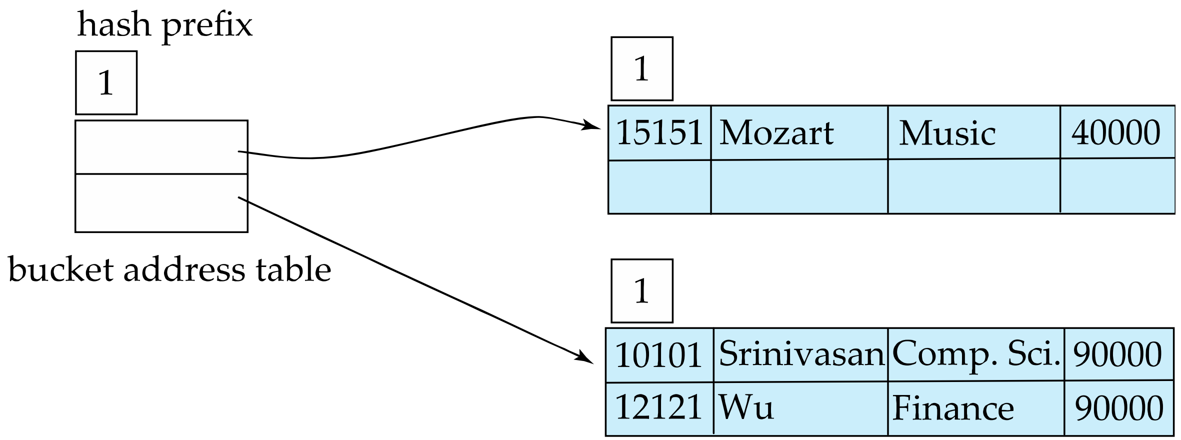
Example (Cont.)

- Initial hash structure; bucket size = 2
- Hash prefix = global depth



Example (Cont.)

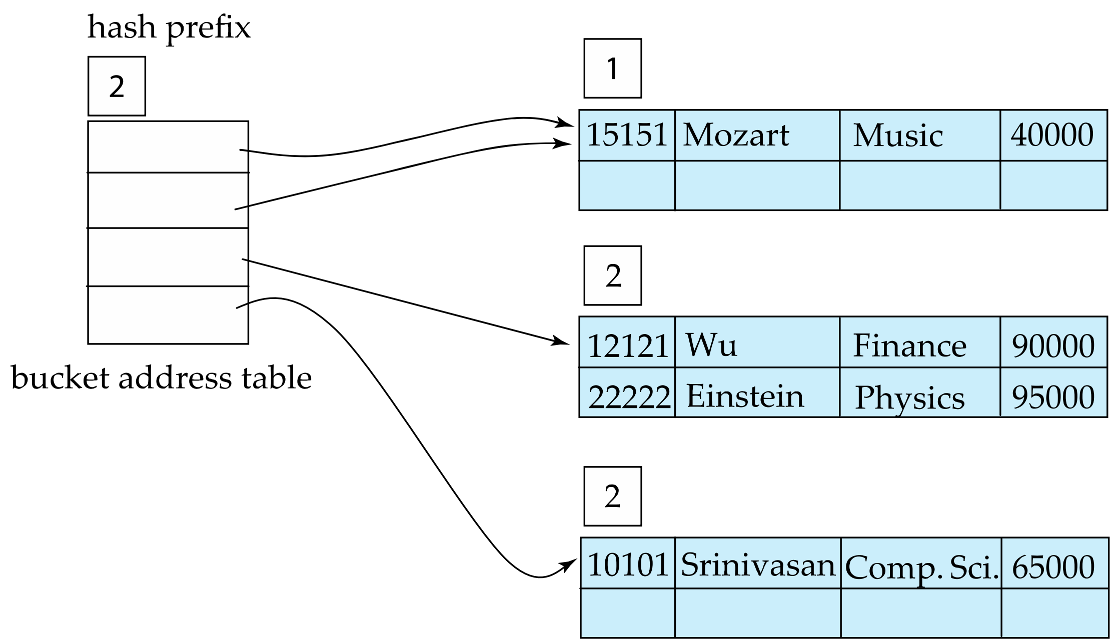
- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records



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.)

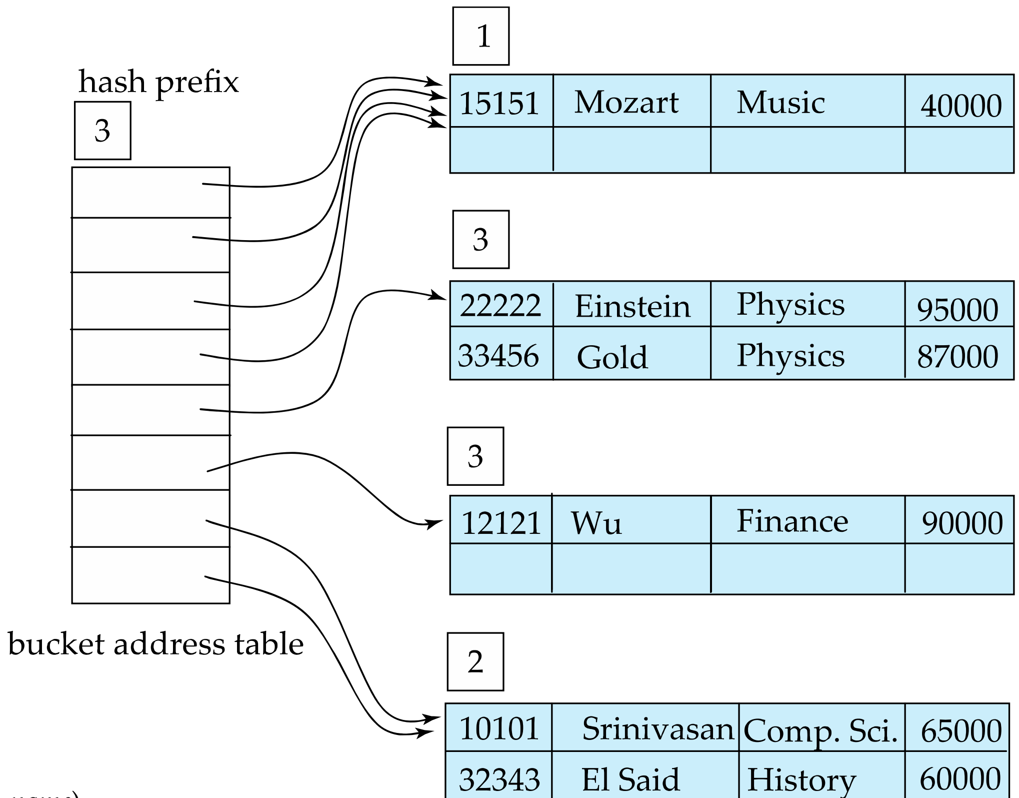
- Hash structure after insertion of Einstein record



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.)

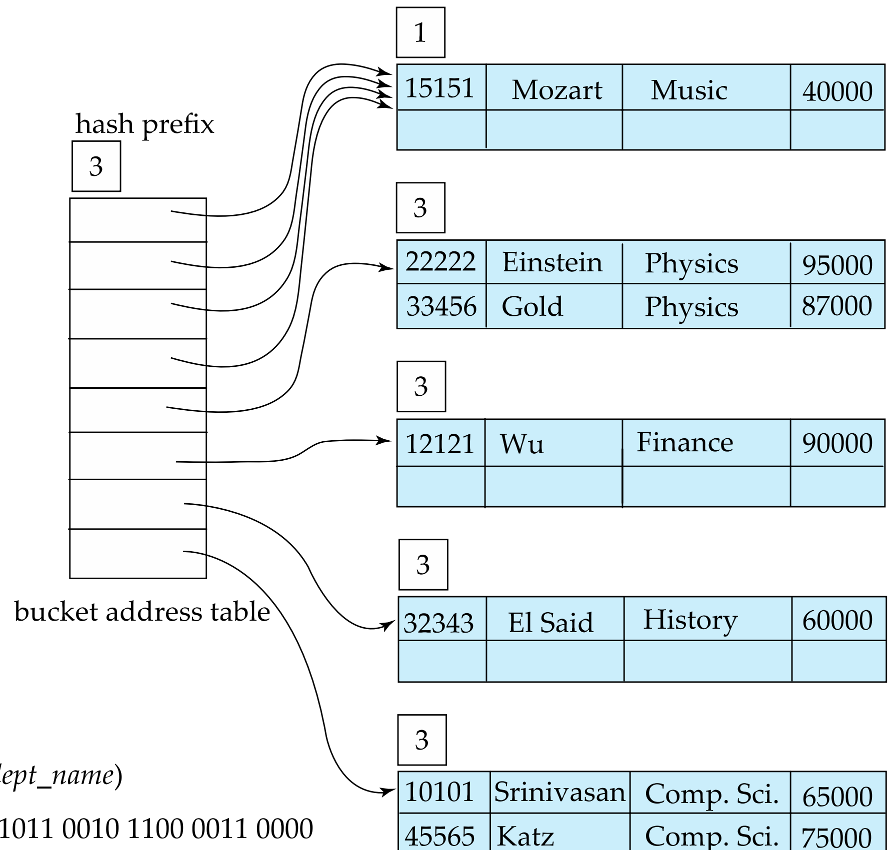
- Hash structure after insertion of Gold and El Said records



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

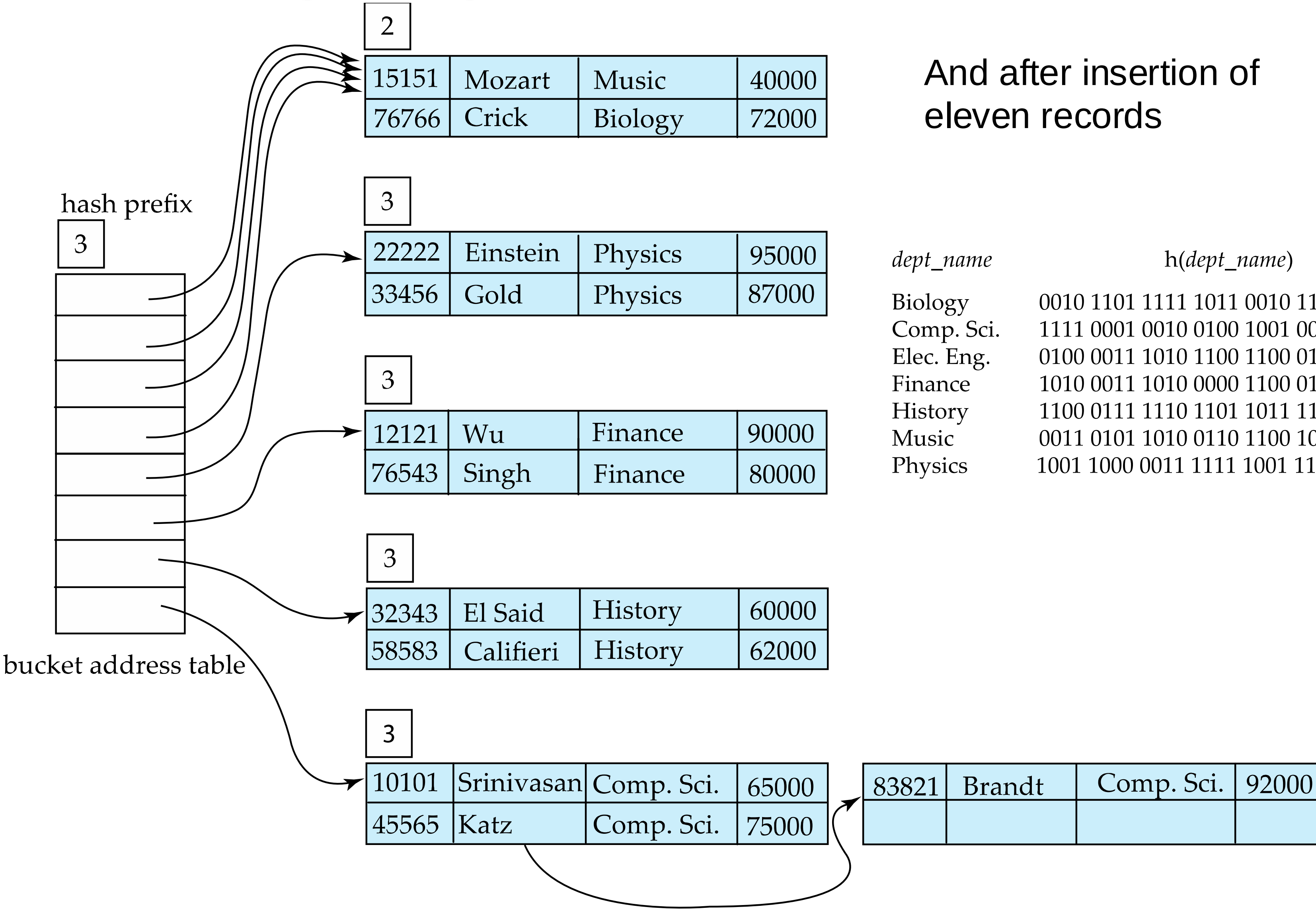
Example (Cont.)

- Hash structure after insertion of Katz record



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

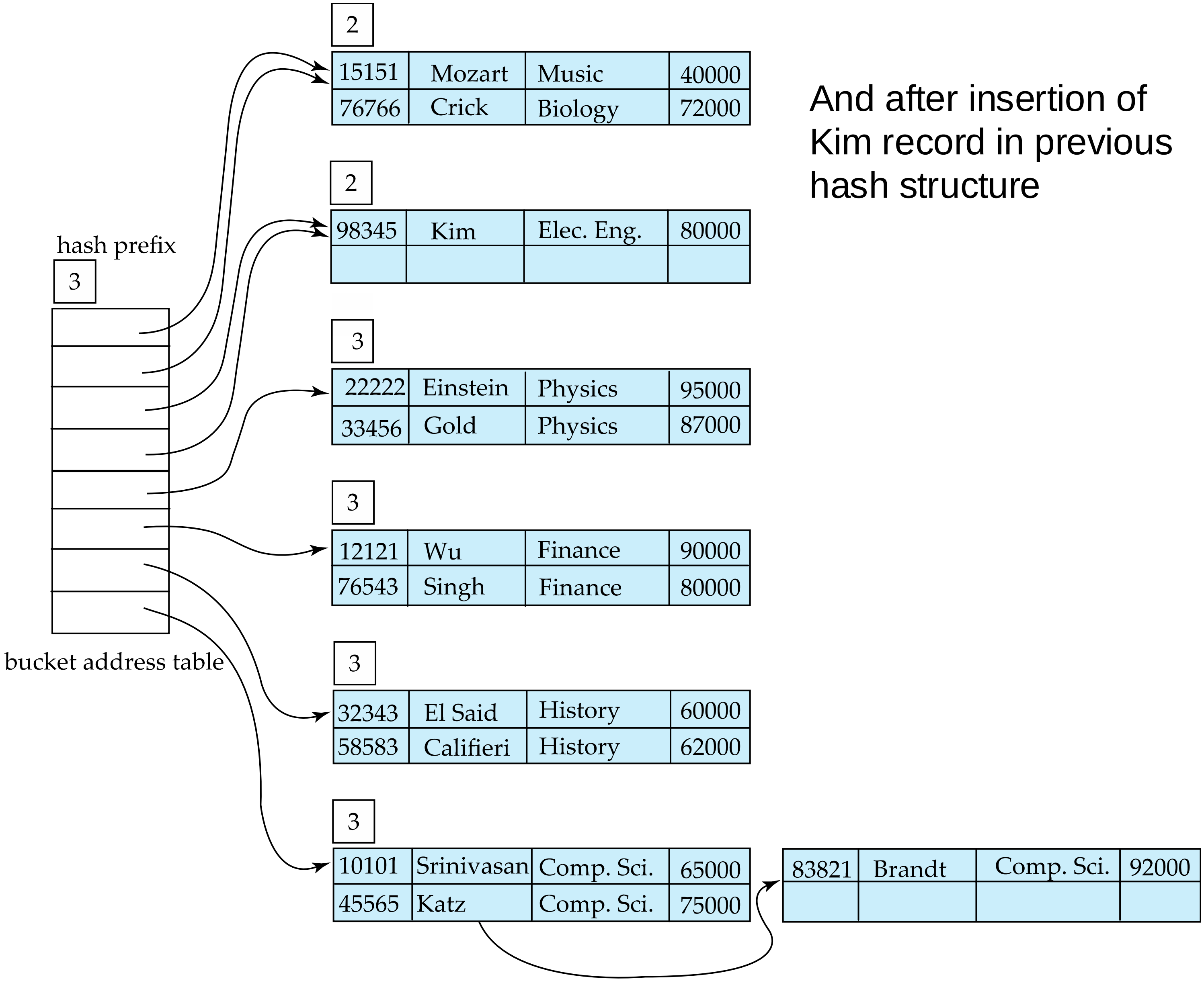
Example (Cont.)



And after insertion of eleven records

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

Example (Cont.)



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

Discussions

- Why is a hash structure not the best choice for a search key on which range queries are likely?

Outline

- Write Optimized Indices
- Hashing
  - Extendible Hashing
- **String Index**
- Spatial Index

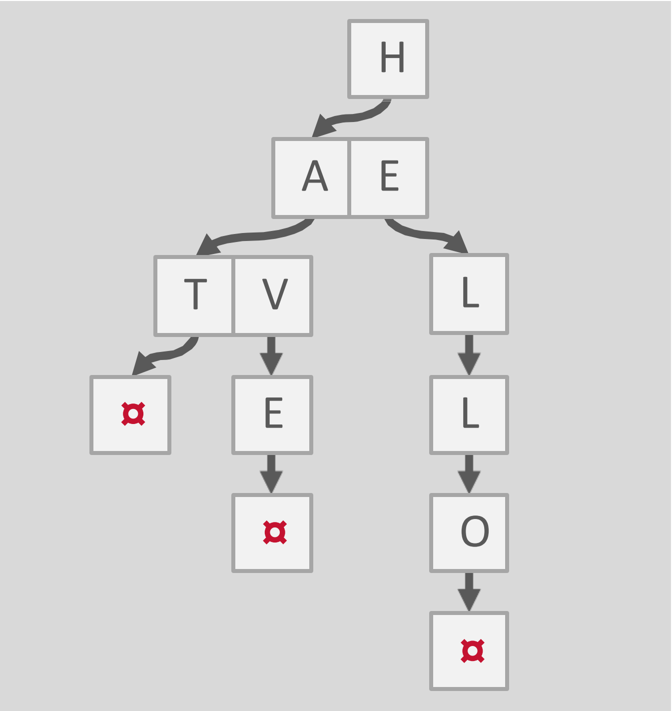
Trie Index

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 not depend on existing keys or insertion order.
→ Does not 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.

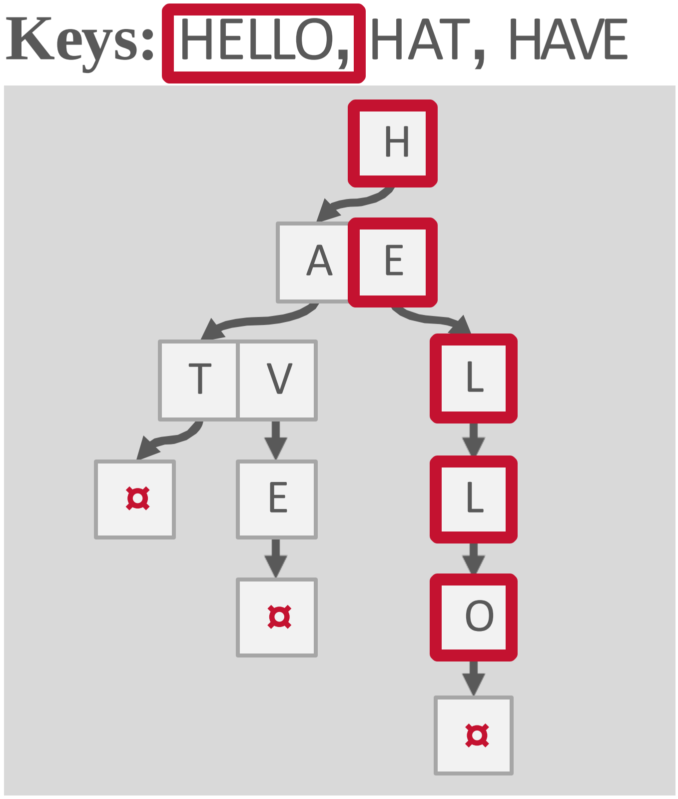
→ Does not depend on existing keys or insertion order.

→ Does not require rebalancing operations.

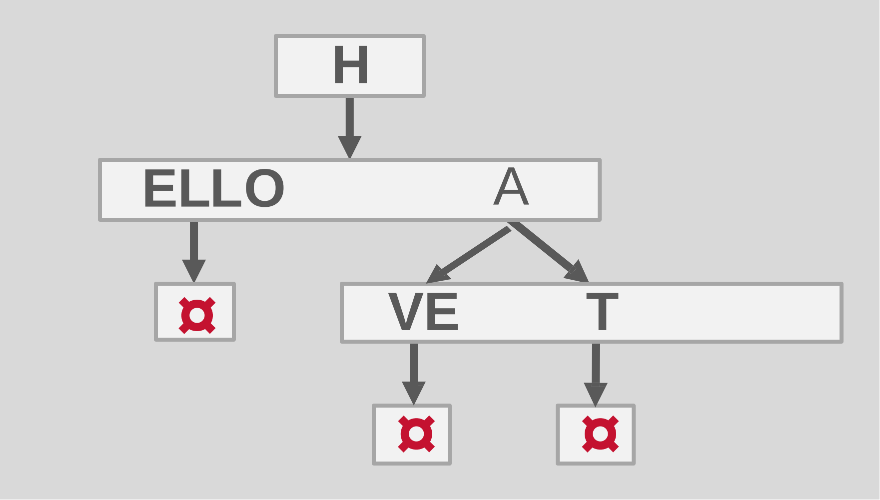
All operations have **O(k)** complexity where **k** is the length of the key.

→ Path to a leaf node represents a key.

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



Radix Tree

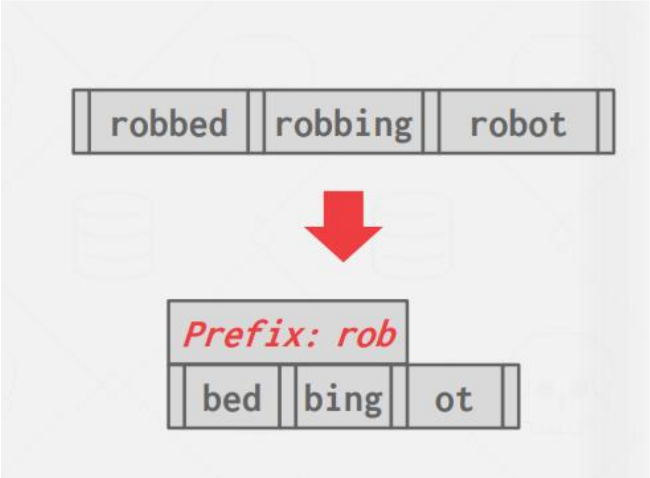


Vertically compressed trie that compacts nodes with a single child.
→ Also known as ***Patricia Tree***.

Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

String Index

- Suppose you have to create a B+-tree index on a large number of names, where the maximum size of a name may be quite large (say 40 characters) and the average name is itself large (say 10 characters). Explain how prefix compression can be used to maximize the average fanout of nonleaf nodes.



Prefix compression

String Index

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g., “Silas” and “Silberschatz” can be separated by “Silb”
  - Keys in leaf node can be compressed by sharing common prefixes

Outline

- Write Optimized Indices
- Hashing
  - Extendible Hashing
- String Index
- **Spatial Index**

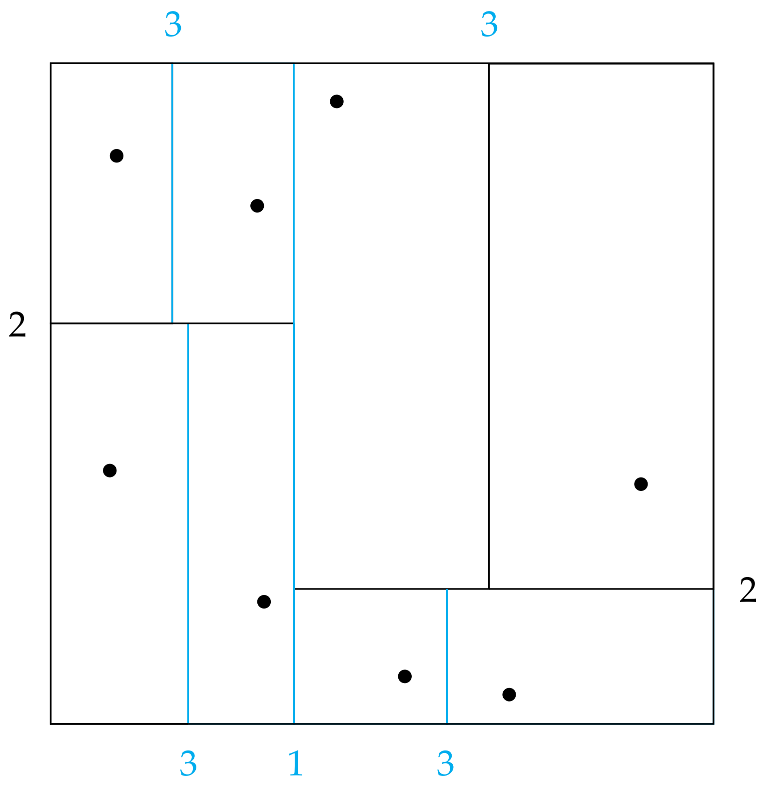
Spatial Data

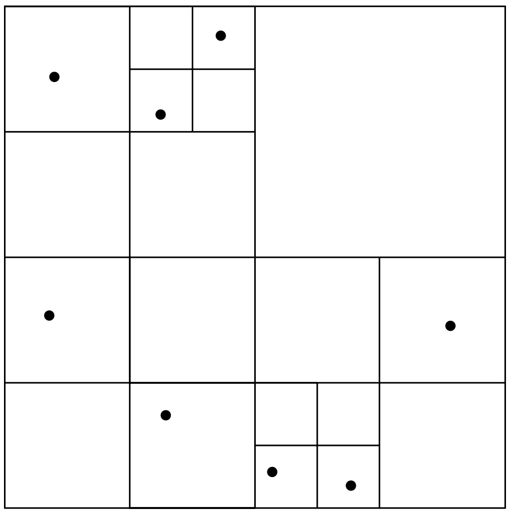
- Databases can store data types such as lines, polygons, in addition to raster images
- **Nearest neighbor queries**
- **Range queries**
- **Spatial join**

Spatial Data

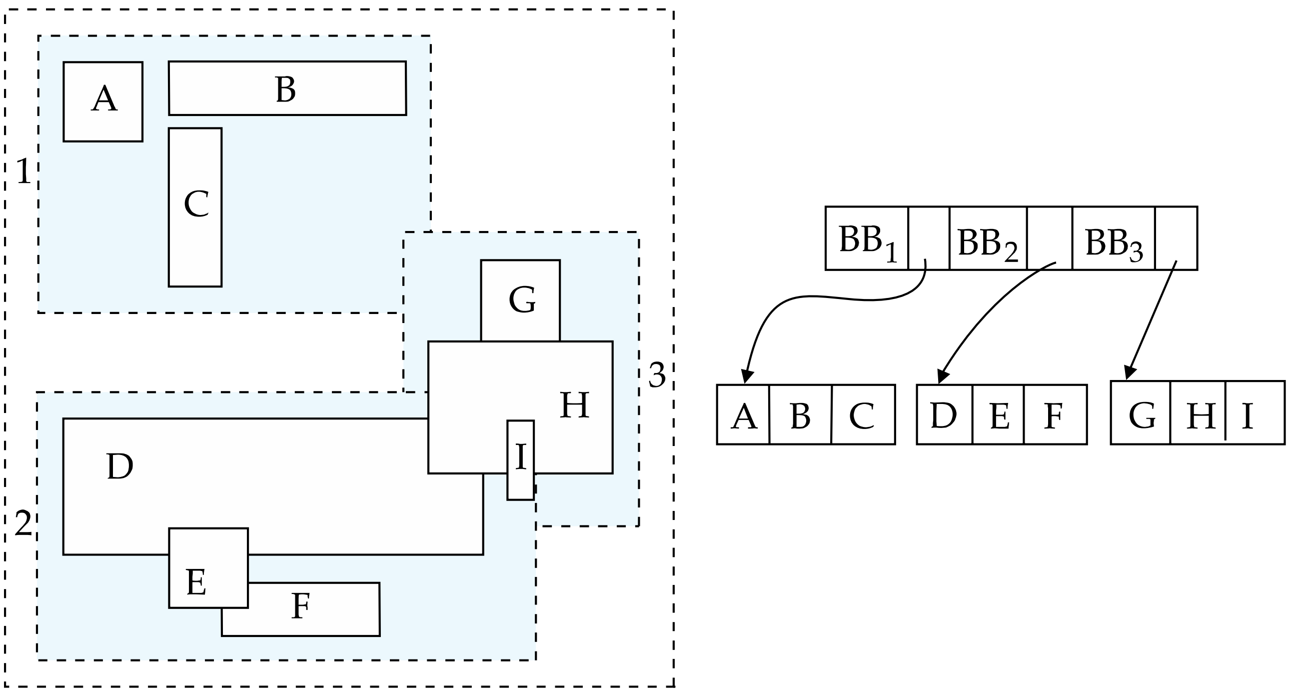
- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

KD Tree and Quad Tree



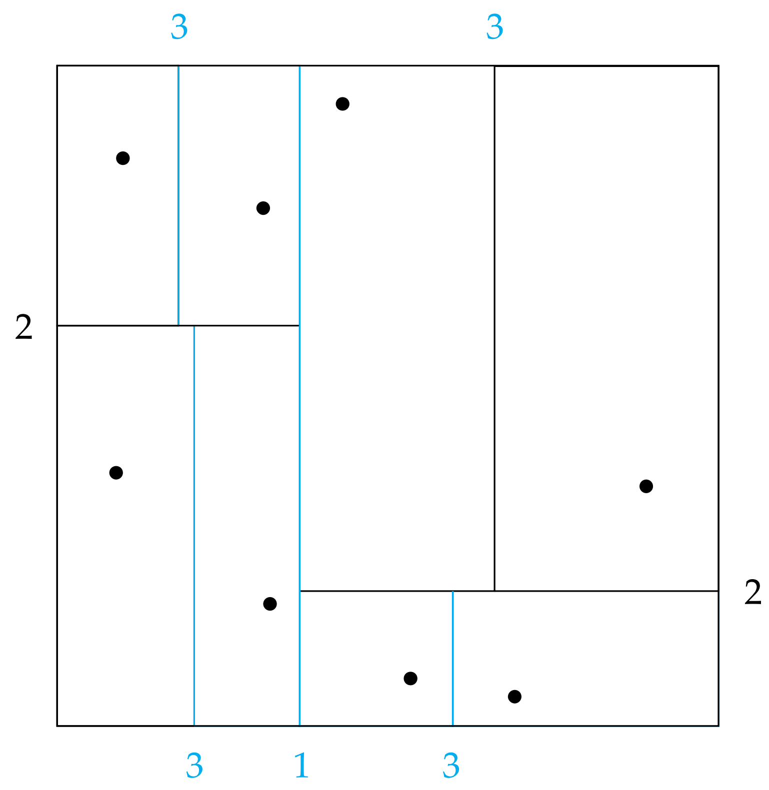


R-Tree



Indexing of Spatial Data

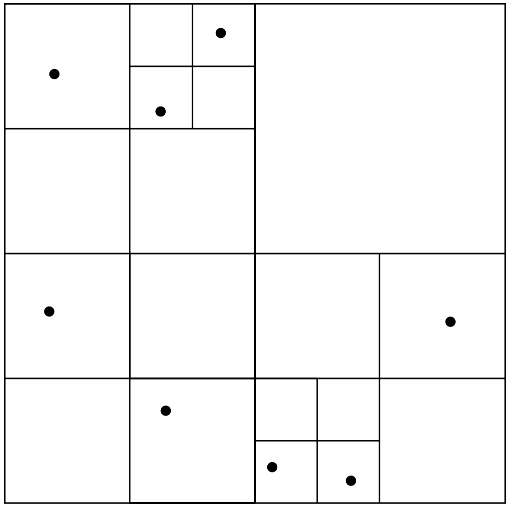
- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).

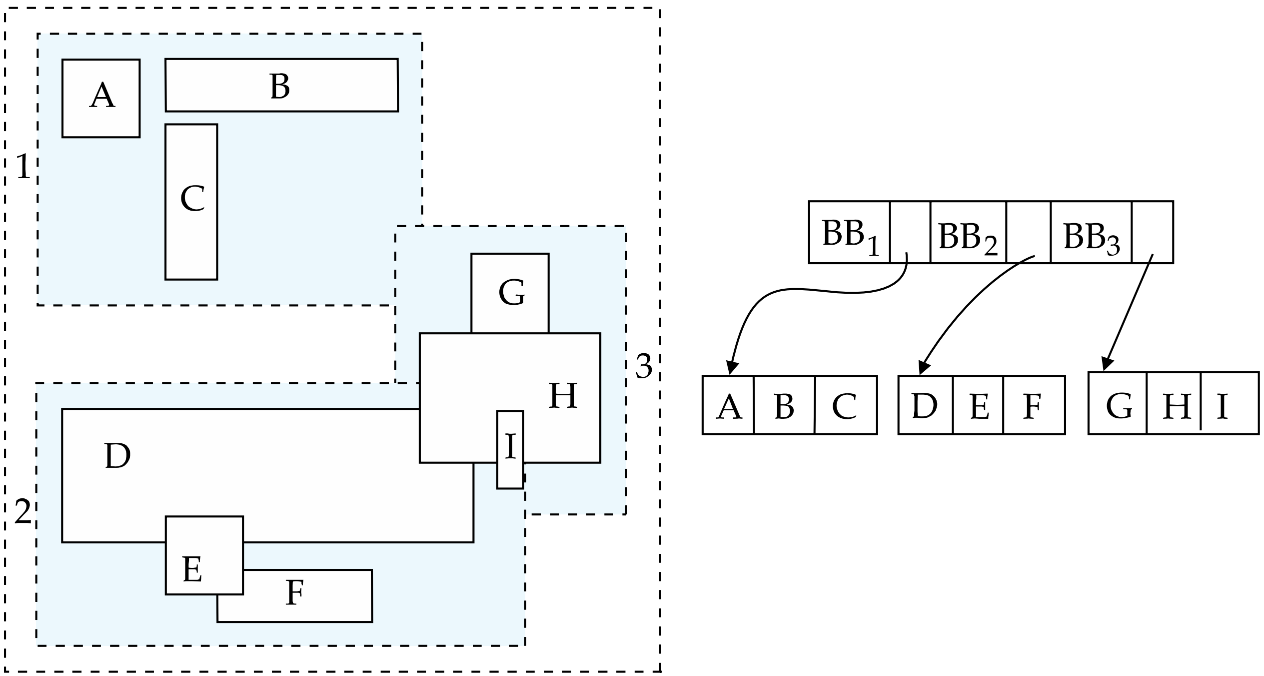


R-Trees

- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an
  N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

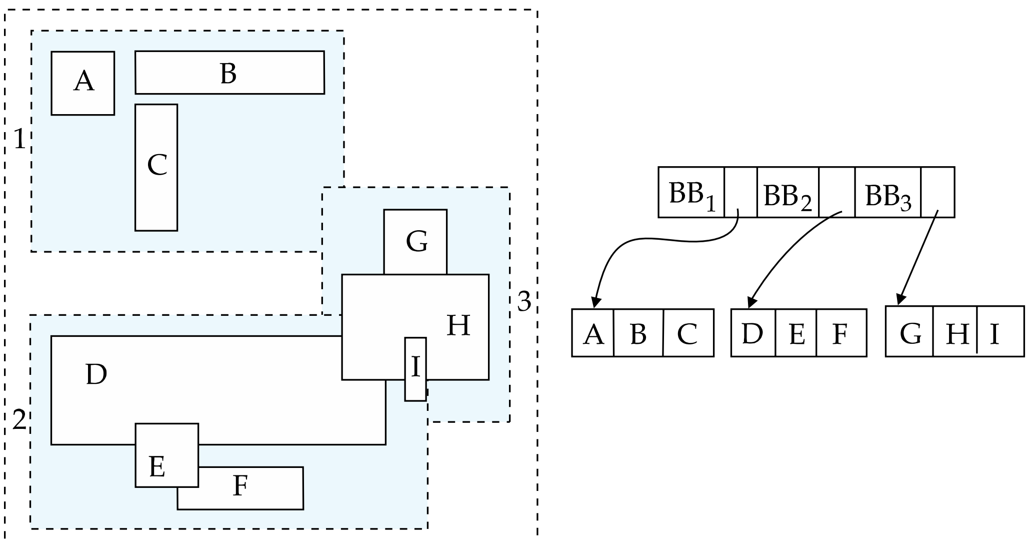
Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.



Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.

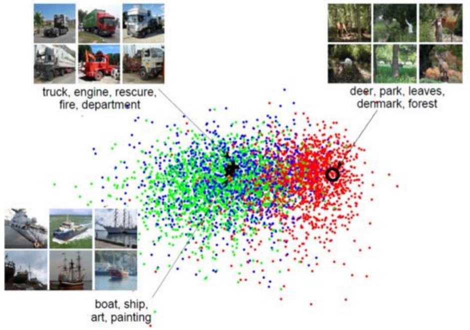


Discussion

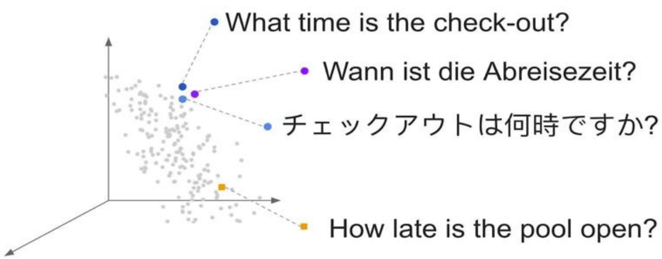
- Suppose you have a spatial database that supports region queries with circular regions, but not nearest-neighbor queries. Describe an algorithm to find the nearest neighbor by making use of multiple region queries.

Nearest Neighbor Search: Representation Learning

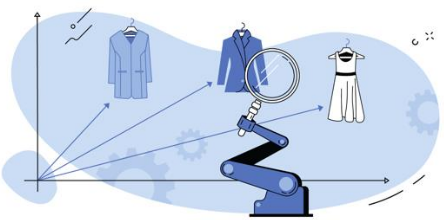
- Objects => High dimensional feature vectors
- Texts, documents, images, video, audio, graphs



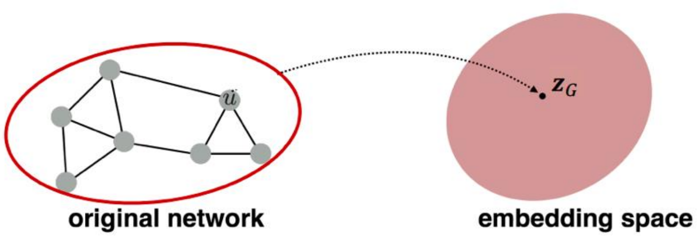
CLIP



word2vec, doc2vec



item2vec, user2vec

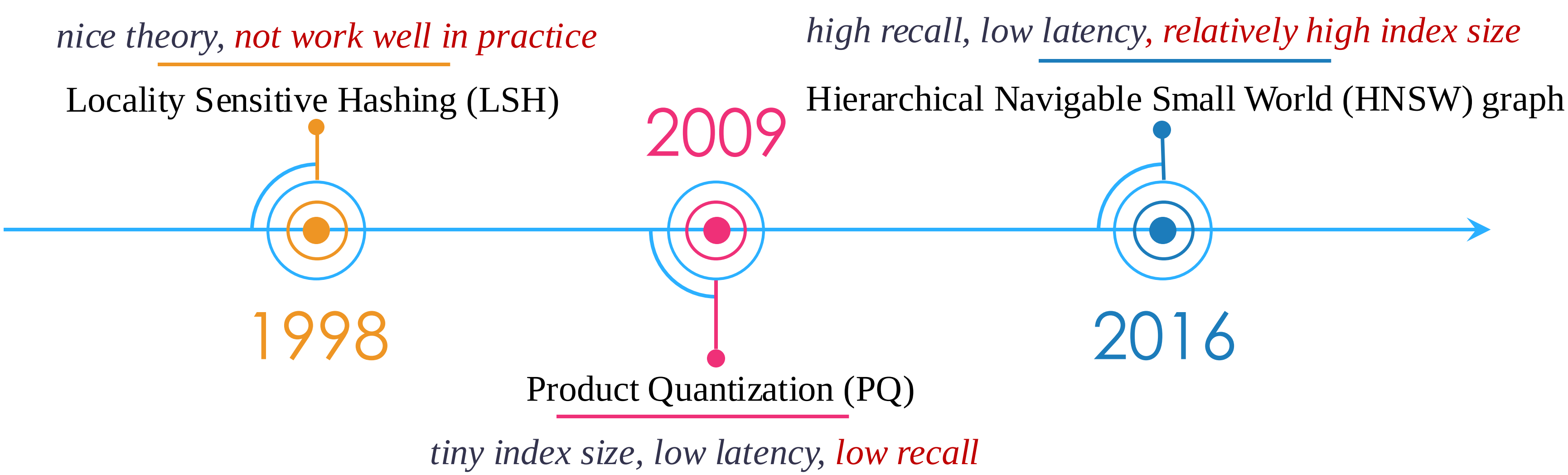


node2vec, graph2vec

https://www.youtube.com/watch?v=dN0lsF2cvm4

Approximate Nearest Neighbor Search (ANNS)

- Finding the exact nearest neighbors is hard due to the “curse of dimensionality”.
- Approximate nearest neighbor search is critical in many applications.
  - Vector databases, Retrieval Augmented Generation (RAG), image retrieval, etc.



Pinecone Weaviate milvus qdrant
marqo CloseVector VectorDB turbopuffer <(°O°)>

Outline

- Write Optimized Indices
- Hashing
  - Extendible Hashing
- String Index
- Spatial Index

Review

- Buffer Manager
  - EM Sorting
- Indexes
  - B+Tree
  - Write Optimized Indices
  - Hashing
    - Extendible Hashing
  - String Index
  - Spatial Index

FIN

Any questions?

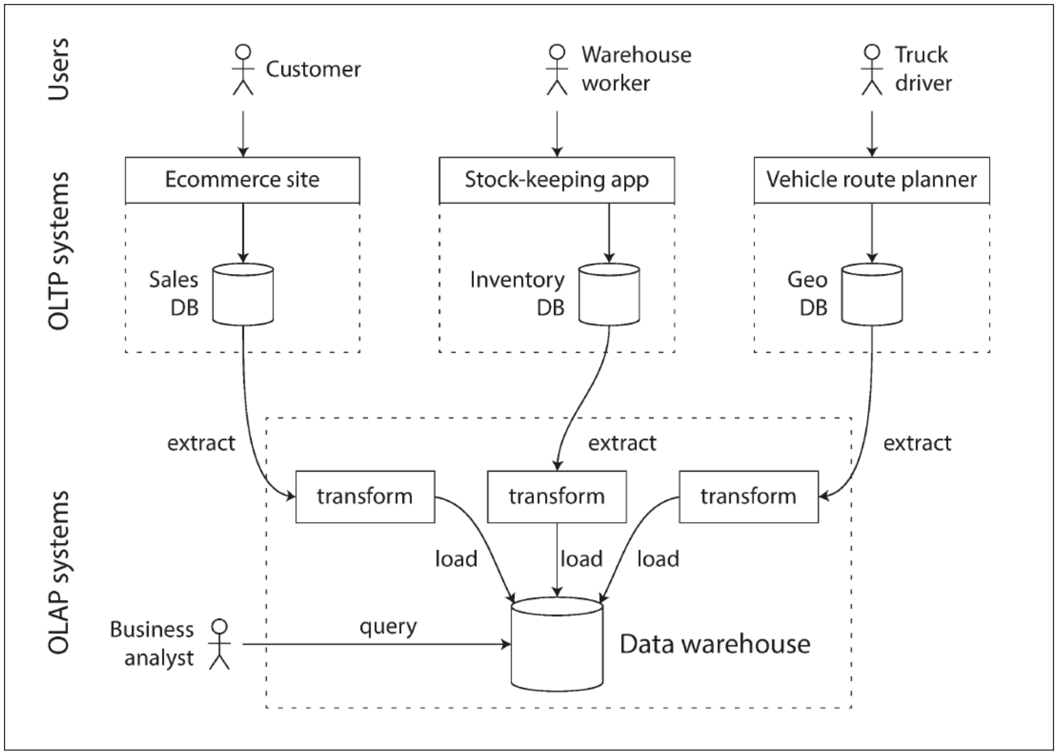


Figure: ETL: Extract Transform Load

Vendors: Expensive

1. Teradata
2. Vertica
3. SAP HANA
4. ParAccel
5. RedShift (Amazon)

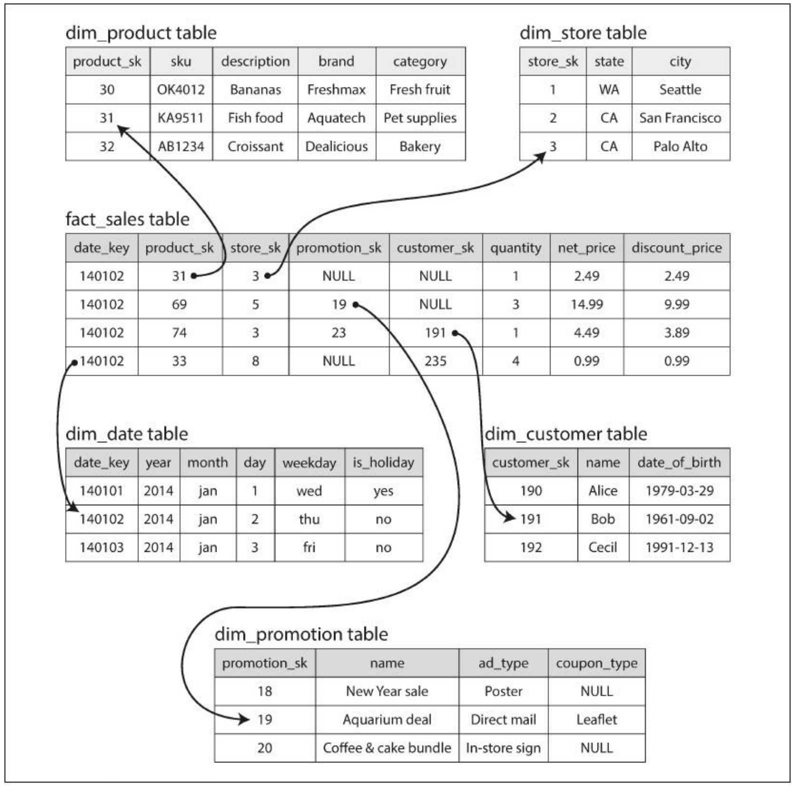
Open Source

- Apache Hive
- Spark SQL
- Cloudera Impala
- Presto (Facebook)
- Apache Tajo and Drill

Schemas for Analytics: Stars and Snowflakes

Star schema.

- Fact table: records for individual events recording who, when, where, why and how of the event, typically 100+ columns.
- Column:
  - An attribute or
  - A foreign key to a dimension table
- Dimension table: a smaller table on one attribute, can also be wide.



dim_product table

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

dim_store table

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

dim_customer table

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

Figure 3-9. Example of a star schema for use in a data warehouse.

Storage and Retrieval

Typical Query: Aggregate over all rows, only relate to a few column.
Example: Get data to analyze whether people are more inclined to buy fresh fruit or candy, depending on the day of the week.

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) as quantity_sold
FROM fact_sales JOIN dim_date JOIN dim_product
WHERE dim_date.year = 2013 AND
  dim_product.category in ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

For fact table with trillions of rows and hundreds of columns, column store is popular.

Storage and Retrieval: Column Store

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Storage and Retrieval: Column Store

Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Figure 3-11. Compressed, bitmap-indexed storage of a single column.

Bit operation. AND, OR, XOR.

- A batch of 64 bits can be computed in one operation.

Bitmap encoding. WHERE product_sk IN (30,68,69):

- Load the bitmaps of product_sk= 30 product_sk= 68 and product_sk= 69, and
- Calculate the bitwise OR.

Storage and Retrieval: Column Store

- Further optimization.
  - Vectorized processing on Single-Instruction-Multi-Data (SIMD)
  - Sort the data in multiple orders for
    - RAID
    - Fast query processing
    - Concurrency

Storage and Retrieval: Aggregation

Cross-tabulation (cross-tab, pivot-table).

clothes_size **all**

color

item_name	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5

Storage and Retrieval: Aggregation

Cross-tabulation (cross-tab, pivot-table).

clothes_size **all**

color

item_name	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

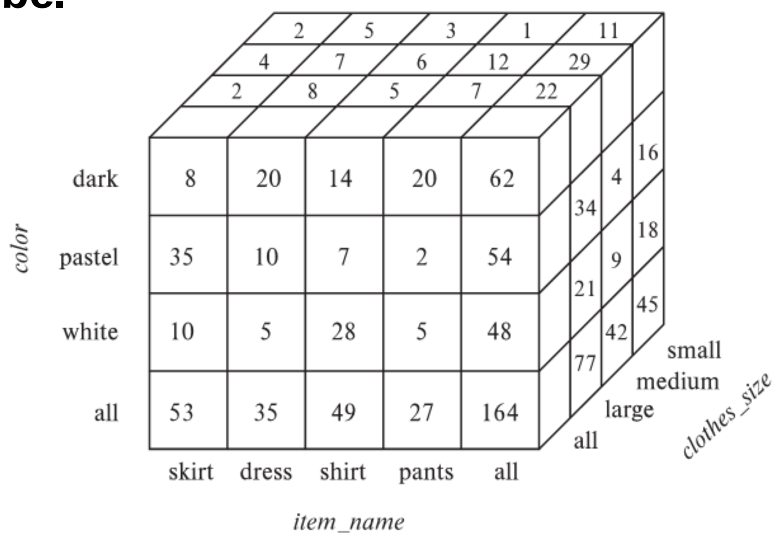
- Values for one of the dimension attributes form the row headers
- Values for another dimension attribute form the column headers
- Other dimension attributes are listed on top
- Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5

Storage and Retrieval: Aggregation

NCE

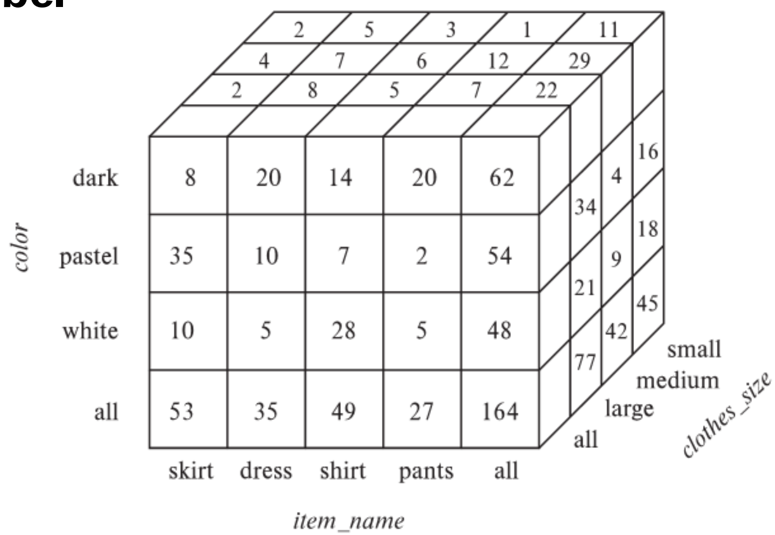
Data Cube.



item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5

Storage and Retrieval: Aggregation

Data Cube.



- A data cube is a multidimensional generalization of a cross-tab
- Can have *n* dimensions; we show 3 above
- Cross-tabs can be used as views on a data cube

item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5

NCE

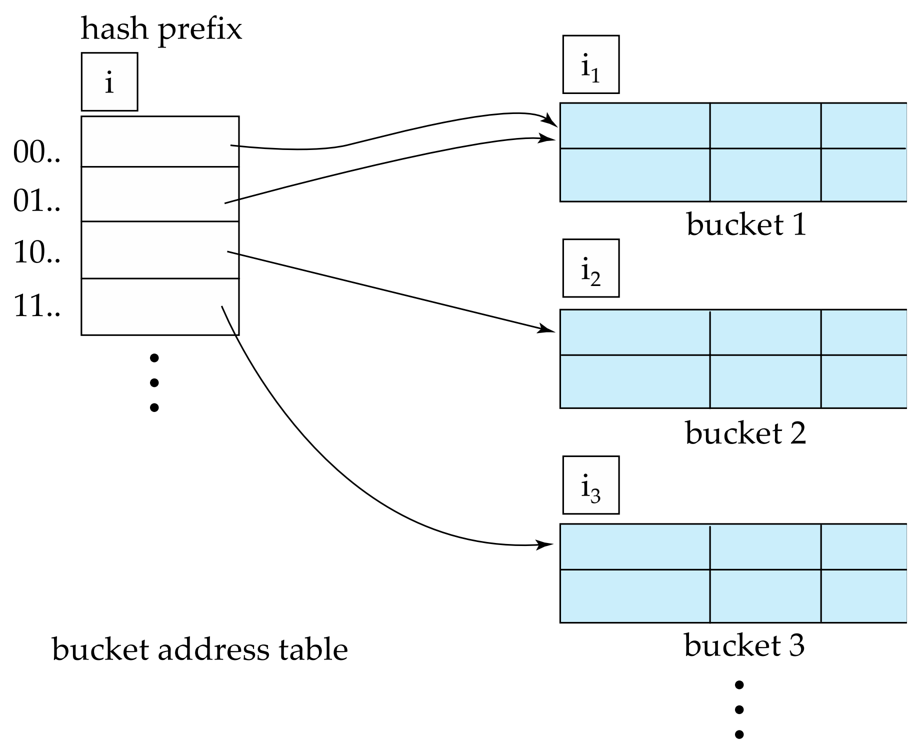
THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE

FIN

Any questions?

Extendable Hash Structure



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

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, $0 \le i \le 32$.
    - 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.

Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
  - All 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