COL362/632 Introduction to Database Management Systems

Indexing – Hash Tables, Bitmaps, & Bloom Filters

Kaustubh Beedkar

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Outline

Hash Index

- Widely used for indexes in main memory
- but, hash file organization is not very widely used
- Given: set K of keys and B buckets
- hash function $h : K \mapsto B$
- Insert: key k, compute $h(k)$ and add entry to the offset
- Use hash function such that
  - Distribution is uniform
  - Distribution is random
- Types of hash indexing
  1. Static Hashing
  2. Dynamic Hashing

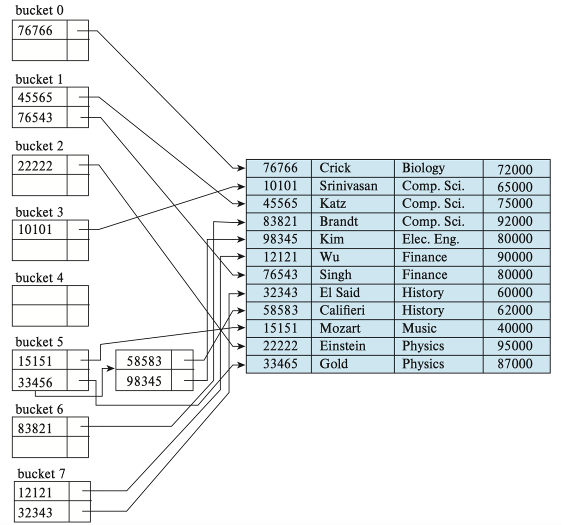
Hashing Index

- Records are stored in buckets
  - linked list of disk blocks
  - linked list of index entries or records (for in-memory indexes)
- Hash file organization
  - Store actual records instead of record pointers

Bucket Overflows

- Occur due to
  - Insufficient buckets
  - Data skew or partitioning skew
- Use overflow chaining to handle multiple records in the same buckets

Example Hash Index



Example Hash Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

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

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

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

bucket 7

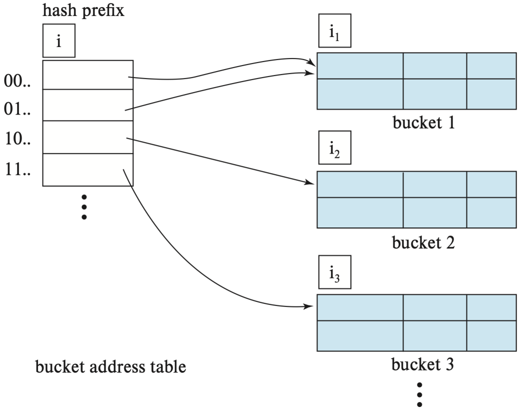
Static Hashing

- Set of buckets is fixed at the time of index creation (problem!)
  - Long overflow chains as database grows
  - Space wastage as database shrinks
  - Expensive reorganization

Extendible Hashing

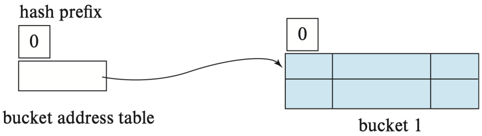
- Type of dynamic hashing
- Keep directory of pointers to buckets
- Reorganize the index by doubling the directory

Hash Data structure



- bucket address table: directory of pointers to buckets
- $hash(key) = b$-bit binary integer (typically $b = 32$)
- use i higher order bits as an offset
- use i bits of $hash(key)$ to determine bucket
- each bucket associated with i_j as length of common hash prefix

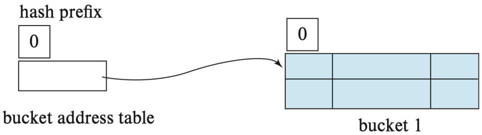
Index Construction



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

Index Construction

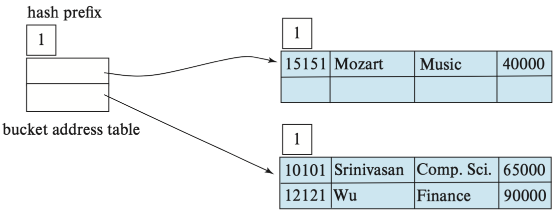
1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)



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

Index Construction

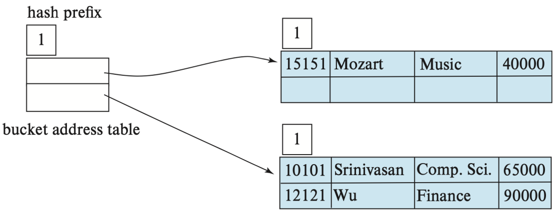
1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)



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

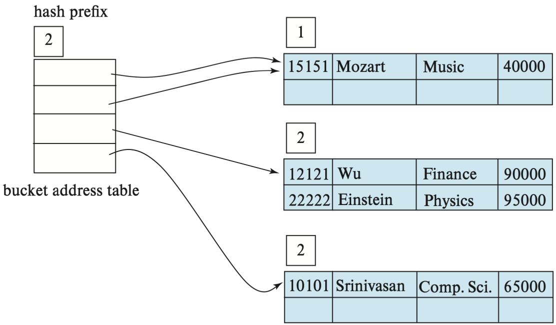
Index Construction

1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)



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

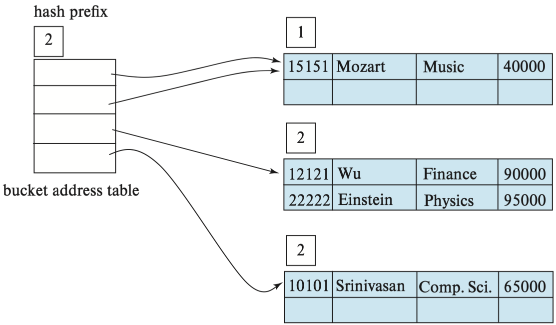
1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)



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

Index Construction

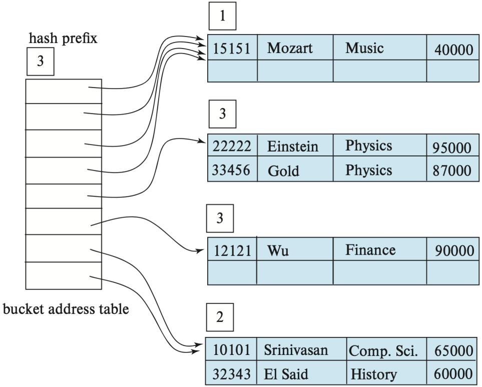
1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)
5. (32343, El Said, History, 60000)
6. (33456, Gold, Physics, 87000)



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

Index Construction

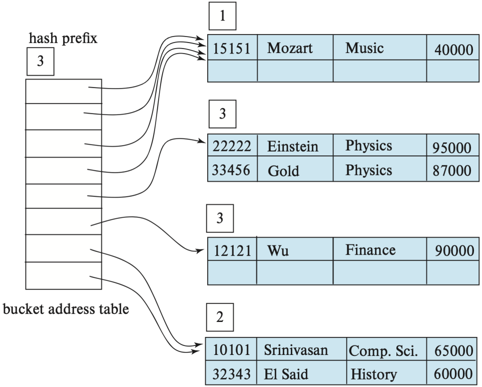
1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)
5. (32343, El Said, History, 60000)
6. (33456, Gold, Physics, 87000)



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

Index Construction

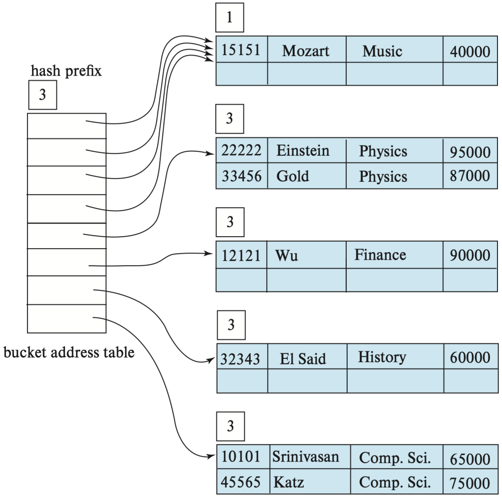
1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)
5. (32343, El Said, History, 60000)
6. (33456, Gold, Physics, 87000)
7. (45565, Katz, Comp. Sci., 75000)



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

Index Construction

1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)
5. (32343, El Said, History, 60000)
6. (33456, Gold, Physics, 87000)
7. (45565, Katz, Comp. Sci., 75000)

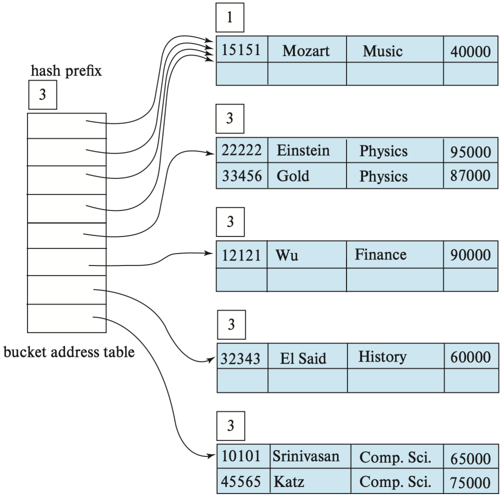


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

Index Construction

1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)
5. (32343, El Said, History, 60000)
6. (33456, Gold, Physics, 87000)
7. (45565, Katz, Comp. Sci., 75000)
8. (58583, Califieri, History, 62000)
9. (76543, Singh, Finance, 80000)
10. (76766, Crick, Biology, 72000)
11. (83821, Brandt, Comp. Sci., 92000)

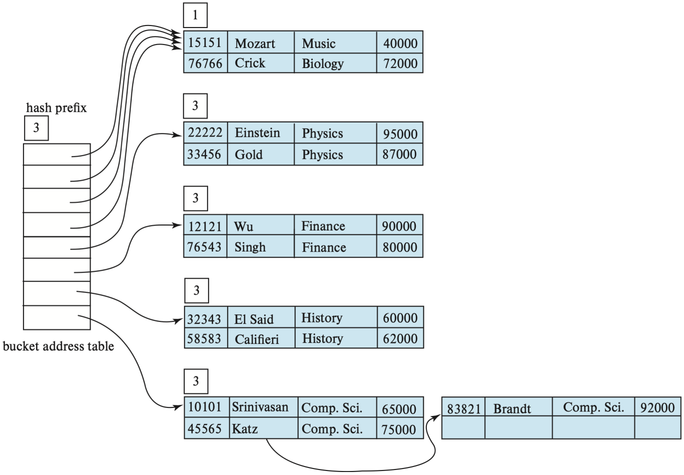
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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



Index Construction

1. (10101, Srinivasan, Comp. Sci., 65000)
2. (12121, Wu, Finance, 90000)
3. (15151, Mozart, Music, 40000)
4. (22222, Einstein, Physics, 95000)
5. (32343, El Said, History, 60000)
6. (33456, Gold, Physics, 87000)
7. (45565, Katz, Comp. Sci., 75000)
8. (58583, Califieri, History, 62000)
9. (76543, Singh, Finance, 80000)
10. (76766, Crick, Biology, 72000)
11. (83821, Brandt, Comp. Sci., 92000)

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



Outline

Bitmap Index

- A **bitmap index** for a field F is a collection of bit-vectors of length n, one for each possible value that may appear in the field F
- Example: Book table:

RowID	BookID	Title	Binding	Language
1	9436	Winnie the Pooh	Hardcover	English
2	1029	Le Petit Prince	Paperback	French
3	8733	Alice in Wonderland	Paperback	English
4	2059	Wind in the Willows	Hardcover	English
5	5995	A Bear Called Paddington	Paperback	English
6	1031	Pierre Lapin	Hardcover	French
7	3984	Le avventure di Pinnochio	Hardcover	Italian

- Bitmap indices for Binding and Language have the following two and three position bitmaps:

```
Binding:                      Language:
  Hardcover:  1001011            English:    1011100
  Paperback:  0110100            French:     0100010
                                 Italian:    0000001
```

- A bitmap index makes it very fast to locate rows with a certain value.

Bitmap Index

- It is also possible to combine bitmap indexes
- Example: to find all hardcover books in English

```
Binding:
  Hardcover:  1001011
  Paperback:  0110100              1001011
Language:                      AND 1011100
  English:    1011100           =  1001000
  French:     0100010
  Italian:    0000001
```

- OR, NOT operations are also possible
- When there are few values, a bitmap index occupies little space
- It is also possible to make the bitmap index occupy less space: skip trailing zeros, various kinds of compression

Join Index

- A **join index** is another means of speeding up query performance
- Like a materialized view, a join index represents the pre-computation of a query $\rightarrow$ **a join query**
- Unlike a materialized view, a join index contains pointers to the rows in the argument tables that satisfy the join predicate.
- A join index is capable of speeding up **certain queries** dramatically, while offering no speed up of other queries.
- Instead of storing list of pointers to rows, position bitmaps can be used.

Example: Join Index

Book table

RowID	BookID	Title	Genre
1	**7493**	Tropical Food	Cooking
2	9436	Winnie the Pooh	Childrens' books
3	9948	Gone with the Wind	Fiction
4	9967	Italian Food	Cooking

Sales table

RowID	BookID	ShopID	SalesID	DayID	Count	Price
1	9436	854	1021	2475	2	30
2	**7493**	854	1021	2475	1	20
3	9948	876	2098	3456	1	20
4	**7493**	876	2231	3456	2	40
5	**7493**	876	3049	2475	1	20
6	9436	854	3362	3569	2	30
7	9967	731	3460	3569	1	35
8	**7493**	731	3460	3569	1	15
9	9948	731	3460	3569	1	15

Join index for Book and sales

list of pointers

Book_RowID	Sales_RowID
1	**(2,4,5,8)**
2	(1,6)
3	(3,9)
4	(7)

position bitmaps

Book_RowID	Sales_RowID
1	**010110010**
2	100001000
3	001000001
4	000000100

Outline

Set membership queries

Given: a set M of n keys

- Each from some finite domain U

Sought: a data structure that supports **set membership queries**: is $k \in M$

- Common problem: caches, dictionaries, databases, ...
- Goal: make data structure as small as possible

Does this help? Not really, can show

- Essentially need to store M
- $\Omega(n \log(|U|/n))$ space

Approximate set membership

- Suppose we can tolerate a small amount of error
  - Few **false positives** acceptable
    answer is yes, but $k \notin M$
  - No **false negatives** allowed
    answer is no, but $k \in M$

- Now, we can do better: **Bloom Filters**
  - Bit vector of m bits, initialized 0
  - Hash function $h : U \mapsto \{0, \ldots, m-1\}$
  - Insert key k: set bit $h(k)$
  - Query key k: answer yes, if bit $h(k)$ set, else no

Bloom Filters in practice

- Need to ensure low **false positive rate (FPR)**
  - Probability of false positive when asking for random $k \notin M$
  - use s hash functions instead of just one
  - Idea: Hash each key s times
  - Insert: set all s bits
  - Query: test whether all s bits $= 1$
  - Both operations take $O(s)$ time
- Assumptions
  - Storage cost of hash functions negligible
  - Hash values are random (independent, uniform)
  - FPR depends only on m, n, s
  - Theoretical analysis
    some other time (look out for advanced data management courses next semester)

Summary

Evaluating an Index

- Which access type does the index support?
- Complexity of accessing a record
- Complexity of inserting a record
- Complexity of deleting a record
- Space complexity

Common Practice

- For point queries use hash index
- For point and range queries use B-tree
- Additionally, for multiple selections use Bitmaps and
- for set containment queries (with acceptable errors) use Bloom filters