CMSC 206: Database Management Systems

Storage and Indexing

Thomas Laurent

07/11/2020

University of the Philippines Open University

Table of contents

Introduction

Introduction

Last week we saw that the DBMS is somewhat smart and tries to optimise our queries by executing operations in a certain order. We also saw that the way we write our queries can help performance.

This week we will see another important tool for performance in a DB: indexing.

Storage

Different kinds of storage

The information system has access to different kinds of storage that come with different speeds and that are more or less suited to different operations.

The rough idea is that the larger the storage, the slower it is, as fast storage is expensive.

The different kinds of storage include:

- RAM, very fast, expensive, and relatively small. Volatile.
- SSD, faster and more expensive than HDD. Non volatile.
- HDD, slow and relatively cheap. Non volatile.
- Magnetic tape, slowest and largerst, usually for archiving purposes. Non volatile.

Cost of operations

No matter the storage methods, and no matter how the storage is physically organised (e.g. heap file, sorted file, hashed file, ...), reading and writing always has a (time) cost. So we want to minimise the operations needed by our DB.

Indexes will help minimise the reads necessary when searching for data, i.e. running queries.

Indexes

Introduction

So, indexes help us speed up our queries, great! What are they, though? They are basically a data structure that lets us quickly know where records that have a certain value for a column are stored.

Indexes can be found in books: they let us quickly look up where to find some terms in the book without scanning the whole book. The idea here is exactly the same.

Different kinds of indexes

There are different types of indexes that suit different needs.

We can build indexes on a single column or on multiple columns.

We can use different types of indexes: single- or multi- level, hash index (good for equality evaluation) or tree index (good for range queries)...

More details are provided in the additional slides on Moodle.

The other side of the coin

Indexes seem perfect if they speed our queries! Why not just built indexes on every column in our database then and have the fastest DB ever?

Indexes do speed up query execution (most times, for some types of queries an index would not help) but they also have a cost:

- An index is data, which means it must be stored.
- An index must be up to date to be of any use, which means we need to update it every time we insert or delete some records in the DB. This means that although we may be speeding up our queries, we are slowing down other operations.

Indexes in practice

Default indexes

You have been using indexes without knowing it.

PostgreSQL, and most DBMS, create an index by default on the primary key columns, as the primary key is often used to search for records or to join tables.

You can see this index for yourself in psql by typing `\d table_name`.

Making indexes

Syntax

```
CREATE INDEX index_name ON table_name (column_names)
DROP INDEX index_name
```

We create indexes with the CREATE INDEX command. The basic usage only needs the table name and the column names to build the index on. Each DBMS then lets us specify options for the index, such as the index method (hashing, tree, ...) we want to use or the order of the index for example.

We can get rid of an index when it is not needed anymore by using the DROP INDEX command.

When to use indexes?

Some of your colleagues from previous years found interesting resources on the question here and here. The main idea is: we want to create indexes when they are worth it, i.e when we really will use them in queries.

Some rules of thumb for when we might need an index:

- a column is often used to select record (in the where clause)
- a column is often used to join tables (in a join clause)
- a table is big enough (if I have two records in my table, reading my index is as time consuming as reading my table...)

Outline

- Introduction
- Types of single-level ordered indexes
- Multilevel indexes
- Dynamic multilevel indexes using B-trees and B^+-trees
- Indexes on multiple keys
- Other types of indexes
- Some general issues concerning indexes
- Physical database design in relational databases

Introduction – What is an index

- Secondary access path method
- Speed up access to files
- Maps indexing field values disk locations
- Different types: single/multi –level or –field
- Example:

Adanedhel 'Elf-Man', name given to Túrin in Nargothrond. 258
Adunakhôr 'Lord of the West', name taken by the nineteenth King of Númenor, the first to do so in the Adûnaic (Númenórean) tongue; his name in Quenya was Herunúmen. 330
Adurant The sixth and most southerly of the tributaries of Gelion in Ossiriand. The name means 'double stream', referring to its divided course about the island of Tol Galen. 147, 229, 290
Aeglos 'Snow-point', the spear of Gil-galad. 364
Aegnor The fourth son of Finarfin, who with his brother Angrod held the northern slopes of Dorthonion; slain in the Dagor Bragollach. The name means 'Fell Fire', 64,94, 141, 180-82
Aelin-uial 'Meres of Twilight', where Aros flowed into Sirion. 133,145, 203, 267, 285
Aerandir 'Sea-wanderer', one of the three mariners who accompanied Eärendil on his voyages. 307
Aerin A kinswoman of Húrin in Dor-lómin; taken as wife by Brodda the Easterling; aided Morwen after the Nirnaeth Arnoediad. 243, 264

TYPES OF SINGLE-LEVEL ORDERED INDEXES

Single-level ordered indexes

- Index on **single indexing field**
- **Ordered** by indexing field value
  - ⇒Possible to do a *binary search*
- Much smaller than the data file

Primary indexes

- File physically ordered on **key field**
- One index entry **per block** in the data file: **sparse** index
  - <K,P>: key and pointer
  - Key: primary key value of the **first** record of the the block
  - Value: pointer to the block
- The first record of each block is called **anchor record**
- Much smaller than data => Less blocks
  - ⇒ Faster to search through
- Insertions and deletions cause a lot of overhead

1: or last, depending on implementation

Clustering indexes

- File physically ordered on **non-key field** (*clustering field*)
- One index entry **per clustering field value** in the data file: **sparse** index
  - Key: ordering field value
  - Value: pointer to the first block where this value appears
- Much smaller than data => Less blocks
  - ⇒ Faster to search through
- Insertions and deletions cause a lot of overhead
  - ⇒Block reservation: block(s) reserved for each clustering field value

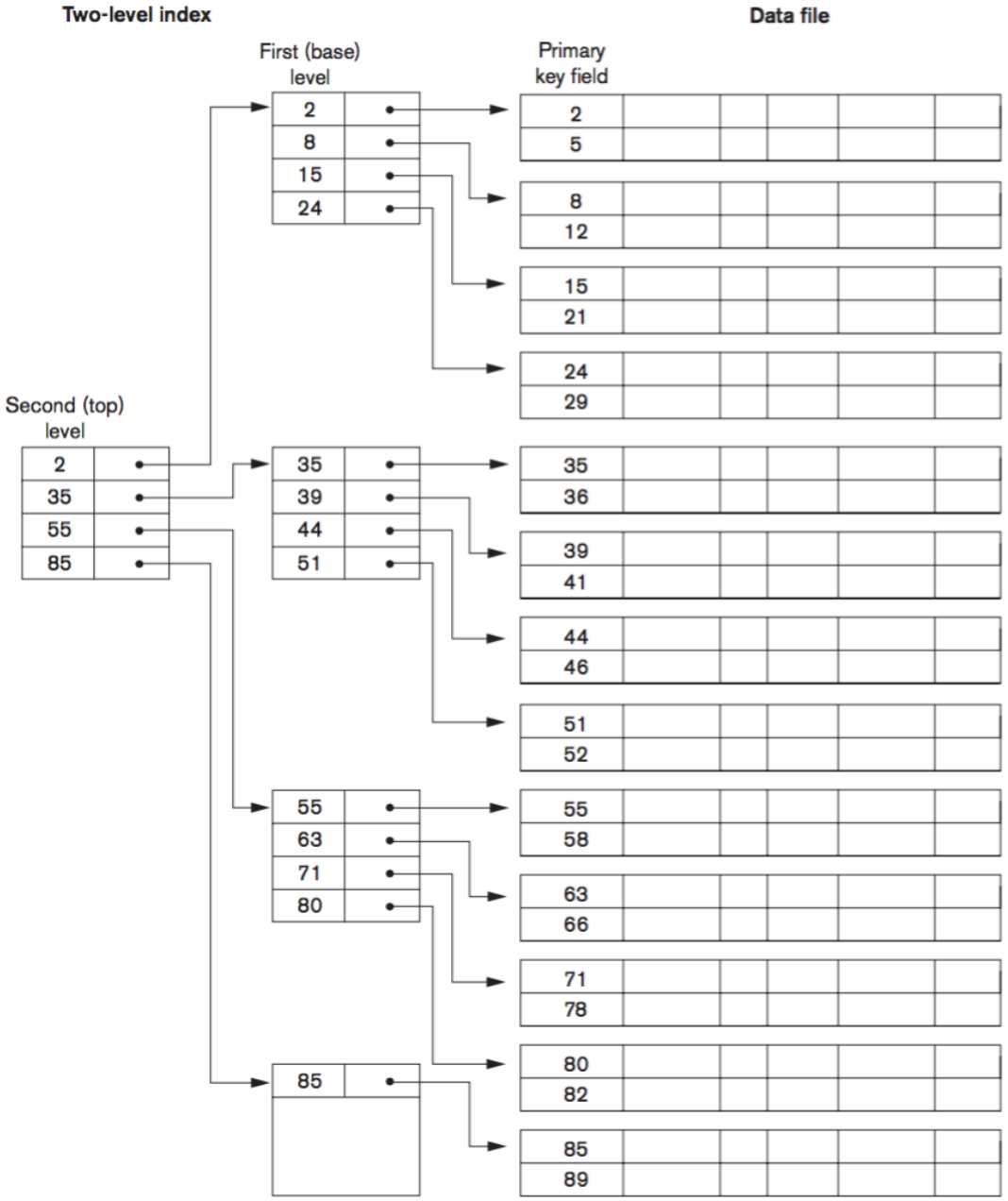
Secondary indexes

- Primary access already exists
- Can create many secondary indexes
- Can be sparse **or** dense
  - Key: non-ordering field value
  - Value: pointer a block or a record
- Date is unordered but index is
  - ⇒Index is searchable by binary search
  - ⇒Provides a **logical** ordering of the data
- Needs more storage space and search time than primary
- Index on non-key, non ordering field: different techniques

MULTILEVEL INDEXES


LEONARDO DICAPRIO
KEN WATANABE
JOSEPH GORDON-LEVITT
MARION COTILLARD
ELLEN PAGE
TOM HARDY
CILLIAN MURPHY
TOM BERENGER
AND MICHAEL CAINE
A FILM BY CHRISTOPHER NOLAN
インセプション
CONNECTION
52 WEST
DVD VIDEO
“★★★★ めくるめく迷宮。息を呑むような離れ業”
– Roger Ebert, CHICAGO SUN-TIMES

Figure 17.6
A two-level primary index resembling ISAM (indexed sequential access method) organization.



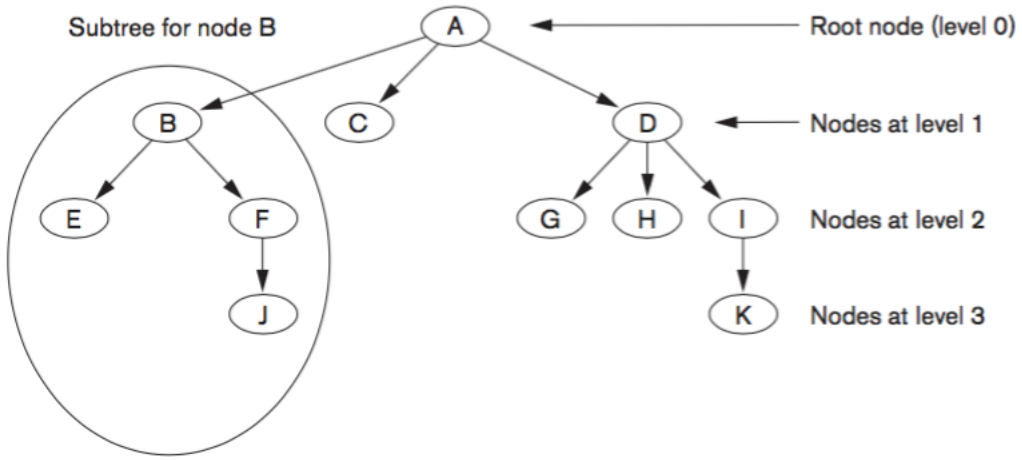
Multilevel indexes

- First level:
  - Any type of index: primary, clustering, secondary
  - Must have **distinct values**
  - Must have **fixed length entry**
- Next levels: primary indexes
- Still overhead for insertions and deletions
  - ⇒Dynamic multilevel index: some space reserved for changes
- Example of multilevel index: IBM ISAM.

DYNAMIC MULTILEVEL INDEXES USING B-TREES AND B^+-TREES

Trees

Figure 17.7
A tree data structure that shows an unbalanced tree.



(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

Search trees

Search Trees. A search tree is slightly different from a multilevel index. A **search tree of order** p is a tree such that each node contains *at most* $p - 1$ search values and p pointers in the order $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$. Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some ordered set of values. All search values are assumed to be unique.[8] Figure 17.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 17.8).

Search trees

- We assume that the search field is unique (a key)
- Challenges:
  - Keep tree **balanced**
  - Minimize changes to tree for each modification in the data
- Balanced tree: all leafs nodes on same level

B-Trees

- Node : $<P_1, <K_1,P_{r1}>,P_2,<K_2, P_{r2}>,\ldots,<K_{q-1},P_{rq-1}>,P_q>$
  - P_i: **tree pointer** to a node of the tree
  - K_i: key field value
  - P_{ri}: **data pointer** to the record with value K_i
- Constraints
  - $K1<K2<\ldots<K_{q-1}$
  - All nodes except root and leaves have at least $\lceil (p/2) \rceil$ tree pointers
  - Root has at least 2 tree pointers, except if it's the only node
  - The tree is balanced
  - For leaf nodes every tree pointer is NULL
- Rules for splitting nodes when full
  - See *Ramakrishnan and Gehrke* (2003)

B^+-Trees

- No data pointers in internal nodes
  - ⇒Can fit more info in a node: bigger p
- Data pointers store only in **leaf nodes**
- Leaf nodes have a pointer to the next leaf node
- Leaf and internal nodes have different stuctures
  - ⇒p and p_{leaf}
- Insertion, deletion and update algorithms
  - Detailed by Toyama sensei in his class

INDEXES ON MULTIPLE KEYS

Indexes on multiple keys - motivation

- How to use indexes for multi-attributes queries ?

1. Use one index and search through the result
2. Use all indexes and combine results

- We have to retrieve a lot of useless data

⇒Indexes on multiple keys at once

Ordered index on multiple attributes

- Same as with single attribute but using a complex key as sorting attribute (e.g. <Dno,Age>)
- Use lexicographic order to sort the index
- Key can become large

Partitioned hashing

- Extension of static external hashing (ch.16)
- Only works for equality comparison queries
- A hash function is created for every attribute we want to use
- The different hashes are concatenated to obtain bucket address
- Drawbacks:
  - Only equality
  - Order is not kept

Grid files

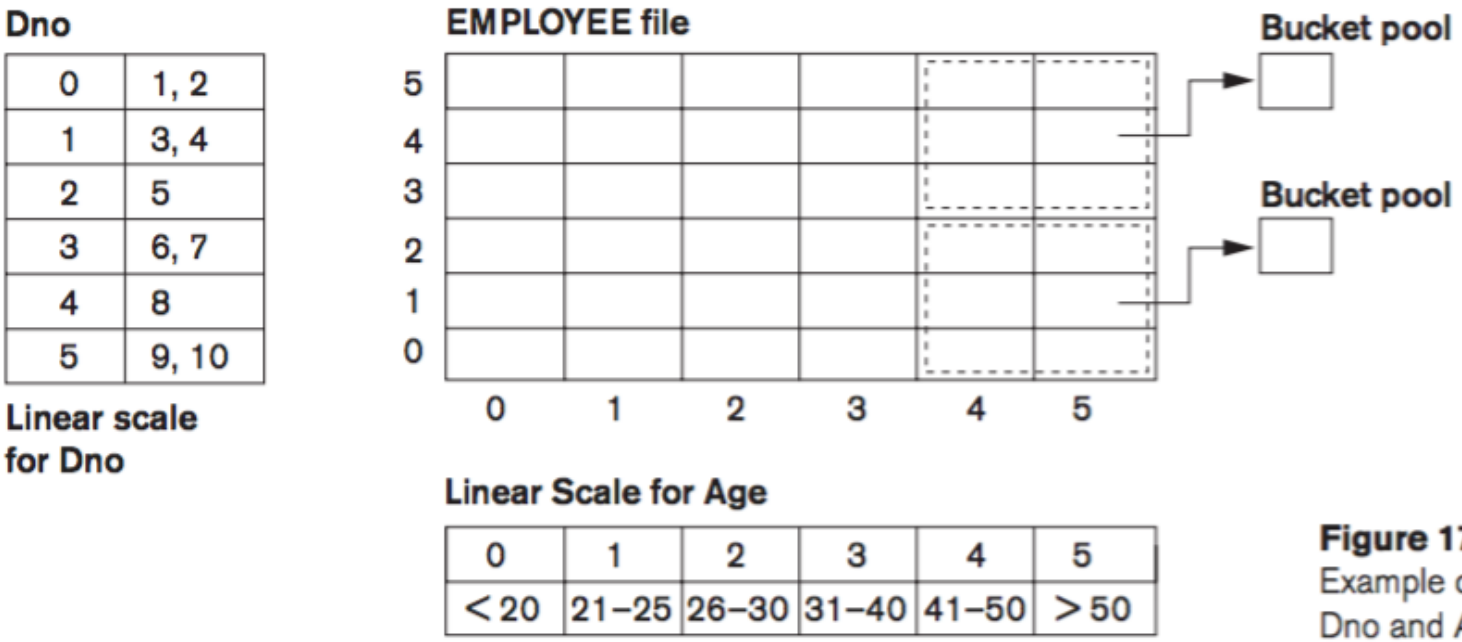
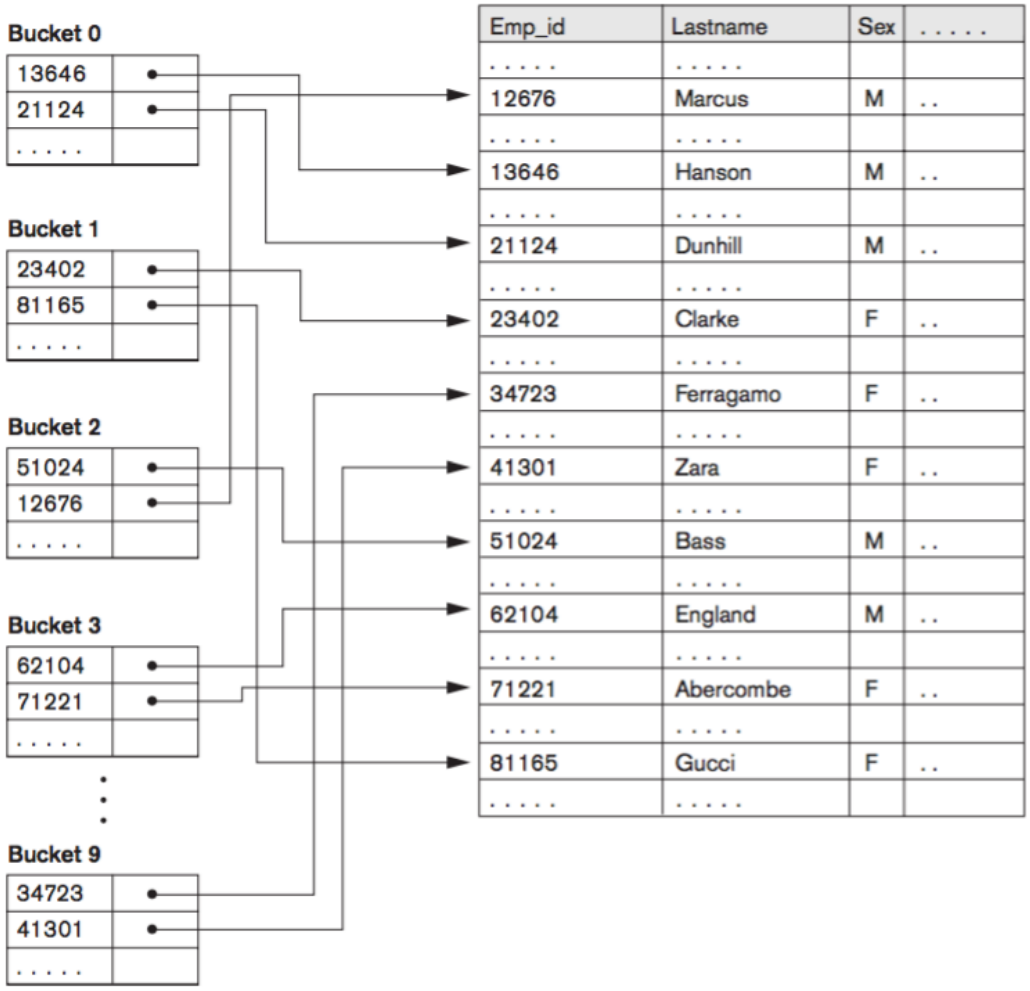


Figure 17.14
Example of a grid array on Dno and Age attributes.

OTHER TYPES OF INDEXES

Hash index

Figure 17.15
Hash-based indexing.



Bitmap indexes

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

Figure 17.16
Bitmap indexes for Sex and Zipcode.

Bitmap indexes

- Advantages:
  - Good for logical operations
  - Efficient in terms of space taken
  - Good if attributes have a high selectivity
- Drawbacks:
  - Update cost: vectors AND ids
    - ⇒Existence bitmap
- Can be used in leaf nodes of B^+-trees

Function based indexes

- Create index based on function of attribute (e.g UPPER(Lname))
- The index is used when queries are made with this function
- Enables us to introduce conditions (e.g. unique)

SOME GENERAL ISSUES CONCERNING INDEXING

Logical versus physical indexes

- Physical indexes: the pointer points to the physical location of the record on disk
  - ⇒Have to update each time the record is moved
- Logical indexes:
  - No pointer
  - Pointer replaced with primary file organization key
    - ⇒We have to search the primary file organization after the index

Index creation

- Many RDMS allow the creation of indexes
- Secondary indexes can be created dynamically by bulk loading : create all the entries and then load them, creating the other index levels simultaneously
- Primary and clustering indexes are harder to generate dynamically because the data must be ordered
- Strings are complicated to index because of their *variable length* and potential for being long

Tuning indexes

- The DB admin must consider when to create an index an what index to create:
  - Are queries execution slowly ?
  - Are indexes not used ?
  - Are indexes being updating too often ?
  - What type of index to use ?
- In some cases rebuilding the index can help performance
  - Deal with overflow/ unused space

Additional issues related to storage of relational indexes

- Using an index for managing constraints and duplicates
- Inverted files and other access methods
- Using indexing hints in queries
- Column-based storage of relations

PHYSICAL DATABASE DESIGN IN RELATIONAL DATABASES

Factors that influence physical database design

- Analyzing the database queries and transactions
- Analyzing the expected frequency of invocation of queries and transactions
- Analyzing the time constraints of queries and transactions
- Analyzing the expected frequencies of update operations
- Analyzing the uniqueness constraints on attributes

Physical database design decisions

- Whether to index an attribute
- What attribute(s) to index on
- Whether to set up a clustered index
- Whether to use a hash index over a tree index
- Whether to use dynamic hashing for the file

Conclusion

- Indexes speed access to data
- Different types of indexes for different situations
- You can’t just put indexes everywhere, you have to think
- Designing databases is a job