Database Applications
Lecture 2: Indexing

Santha Sumanasekara
July 2022

RMIT UNIVERSITY

1

1

Menti-time!

Menti.com
Code:

RMIT UNIVERSITY

2

Database Indexing

- Overview
- Examples of index-based queries -- Demos
- Different indexing methods used in databases
  - B+ -- Tree Index
  - Hash Index
  - Bitmap Index
- Disadvantages of Indexing

RMIT UNIVERSITY

3

3

Why is database optimisation important?

- As data size grows, query performance degrades.
  - In many applications, growth is not even linear. Recall the exponential growth of Facebook users or Amazon catalog.
- Hardware improvements often lag the growth of data (do not expect exponential growth of SSD (solid state disk) speeds over next decade.
- Disk Access time is critical!

RMIT UNIVERSITY

4

4

Disk Access!

- First rule of database performance:
  Disk access is the most expensive thing databases do!
- Accessing data in memory can be 10-100ns
- Accessing data on disk can be up to 10s of ms
  - *That's 5-6 orders of magnitude difference!*
  - Even solid-state drives are 10s-100s of μs (1000x slower)
- Unfortunately, disk IO is usually unavoidable
  - Usually the data simply doesn't fit into memory
  - The data needs to be persistent for when the DB is shut down, or when the server crashes, etc.
- DBs work very hard to minimise the amount of disk IO.

RMIT UNIVERSITY

5

5

Disk Access!

This table has 7 million rows – on-time performance of US flights in a calendar year.

- Run the following query twice:

```
SELECT COUNT(*)
   FROM ontime;
```

- Did you see any difference in runtime?
  - Why?

RMIT UNIVERSITY

6

6

Minimising Disk Access!

- Consider the following query:

```
SELECT * FROM ontime WHERE serialnum = 12345;
```

- these are 2 different ways that SQL could find the results:
  - **Do a "full table scan":** look at every single row in the table, return the matching rows.
  - **Make a copy of the table sorted by `serialnum`**, then do a "**binary search"** to find the row where the `serialnum` is 12345.

RMIT UNIVERSITY

7

7

Minimising Disk Access!

Will show "how" later!

- Consider the following query:

```
SELECT * FROM ontime WHERE serialnum = 12345;
```

- Which one is faster?
- It depends on the data, and on how often the query will be executed.
- If the table is only 10 rows long, then a full table scan only requires looking at 10 rows, and the first plan would work out well.
- However our `ontime` table has 7 million rows!
- It would be faster to do a binary search on a sorted table - we only need 23 lookups to find a value in 7 million rows.
- But, sorting of 7 million rows is expensive!

RMIT UNIVERSITY

8

8

Minimising Disk Access!

- Consider the following query:

```
SELECT * FROM ontime WHERE serialnum = 12345;
```

- Apparently, both methods are not very efficient!
- Building and using an **Index** is a very promising approach this problem.

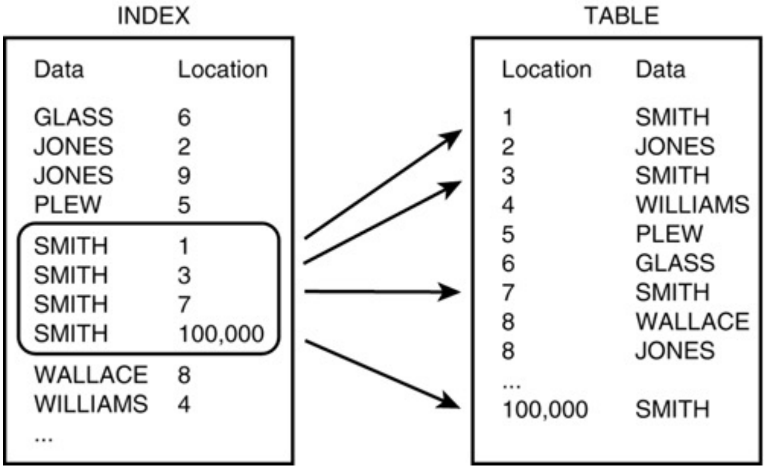
RMIT UNIVERSITY

9

9

Minimising Disk Access!

- An (over-)simplified illustration of how indexing help reduce disk accesses.



RMIT UNIVERSITY

10

10

Minimising Disk Access!

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file

RMIT UNIVERSITY

11

11

Minimising Disk Access!

- For mathematically inclined:

Given that an index record contains only the indexed field and a pointer to the original record, it will be smaller than the multi-field record that it points to. So the index itself requires fewer disk blocks than the original table, which therefore requires fewer block accesses to iterate through. The schema for an index on the *SerialNum* field is outlined below;

Fieldname	Data type	size
SerialNum	serial	4 bytes
(record pointer)	Special	4 bytes

Note: Pointers in MySQL are 2, 3, 4 or 5 bytes in length depending on the size of the table.

Given our sample database of r = 7,000,000 records with an index record length of R = 8 bytes and using the default block size B = 1,024 bytes. The blocking factor of the index would be bfr = (B/R) = 1024/8 = 128 records per disk block. The total number of blocks required to hold the index is N = (r/bfr) = 7000000/128 = 54688 blocks.
Now a search using the *SerialNum* field can utilize the index to increase performance. This allows for a binary search of the index with an average of log2 54688 = 15.73 = 16 block accesses. To find the address of the actual record, which requires a further block access to read, bringing the total to 16 + 1 = 17 block accesses, a far cry from the 7,000,000 block accesses required to find a *SerialNum* match in the non-indexed table.

RMIT UNIVERSITY

12

12

Demo Time!

Let’s try out a query with and without an index to help us.

RMIT UNIVERSITY

13

Demo Time!

- Consider the following query:

```
SELECT * FROM ontime
   WHERE serialnum = 12345;
```

```
SELECT * FROM ontime
   WHERE tailnum ='N305SW';
```

- There is an index on talinum attribute.
- So, theoretically, the first query should use a full table scan, and the second query should utilise the index.

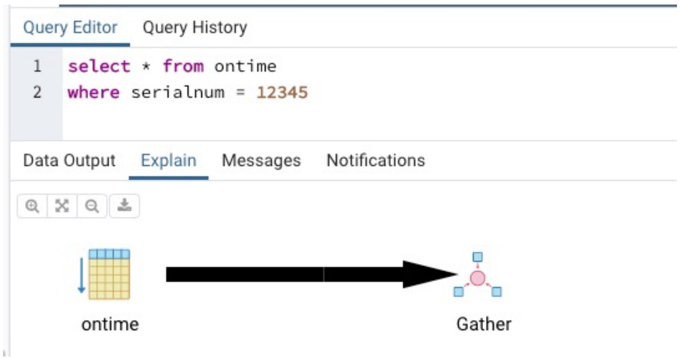
RMIT UNIVERSITY

14

14

Demo Time!

```
SELECT * FROM ontime
   WHERE serialnum = 12345;
```



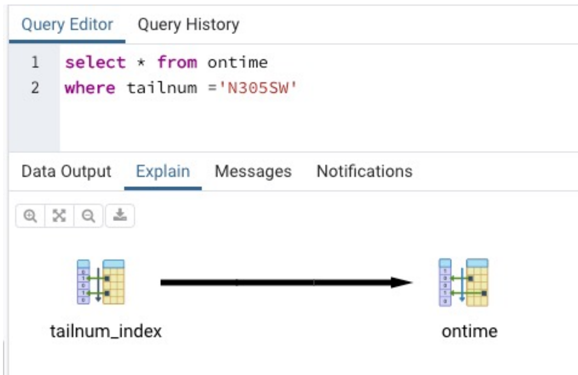
RMIT UNIVERSITY

15

15

Demo Time!

```
SELECT * FROM ontime
   WHERE tailnum ='N305SW';
```



RMIT UNIVERSITY

16

16

Demo Time – What is an Index-only Scan query?

```
SELECT tailnum FROM ontime
   WHERE tailnum LIKE 'N305%';
```

- Many databases support *index-only scans*, which can answer queries from an index alone without any table access.
- Oracle and SQL Server can do index-only scans.
- PostgreSQL has partial support (depends on index type).

RMIT UNIVERSITY

17

Demo Time – What is an Index-only Scan query?

```
SELECT tailnum FROM ontime
   WHERE tailnum LIKE 'N305%';
```

RMIT UNIVERSITY

18

Building Indexes -- SQL

- Consider building an index on `SerialNum` attribute.

```
CREATE INDEX serial_index ON ontime(serialnum);
```

- "serial_index" is a label given to the index.
- It is possible to build an index on more than one attribute (a composite index).
- Say if we do lot of queries on dates, we can build a date index.

```
CREATE INDEX date_index ON
                    ontime(year, month, dayofmonth);
```

RMIT UNIVERSITY

19

19

Index Characteristics: Ordered vs. Hash

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order

```
CREATE INDEX serial_index
      ON ontime(serialnum);
```

  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

```
CREATE INDEX year_index
      ON ontime USING HASH(year);
```

RMIT UNIVERSITY

20

20

Index Characteristics: dense vs. sparse

- The ordered indexes can be built in two variations:
  - A **dense index** includes every single value from the source column(s). Faster lookups, but a larger space overhead.
  - A **sparse index** only includes some of the values. Lookups require searching more records, but index is smaller.

RMIT UNIVERSITY

21

21

Index Characteristics: dense

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

22

22

Index Characteristics: sparse

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

RMIT UNIVERSITY

23

23

Index Implementations

- Virtually all databases provide ordered indexes, using some kind of balanced tree structure
  - B+-tree and B-tree indexes, typically referred to as “b tree” indexes
- Some databases also provide hash indexes
  - More complex to manage than ordered indexes, so not very common in open-source databases
- Several other kinds of indexes as well:
  - Bitmap indexes – to speed up queries on multiple keys
    - Also less common in open-source databases
  - R-tree indexes – to make spatial queries very fast
    - With ubiquity of geospatial data, quite common these days

RMIT UNIVERSITY

24

24

Index Implementations – B+ Tree

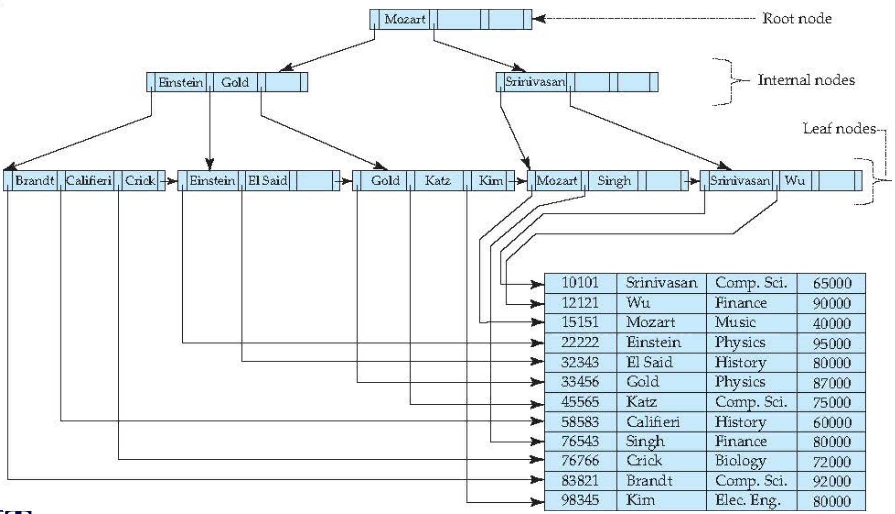
- A *very* widely used ordered index storage format
- Manages a balanced tree structure
  - Every path from root to leaf is the same length
  - Generally remains efficient for selects, even with inserts and deletes occurring
- Can consume significant space, since individual nodes can be up to half empty!
- Index updates for insert and delete can be slow.
  - Tree structure must be updated properly
  - Inserts may require one or more nodes to be split
  - Deletes may require one or more nodes to be merged.
- Performance benefits on queries more than outweigh these costs!
- Queries are straightforward: exact match, partial-match, range, etc

RMIT UNIVERSITY

25

25

Index Implementations – B+ Tree
Tree



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

RMIT UNIVERSITY

26

26

Index Implementations – Hash Index

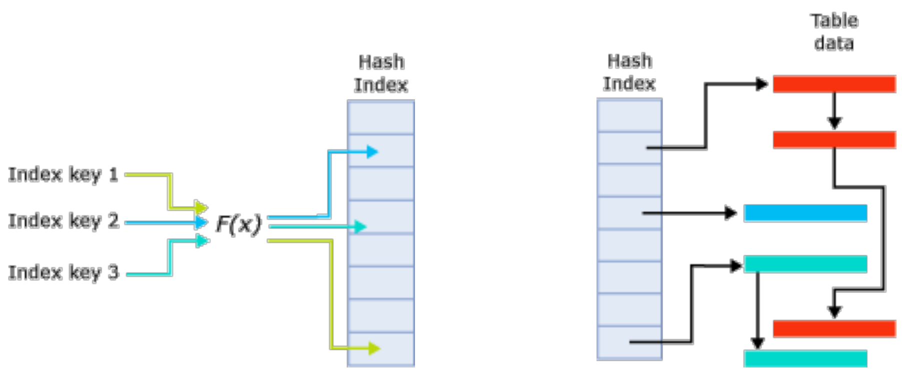
- A hash index consists of a collection of buckets organized in an array. A hash function maps index keys to corresponding buckets in the hash index.
- Multiple index keys may be mapped to the same hash bucket.
- If two index keys are mapped to the same hash bucket, there is a hash collision. A large number of hash collisions can have a performance impact on read operations.
- The in-memory hash index structure consists of an array of memory pointers.

RMIT UNIVERSITY

27

27

Index Implementations – Hash Index



RMIT UNIVERSITY

28

28

B+ Tree Index	Hash Index
`CREATE INDEX name ON table(attributes);`	`CREATE INDEX name ON table USING HASH(attributes);`
Works with all kinds of queries: exact match, partial match, range, etc	Works with exact match only
Generally takes up disk space, and based on the choice of the key, can be large	Small(ish) enough to maintain in-memory, and as a result, very fast
Not sensitive to data spread (even mostly skewed data produce balanced ++ trees.	If (key) data spread is skewed, there can be lot of collisions and as a result, performance can degrade.

Verifying Indexing Usage

- Very important to verify that your new index is actually being used!
- Some times (cost-based) query optimiser decides against the use of an index! It might have determined a full table scan can be cheaper.
- Consider the following query:

```
SELECT tailnum FROM ontime
   WHERE year >= 2008;
```

- Can you guess why the optimiser made that choice?

RMIT UNIVERSITY

Indexing – Disadvantages

- Indexes impose an overhead in both space and time
- Speeds up "SELECT"s, but slows down all modifications
- Large keys seriously degrade index performance.
  - Example: B+ trees
  - Biggest benefit is very large branching factor of each node
  - Large key-values will dramatically reduce the branching factor, deepening the tree and increasing IO costs
- Adding indexes on frequently changing fields
  - Any drawbacks to putting an index on account balances?
  - Account balances change all the time.
  - Will definitely incur a performance penalty on updates

RMIT UNIVERSITY

31

31

Menti-time!

Menti.com
Code:

RMIT UNIVERSITY

32

Questions?

RMIT UNIVERSITY

33