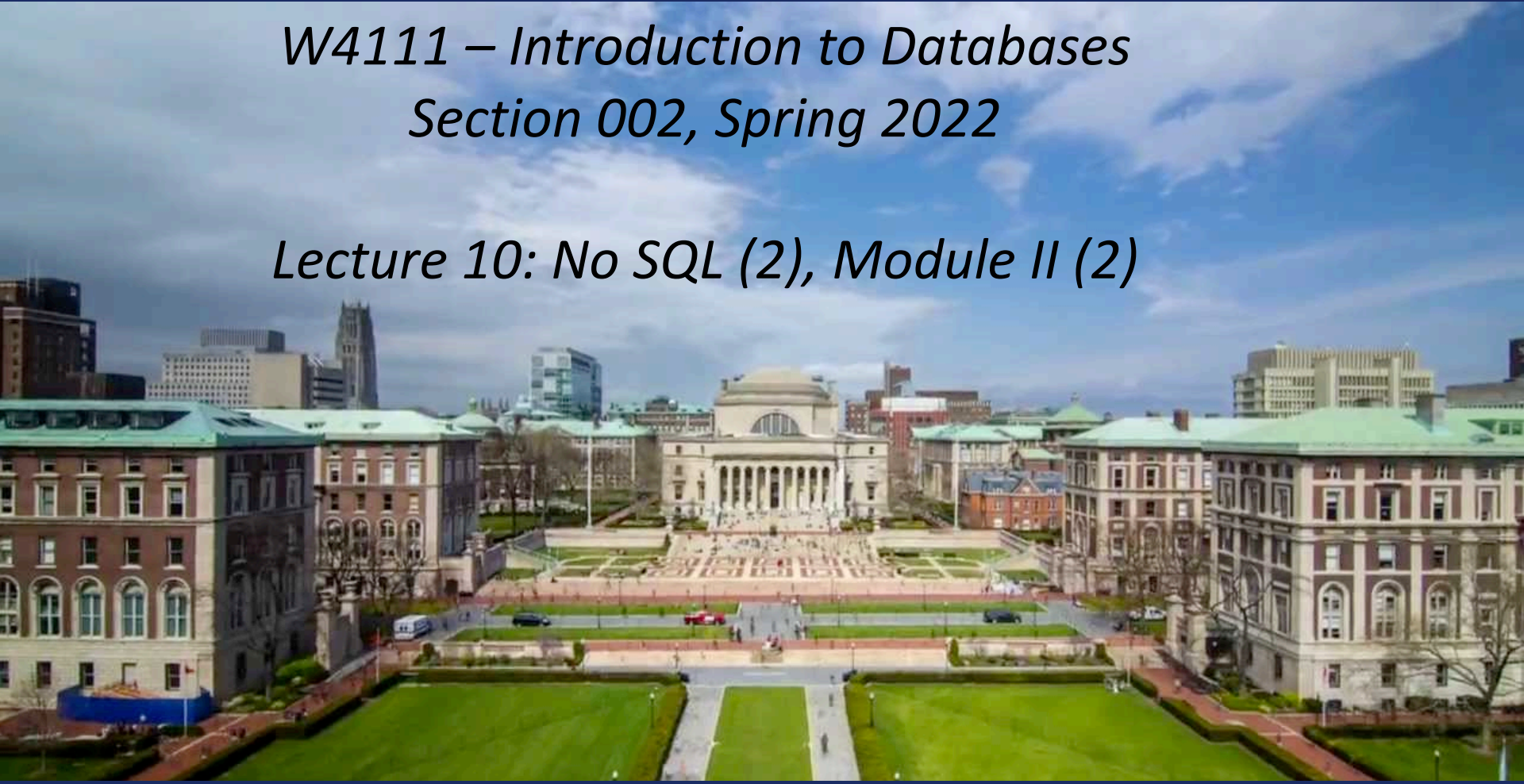
W4111 – Introduction to Databases
Section 002, Spring 2022

Lecture 10: No SQL (2), Module II (2)

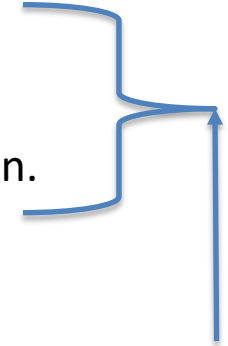


COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Contents

Today's Contents

- Homework and exams.
- Agenda update – Parallel coverage of
  - Module II: DBMS internal architecture and implementation.
  - Module III: NoSQL.
- Module II:
  - Major subsystems reminder.
  - Major subsystems summary.
  - Database disks and files.
- Module III:
  - Overview and NoSQL concepts.
  - Graph databases and Neo4j.



I am going to cover module II in chunks over multiple lectures:

- Material is on the *required* syllabus.
- Interesting and fascinating.
- But, No SQL is more broadly applicable to how students will use databases.

Homework and Exams

Homework and Exams

- The previous schedule I published was:
  - Midterm: 20-MAR to 27-MAR.
  - HW3: 27-MAR to 09-APR.
  - HW4: 09-APR to 23-APR.
  - HW5: 24-APR to 03-MAY.
  - Final: 03-MAY to 10-MAY.

Dumping a HW on you now, right after the midterm would not have been very nice.

You need a break.

- We will go to two more HWs, not 3.
  I will come up with a schedule this weekend.

Module II (2)

Course Modules – Reminder

Course Overview

Each section of W4111 is slightly different based on student interest and professor's focus. There is a common, core syllabus. Professors cover topics in different orders and grouping based on teaching style.

This section of W4111 has four modules:

- **Foundational concepts (50% of semester):** This module covers concepts like data models, relational model, relational databases and applications, schema, normalization, ... The module focuses on the relational model and relational databases. The concepts are critical and foundational for all types of databases and data centric applications.
- **Database management system architecture and implementation (10%):** This module covers the software architecture, algorithms and implementation techniques that allow databases management systems to deliver functions. Topics include memory hierarchy, storage systems, caching/buffer pools, indexes, query processing, query optimization, transaction processing, isolation and concurrency control.
- **NoSQL – "Not Only SQL" databases (20%):** This module provides motivation for "NoSQL" data models and databases, and covers examples and use cases. The module also includes cloud databases and databases-as-a-service.
- **Data Enabled Decision Support (20%):** This module covers data warehouses, data import and cleanse, OLAP, Pivot Tables, Star Schema, reporting and visualization, and provides and overview of analysis techniques, e.g. clustering, classification, analysis, mining.

Module II – DBMS Architecture and Implementation Overview and Reminder

Module II – DBMS Architecture and Implementation

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Module II – DBMS Architecture and Implementation

2. Give users the ability to *query* the data (a "query" is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.

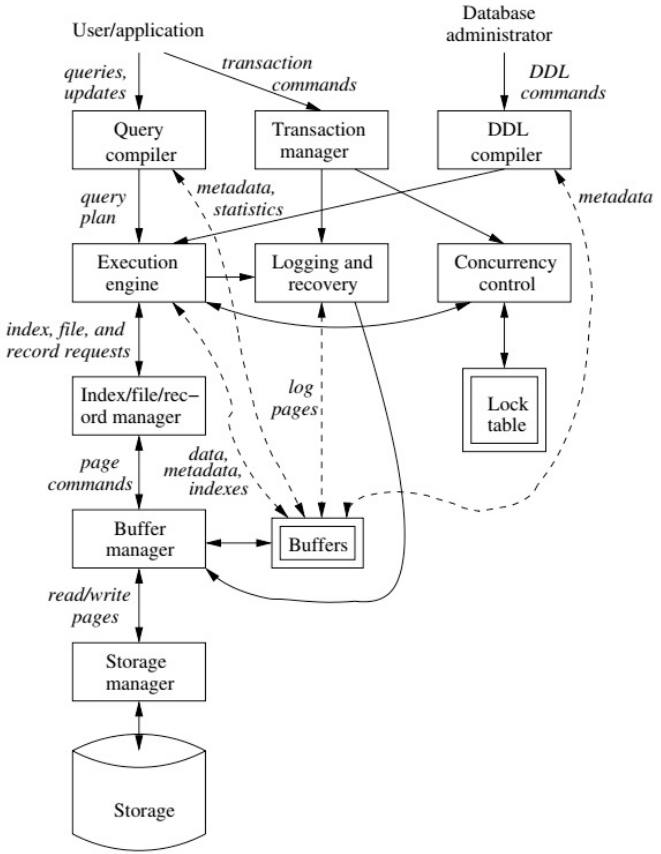
3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.

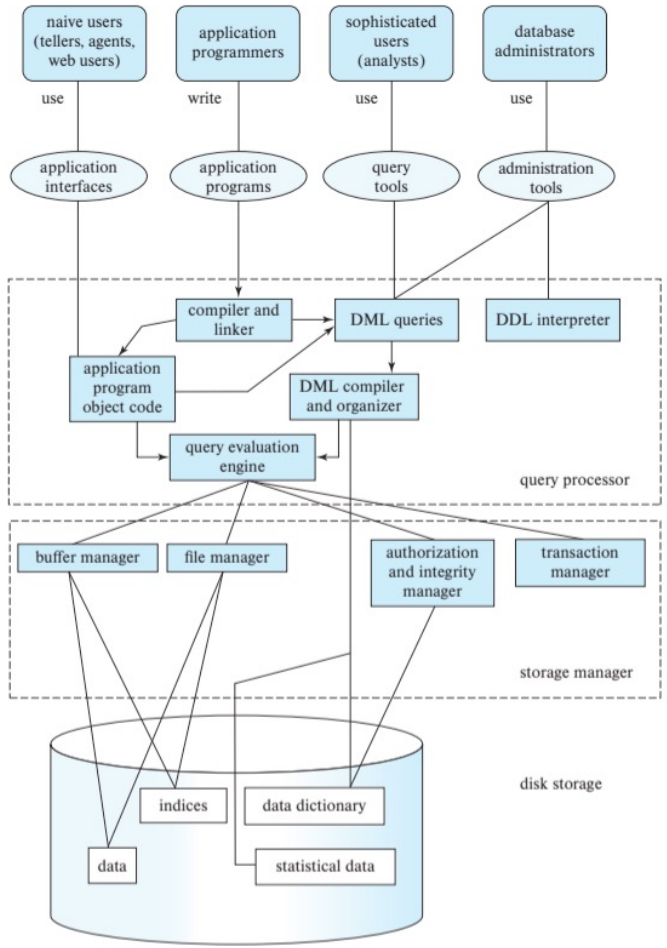
4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.

5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

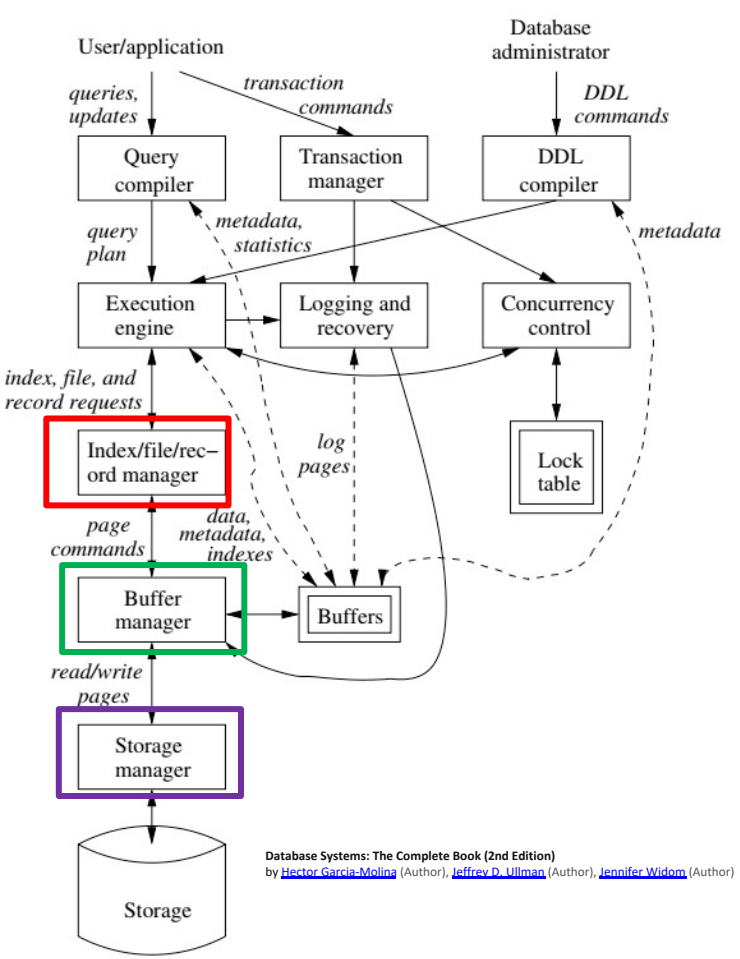
DBMS Arch.





Data Management

- Find things quickly.
- Access things quickly.
- Load/save things quickly.



Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Data Storage Structures
(Database Systems Concepts, V7, Ch. 13)
Cont

Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- **B^+-tree file organization**
  - Ordered storage even with inserts/deletes
  - More on this in Chapter 14
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
  - More on this in Chapter 14

Heap File Organization

- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated
- Important to be able to efficiently find free space within file
- **Free-space map**
  - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
  - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

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

  - Can have second-level free-space map
  - In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6

- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)

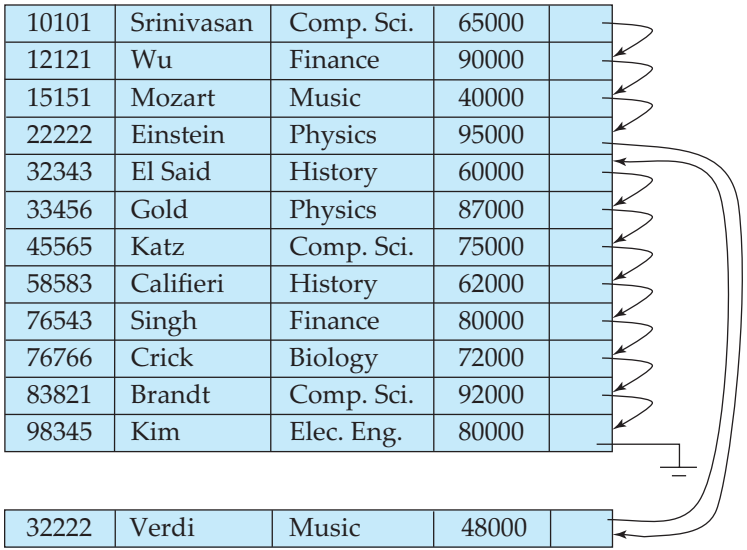
Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

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	

Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



Partitioning

- **Table partitioning**: Records in a relation can be partitioned into smaller relations that are stored separately
- E.g., *transaction* relation may be partitioned into *transaction_2018, transaction_2019, etc.*
- Queries written on *transaction* must access records in all partitions
  - Unless query has a selection such as *year*=2019, in which case only one partition in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk

Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

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

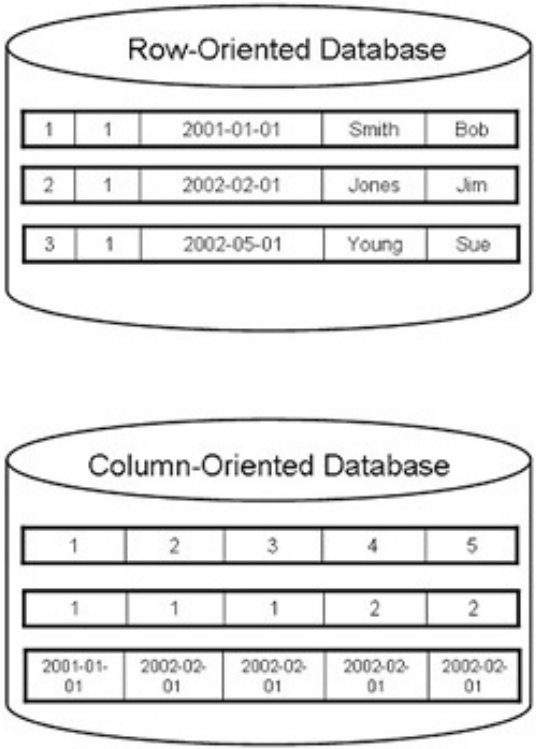
Columnar Representation

- Benefits:
  - Reduced IO if only some attributes are accessed
  - Improved CPU cache performance
  - Improved compression
  - **Vector processing** on modern CPU architectures
- Drawbacks
  - Cost of tuple reconstruction from columnar representation
  - Cost of tuple deletion and update
  - Cost of decompression
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
  - Called **hybrid row/column stores**

Row vs Column

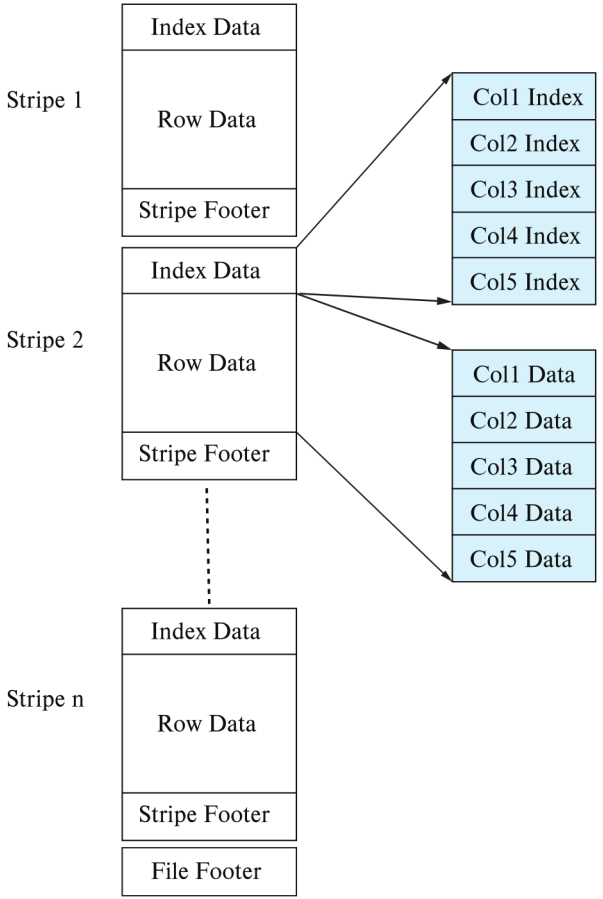
- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for diffe types of query, e.g.
  - Columnar is extremely powerful for B
    - Aggregation ops, e.g. SUM, AVG
    - PROJECT (do not load all of the row) t
  - Row is powerful for OLTP. Transaction typically create and retrieve
    - One row at a time
    - All the columns of a single row.

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura



Columnar File Representation

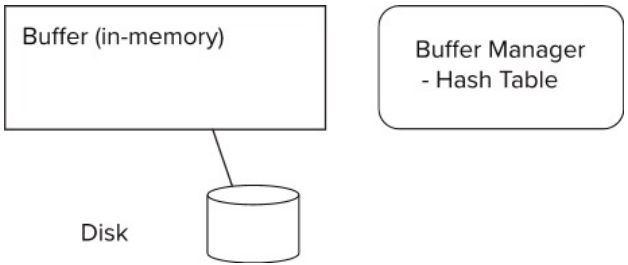
- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:



Memory and Buffer Pools

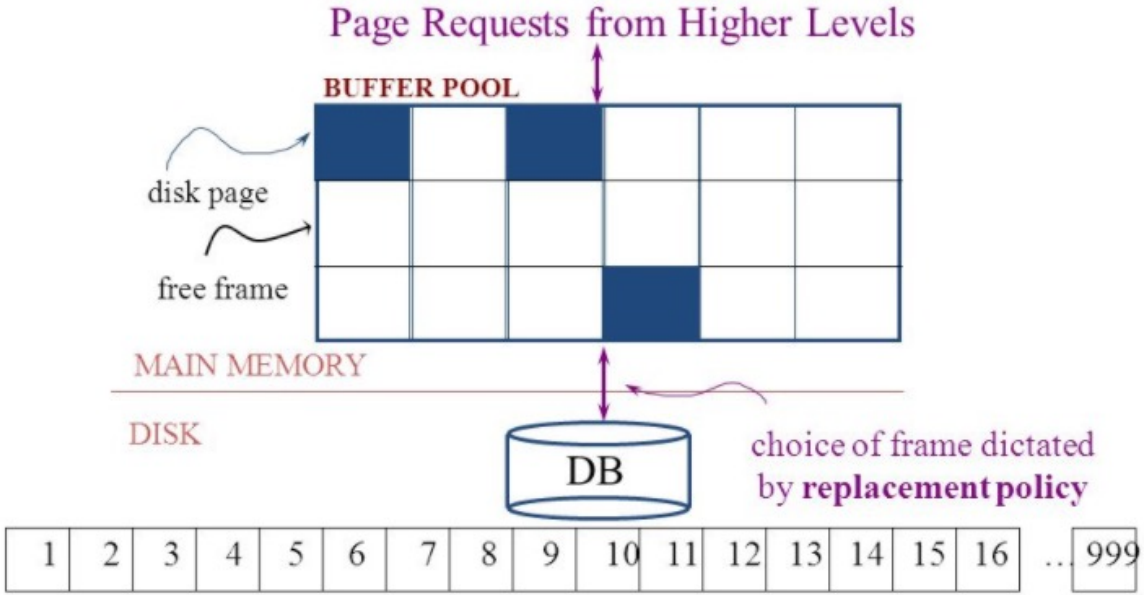
Storage Access

- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



The Logical Concept

- The DBMS and queries can only manipulate in-memory blocks and records.
- A very, very, very small fraction of all blocks fit in memory.



Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  - If the block is already in the buffer, buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
  - **Pin** done before reading/writing data from a block
  - **Unpin** done when read /write is complete
  - Multiple concurrent pin/unpin operations possible
    - Keep a pin count, buffer block can be evicted only if pin count = 0
- **Shared and exclusive locks on buffer**
  - Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
  - Readers get shared lock, updates to a block require exclusive lock
  - **Locking rules:**
    - Only one process can get exclusive lock at a time
    - Shared lock cannot be concurrently with exclusive lock
    - Multiple processes may be given shared lock concurrently

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
  - Idea behind LRU – use past pattern of block references as a predictor of future references
  - LRU can be bad for some queries
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loops

```
for each tuple tr of r do
  for each tuple ts of s do
    if the tuples tr and ts match …
```

Buffer-Replacement Policies (Cont.)

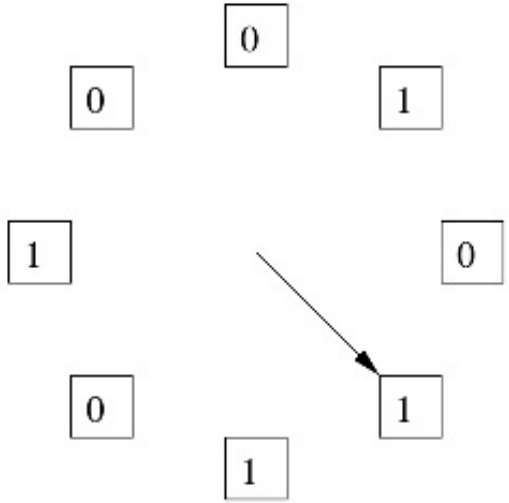
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager may reorder writes
  - Can lead to corruption of data structures on disk
    - E.g., linked list of blocks with missing block on disk
    - File systems perform consistency check to detect such situations
  - Careful ordering of writes can avoid many such problems

Replacement Policy

- The *replacement policy* is one of the most important factors in database management system implementation and configuration.
- A very simple, introductory explanation is (https://en.wikipedia.org/wiki/Cache_replacement_policies).
  - There are a lot of possible policies.
  - The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm/simply optimal replacement policy or the clairvoyant algorithm.
- All implementable policies are an attempt to approximate knowledge of the future based on knowledge of the past.
- Least Recently Used is based on the simplest assumption
  - The information that will not be needed for the longest time.
  - Is the information that has not been accessed for the longest time.

The “Clock Algorithm”

- LRU is (perceived to be) expensive
  - Maintain timestamp for each block.
  - Update and resort blocks on access.
- The “Clock Algorithm” is a less expensive approximation.
  - Arrange the frames (places blocks can go) into a logical circle like the seconds on a clock face.
  - Each frame is marked 0 or 1.
    - Set to 1 when block added to frame.
    - Or when application accesses a block in frame.
  - Replacement choice
    - Sweep second hand clockwise one frame at a time.
    - If bit is 0, choose for replacement.
    - If bit is 1, set bit to zero and go to next frame.
- The basic idea is. On a clock face
  - If the second hand is currently at 27 seconds.
  - The 28 second tick mark is “the least recently touched mark.”



Replacement Algorithm

The algorithms are more sophisticated in the real world, e.g.

- "Scans" are common, e.g. go through a large query result in order (will be more clear when discussing cursors).
  - The engine knows the current position in the result set.
  - Uses the sort order to determine which records will be accessed soon.
  - Tags those blocks as not replaceable.
  - (A form of clairvoyance).
- Not all users/applications are equally "important."
  - Classify users/applications into priority 1, 2 and 3.
  - Sub-allocate the buffer pool into pools P1, P2 and P3.
  - Apply LRU within pools and adjust pool sizes based on relative importance.
  - This prevents
    - A high access rate, low-priority application from taking up a lot of frames
    - Result in low access, high priority applications not getting buffer hits.

Optimization of Disk Block Access (Cont.)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in Chapter 19)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM or flash buffer immediately
  - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
  - Used exactly like nonvolatile RAM
    - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NV-RAM or log disk
  - Reordering without journaling: risk of corruption of file system data

Indexes

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to 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

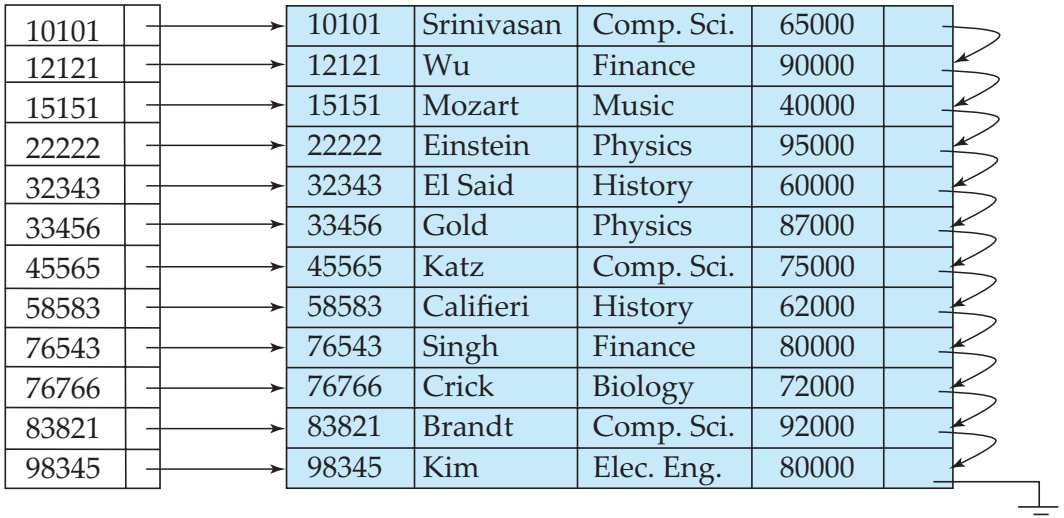
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index.**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.

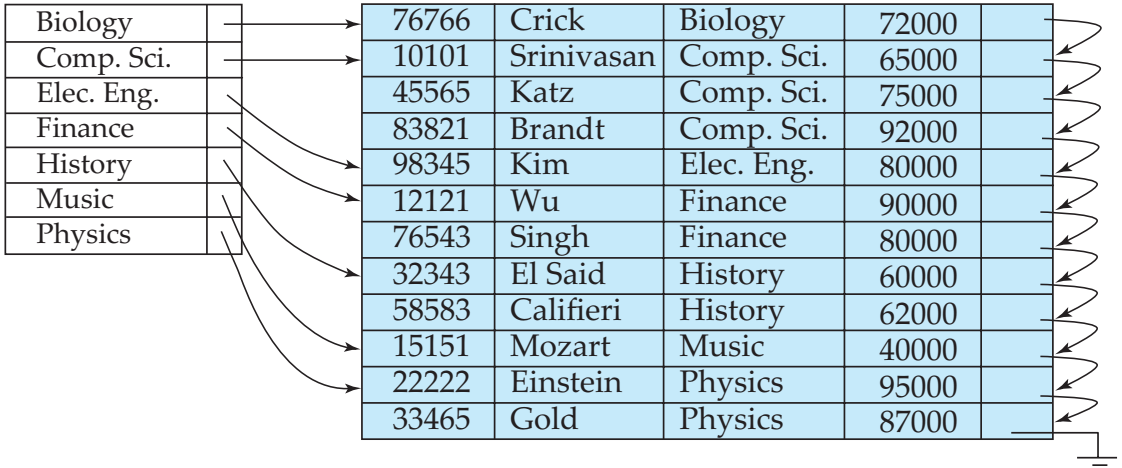
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



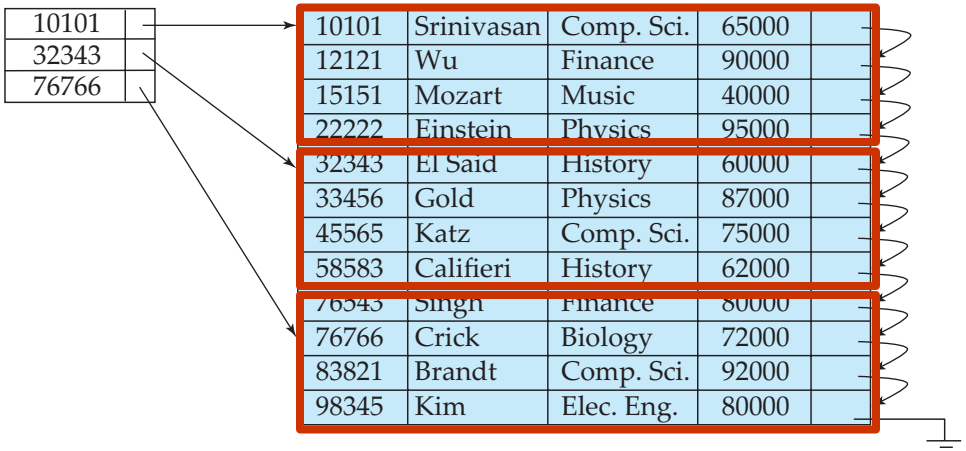
Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



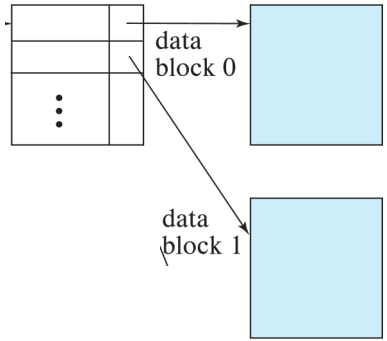
Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



Sparse Index Files (Cont.)

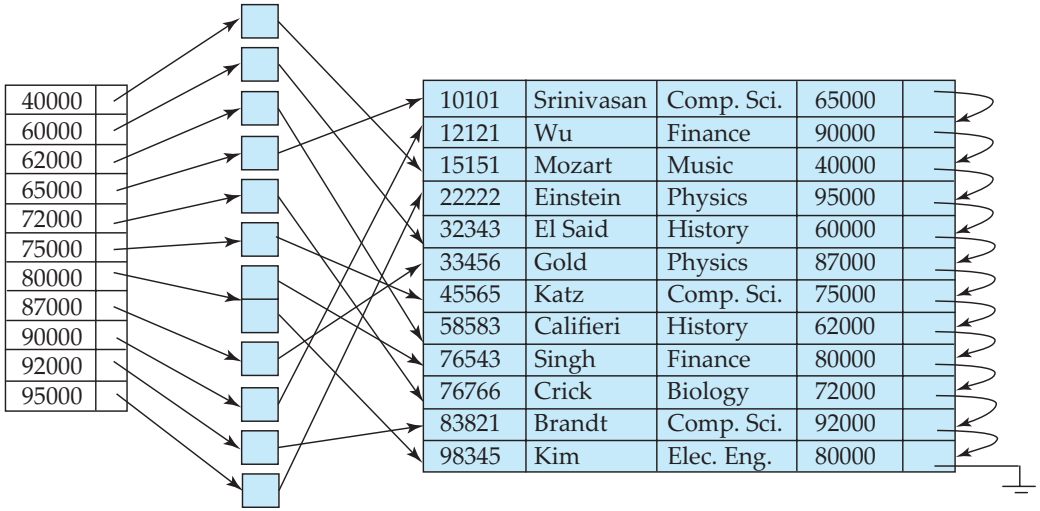
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**:
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



  - For unclustered index: sparse index on top of dense index (multilevel index)

Secondary Indices Example

- Secondary index on salary field of instructor



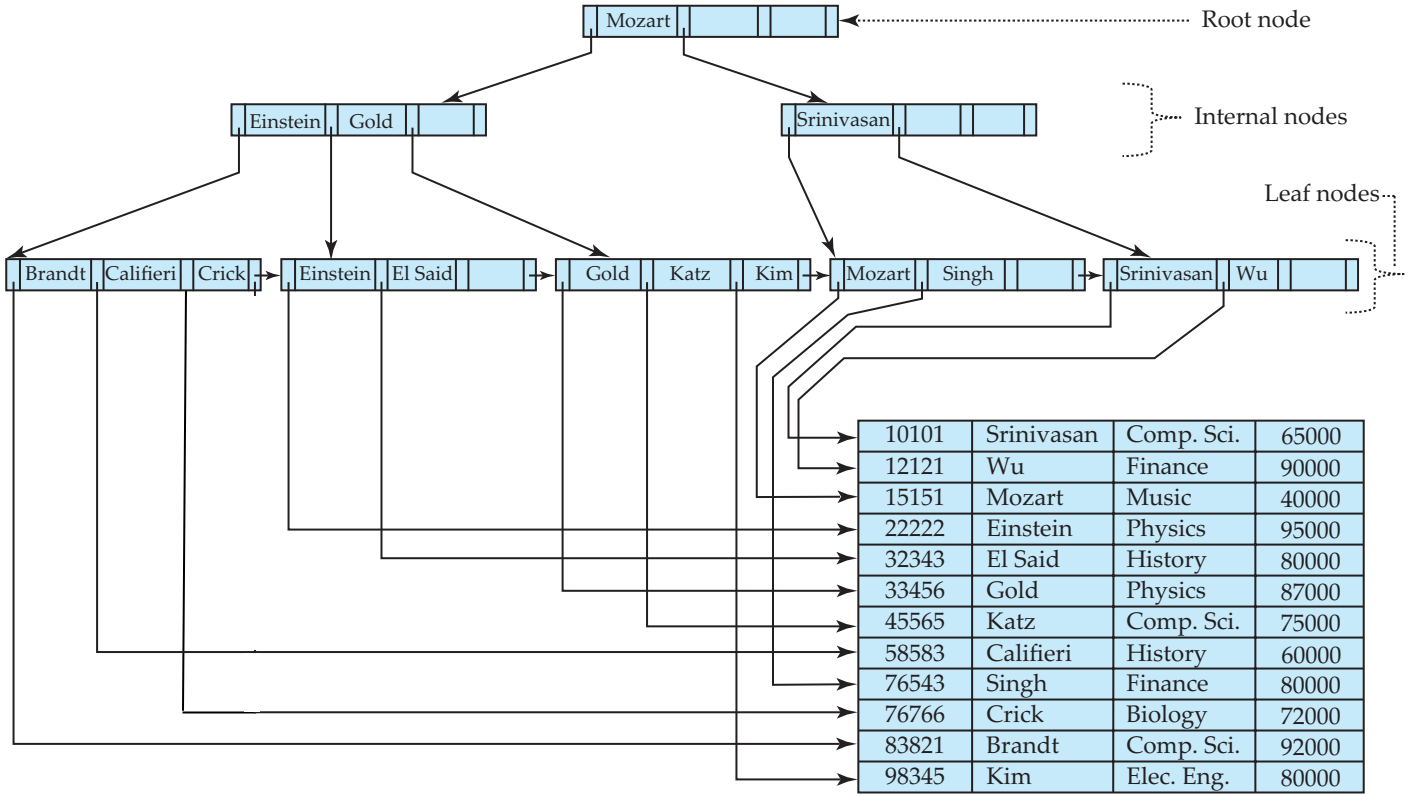
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Indices on Multiple Keys

- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name, ID*)
  - Values are sorted lexicographically
    - E.g. (John, 12121) < (John, 13514) and (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name, ID*)

- (nameLast, nameFirst, birthyear)
  - nameLast [nameLast = “Ferguson”] [nameLast like “Fer%”]
  - nameLast, nameFirst
  - nameLast, nameFirst, birthyear
- NOT and index on
  - nameFirst, nameLast
  - birthyear
  - nameLast like [%er%]

Example of B+-Tree



B+-Tree Index Files (Cont.)

A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-Tree Node Structure

- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

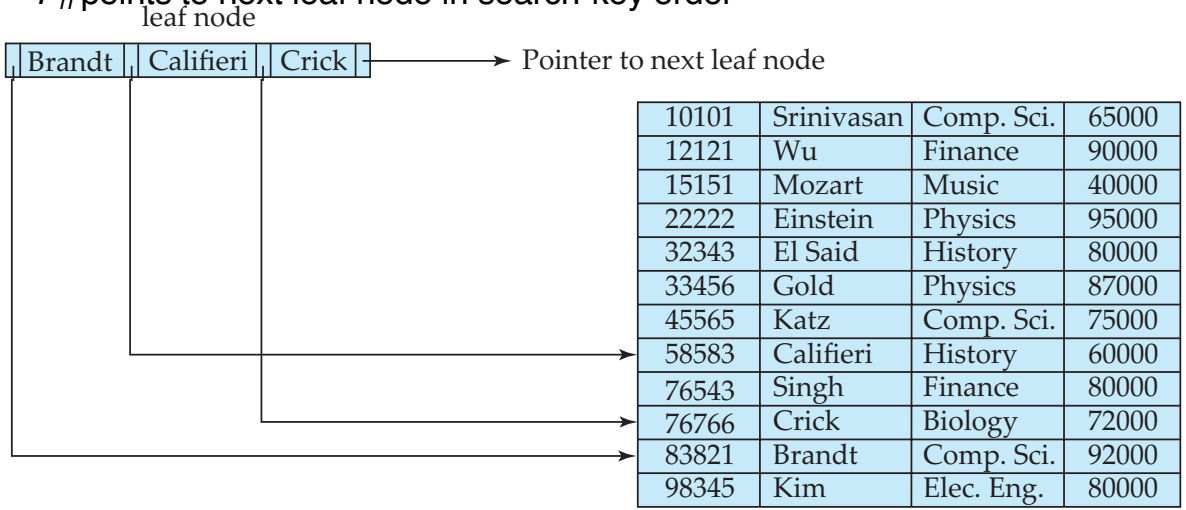
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



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

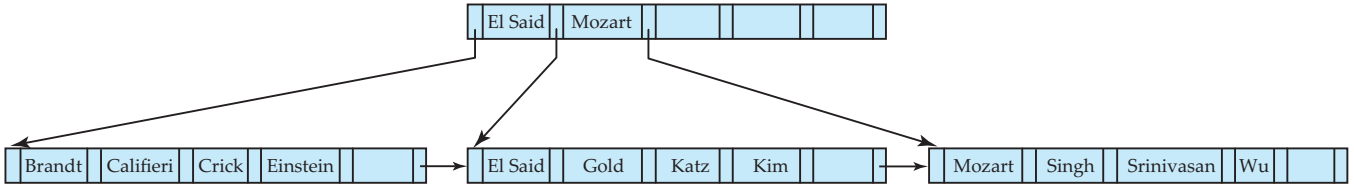
Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
  - General structure

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Example of B+-tree

- B+-tree for *instructor* file ($n = 6$)



- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.

Observations about B^+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2* \lceil n/2 \rceil$ values
  - Next level has at least $2* \lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Show the Simulator

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

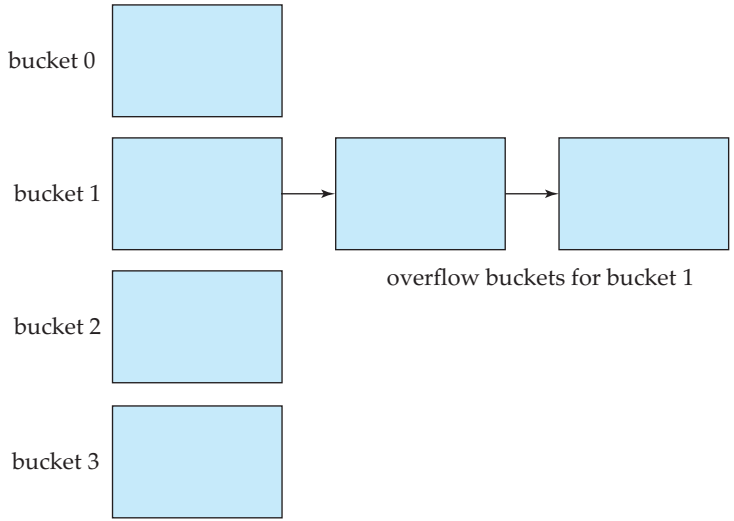
Hashing

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 (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.



Example of Hash File Organization

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

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

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.

Show the Simulator

http://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html
https://opendsa-server.cs.vt.edu/ODSA/AV/Development/hashAV.html

nameLast="Ferguson"
nameLast >= "Ferguson" and nameLast <= "Guthrie"

Select * from professors join students using (uni)

O(N)*O(M) → O(N*M)

O(N)+O(M)+O(1)*O(N) --> O(N+M)

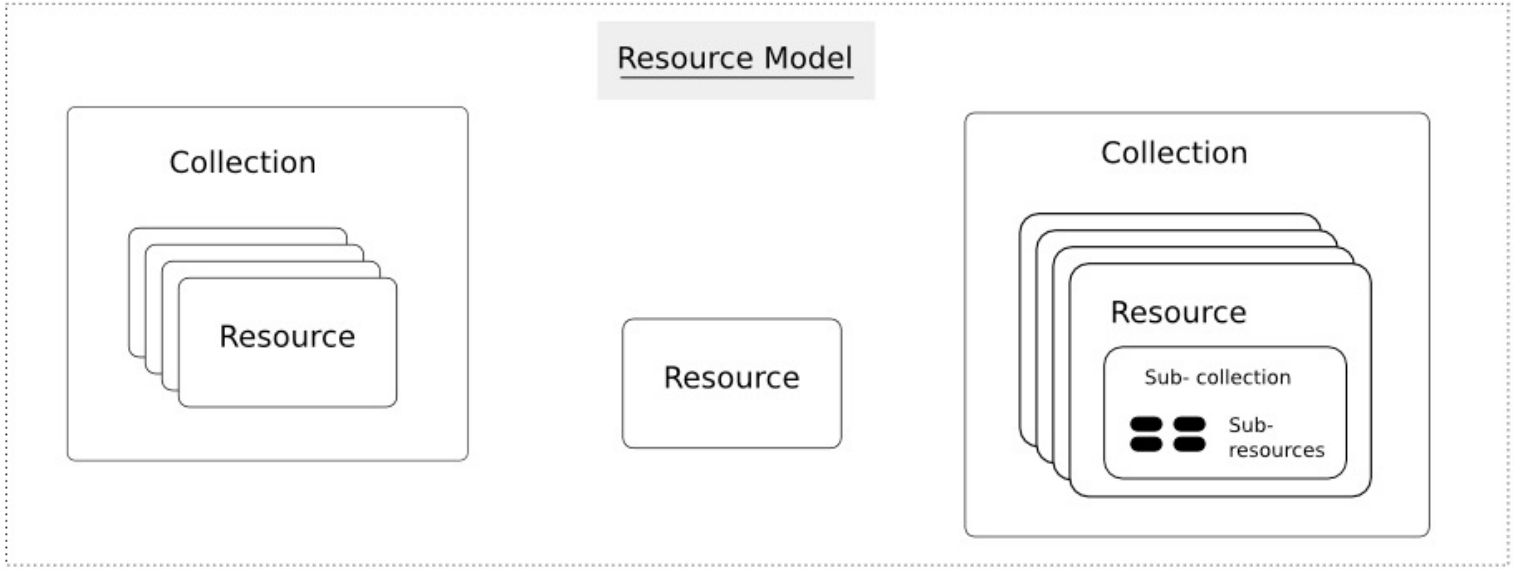
Data Models and REST

Data Modeling Concepts and REST

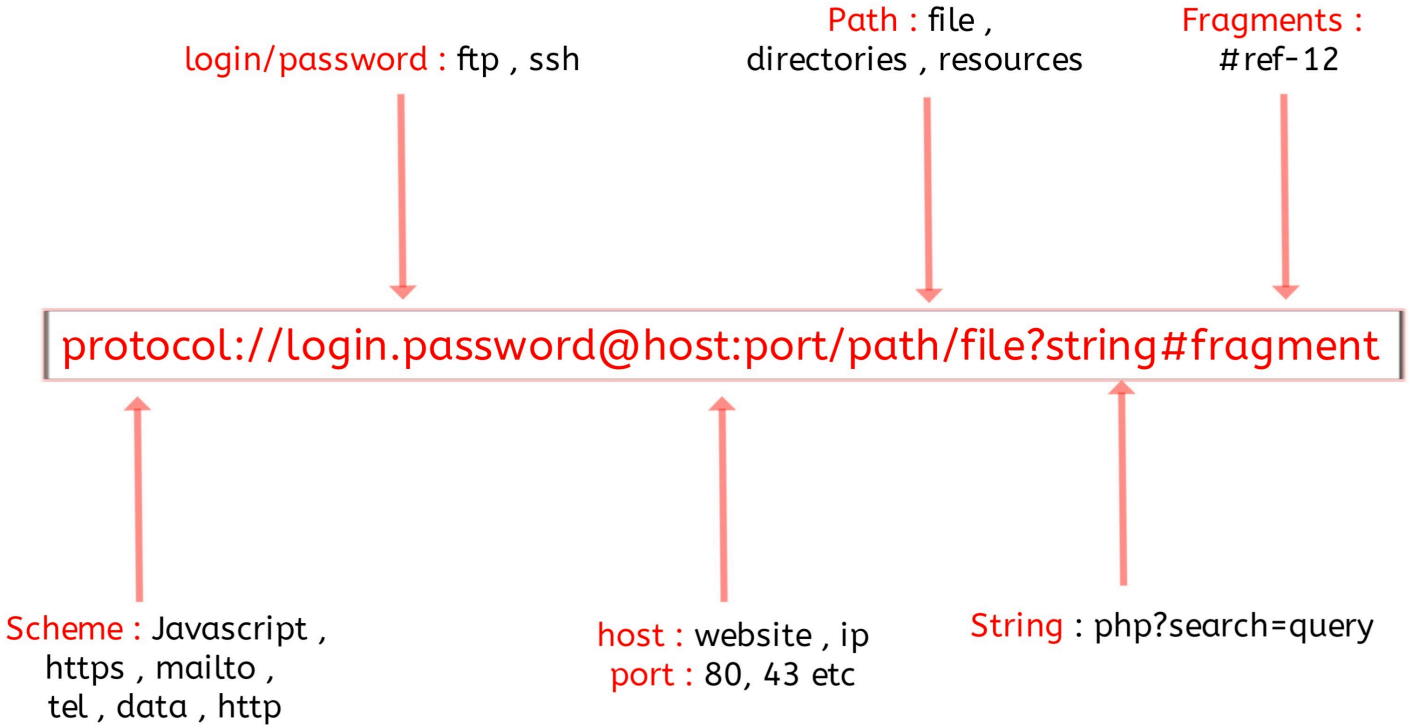
Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

REST and Resources



URLs



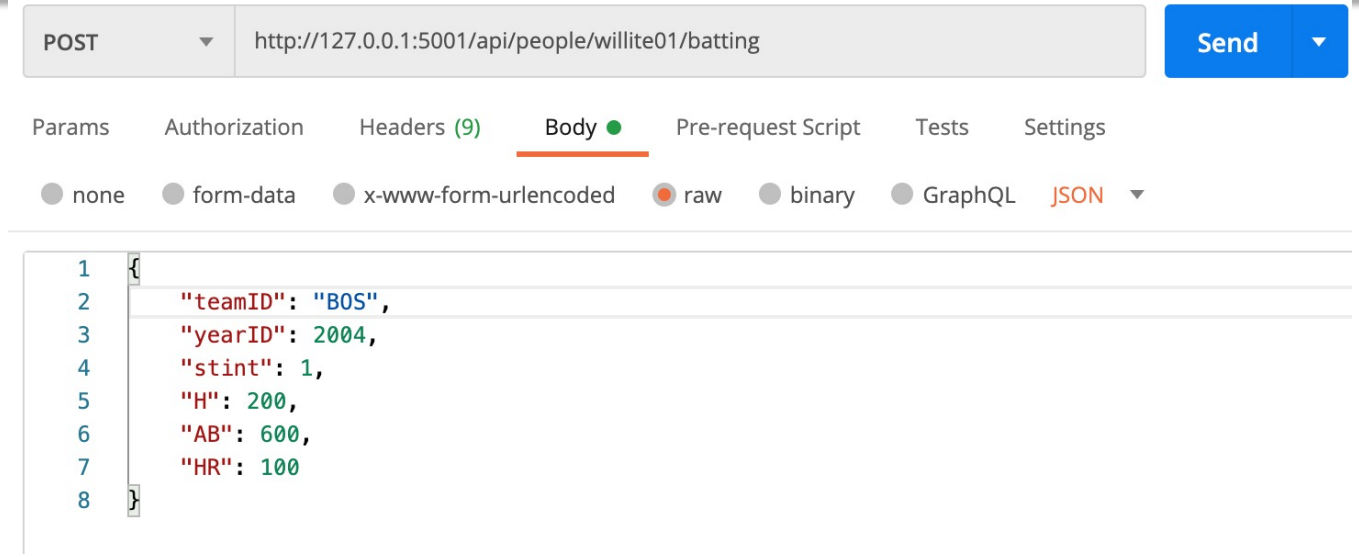
Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...; SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21	
GET	/people/21/batting/2004_1	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21 AND yearID=2004 AND stint=1	

PUT, DELETE, UPDATE

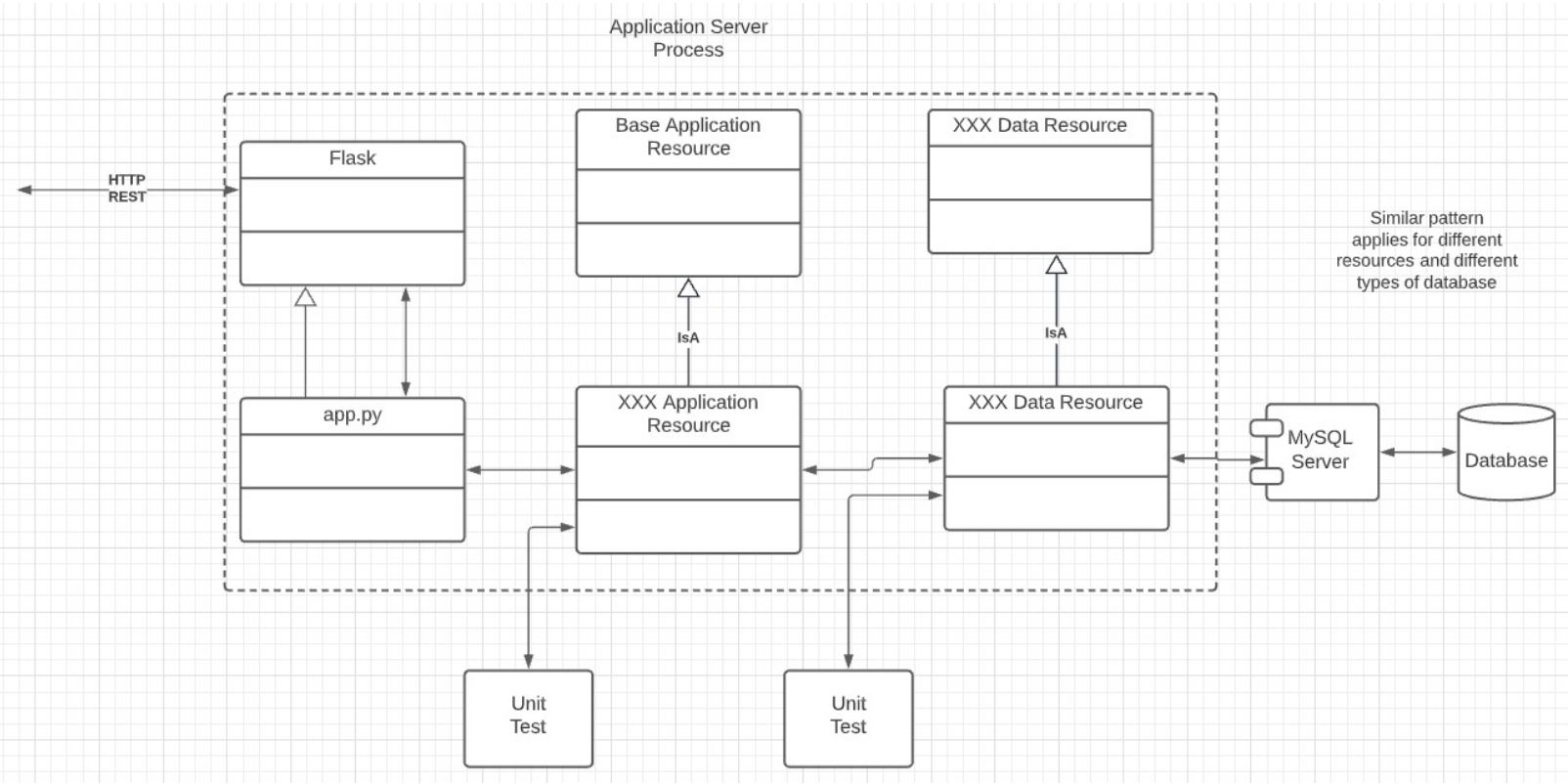
- /people?
  - POST (INSERT)
  - GET (SELECT ... WHERE ...)
- /people/21
  - WHERE peopleID=21
  - DELETE → DELETE WHERE
  - PUT → UPDATE SET ..... WHERE
  - GET SELECT ... WHERE

Simplistic, Conceptual Mapping (Examples)

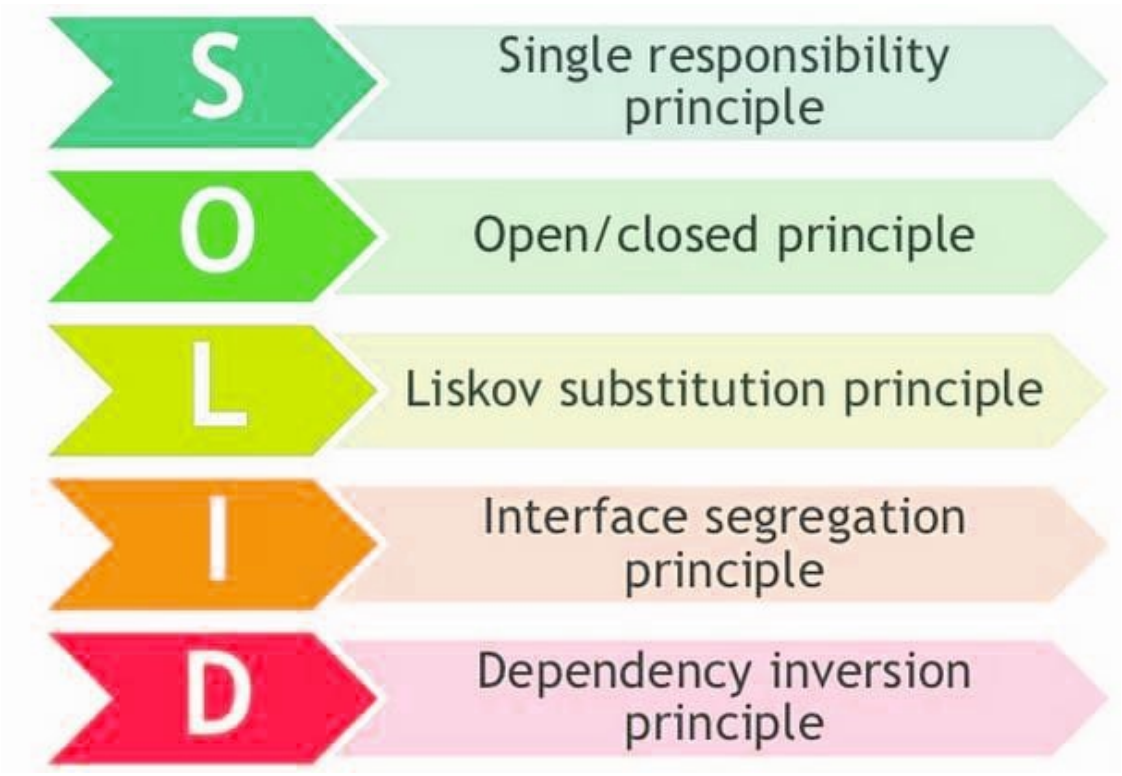



```
INSERT INTO
    batting(playerID, teamID, yearID, stint, H, AB, HR)
    VALUES ("willite01", "BOS", 2004, 1, 200, 600, 100)
```

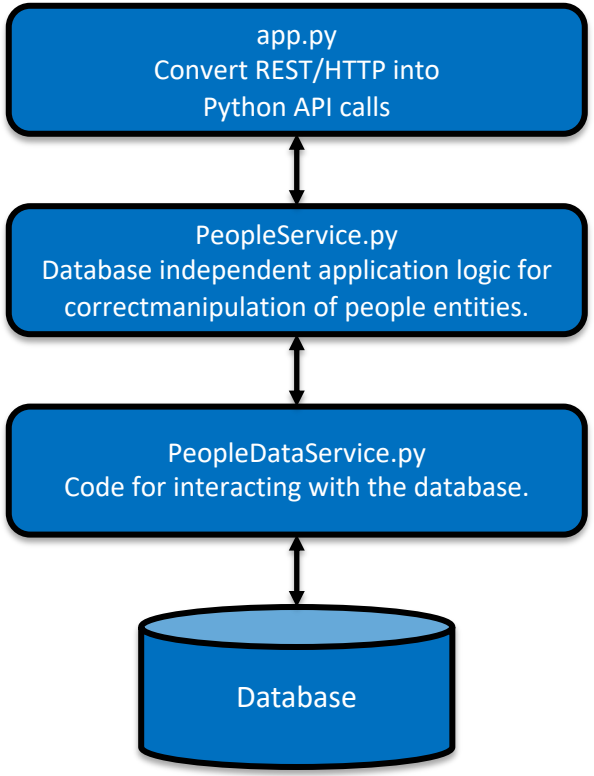
REST Application Structure



SOLID (SW) Design Principle



Single Responsibility

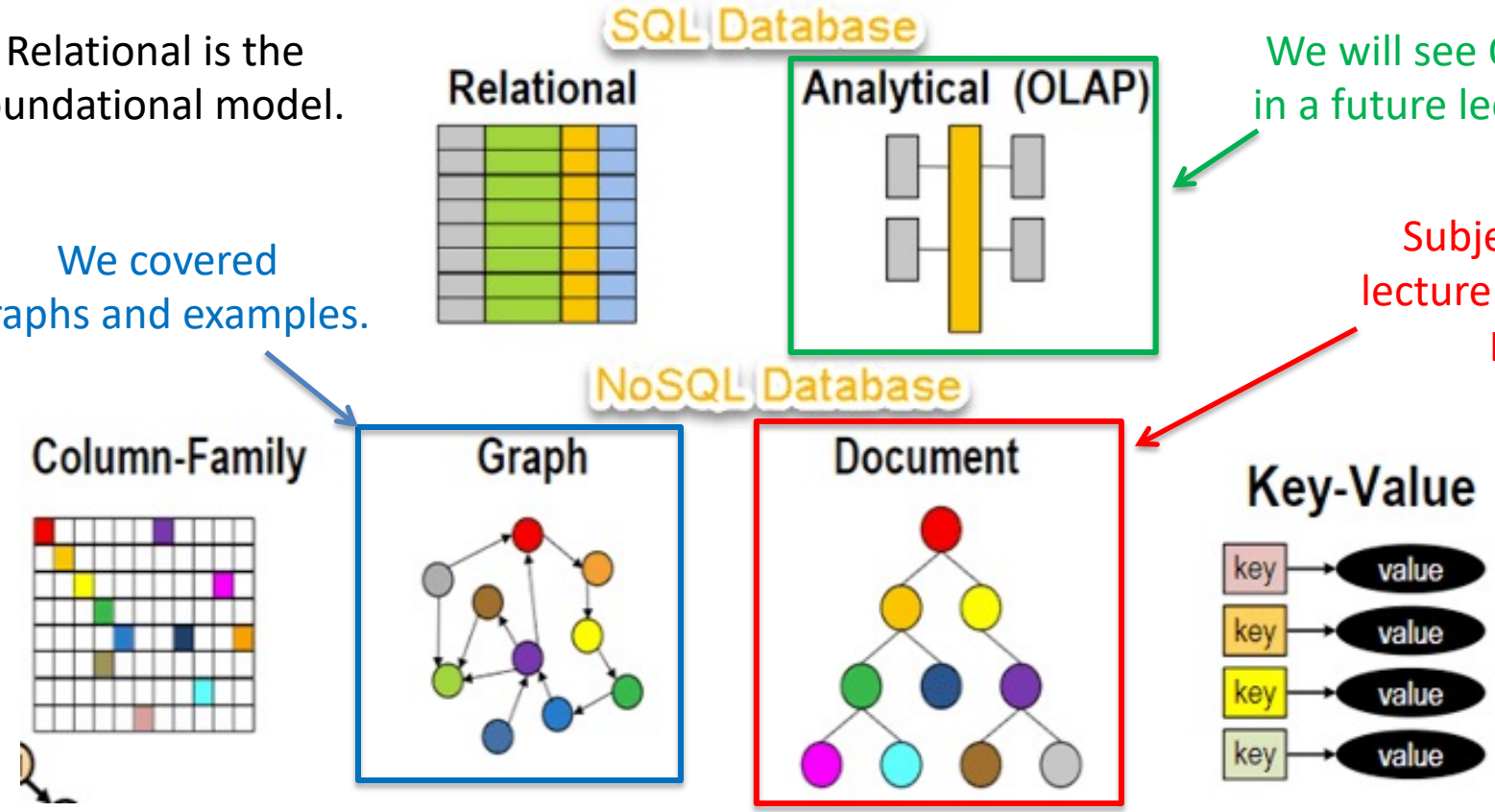


NoSQL

One Taxonomy

Simplistic Classification (https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b)

Relational is the foundational model.



We will see OLAP in a future lecture.

We covered graphs and examples.

Subject of this lecture and part of HW4

Document Databases
MongoDB

MongoDB Overview

Copied from online sources.
If you get to use Google/StackOverflow for code,
I get to use Google/SlideShare for presentations.

MongoDB Concepts

Switch to notebook.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server, Client, Tools, Packages	
mysqld/Oracle	mongod
mysql/sqlplus	mongo
DataGrip	Compass
pymysql	pymongo

Core Operations

Basic Operations:

- Create database
- Create collection
- Create-Retrieve-Update-Delete (CRUD):
  - Create: insert()
  - Retrieve:
    - find()
    - find_one()
  - Update: update()
  - Delete: remove()

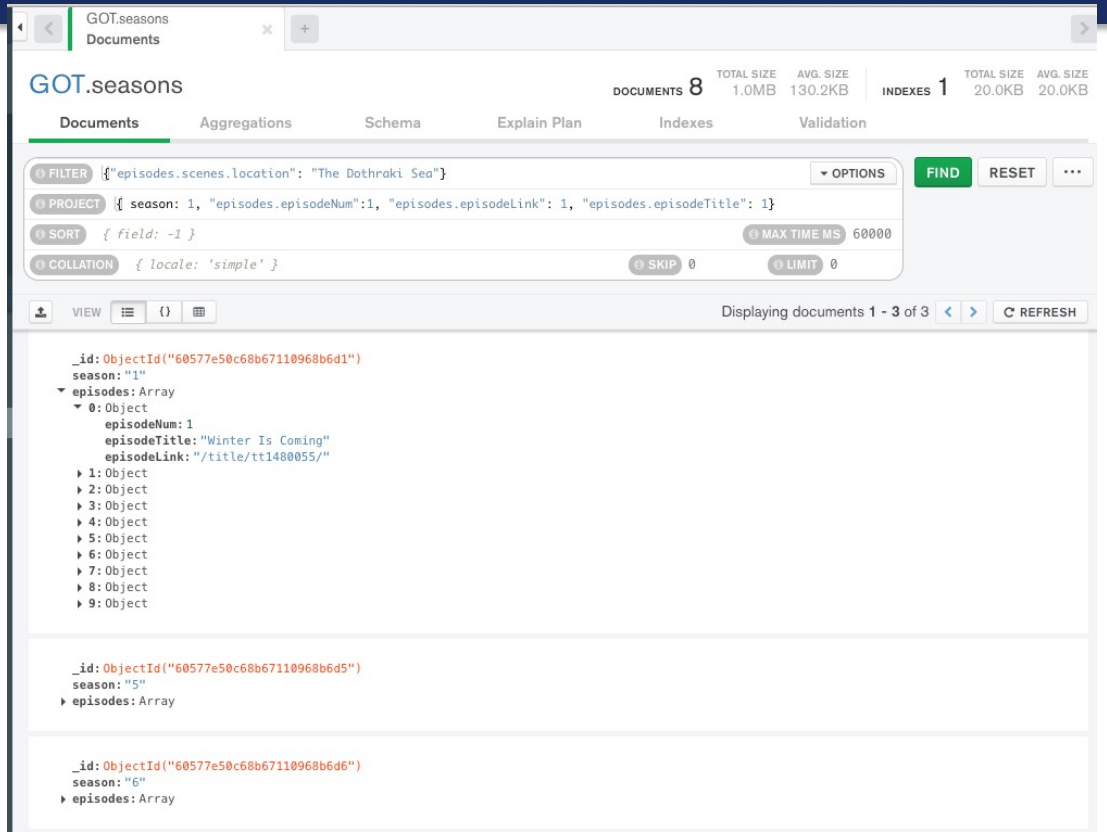
More Advanced Concepts:

- Limit
- Sort
- Aggregation Pipelines
  - Merge
  - Union
  - Lookup
  - Match
  - Merge
  - Sample
  - … …

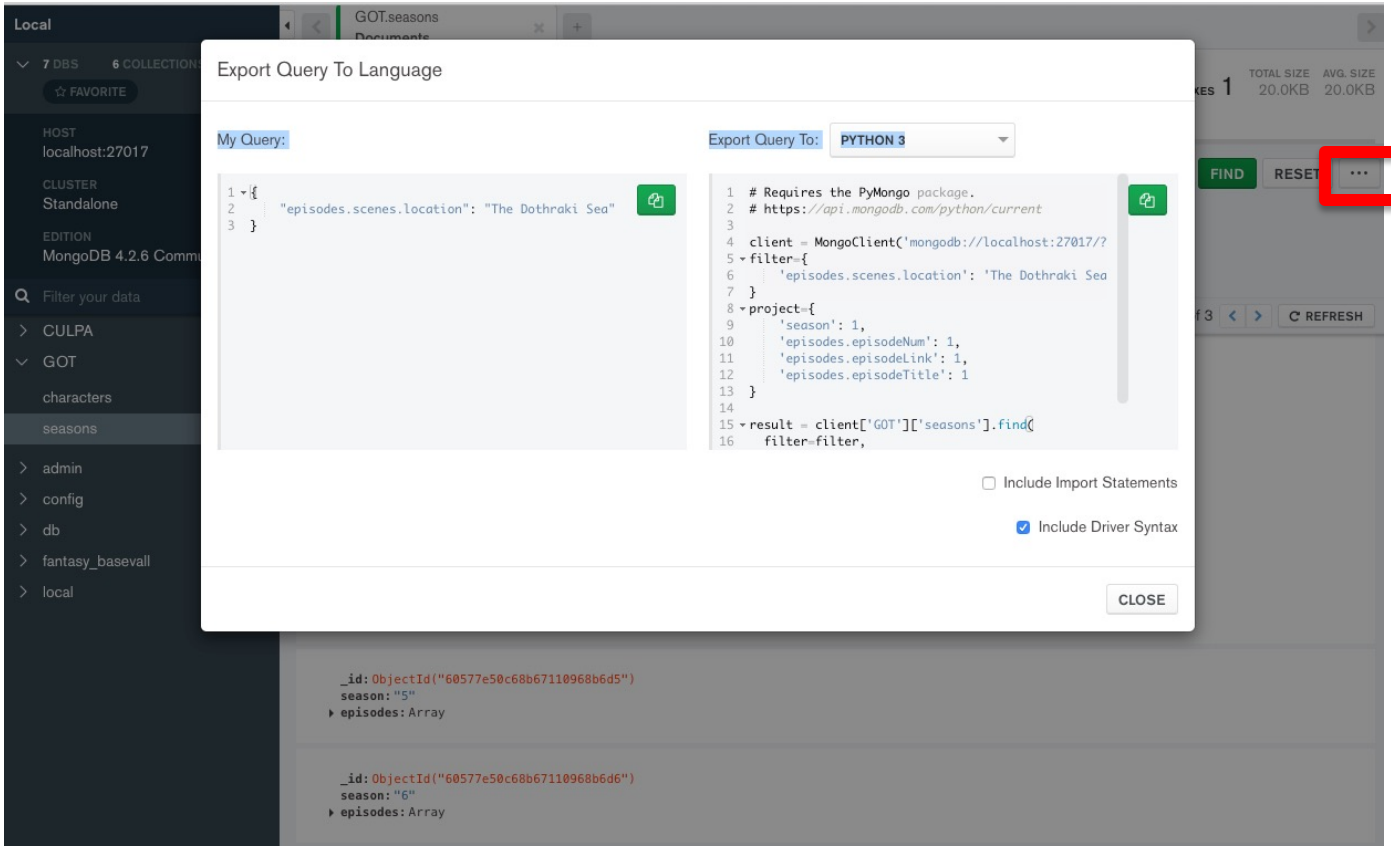
We will just cover the basics for now and may cover more things in HW or other lectures.

find()

- Note:
  - MongoDB use a more pymysql approach, e.g. an API, than pure declarative languages like SQL.
  - The parameters for find() are where the declarative language appears.
- The basic forms of find() and find_one() have two parameters:
  - *filter expression*
  - *Project expression*
- You can use the Compass tool and screen captures for some HW and exam answers.
- What if I want the answer in a Jupyter Notebook?



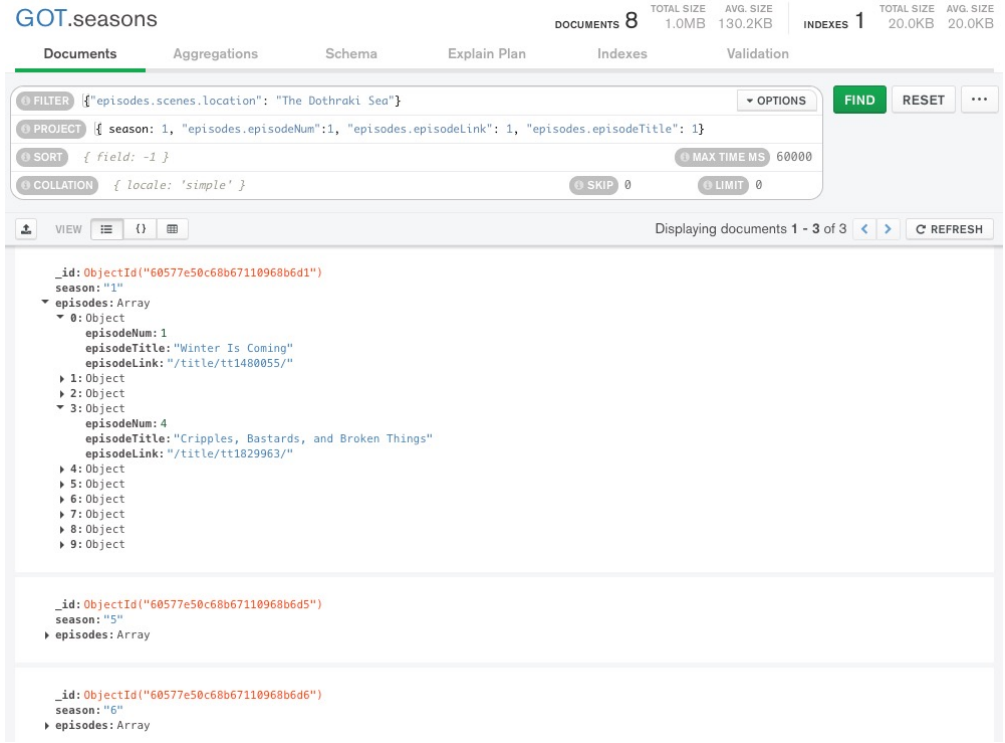
Generate Code



- Choose Export to Language.
- Copy into the notebook.
- The export has an option to include all the connection setup, choosing DB, ...

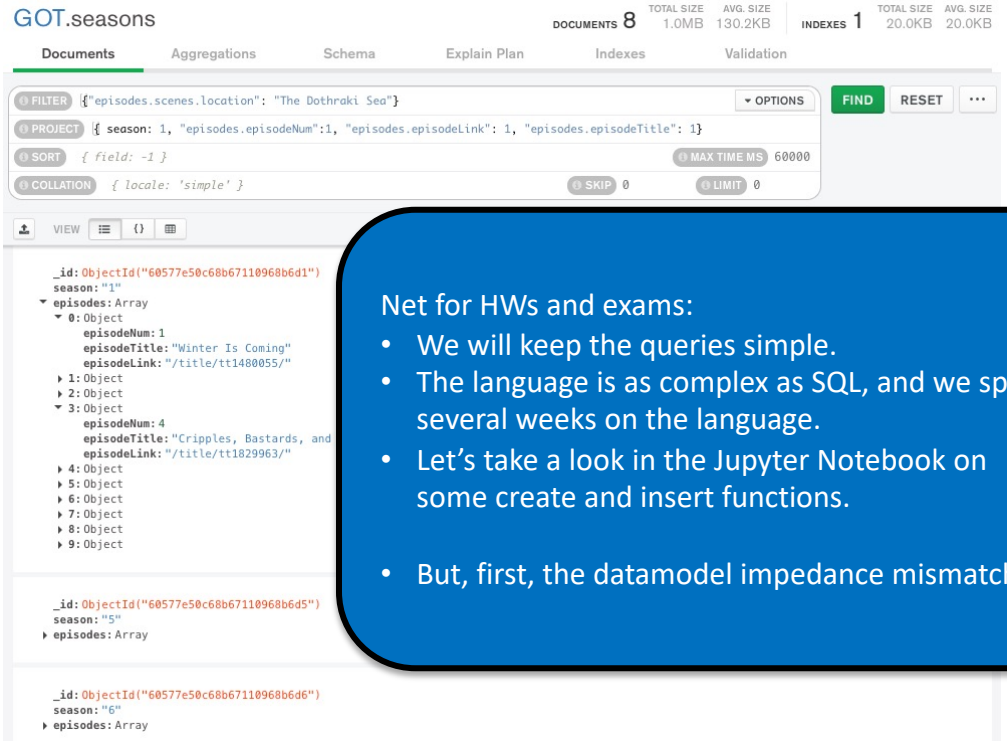
- Switch to Notebook

Result is not Quite You Expect



- The query returns documents that match.
  - The document is “Large” and has and episodes and seasons.
  - If you do a $project requesting episodes/episode content,
    - You get all episodes in the documents that match.
    - Not just the episodes with the scene/location.
    - Projecting array elements from arrays whose elements are arrays is complex and baffling.
- You also get back something (a cursor) that is iterable.

Result is not Quite You Expect



Net for HWs and exams:

- We will keep the queries simple.
- The language is as complex as SQL, and we spent several weeks on the language.
- Let's take a look in the Jupyter Notebook on some create and insert functions.
- But, first, the datamodel impedance mismatch concept.

- The query returns documents that match.
  - The document is "Large" and has d episodes and seasons.
  - u do a $project requesting odes/episode content,
    - You get all episodes in the documents that match.
    - Not just the episodes with the scene/location.
    - Projecting array elements from arrays whose elements are arrays is complex and baffling.
- You also get back something (a cursor) that is iterable.

Datamodel Impedance Match/Mismatch

Document Format:


```
{
  "orderNumber": 10100,
  "customerNumber": 363,
  "orderDate": "2003-01-06",
  "requiredDate": "2003-01-13",
  "shippedDate": "2003-01-10",
  "status": "Shipped",
  "orderDetails": [
   {
    "orderLineNumber": 1,
    "productCode": "S24_3969",
    "quantityOrdered": 49,
    "priceEach": "35.29"
   },
   {
    "orderLineNumber": 2,
    "productCode": "S18_2248",
    "quantityOrdered": 50,
    "priceEach": "55.09"
   },
   {
    "orderLineNumber": 3,
    "productCode": "S18_1749",
    "quantityOrdered": 30,
    "priceEach": "136.00"
   },
   {
    "orderLineNumber": 4,
    "productCode": "S18_4409",
    "quantityOrdered": 22,
    "priceEach": "75.46"
   }
  ]
}
```


orderNumber	customerNumber	orderDate	requiredDate	shippedDate	status	productCode	quantityOrder...	priceEach	orderLineNumber
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S24_3969	49	35.29	1
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S18_2248	50	55.09	2
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S18_1749	30	136.00	3
10100	363	2003-01-06	2003-01-13	2003-01-10	Shipped	S18_4409	22	75.46	4

More Natural Mapping

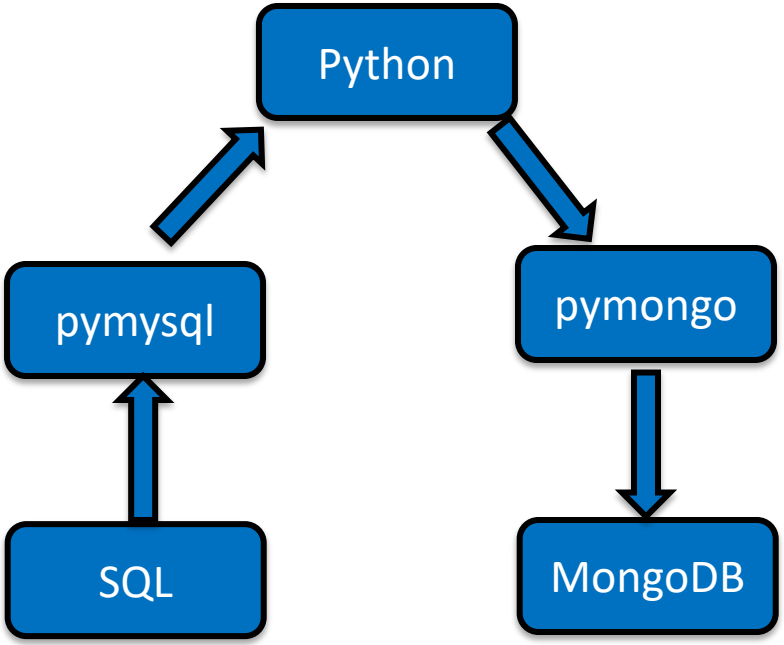
Impedance mismatch is the term used to refer to the problems that occurs due to differences between the database model and the programming language model. The practical relational model has 3 components these are:

1. Attributes and their data types
2. Tuples
3. Tables/collections/sets
4. Relationships

More Fun – Data Types

MongoDB supports many datatypes. Some of them are −

- **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** − This type is used to store arrays or list or multiple values into one key.
- **Timestamp** − ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.
- **Code** − This datatype is used to store JavaScript code into the document.
- **Regular expression** − This datatype is used to store regular expression.



(Some) MongoDB CRUD Operations

- Create:
  - db.collection.insertOne()
  - db.collection.insertMany()
- Retrieve:
  - db.collection.find()
  - db.collection.findOne()
  - db.collection.findOneAndUpdate()
  - … ..
- Update:
  - db.collection.updateOne()
  - db.collection.updateMany()
  - db.collection.replaceOne()
- Delete:
  - db.collection.deleteOne()
  - db.collection.deleteMany()

pymongo maps the camel case to _, e.g.

- findOne()
- find_one()

There are good online tutorials:

- https://www.tutorialspoint.com/python_data_access
- https://www.tutorialspoint.com/mongodb/index.htm

(Some) MongdoDB Pipeline Operators

https://www.slideshare.net/mongodb/s01-e04-analytics

Aggregation operators

- Pipeline and Expression operators

Pipeline	Expression	Arithmetic	Conditional
$match	$addToSet	$add	$cond
$sort	$first	$divide	$ifNull
$limit	$last	$mod	
$skip	$max	$multiply	
$project	$min	$subtract	
$unwind	$avg		Variables
$group	$push		
$geoNear	$sum		$let
$text			$map
$search			

Tip: Other operators for date, time, boolean and string manipulation

MongoDB Checkpoint

- You can see that MongoDB has powerful, sophisticated
  - Operators
  - Expressions
  - Pipelines
- We have only skimmed the surface. There is a lot more:
  - Indexes
  - Replication, Sharding
  - Embedded Map-Reduce support
  - … …
- We will explore a little more in subsequent lectures, homework, … …
- You will have to install MongoDB and Compass for HWs and final exam.

Let’s Do Some Examples

- Switch to the notebook.
- But first, ... How does Don use MongoDB?