

DS 4300

Large Scale Information Storage and Retrieval

Foundations

Mark Fontenot, PhD
Northeastern University

Searching

- Searching is the most common operation performed by a database system
- In SQL, the SELECT statement is arguably the most versatile / complex.
- Baseline for efficiency is **Linear Search**
 - Start at the beginning of a list and proceed element by element until:
 - You find what you're looking for
 - You get to the last element and haven't found it

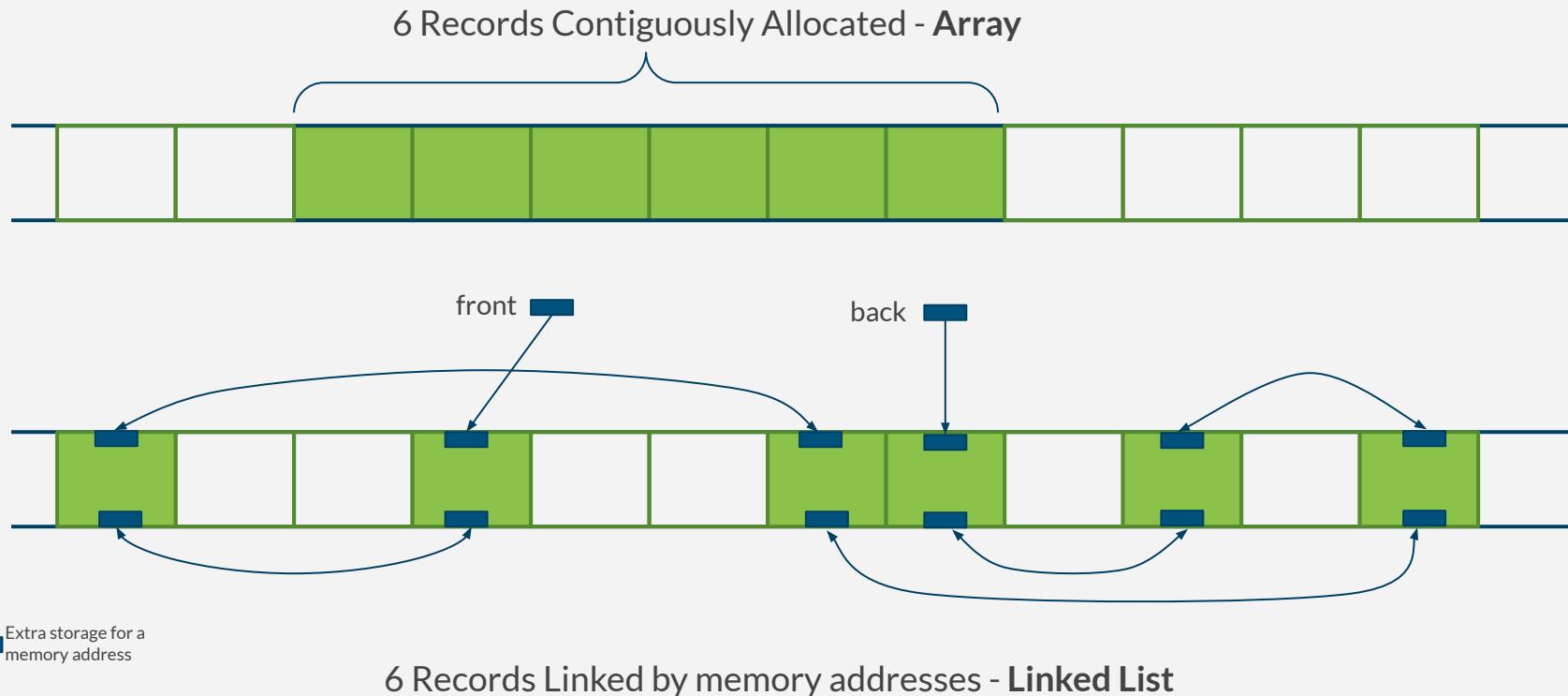
Searching

- **Record** - A collection of values for attributes of a single entity instance; a row of a table
- **Collection** - a set of records of the same entity type; a table
 - Trivially, stored in some sequential order like a list
- **Search Key** - A value for an attribute from the entity type
 - Could be ≥ 1 attribute

Lists of Records

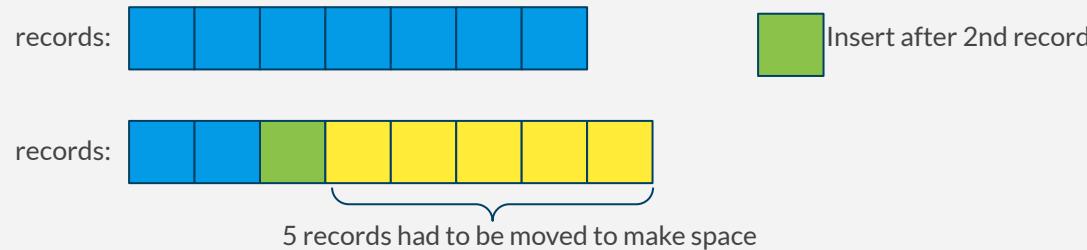
- If each record takes up x bytes of memory, then for n records, we need $n*x$ bytes of memory.
- Contiguously Allocated List
 - All $n*x$ bytes are allocated as a single “chunk” of memory
- Linked List
 - Each record needs x bytes + additional space for 1 or 2 memory addresses
 - Individual records are linked together in a type of chain using memory addresses

Contiguous vs Linked

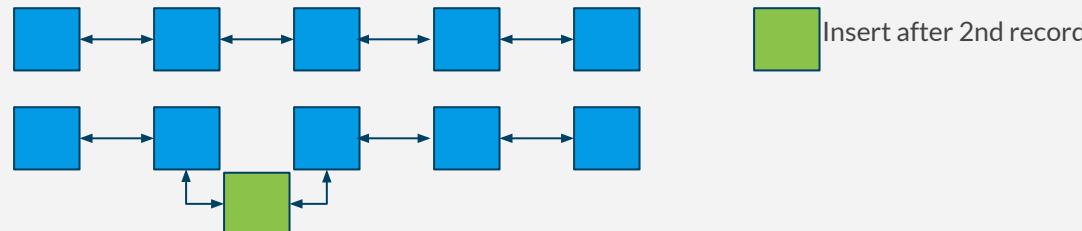


Pros and Cons

- Arrays are faster for random access, but slow for inserting anywhere but the end



- Linked Lists are faster for inserting anywhere in the list, but slower for random access



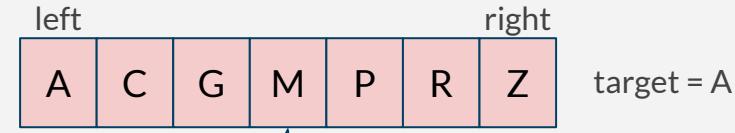
Observations:

- Arrays
 - fast for random access
 - slow for random insertions
- Linked Lists
 - slow for random access
 - fast for random insertions

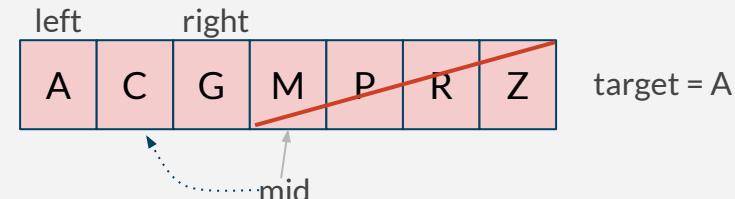
Binary Search

- Input: array of values in sorted order, target value
- Output: the location (index) of where target is located or some value indicating target was not found

```
def binary_search(arr, target)
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```



Since $target < arr[mid]$, we reset $right$ to $mid - 1$.



Time Complexity

- Linear Search
 - Best case: target is found at the first element; only 1 comparison
 - Worst case: target is not in the array; n comparisons
 - Therefore, in the worst case, linear search is $O(n)$ time complexity.
- Binary Search
 - Best case: target is found at mid ; 1 comparison (inside the loop)
 - Worst case: target is not in the array; $\log_2 n$ comparisons
 - Therefore, in the worst case, binary search is $O(\log_2 n)$ time complexity.

Back to Database Searching

- Assume data is stored on disk by column id's value
- Searching for a specific id = fast.
- But what if we want to search for a specific *specialVal*?
 - Only option is linear scan of that column
- Can't store data on disk sorted by both id and specialVal (at the same time)
 - data would have to be duplicated → space inefficient

id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

Back to Database Searching

- Assume data is stored on disk by column id's value
- Search time is proportional to number of rows
- Build an external data structure to support faster searching by *specialVal* than a linear scan.
- Can't store all data in memory and search from memory
 - data would have to be duplicated → space inefficient

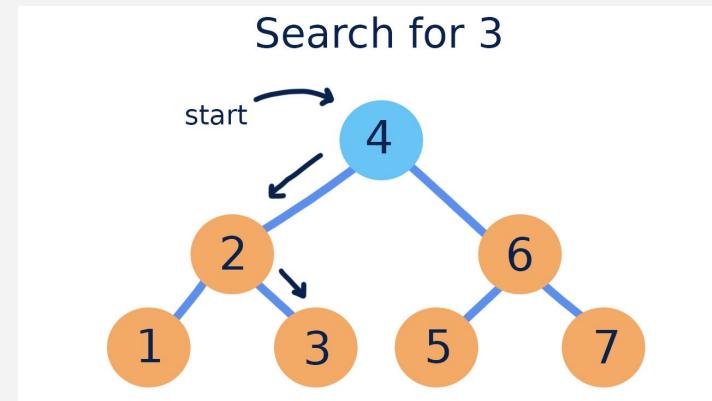
id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

What do we have in our arsenal?

- 1) An array of tuples (specialVal, rowNumber) sorted by specialVal
 - a) We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table
 - b) But, every insert into the table would be like inserting into a sorted array - slow...
- 2) A linked list of tuples (specialVal, rowNumber) sorted by specialVal
 - a) searching for a specialVal would be slow - linear scan required
 - b) But inserting into the table would theoretically be quick to also add to the list.

Something with Fast Insert and Fast Search?

- Binary Search Tree - a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent.



To the Board!

DS 4300

Moving Beyond the Relational Model

Mark Fontenot, PhD
Northeastern University

Benefits of the Relational Model

- (Mostly) Standard Data Model and Query Language
- ACID Compliance (more on this in a second)
 - Atomicity, Consistency, Isolation, Durability
- Works well will highly structured data
- Can handle large amounts of data
- Well understood, lots of tooling, lots of experience

Relational Database Performance

Many ways that a RDBMS increases efficiency:

- indexing (the topic we focused on)
- directly controlling storage
- column oriented storage vs row oriented storage
- query optimization
- caching/prefetching
- materialized views
- precompiled stored procedures
- data replication and partitioning

Transaction Processing

- **Transaction** - a sequence of one or more of the CRUD operations performed as a single, logical unit of work
 - Either the entire sequence succeeds (COMMIT)
 - OR the entire sequence fails (ROLLBACK or ABORT)
- Help ensure
 - Data Integrity
 - Error Recovery
 - Concurrency Control
 - Reliable Data Storage
 - Simplified Error Handling

ACID Properties

- **Atomicity**
 - transaction is treated as an atomic unit - it is fully executed or no parts of it are executed
- **Consistency**
 - a transaction takes a database from one consistent state to another consistent state
 - consistent state - all data meets integrity constraints

ACID Properties

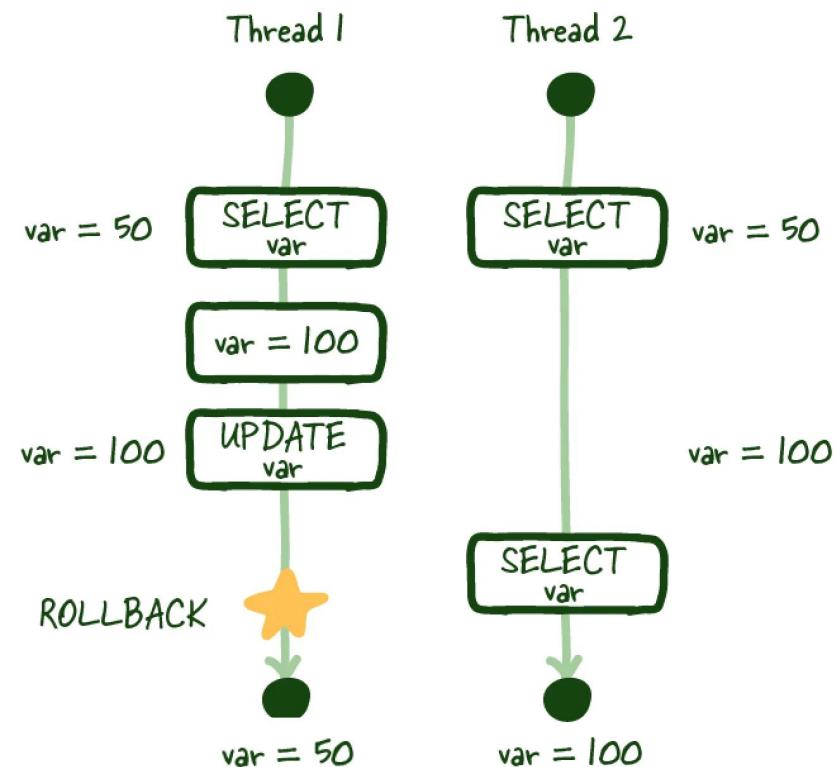
- Isolation

- Two transactions T_1 and T_2 are being executed at the same time but cannot affect each other
- If both T_1 and T_2 are reading the data - no problem
- If T_1 is reading the same data that T_2 may be writing, can result in:
 - Dirty Read
 - Non-repeatable Read
 - Phantom Reads

Isolation: Dirty Read

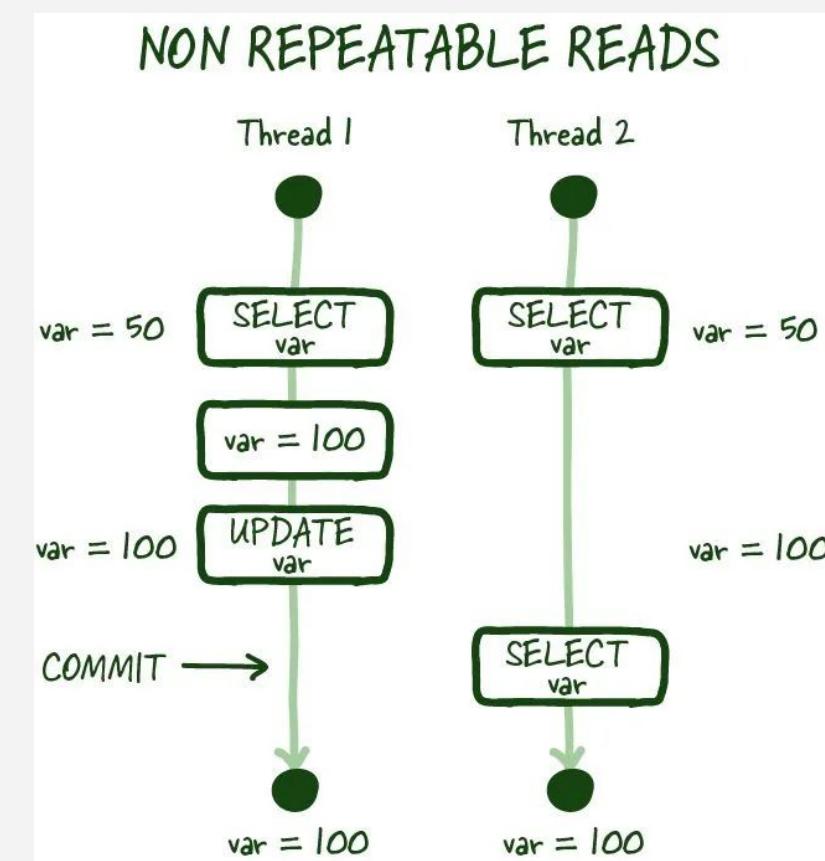
DIRTY READS

Dirty Read - a transaction T_1 is able to read a row that has been modified by another transaction T_2 that hasn't yet executed a COMMIT



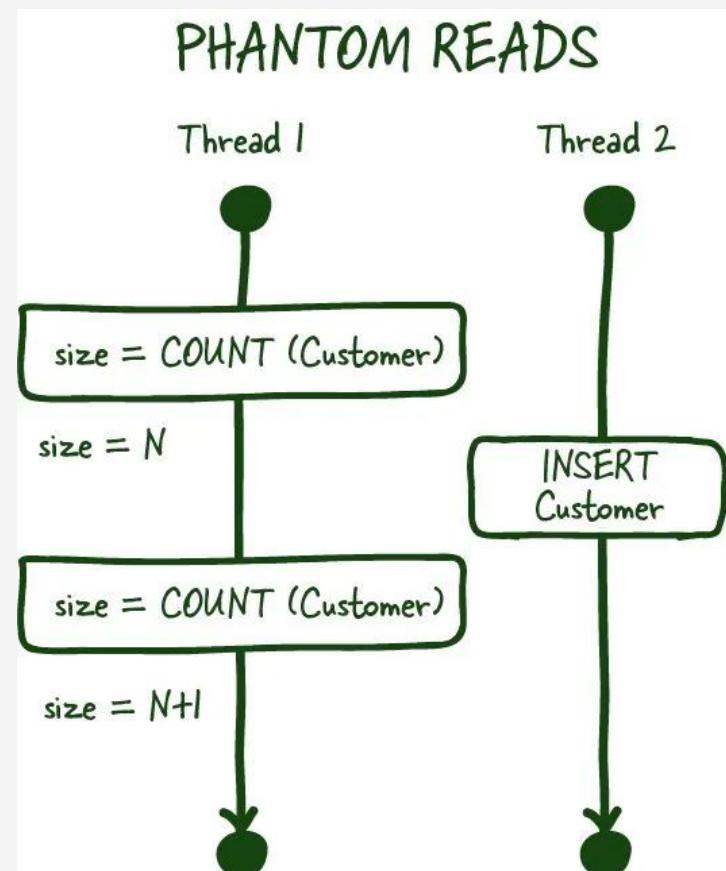
Isolation: Non-Repeatable Read

Non-repeatable Read - two queries in a single transaction T_1 execute a SELECT but get different values because another transaction T_2 has changed data and COMMITTED



Isolation: Phantom Reads

Phantom Reads - when a transaction T_1 is running and another transaction T_2 adds or deletes rows from the set T_1 is using



Example Transaction - Transfer \$\$

```
DELIMITER //

CREATE PROCEDURE transfer(
    IN sender_id INT,
    IN receiver_id INT,
    IN amount DECIMAL(10,2)
)
BEGIN
    DECLARE rollback_message VARCHAR(255)
        DEFAULT 'Transaction rolled back: Insufficient funds';
    DECLARE commit_message VARCHAR(255)
        DEFAULT 'Transaction committed successfully';

    -- Start the transaction
    START TRANSACTION;

    -- Attempt to debit money from account 1
    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;

    -- Attempt to credit money to account 2
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

    -- Continued Next Slide
```

Example Transaction - Transfer \$\$

```
-- Continued from previous slide

-- Check if there are sufficient funds in account 1
-- Simulate a condition where there are insufficient funds
IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
    -- Roll back the transaction if there are insufficient funds
    ROLLBACK;
    SIGNAL SQLSTATE '45000'      -- 45000 is unhandled, user-defined error
        SET MESSAGE_TEXT = rollback_message;
ELSE
    -- Log the transactions if there are sufficient funds
    INSERT INTO transactions (account_id, amount, transaction_type)
        VALUES (sender_id, -amount, 'WITHDRAWAL');
    INSERT INTO transactions (account_id, amount, transaction_type)
        VALUES (receiver_id, amount, 'DEPOSIT');

    -- Commit the transaction
    COMMIT;
    SELECT commit_message AS 'Result';
END IF;
END //

DELIMITER ;
```

ACID Properties

- **Durability**
 - Once a transaction is completed and committed successfully, its changes are permanent.
 - Even in the event of a system failure, committed transactions are preserved
- For more info on Transactions, see:
 - Kleppmann Book Chapter 7

But ...

Relational Databases may not be the solution to all problems...

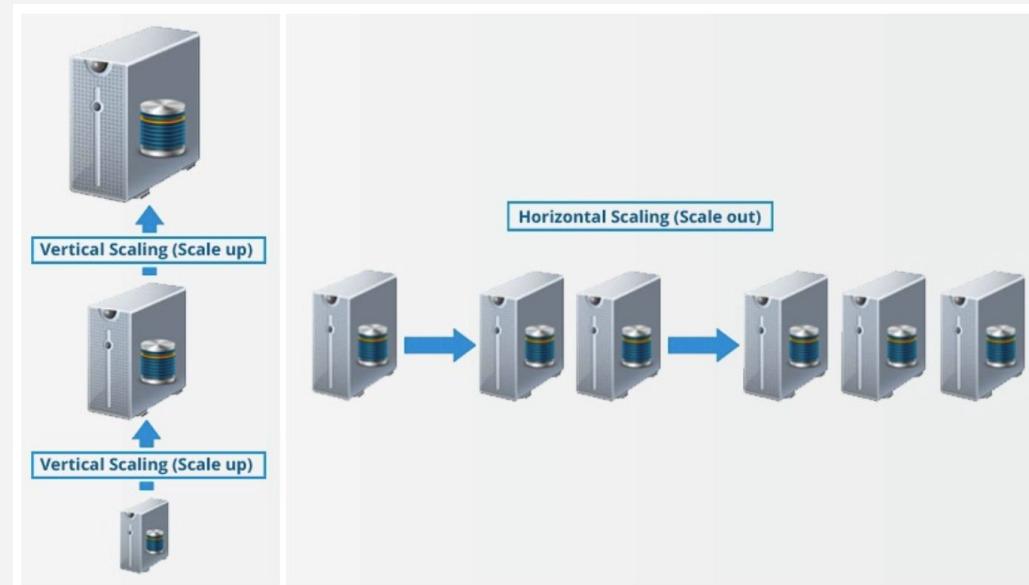
- sometimes, schemas evolve over time
- not all apps may need the full strength of ACID compliance
- joins can be expensive
- a lot of data is semi-structured or unstructured (JSON, XML, etc)
- Horizontal scaling presents challenges
- some apps need something more performant (real time, low latency systems)

Scalability - Up or Out?

Conventional Wisdom: Scale vertically (up, with bigger, more powerful systems) until the demands of high-availability make it necessary to scale out with some type of distributed computing model

But why? Scaling up is easier - no need to really modify your architecture. But there are practical and financial limits

However: There are modern systems that make horizontal scaling less problematic.



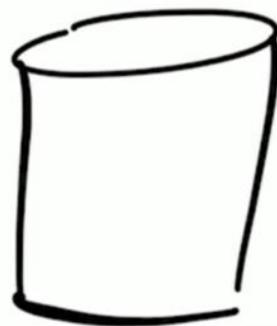
So what? Distributed Data when Scaling Out

A distributed system is “*a collection of independent computers that appear to its users as one computer.*” -Andrew Tennenbaum

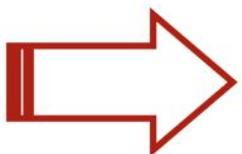
Characteristics of Distributed Systems:

- computers operate concurrently
- computers fail independently
- no shared global clock

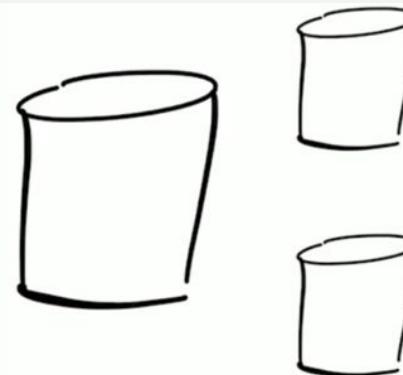
Distributed Storage - 2 Directions



Single
Main
Node

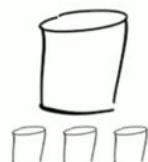


Replication:



Sharding:

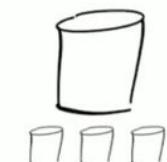
Aaron-
Frances



Frances-
Nancy



Nancy-
Zed

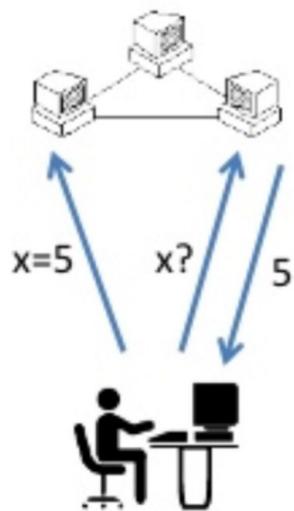


Distributed Data Stores

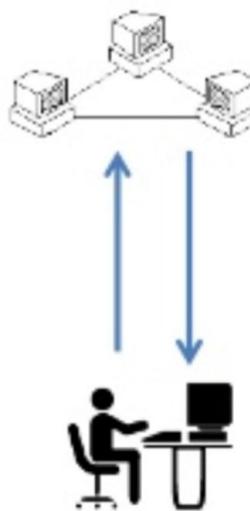
- Data is stored on > 1 node, typically replicated
 - i.e. each block of data is available on N nodes
- Distributed databases can be relational or non-relational
 - MySQL and PostgreSQL support replication and sharding
 - CockroachDB - new player on the scene
 - Many NoSQL systems support one or both models
- But remember: **Network partitioning is inevitable!**
 - network failures, system failures
 - Overall system needs to be **Partition Tolerant**
 - System can keep running even w/ network partition

The CAP Theorem

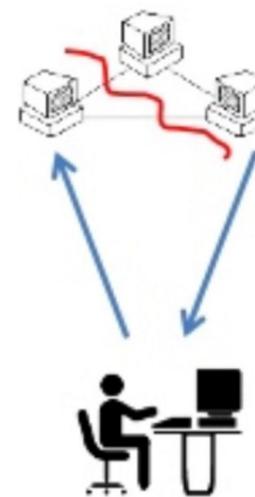
Consistency



Availability



Partition tolerance



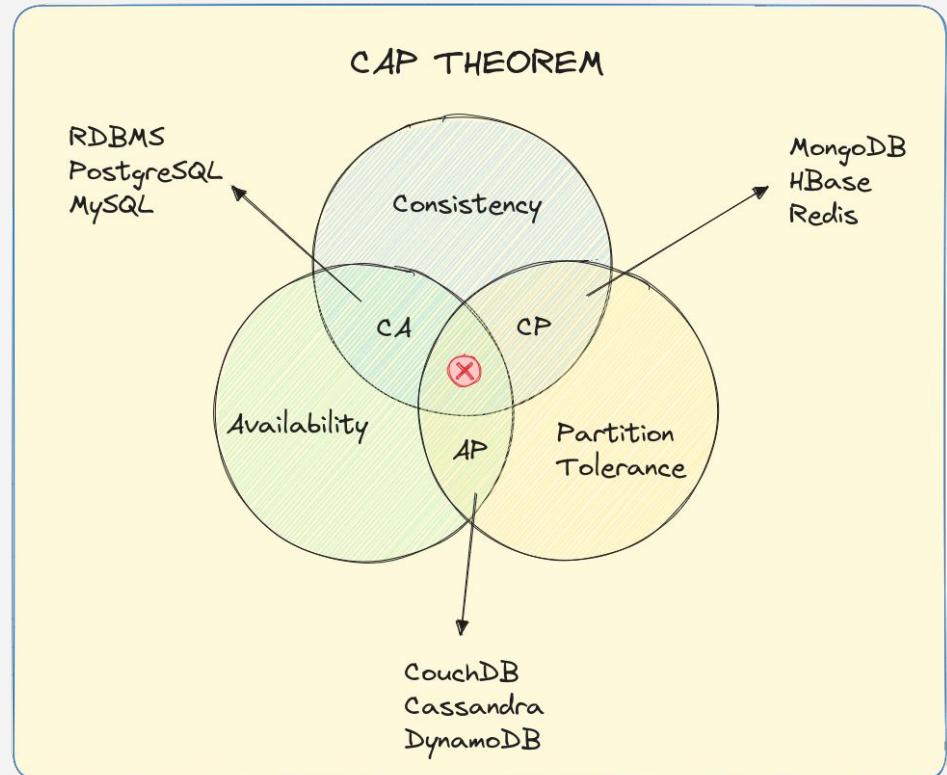
The CAP Theorem

The **CAP Theorem** states that it is impossible for a distributed data store to *simultaneously* provide more than two out of the following three guarantees:

- **Consistency** - Every read receives the most recent write or error thrown
- **Availability** - Every request receives a (non-error) response - but no guarantee that the response contains the most recent write
- **Partition Tolerance** - The system can continue to operate despite arbitrary network issues.

CAP Theorem - Database View

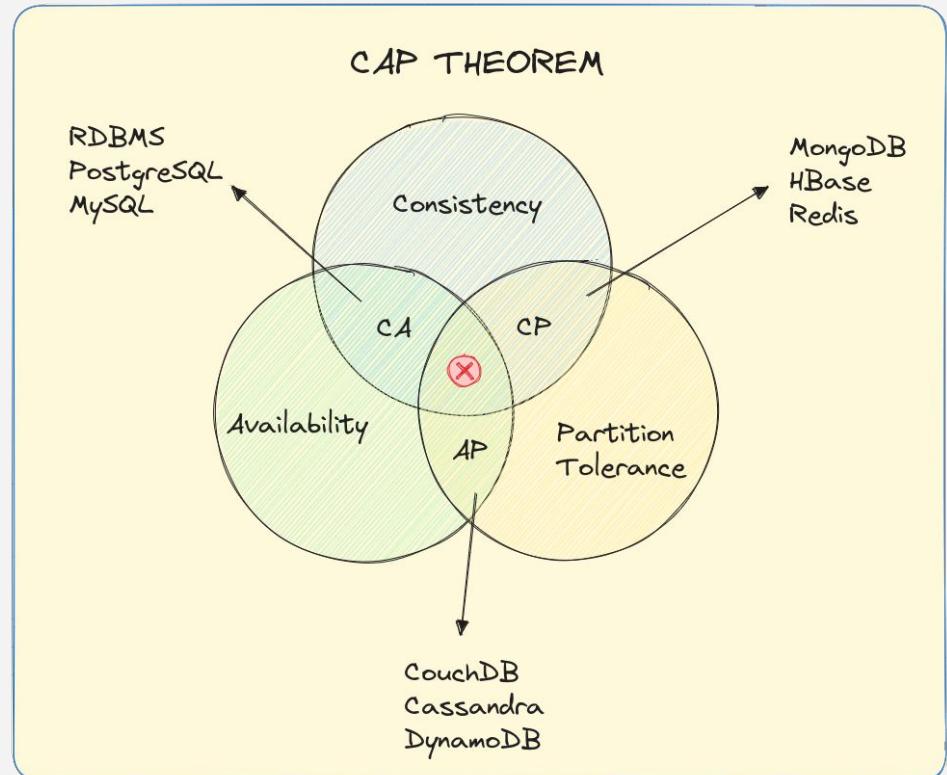
- **Consistency***: Every user of the DB has an identical view of the data at any given instant
- **Availability**: In the event of a failure, the database remains operational
- **Partition Tolerance**: The database can maintain operations in the event of the network's failing between two segments of the distributed system



* Note, the definition of Consistency in CAP is different from that of ACID.

CAP Theorem - Database View

- **Consistency + Availability:** System always responds with the latest data and every request gets a response, but may not be able to deal with network issues
- **Consistency + Partition Tolerance:** If system responds with data from a distributed store, it is always the latest, else data request is dropped.
- **Availability + Partition Tolerance:** System always sends responses based on distributed store, but may not be the absolute latest data.



CAP in Reality

What it is really saying:

- If you cannot limit the number of faults, requests can be directed to any server, and you insist on serving every request, then you cannot possibly be consistent.

But it is interpreted as:

- You must always give up something: consistency, availability, or tolerance to failure.

??
..

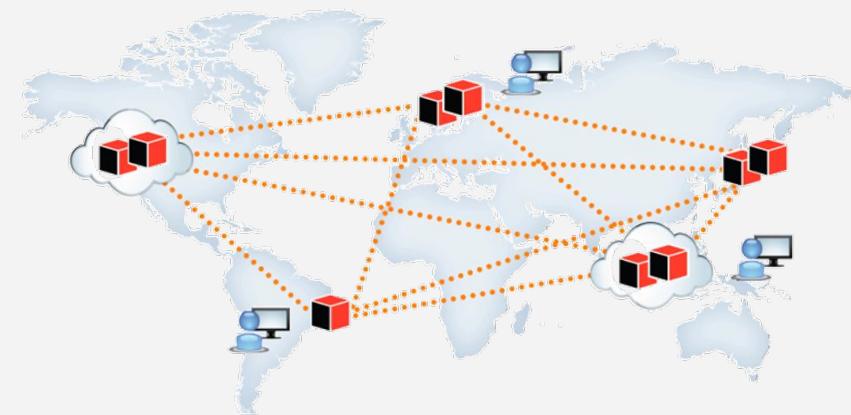
DS 4300

Replicating Data

Mark Fontenot, PhD
Northeastern University

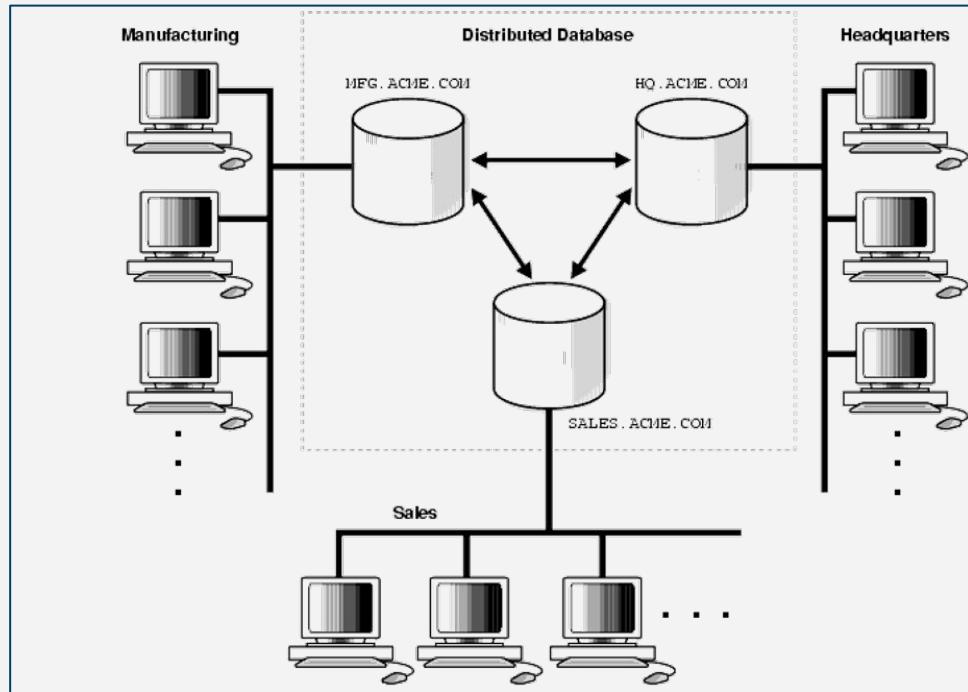
Distributing Data - Benefits

- **Scalability / High throughput:** Data volume or Read/Write load grows beyond the capacity of a single machine
- **Fault Tolerance / High Availability:** Your application needs to continue working even if one or more machines goes down.
- **Latency:** When you have users in different parts of the world you want to give them fast performance too



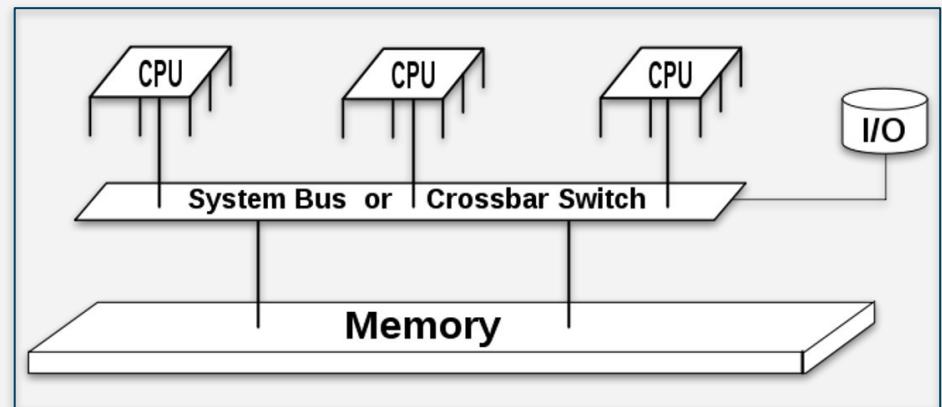
Distributed Data - Challenges

- **Consistency:** Updates must be propagated *across the network*.
- **Application Complexity:** Responsibility for reading and writing data in a distributed environment often falls to the application.



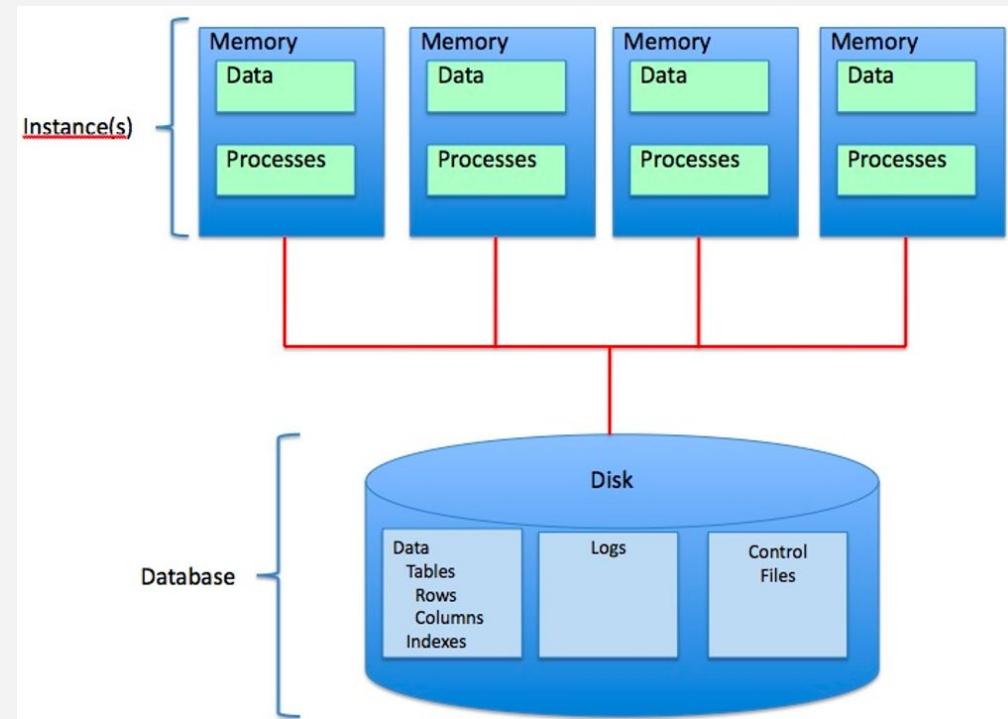
Vertical Scaling - Shared Memory Architectures

- Geographically Centralized server
- Some fault tolerance (via hot-swappable components)



Vertical Scaling - Shared Disk Architectures

- Machines are connected via a fast network
- Contention and the overhead of locking limit scalability (high-write volumes) ... BUT ok for Data Warehouse applications (high read volumes)



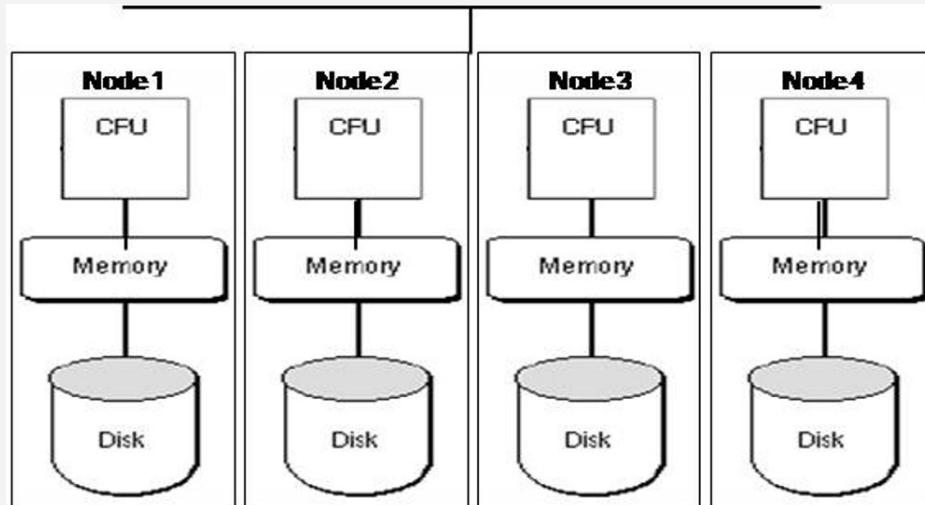
AWS EC2 Pricing - Oct 2024

Instance name	On-Demand hourly rate	vCPU	Memory	Storage	Network performance
t4g.nano	\$0.0042	2	0.5 GiB	EBS Only	Up to 5 Gigabit
t4g.micro	\$0.0084	2	1 GiB	EBS Only	Up to 5 Gigabit
t4g.small	\$0.0168	2	2 GiB	EBS Only	Up to 5 Gigabit
t3.medium	\$0.0416	2	4 GiB	EBS Only	Up to 5 Gigabit
t3.large	\$0.0832	2	8 GiB	EBS Only	Up to 5 Gigabit
t3.xlarge	\$0.1664	4	16 GiB	EBS Only	Up to 5 Gigabit
t3.2xlarge	\$0.3328	8	32 GiB	EBS Only	Up to 5 Gigabit
u-6tb1.112xlarge	\$54.60	448	32768 GiB	EBS Only	100 Gigabit
u-9tb1.112xlarge	\$81.90	448	32768 GiB	EBS Only	100 Gigabit
p5.48xlarge	\$98.32	192	2048 GiB	8 x 3840 GB SSD	3200 Gigabit
u-12tb1.112xlarge	\$109.20	448	12288 GiB	EBS Only	100 Gigabit

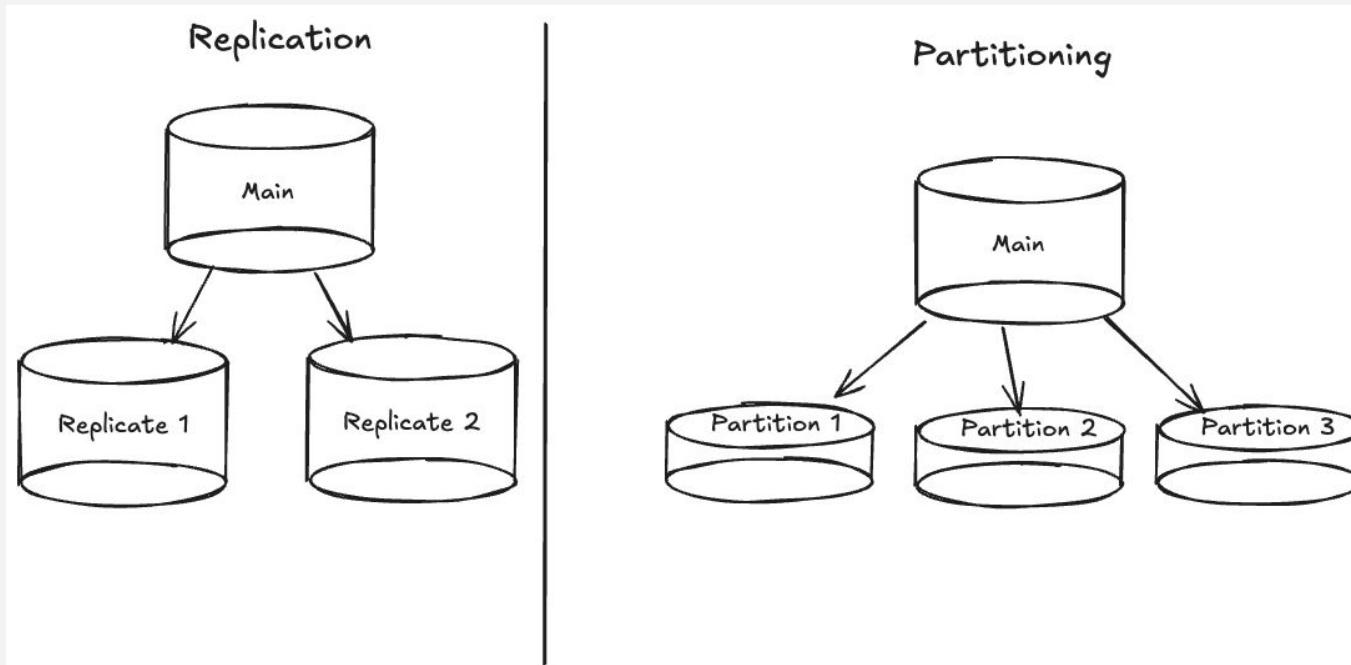
> \$78,000/month

Horizontal Scaling - Shared Nothing Architectures

- Each node has its own CPU, memory, and disk
- Coordination via application layer using conventional network
- Geographically distributed
- Commodity hardware



Data - Replication vs Partitioning



Replicates have
same data as Main

Partitions have a
subset of the data

Replication

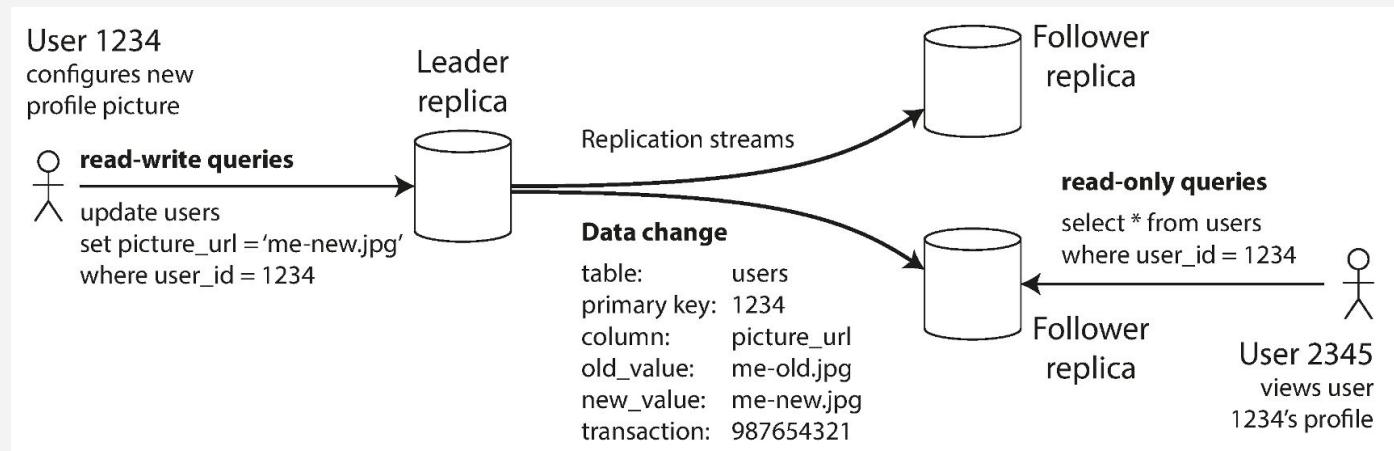
Common Strategies for Replication

- Single leader model
- Multiple leader model
- Leaderless model

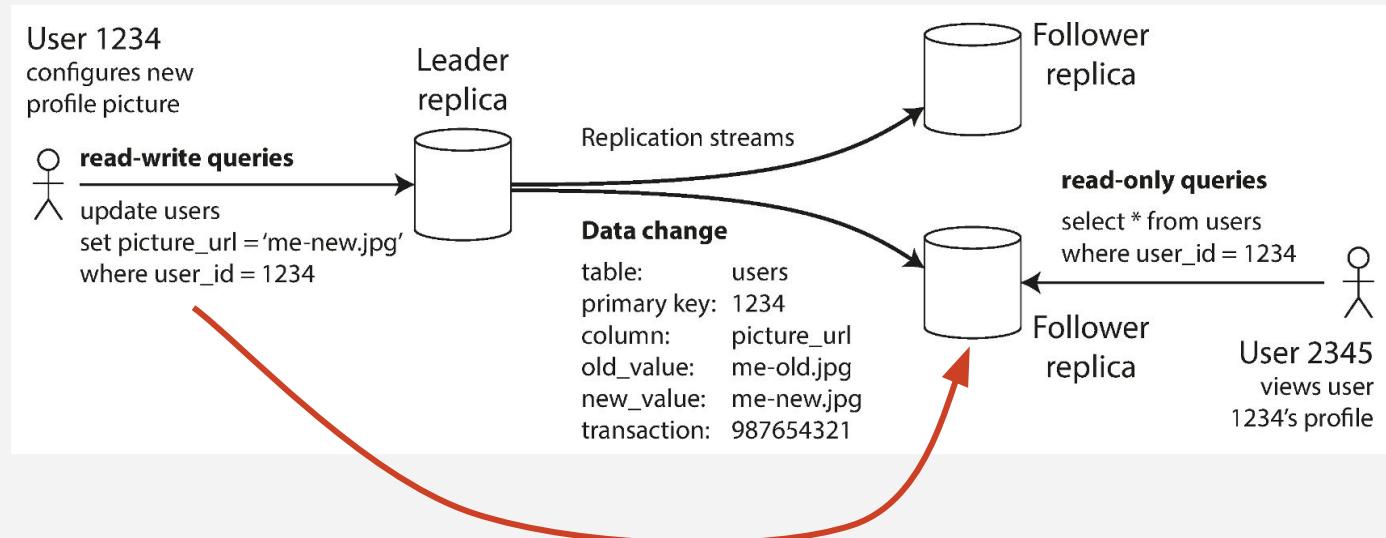
Distributed databases usually adopt one of these strategies.

Leader-Based Replication

- All writes from clients go to the leader
- Leader sends replication info to the followers
- Followers process the instructions from the leader
- Clients can read from either the leader or followers



Leader-Based Replication



This write could NOT be sent to one of the followers... only the leader.

Leader-Based Replication - Very Common Strategy

Relational:

- MySQL,
- Oracle,
- SQL Server,
- PostgreSQL

NoSQL:

- MongoDB,
- RethinkDB (realtime web apps),
- Espresso (LinkedIn)

Messaging Brokers: Kafka, RabbitMQ

How Is Replication Info Transmitted to Followers?

Replication Method	Description
Statement-based	Send INSERT, UPDATE, DELETEs to replica. Simple but error-prone due to non-deterministic functions like now(), trigger side-effects, and difficulty in handling concurrent transactions.
Write-ahead Log (WAL)	A byte-level specific log of every change to the database. Leader and all followers must implement the same storage engine and makes upgrades difficult.
Logical (row-based) Log	For relational DBs: Inserted rows, modified rows (before and after), deleted rows. A transaction log will identify all the rows that changed in each transaction and how they changed. Logical logs are decoupled from the storage engine and easier to parse.
Trigger-based	Changes are logged to a separate table whenever a trigger fires in response to an insert, update, or delete. Flexible because you can have application specific replication, but also more error prone.

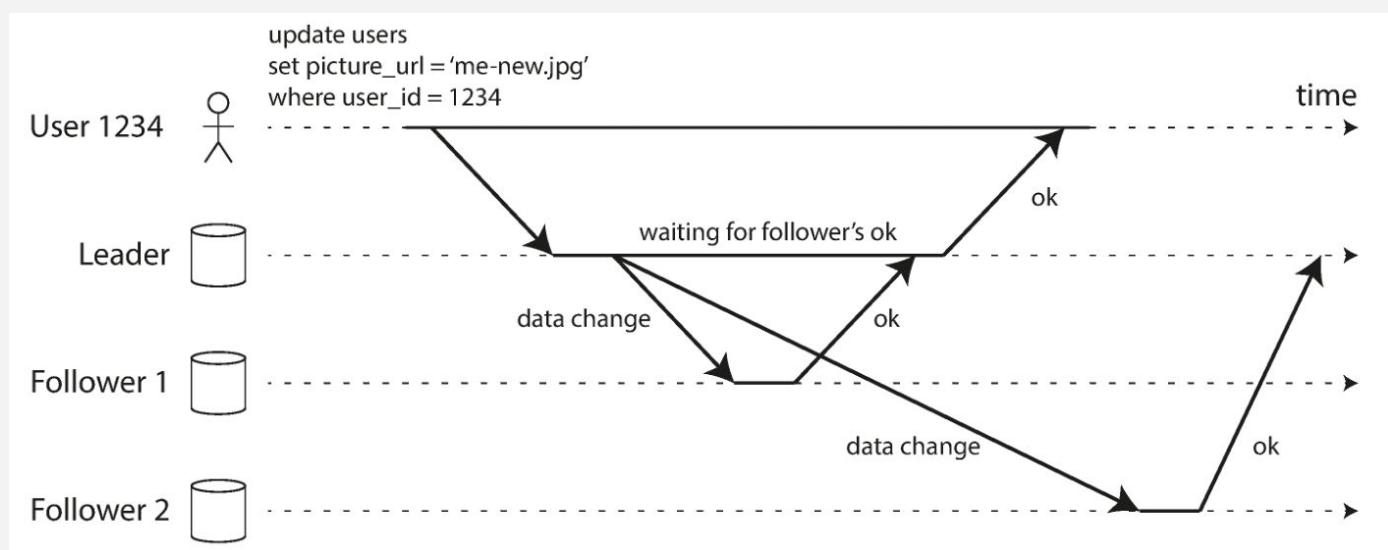
Synchronous vs Asynchronous Replication

Synchronous: Leader waits for a response from the follower

Asynchronous: Leader doesn't wait for confirmation.

Synchronous:

Asynchronous:



What Happens When the Leader Fails?



Challenges: How do we pick a new Leader Node?

- Consensus strategy – perhaps based on who has the most updates?
- Use a controller node to appoint new leader?

AND... *how do we configure clients to start writing to the new leader?*

What Happens When the Leader Fails?



More Challenges:

- If asynchronous replication is used, new leader may not have all the writes
How do we recover the lost writes? Or do we simply discard?
- After (if?) the old leader recovers, how do we avoid having multiple leaders receiving conflicting data? (Split brain: no way to resolve conflicting requests.)
- Leader failure detection. Optimal timeout is tricky.

Replication Lag

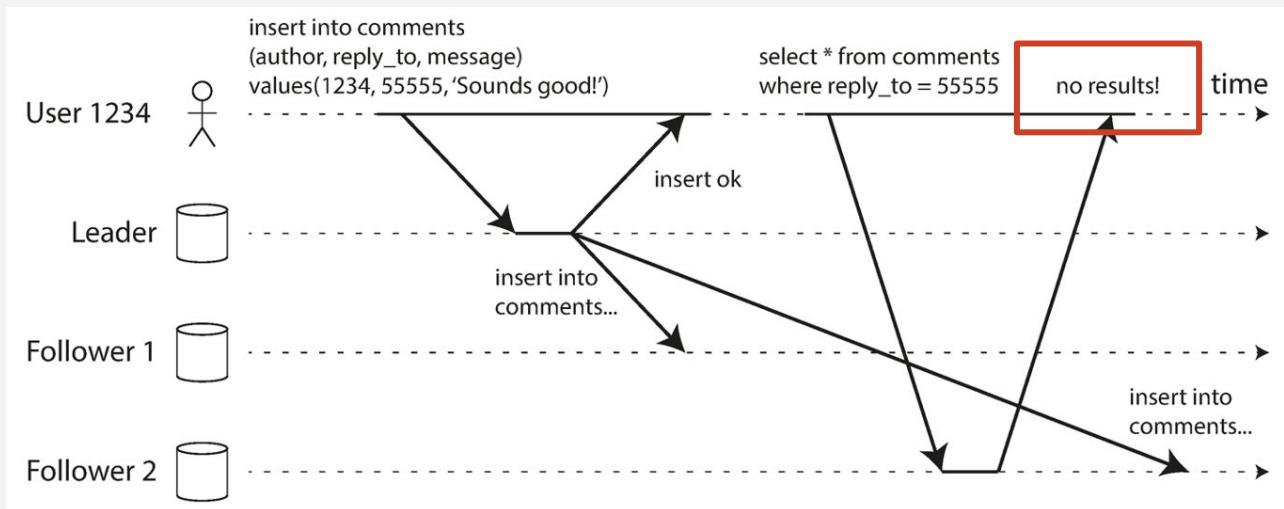
Replication Lag refers to the time it takes for writes on the leader to be reflected on all of the followers.

- **Synchronous replication:** Replication lag causes writes to be slower and the system to be more brittle as num followers increases.
- **Asynchronous replication:** We maintain availability *but at the cost of delayed or eventual consistency*. This delay is called the *inconsistency window*.

Read-after-Write Consistency

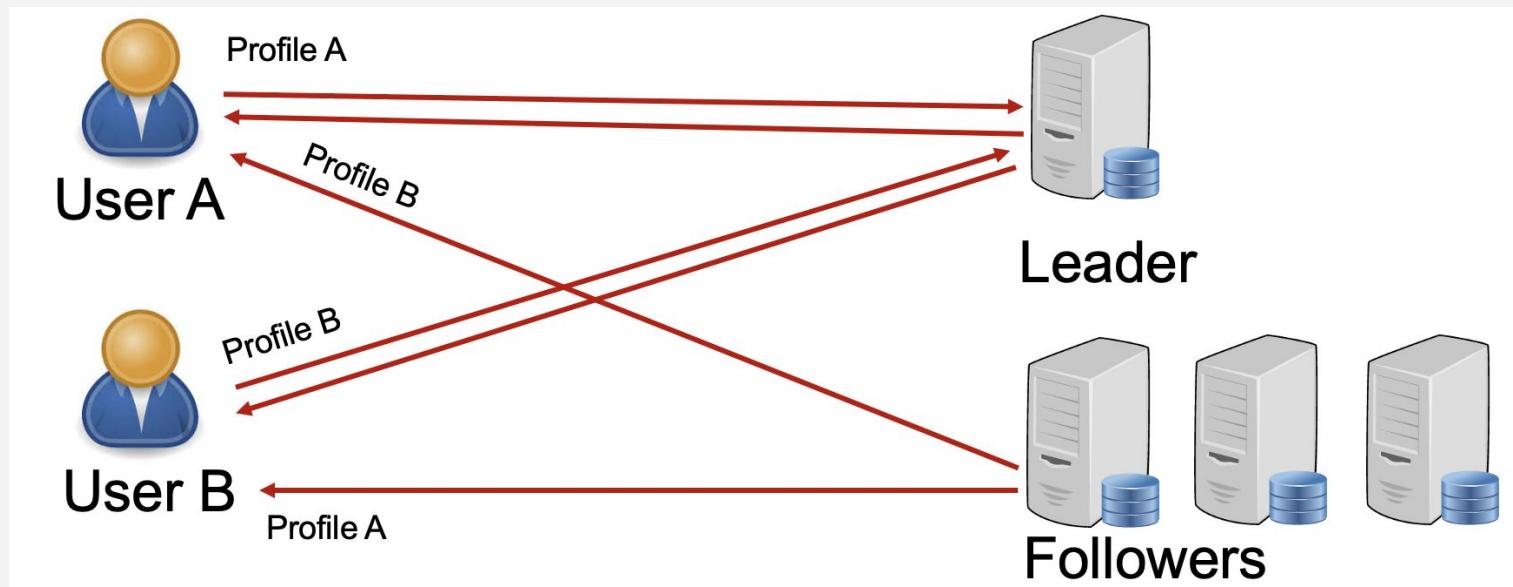
Scenario - you're adding a comment to a Reddit post... after you click Submit and are back at the main post, your comment should show up for you.

- Less important for other users to see your comment as immediately.



Implementing Read-After-Write Consistency

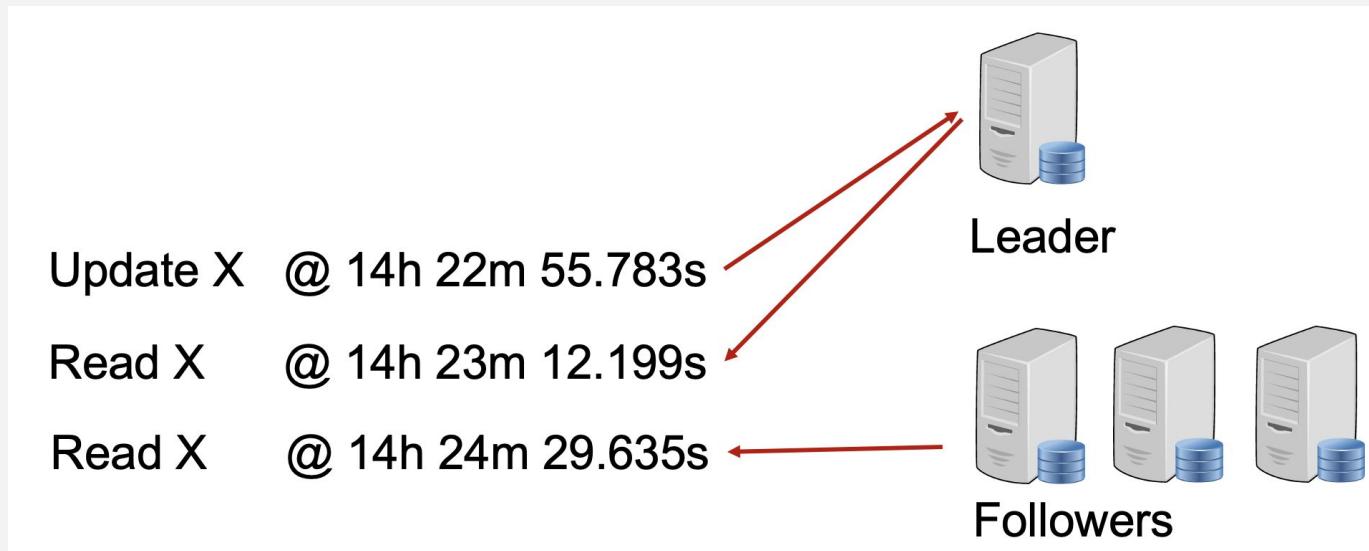
Method 1: Modifiable data (from the client's perspective) is always read from the leader.



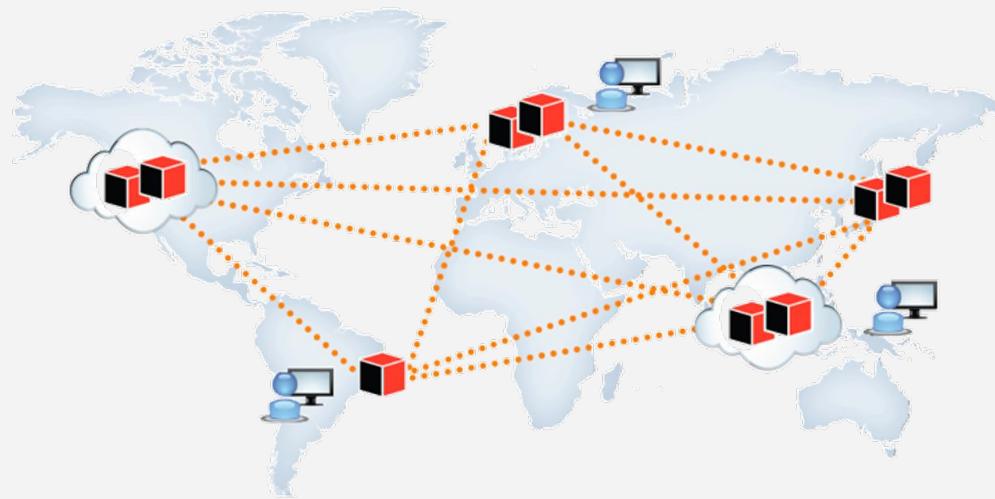
Implementing Read-After-Write Consistency

Method 2: Dynamically switch to reading from leader for “recently updated” data.

- For example, have a policy that all requests within one minute of last update come from leader.



But... This Can Create Its Own Challenges

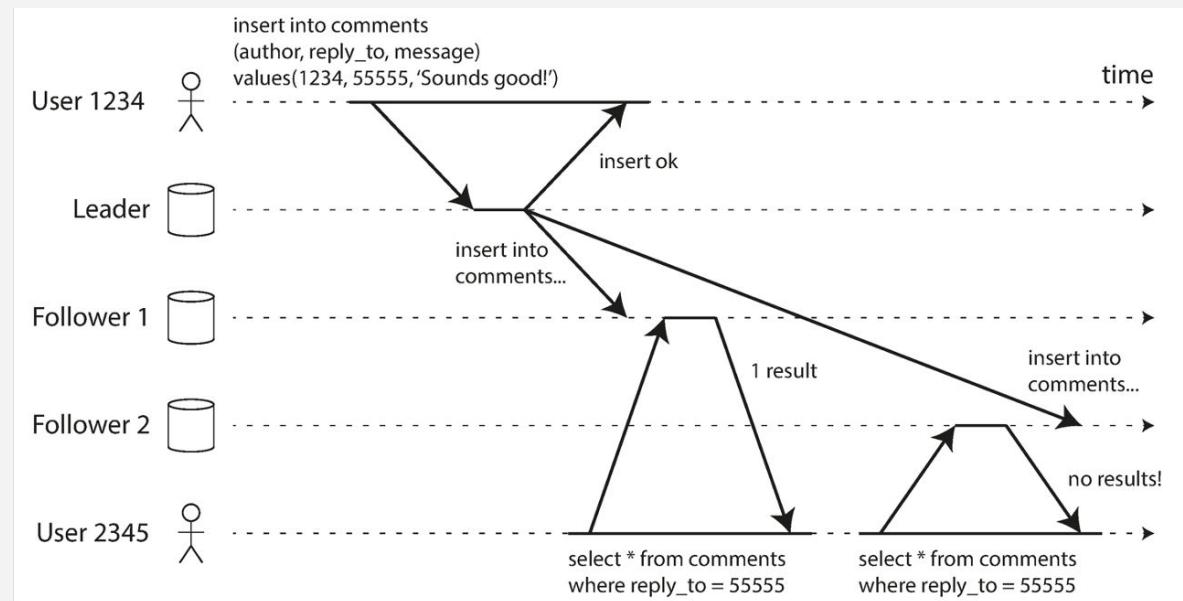


We created followers so they would be proximal to users. BUT... now we have to route requests to distant leaders when reading modifiable data?? :(

Monotonic Read Consistency

Monotonic read anomalies:
occur when a user reads
values out of order from
multiple followers.

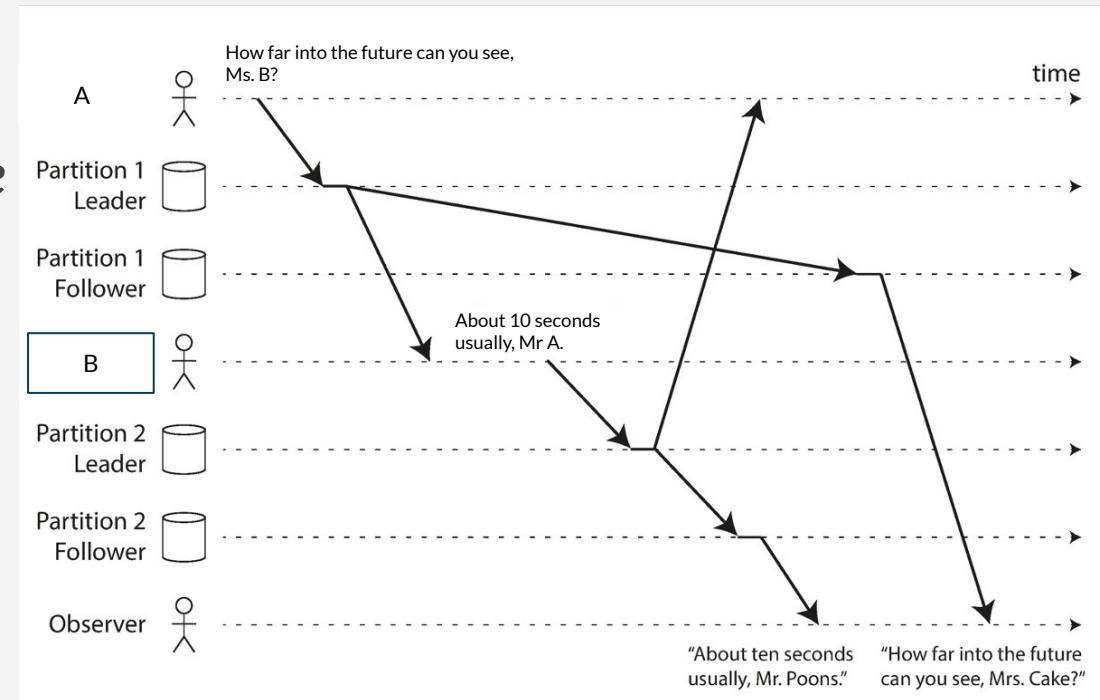
Monotonic read consistency:
ensures that when a user
makes multiple reads, they
will not read older data after
previously reading newer
data.



Consistent Prefix Reads

Reading data out of order can occur if different partitions replicate data at different rates. There is *no global write consistency*.

Consistent Prefix Read Guarantee - ensures that if a sequence of writes happens in a certain order, anyone reading those writes will see them appear in the same order.



??
..

DS 4300

Large Scale Information Storage and Retrieval

B+ Tree Walkthrough

Mark Fontenot, PhD
Northeastern University

Insert: 42, 21, 63, 89

B+ Tree : $m = 4$

21	42	63	89
----	----	----	----

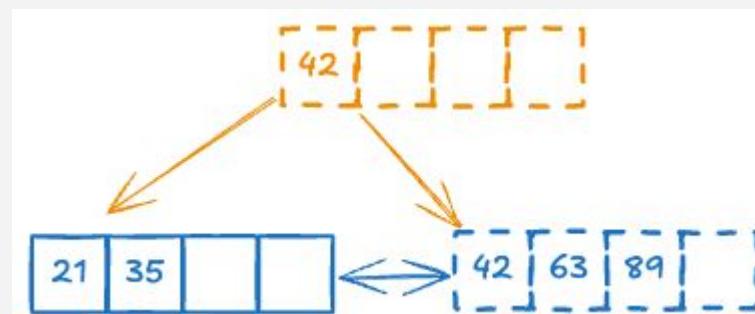
- Initially, the first node is a leaf node AND root node.
- 21, 42, ... represent keys of some set of K:V pairs
- Leaf nodes store keys and data, although data not shown
- Inserting another key will cause the node to split.

Insert: 35

B+ Tree : $m = 4$

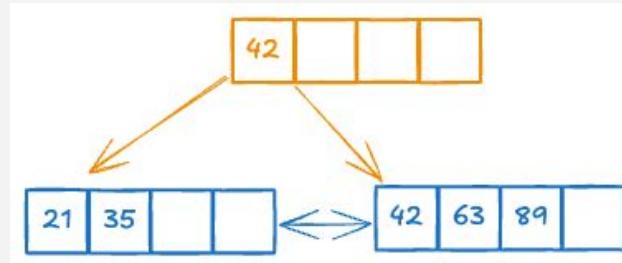


- Leaf node needs to split to accommodate 35. New leaf node allocated to the right of existing node
- 5/2 values stay in original node; remaining values moved to new node
- Smallest value from new leaf node (42) is copied up to the parent, which needs to be created in this case. It will be an *internal* node.

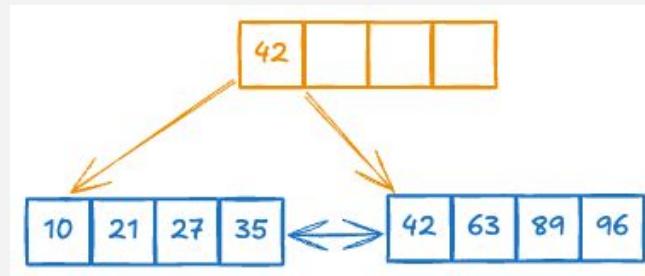


Insert: 10, 27, 96

B+ Tree : $m = 4$

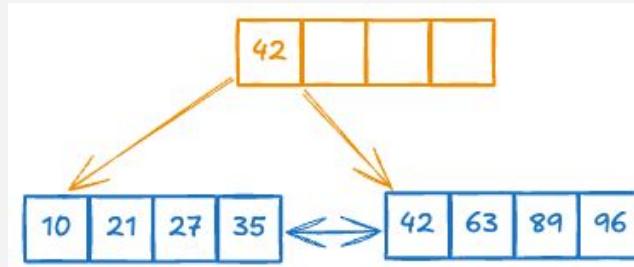


- The insert process starts at the root node. The keys of the root node are searched to find out which child node we need to descend to.
 - EX: 10. Since $10 < 42$, we follow the pointer to the left of 42
- Note - none of these new values cause a node to split

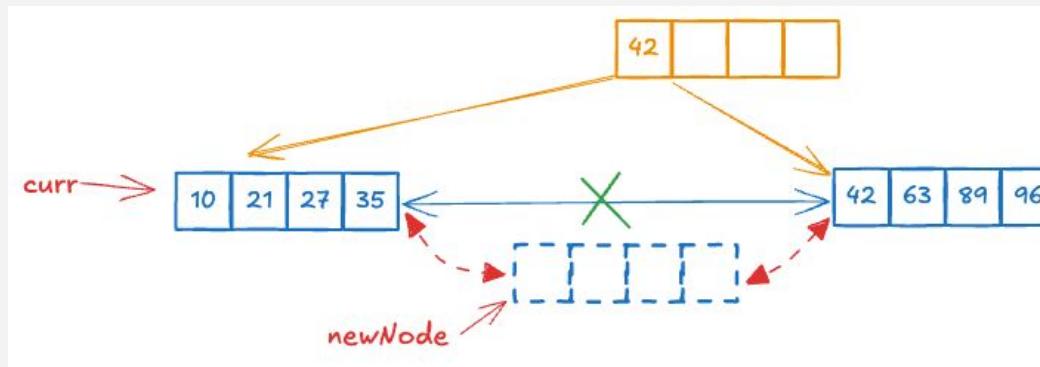


B+ Tree : $m = 4$

Insert: 30

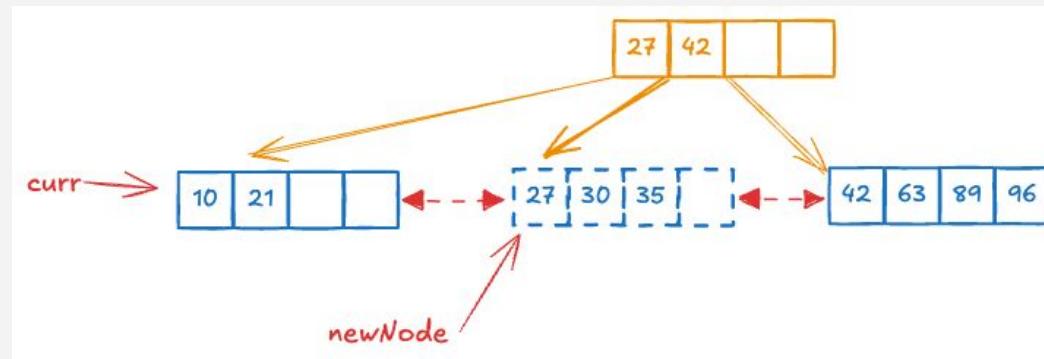


- Starting at root, we descend to the left-most child (we'll call *curr*).
 - *curr* is a leaf node. Thus, we insert 30 into *curr*.
 - BUT *curr* is full. So we have to split.
 - Create a new node to the right of *curr*, temporarily called *newNode*.
 - Insert *newNode* into the doubly linked list of leaf nodes.

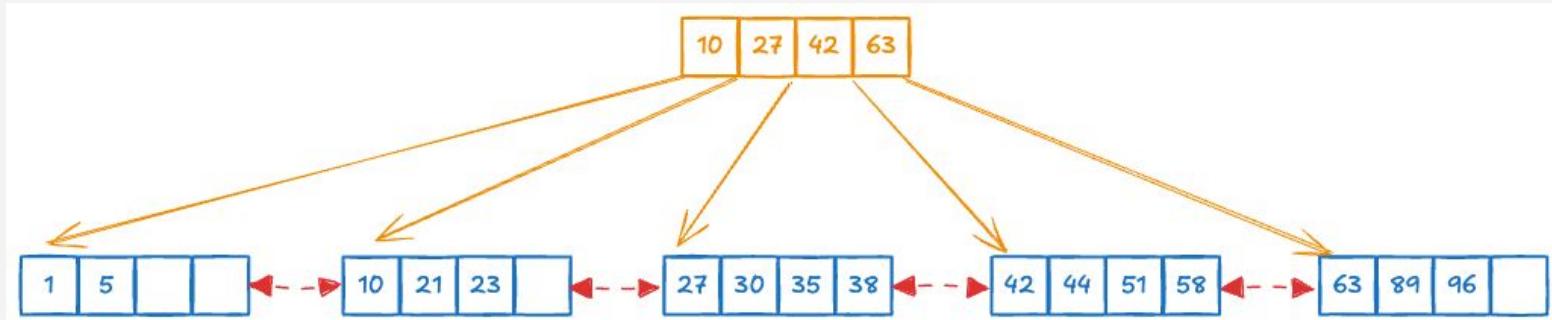


Insert: 30 cont'd.

- re-distribute the keys
- copy the smallest key (27 in this case) from newNode to parent; rearrange keys and pointers in parent node.
- Parent of newNode is also root. So, nothing else to do.



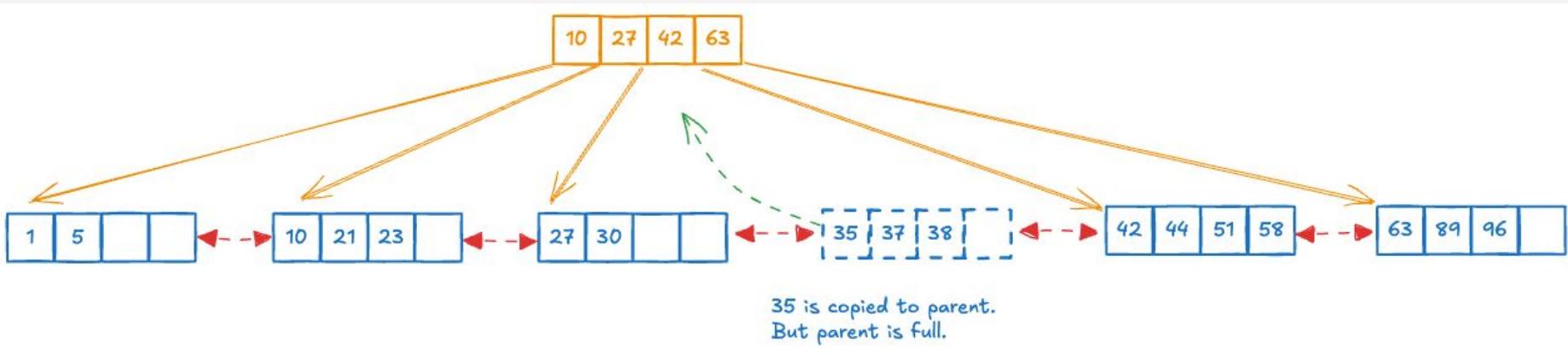
Fast forward to this state of the tree...



- Observation: The root node is full.
 - The next insertion that splits a leaf will cause the root to split, and thus the tree will get 1 level deeper.

Insert 37. Step 1

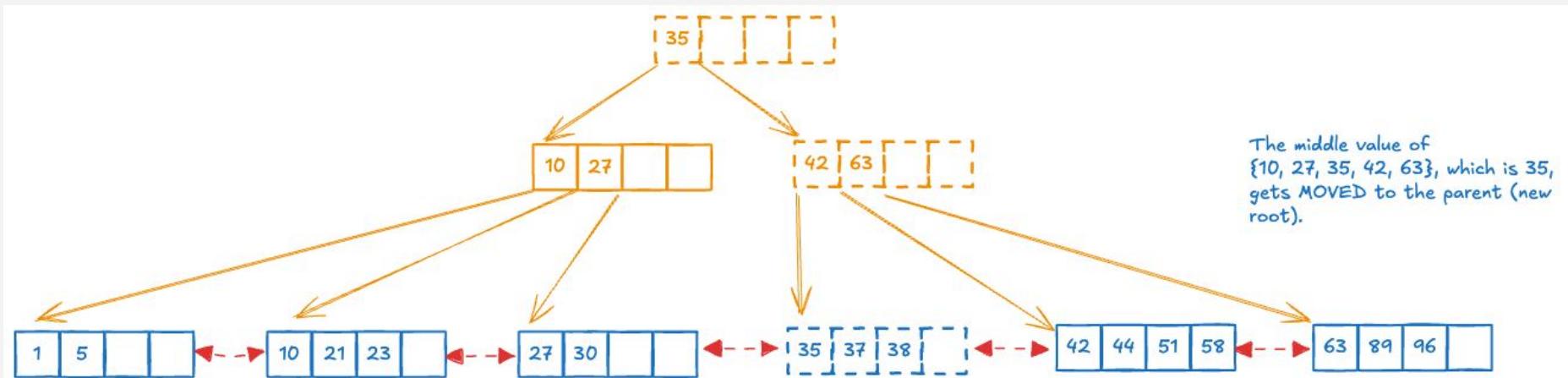
B+ Tree : $m = 4$



Insert 37. Step 2.

B+ Tree : $m = 4$

- When splitting an internal node, we **move** the middle element to the parent (instead of copying it).
- In this particular tree, that means we have to create a new internal node which is also now the root.



DS 4300

NoSQL & KV DBs

Mark Fontenot, PhD
Northeastern University

Distributed DBs and ACID - Pessimistic Concurrency

- ACID transactions
 - Focuses on “data safety”
 - considered a pessimistic concurrency model because it assumes one transaction has to protect itself from other transactions
 - IOW, it assumes that if something can go wrong, it will.
 - Conflicts are prevented by locking resources until a transaction is complete (there are both read and write locks)
 - Write Lock Analogy → borrowing a book from a library... If you have it, no one else can.

See <https://www.freecodecamp.org/news/how-databases-guarantee-isolation> for more for a deeper dive.

Optimistic Concurrency

- Transactions do not obtain locks on data when they read or write
- *Optimistic* because it assumes conflicts are unlikely to occur
 - Even if there is a conflict, everything will still be OK.
- But how?
 - Add last update timestamp and version number columns to every table... read them when changing. THEN, check at the end of transaction to see if any other transaction has caused them to be modified.

Optimistic Concurrency

- Low Conflict Systems (backups, analytical dbs, etc.)
 - Read heavy systems
 - the conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict.
 - So, optimistic concurrency works well - allows for higher concurrency
- High Conflict Systems
 - rolling back and rerunning transactions that encounter a conflict → less efficient
 - So, a locking scheme (pessimistic model) might be preferable

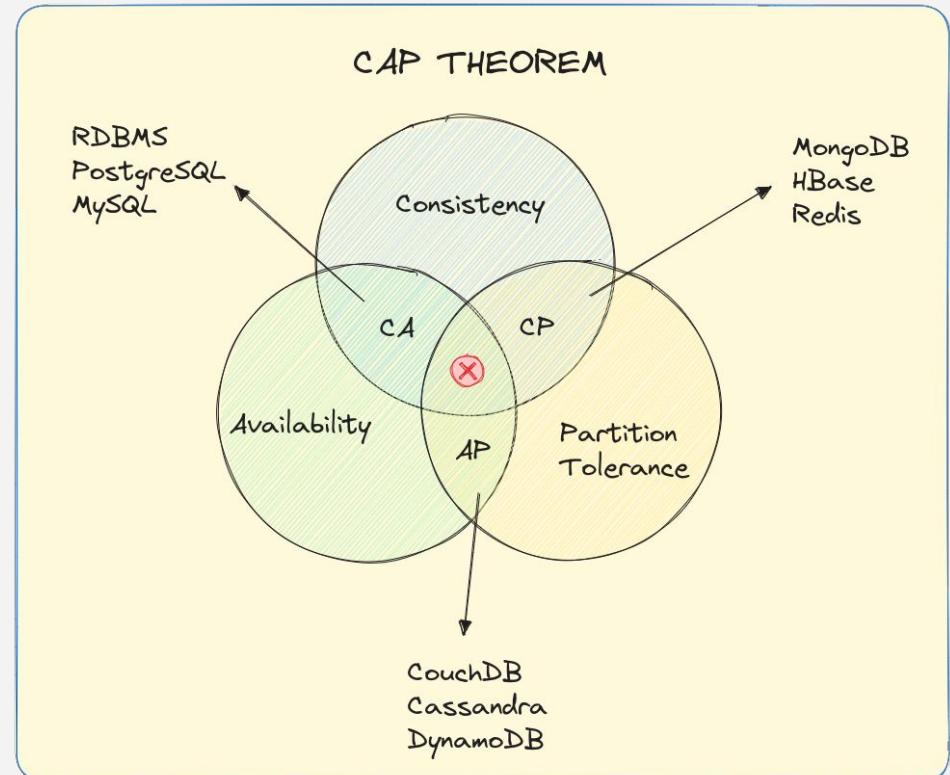
NoSQL

- “NoSQL” first used in 1998 by Carlo Strozzi to describe his relational database system that *did not use SQL*.
- More common, modern meaning is “Not Only SQL”
- *But*, sometimes thought of as non-relational DBs
- Idea originally developed, in part, as a response to processing unstructured web-based data.

CAP Theorem Review

You can have 2, but not 3, of the following:

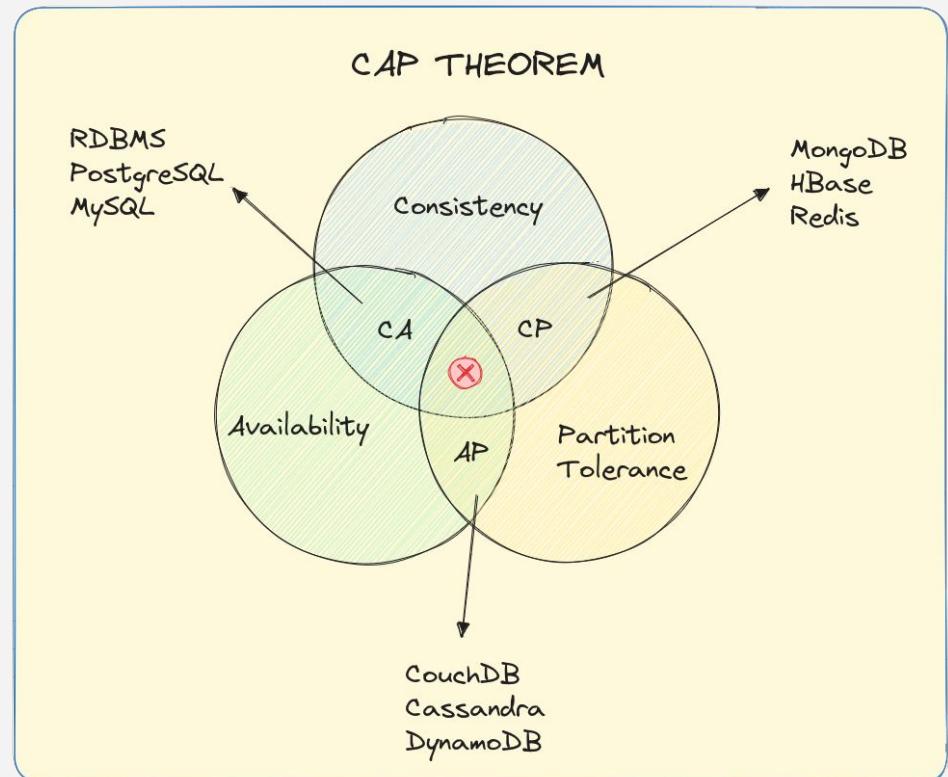
- **Consistency***: Every user of the DB has an identical view of the data at any given instant
- **Availability**: In the event of a failure, the database system remains operational
- **Partition Tolerance**: The database can maintain operations in the event of the network's failing between two segments of the distributed system



* Note, the definition of Consistency in CAP is different from that of ACID.

CAP Theorem Review

- **Consistency + Availability:** System always responds with the latest data and every request gets a response, but may not be able to deal with network partitions
- **Consistency + Partition Tolerance:** If system responds with data from the distrib. system, it is always the latest, else data request is dropped.
- **Availability + Partition Tolerance:** System always sends responses based on distributed store, but may not be the absolute latest data.



ACID Alternative for Distrib Systems - BASE

- **Basically Available**

- Guarantees the availability of the data (per CAP), but response can be “failure”/“unreliable” because the data is in an inconsistent or changing state
- System appears to work most of the time

ACID Alternative for Distrib Systems - BASE

- **Soft State** - The state of the system could change over time, even w/o input. Changes could be result of *eventual consistency*.
 - Data stores don't have to be write-consistent
 - Replicas don't have to be mutually consistent

ACID Alternative for Distrib Systems - BASE

- Eventual Consistency - The system will eventually become consistent
 - All writes will eventually stop so all nodes/replicas can be updated

Categories of NoSQL DBs - Review

First Up → Key-Value Databases

Key Value Stores

key = value

- Key-value stores are designed around:
 - *simplicity*
 - the data model is extremely simple
 - comparatively, tables in a RDBMS are very complex.
 - lends itself to simple CRUD ops and API creation

Key Value Stores

key = value

- Key-value stores are designed around:
 - *speed*
 - usually deployed as in-memory DB
 - retrieving a *value* given its *key* is typically a O(1) op b/c hash tables or similar data structs used under the hood
 - no concept of complex queries or joins... they slow things down

Key Value Stores

key = value

- Key-value stores are designed around:
 - *scalability*
 - Horizontal Scaling is simple - add more nodes
 - Typically concerned with *eventual consistency*, meaning in a distributed environment, the only guarantee is that all nodes will *eventually* converge on the same value.

KV DS Use Cases

- EDA/Experimentation Results Store
 - store intermediate results from data preprocessing and EDA
 - store experiment or testing (A/B) results w/o prod db
- Feature Store
 - store frequently accessed feature → low-latency retrieval for model training and prediction
- Model Monitoring
 - store key metrics about performance of model, for example, in real-time inferencing.

KV SWE Use Cases

- Storing Session Information
 - everything about the current *session* can be stored via a single PUT or POST and retrieved with a single GET VERY Fast
- User Profiles & Preferences
 - User info could be obtained with a single GET operation... language, TZ, product or UI preferences
- Shopping Cart Data
 - Cart data is tied to the user
 - needs to be available across browsers, machines, sessions
- Caching Layer:
 - In front of a disk-based database

Redis DB

- Redis (Remote Directory Server)
 - Open source, in-memory database
 - Sometimes called a data structure store
 - Primarily a KV store, but can be used with other models: Graph, Spatial, Full Text Search, Vector, Time Series
 - From db-engines.com Ranking of KV Stores:

Rank			DBMS	Database Model	Score		
Oct 2024	Sep 2024	Oct 2023			Oct 2024	Sep 2024	Oct 2023
1.	1.	1.	Redis	Key-value, Multi-model	149.63	+0.20	-13.33
2.	2.	2.	Amazon DynamoDB	Multi-model	71.85	+1.78	-9.07
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	24.50	-0.47	-9.80
4.	4.	4.	Memcached	Key-value	17.79	+0.95	-3.05
5.	5.	5.	etcd	Key-value	7.17	+0.12	-1.57
6.	↑7.	↑8.	Aerospike	Multi-model	5.57	+0.41	-0.86
7.	↓6.	↓6.	Hazelcast	Key-value, Multi-model	5.57	-0.16	-2.60
8.	8.	↓7.	Fhcache	Key-value	4.76	-0.03	-1.79

- It is considered an in-memory database system, but...
 - Supports durability of data by: a) essentially saving snapshots to disk at specific intervals or b) append-only file which is a journal of changes that can be used for roll-forward if there is a failure
- Originally developed in 2009 in C++
- Can be very fast ... > 100,000 SET ops / second
- Rich collection of commands
- Does NOT handle complex data. No secondary indexes.
Only supports lookup by Key.

Redis Data Types

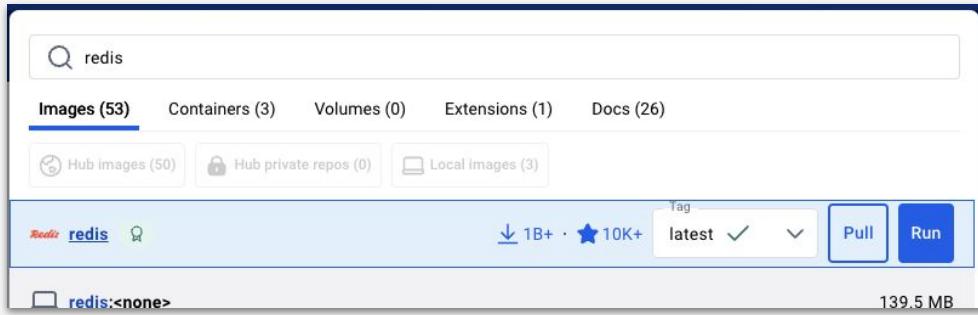
Keys:

- usually strings but can be any binary sequence

Values:

- Strings
- Lists (linked lists)
- Sets (unique unsorted string elements)
- Sorted Sets
- Hashes (string → string)
- Geospatial data

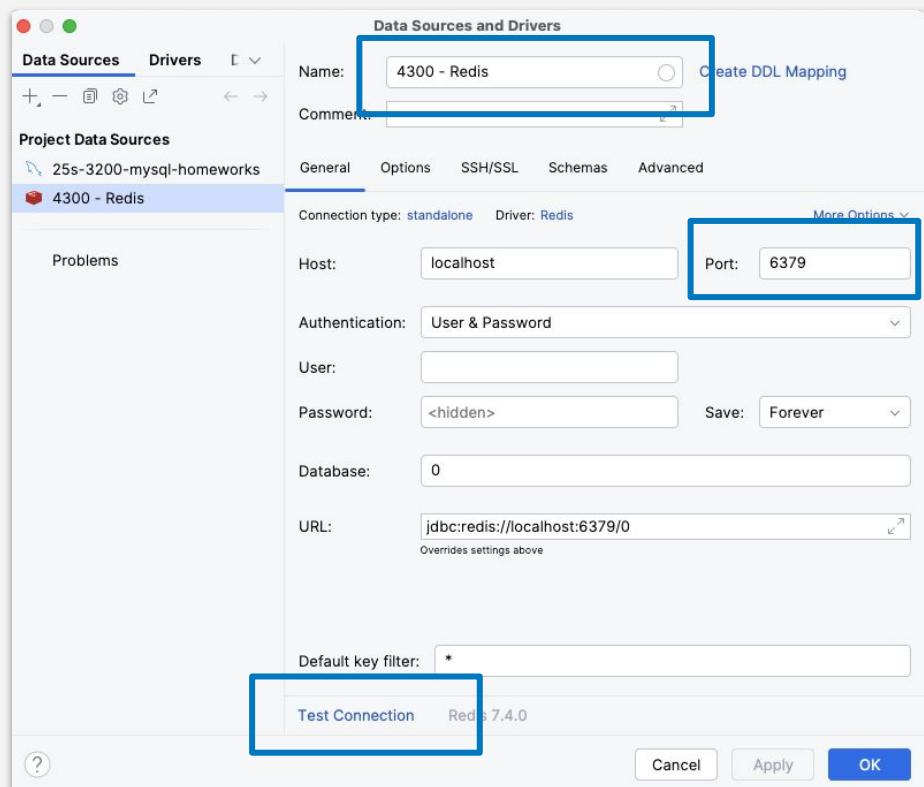
Setting Up Redis in Docker



- In Docker Desktop, search for Redis.
- Pull/Run the latest image (see above)
 - Optional Settings: add **6379** to Ports to expose that port so we can connect to it.
- Normally, you **would not** expose the Redis port for security reasons
 - If you did this in a prod environment, major security hole.
 - Notice, we didn't set a password...

Connecting from DataGrip

- File > New > Data Source > Redis
- Give the Data Source a Name
- Make sure the port is 6379
- Test the connection 



Redis Database and Interaction

- Redis provides 16 databases by default
 - They are numbered 0 to 15
 - There is no other name associated
- Direct interaction with Redis is through a set of commands related to setting and getting k/v pairs (and variations)
- Many language libraries available as well.

```
SET ds4300 "I Love AVL Trees!"
```

```
SET cs3200 "I Love SQL!"
```

```
KEYS *
```

```
GET cs3200
```

```
DEL ds4300
```

Foundation Data Type - String

- Sequence of bytes - text, serialized objects, bin arrays
- Simplest data type
- Maps a string to another string
- Use Cases:
 - caching frequently accessed HTML/CSS/JS fragments
 - config settings, user settings info, token management
 - counting web page/app screen views OR rate limiting

Some Initial Basic Commands

- **SET** /path/to/resource 0
SET user:1 "John Doe"
GET /path/to/resource
EXISTS user:1
DEL user:1
KEYS user*
- **SELECT** 5
 - select a different database

Some Basic Commands

- **SET** someValue 0
- **INCR** someValue #increment by 1
- **INCRBY** someValue 10 #increment by 10
- **DECR** someValue #decrement by 1
- **DECRBY** someValue 5 #decrement by 5
 - INCR parses the value as int and increments (or adds to value)
- **SETNX** key value
 - only sets value to key if key does not already exist

Hash Type

- Value of KV entry is a collection of *field-value* pairs
- Use Cases:
 - Can be used to represent basic objects/structures
 - number of field/value pairs per hash is $2^{32}-1$
 - practical limit: available system resources (e.g. memory)
 - Session information management
 - User/Event tracking (could include TTL)
 - Active Session Tracking (all sessions under one hash key)

Hash Commands

```
HSET bike:1 model Demios brand Ergonom price 1971
```

```
HGET bike:1 model
```

```
HGET bike:1 price
```

```
HGETALL bike:1
```

```
HMGET bike:1 model price weight
```

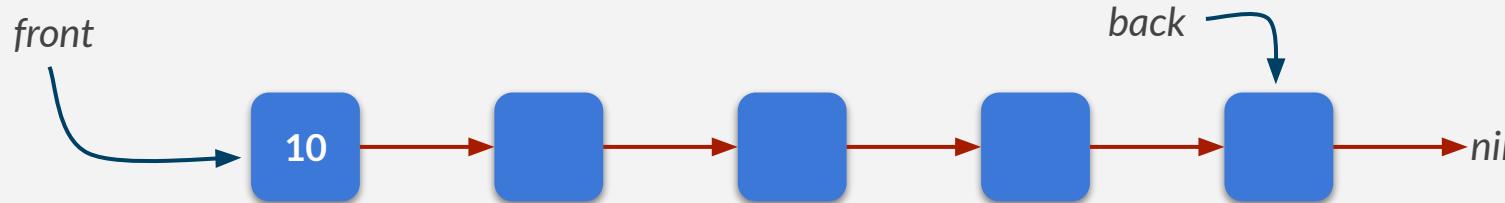
```
HINCRBY bike:1 price 100
```

What is returned?

List Type

- Value of KV Pair is **linked lists** of string values
- Use Cases:
 - implementation of stacks and queues
 - queue management & message passing queues (producer/consumer model)
 - logging systems (easy to keep in chronological order)
 - build social media streams/feeds
 - message history in a chat application
 - batch processing by queueing up a set of tasks to be executed sequentially at a later time

Linked Lists Crash Course



- Sequential data structure of linked nodes (instead of contiguously allocated memory)
- Each node points to the next element of the list (except the last one - points to *nil/null*)
- $O(1)$ to insert new value at front or insert new value at end

List Commands - Queue

Queue-like Ops

LPUSH bikes:repairs bike:1

LPUSH bikes:repairs bike:2

RPOP bikes:repairs

RPOP biles:repairs

List Commands - Stack

Stack-like Ops

```
LPUSH bikes:repairs bike:1
```

```
LPUSH bikes:repairs bike:2
```

```
LPOP bikes:repairs
```

```
LPOP biles:repairs
```

List Commands - Others

Other List Ops

LLEN mylist

LPUSH mylist "one"
LPUSH mylist "two"
LPUSH mylist "three"

LRANGE <key> <start> <stop>

LRANGE mylist 0 3

LRANGE mylist 0 0

LRANGE mylist -2 -1

1) "three"
2) "two"
3) "one"

1) "three"

1) "two"
2) "one"

JSON Type

- Full support of the JSON standard
- Uses JSONPath syntax for parsing/navigating a JSON document
- Internally, stored in binary in a tree-structure → fast access to sub elements

Set Type

- Unordered collection of unique strings (members)
- Use Cases:
 - track unique items (IP addresses visiting a site, page, screen)
 - primitive relation (set of all students in DS4300)
 - access control lists for users and permission structures
 - social network friends lists and/or group membership
- Supports set operations!!

Set Commands

```
SADD ds4300 "Mark"
```

```
SADD ds4300 "Sam"
```

```
SADD cs3200 "Nick"
```

```
SADD cs3200 "Sam"
```

```
SISMEMBER ds4300 "Mark"
```

```
SISMEMBER ds4300 "Nick"
```

```
SCARD ds4300
```

Set Commands

SADD ds4300 "Mark"
SADD ds4300 "Sam"
SADD cs3200 "Nick"
SADD cs3200 "Sam"

SCARD ds4300

SINTER ds4300 cs3200

SDIFF ds4300 cs3200

SREM ds4300 "Mark"

SRANDMEMBER ds4300

??
?

DS 4300

Redis in Docker Setup

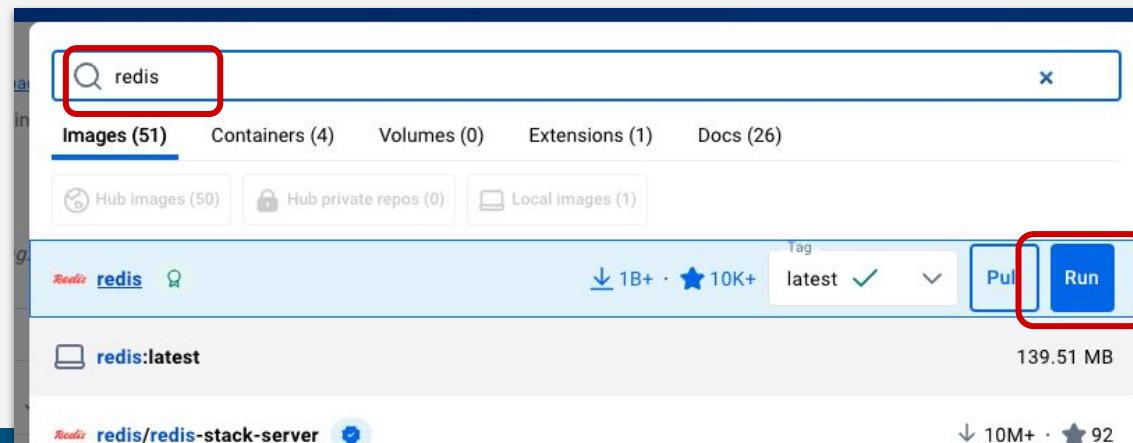
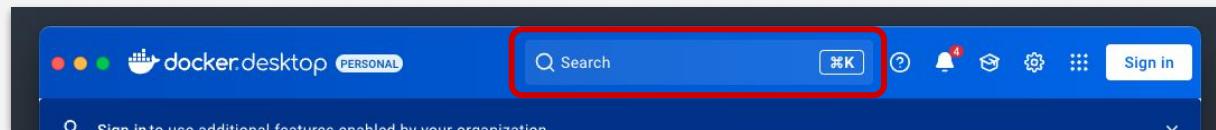
Mark Fontenot, PhD
Northeastern University

Pre-Requisites

- You have installed Docker Desktop
- You have installed Jetbrains DataGrip

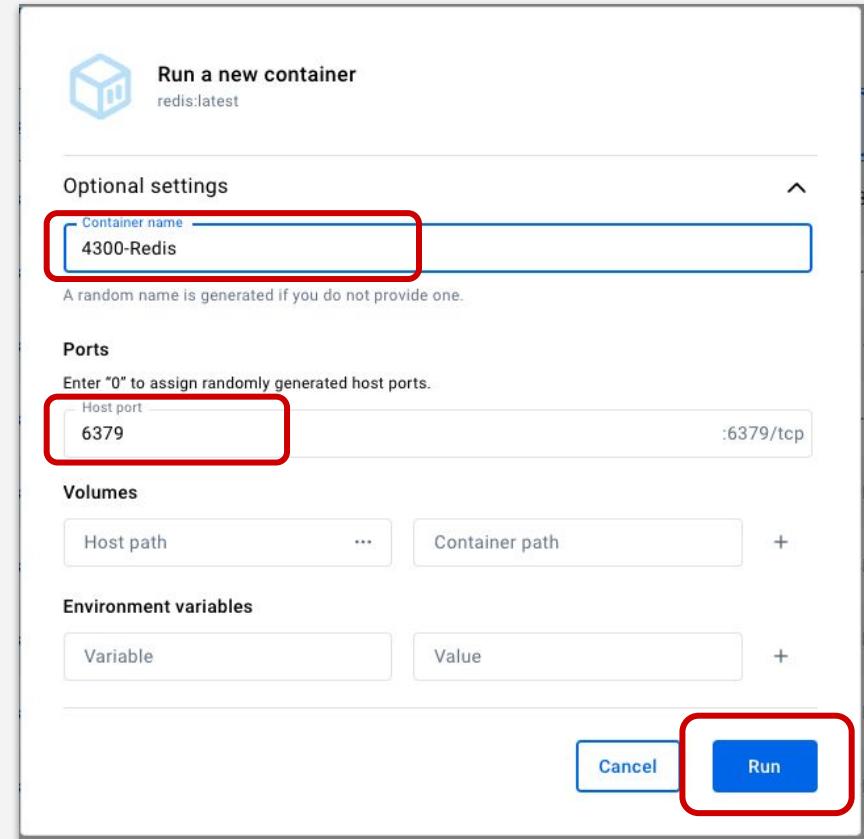
Step 1 - Find the Redis Image

- Open Docker Desktop
- Use the Built In search to find the Redis Image
- Click Run



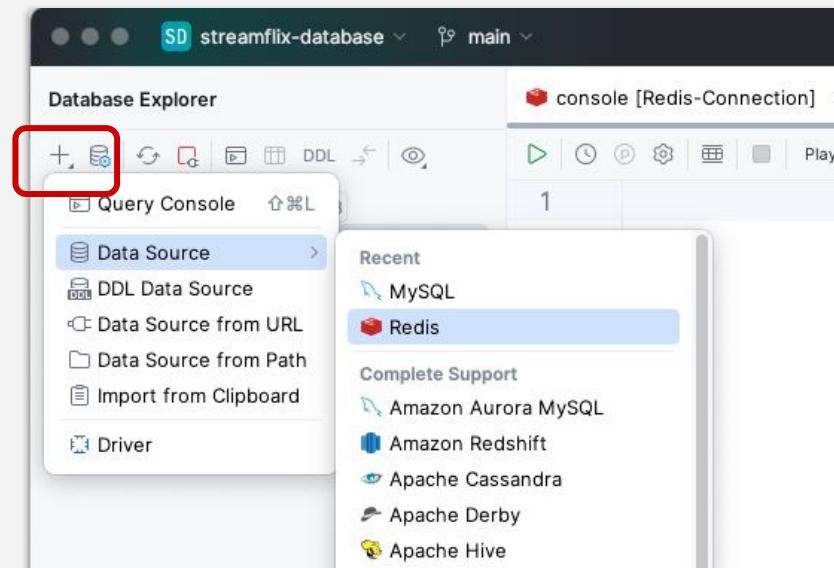
Step 2 - Configure & Run the Container

- Give the new container a name
- Enter **6379** in Host Port field
- Click Run
- *Give Docker some time to download and start Redis*



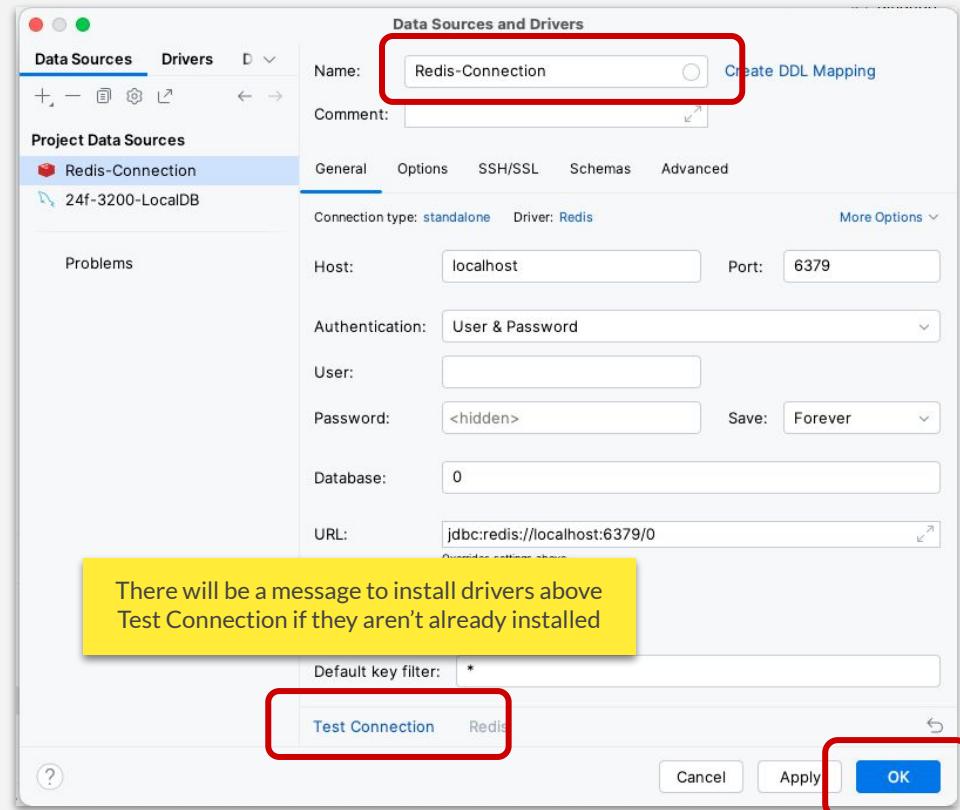
Step 3 - Set up Data Source in DataGrip

- Start DataGrip
- Create a new Redis Data Source
 - You can use the + in the Database Explorer OR
 - You can use New from the File Menu



Step 4 - Configure the Data Source

- Give the data source a name
- Install Drivers if needed (message above Test Connection)
- Test the Connection to Redis
- Click OK if connection test was successful



DS 4300

Redis + Python

Mark Fontenot, PhD
Northeastern University

Redis-py

- Redis-py is the standard client for Python.
- Maintained by the Redis Company itself
- GitHub Repo: [redis/redis-py](#)
- In your 4300 Conda Environment:

```
pip install redis
```

Connecting to the Server

```
import redis
redis_client = redis.Redis(host='localhost',
                           port=6379,
                           db=2,
                           decode_responses=True)
```

- For your Docker deployment, host could be *localhost* or *127.0.0.1*
- Port is the port mapping given when you created the container (probably the default 6379)
- db is the database 0-15 you want to connect to
- decode_responses → data comes back from server as bytes. Setting this true converter them (decodes) to strings.

Redis Command List

- Full List > [here](#) <
- Use Filter to get to command for the particular data structure you're targeting (list, hash, set, etc.)
- Redis.py Documentation > [here](#) <
- The next slides are not meant to be an exhaustive list of commands, only some highlights. Check the documentation for a complete list.

String Commands

```
# r represents the Redis client object  
r.set('clickCount:/abc', 0)  
val = r.get('clickCount:/abc')  
r.incr('clickCount:/abc')  
ret_val = r.get('clickCount:/abc')  
print(f'click count = {ret_val}')
```

String Commands - 2

```
# r represents the Redis client object
redis_client.mset({'key1': 'val1',
                    'key2': 'val2',
                    'key3': 'val3'})
print(redis_client.mget('key1',
                        'key2',
                        'key3'))
# returns as list ['val1', 'val2', 'val3']
```

String Commands - 3

- set(), mset(), setex(), msetnx(), setnx()
- get(), mget(), getex(), getdel()
- incr(), decr(), incrby(), decrby()
- strlen(), append()

List Commands - 1

```
# create list: key = 'names'  
# values = ['mark', 'sam', 'nick']  
redis_client.rpush('names',  
                   'mark', 'sam', 'nick')  
  
# prints ['mark', 'sam', 'nick']  
print(redis_client.lrange('names', 0, -1))
```

List Commands - 2

- `lpush()`, `lpop()`, `lset()`, `lrem()`
- `rpush()`, `rpop()`
- `lrange()`, `llen()`, `lpos()`
- Other commands include moving elements between lists, popping from multiple lists at the same time, etc.

Hash Commands - 1

```
redis_client.hset('user-session:123',  
    mapping={'first': 'Sam',  
              'last': 'Uelle',  
              'company': 'Redis',  
              'age': 30  
})
```

```
# prints:  
#{'name': 'Sam', 'surname': 'Uelle', 'company': 'Redis', 'age': '30'}  
print(redis_client.hgetall('user-session:123'))
```

Hash Commands - 2

- hset(), hget(), hgetall()
- hkeys()
- hdel(), hexists(), hlen(), hstrlen()

Redis Pipelines

- Helps avoid multiple related calls to the server → less network overhead

```
r = redis.Redis(decode_responses=True)
pipe = r.pipeline()

for i in range(5):
    pipe.set(f"seat:{i}", f"#{i}")

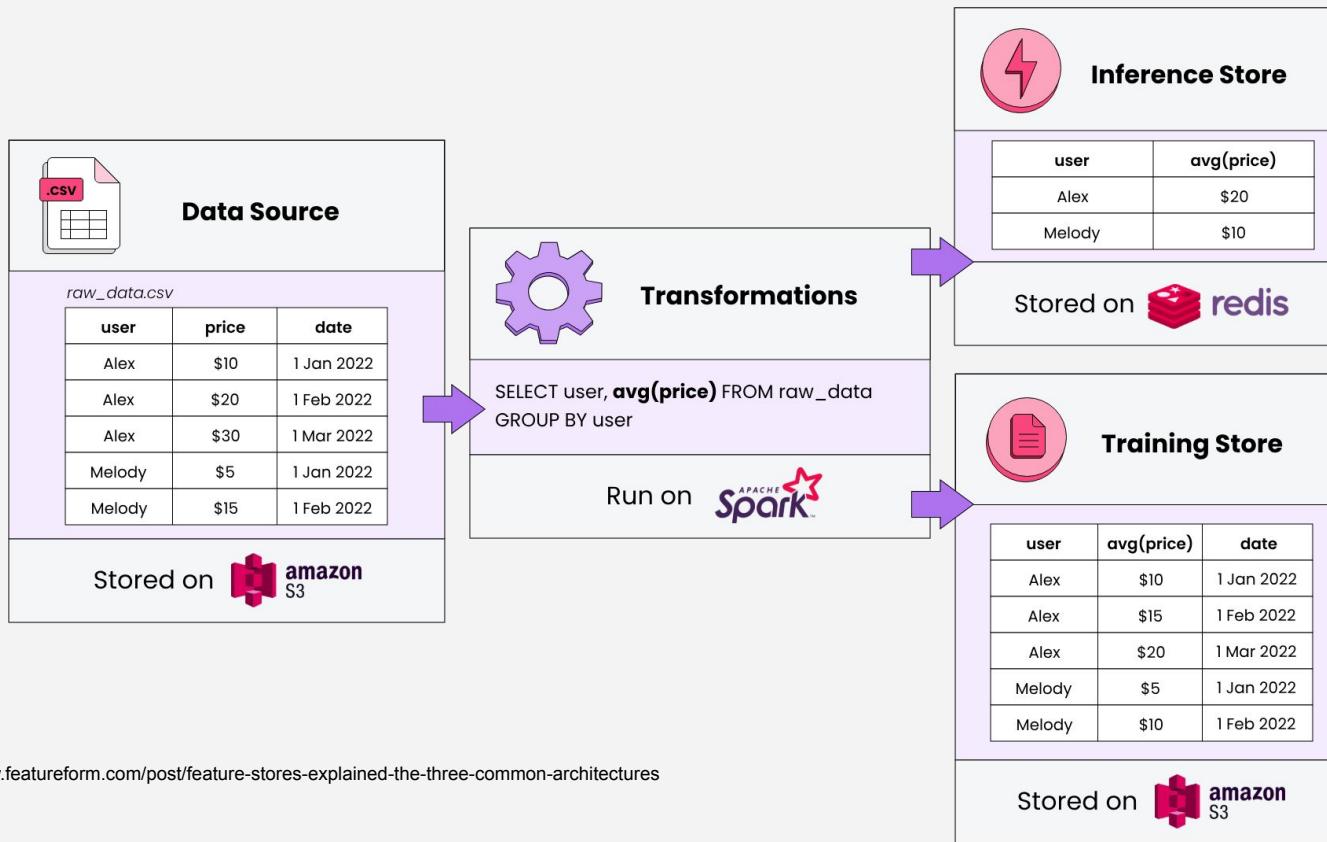
set_5_result = pipe.execute()
print(set_5_result) # >>> [True, True, True, True, True]

pipe = r.pipeline()

# "Chain" pipeline commands together.
get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
print(get_3_result) # >>> ['#0', '#3', '#4']
```

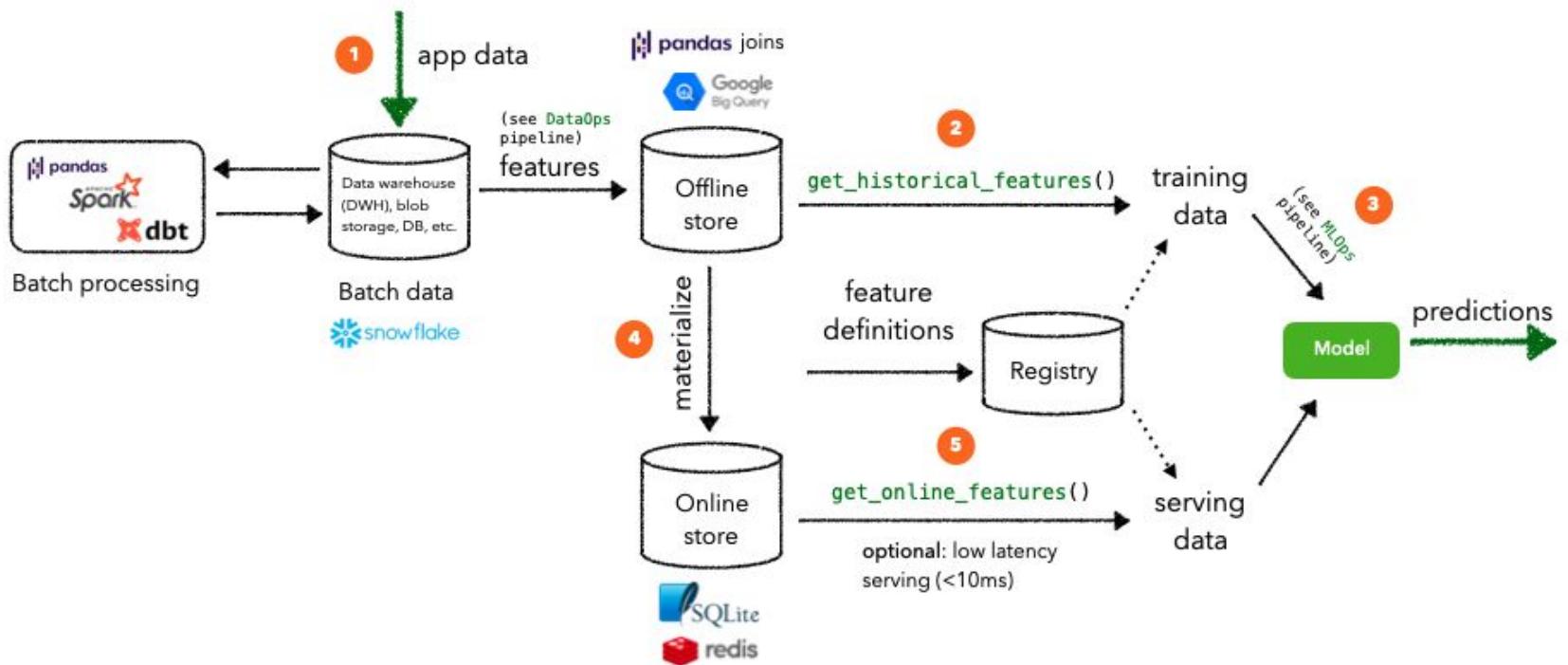
Redis in Context

Redis in ML - Simplified Example



Source: <https://www.featureform.com/post/feature-stores-explained-the-three-common-architectures>

Redis in DS/ML



Source: <https://madewithml.com/courses/mlops/feature-store/>

DS 4300

Document Databases & MongoDB

Mark Fontenot, PhD
Northeastern University

Document Database

A Document Database is a non-relational database that stores data as structured documents, usually in JSON.

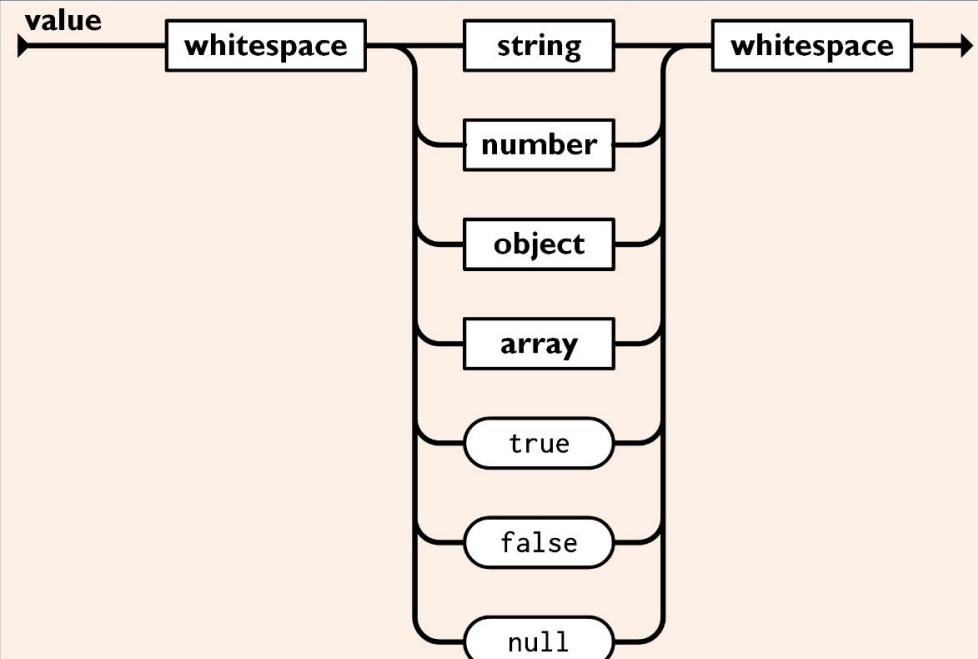
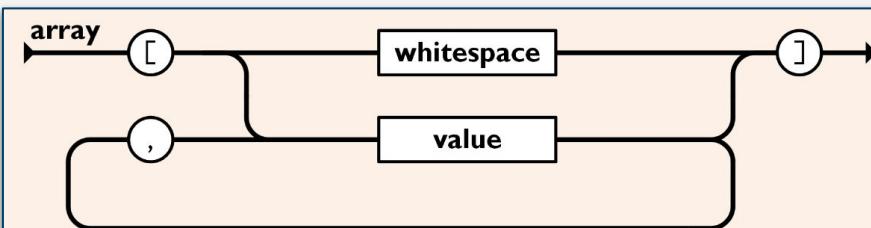
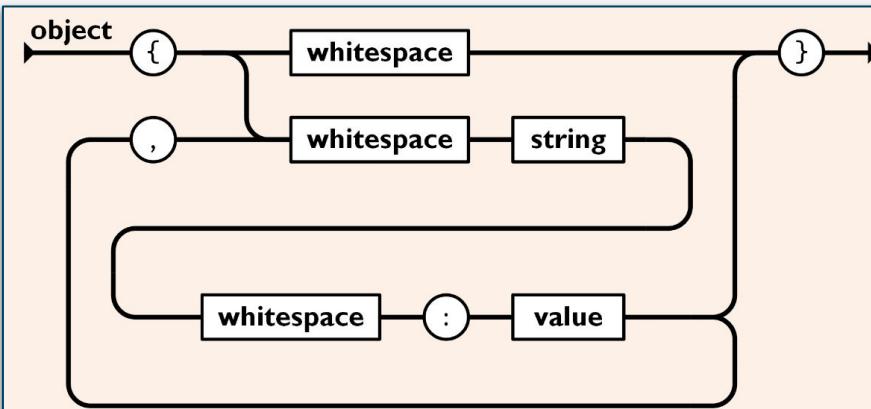
They are designed to be *simple, flexible, and scalable*.

```
{  
  "orders": [  
    {  
      "orderno": "748745375",  
      "date": "June 30, 2088 1:54:23 AM",  
      "trackingno": "TN0039291",  
      "custid": "11045",  
      "customer": {  
        "custid": "11045",  
        "fname": "Sue",  
        "lname": "Hatfield",  
        "address": "1409 Silver Street",  
        "city": "Ashland",  
        "state": "NE",  
        "zip": "68003"  
      }  
    }  
  ]  
}
```

What is JSON?

- **JSON (JavaScript Object Notation)**
 - a lightweight data-interchange format
 - It is easy for humans to read and write.
 - It is easy for machines to parse and generate.
- **JSON is built on two structures:**
 - A **collection of name/value pairs**. In various languages, this is operationalized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
 - An **ordered list of values**. In most languages, this is operationalized as an array, vector, list, or sequence.
- These are two *universal data structures* supported by virtually all modern programming languages
 - Thus, JSON makes a great data interchange format.

JSON Syntax



Binary JSON? BSON

- **BSON → Binary JSON**

- binary-encoded serialization of a JSON-like document structure
- supports extended types not part of basic JSON (e.g. Date, BinaryData, etc)
- **Lightweight** - keep space overhead to a minimum
- **Traversable** - designed to be easily traversed, which is vitally important to a document DB
- **Efficient** - encoding and decoding *must* be efficient
- Supported by many modern programming languages

```
{"hello": "world"} →  
 \x16\x00\x00\x00          // total document size  
 \x02                      // 0x02 = type String  
 hello\x00                  // field name  
 \x06\x00\x00\x00world\x00  // field value  
 \x00                      // 0x00 = type E00 ('end of object')
```

XML (eXtensible Markup Language)

- Precursor to JSON as data exchange format
- XML + CSS → web pages that separated content and formatting
- Structurally similar to HTML, but tag set is extensible

```
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
</CATALOG>
```

XML-Related Tools/Technologies

- **Xpath** - a syntax for retrieving specific elements from an XML doc
- **Xquery** - a query language for *interrogating* XML documents; the *SQL* of XML
- **DTD** - Document Type Definition - a language for describing the allowed structure of an XML document
- **XSLT** - eXtensible Stylesheet Language Transformation - tool to transform XML into other formats, including non-XML formats such as HTML.

Why Document Databases?

- Document databases address the *impedance mismatch* problem between object persistence in OO systems and how relational DBs structure data.
 - OO Programming → Inheritance and Composition of types.
 - How do we save a complex object to a relational database?
We basically have to deconstruct it.
- The structure of a document is *self-describing*.
- They are well-aligned with apps that use JSON/XML as a transport layer

MongoDB

MongoDB

- Started in 2007 after Doubleclick was acquired by Google, and 3 of its veterans realized the limitations of relational databases for serving > 400,000 ads per second
- MongoDB was short for *Humongous Database*
- MongoDB Atlas released in 2016 → documentdb as a service

MongoDB Structure

Database

Collection A

Document 1

Document 2

Document 3

Collection B

Document 1

Document 2

Document 3

Collection C

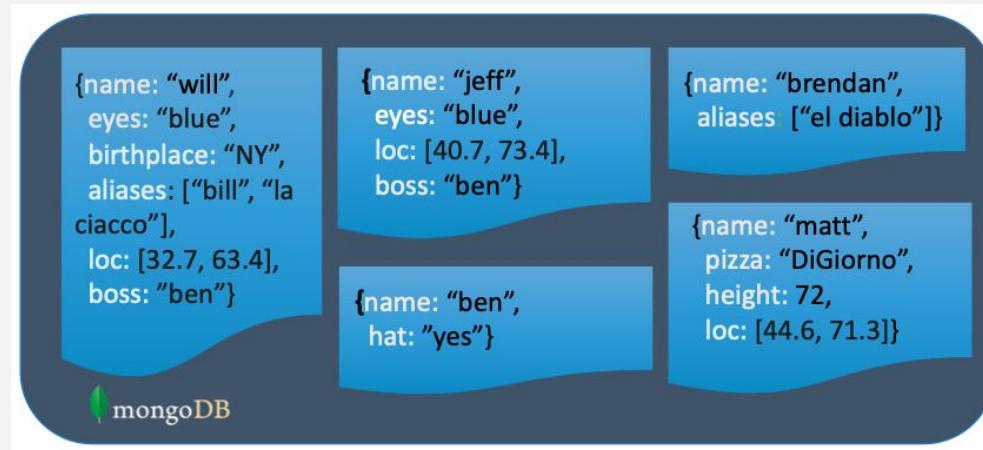
Document 1

Document 2

Document 3

MongoDB Documents

- No predefined schema for documents is needed
- Every document in a collection could have different data/schema



Relational vs Mongo/Document DB

RDBMS	MongoDB
Database	Database
Table/View	Collection
Row	Document
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference

MongoDB Features

- Rich Query Support - robust support for all CRUD ops
- Indexing - supports primary and secondary indices on document fields
- Replication - supports replica sets with automatic failover
- Load balancing built in

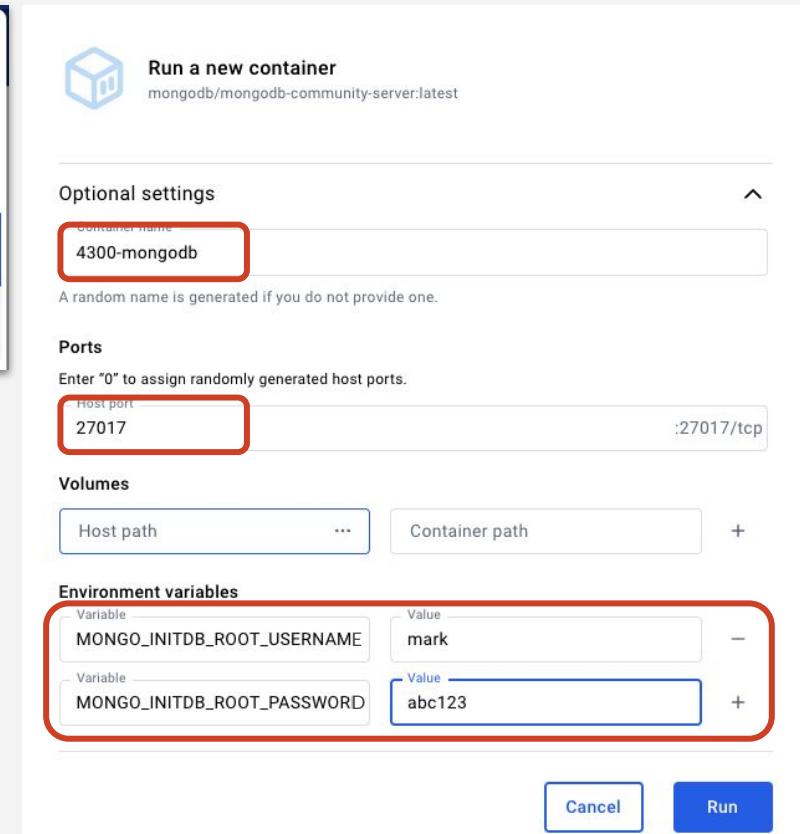
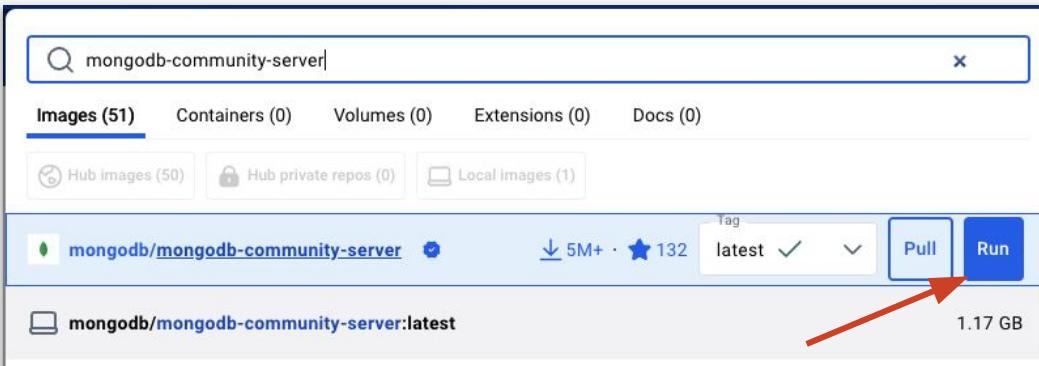
MongoDB Versions

- MongoDB Atlas
 - Fully managed MongoDB service in the cloud (DBaaS)
- MongoDB Enterprise
 - Subscription-based, self-managed version of MongoDB
- MongoDB Community
 - source-available, free-to-use, self-managed

Interacting with MongoDB

- **mongosh** → MongoDB Shell
 - CLI tool for interacting with a MongoDB instance
- MongoDB Compass
 - free, open-source GUI to work with a MongoDB database
- DataGrip and other 3rd Party Tools
- Every major language has a library to interface with MongoDB
 - PyMongo (Python), Mongoose (JavaScript/node), ...

Mongodb Community Edition in Docker



- Create a container
- Map host:container port 27017
- Give initial username and password for superuser

MongoDB Compass

- GUI Tool for interacting with MongoDB instance
- Download and install from > [here](#) <.

New Connection

Manage your connection settings

URI **Edit Connection String**

Name **Color**

Favorite this connection
Favoriting a connection will pin it to the top of your list of connections

Advanced Connection Options

Advanced Connection Options

General **Authentication** **TLS/SSL** **Proxy/SSH** **In-Use Encryption** **Advanced**

Authentication Method

Username/Password **OIDC** **X.509** **Kerberos** **LDAP** **AWS IAM**

Username Optional

Password Optional

Authentication Database Optional

Authentication Mechanism

Default **SCRAM-SHA-1** **SCRAM-SHA-256**

Save **Connect** **Save & Connect**

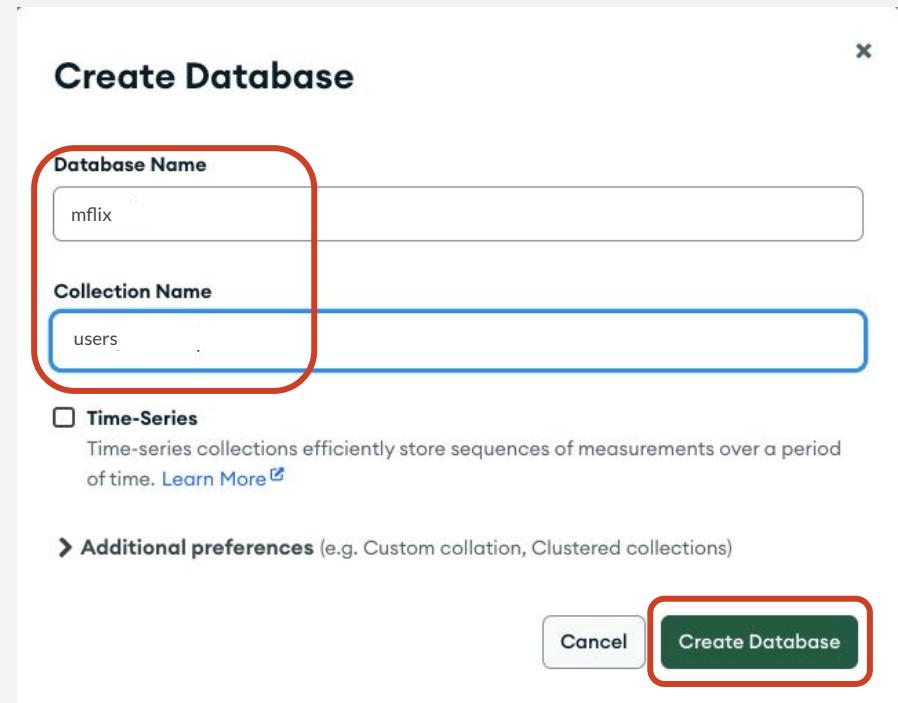


Load MFlix Sample Data Set

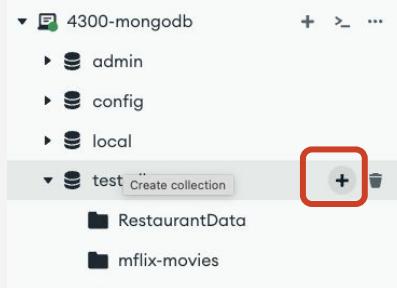
- In Compass, create a new Database named **mflix**
- Download [mflix sample dataset](#) and unzip it
- Import JSON files for users, theaters, movies, and comments into new collections in the mflix database

Creating a Database and Collection

To Create a new DB:



To Create a new Collection:



mongosh - Mongo Shell

- `find(...)` is like `SELECT`

```
collection.find({ ____ }, { ____ })
```

filters

projections

mongosh - find()

- SELECT * FROM users;

```
use mflix  
  
db.users.find()
```

-

```
SELECT *  
FROM users  
WHERE name = "Davos Seaworth";
```

mongosh - find()

filter

```
db.users.find({ "name": "Davos Seaworth" })
```

```
< {  
    _id: ObjectId('59b99dbecfa9a34dcd7885c9'),  
    name: 'Davos Seaworth',  
    email: 'liam_cunningham@gameofthron.es',  
    password: '$2b$12$jbgNowG97LHNIm4axwXDz.tkFITsmw/aylIY/lZDaJRgnHZjB029e'  
}
```

mongosh - find()

- SELECT *
FROM movies
WHERE rated in ("PG", "PG-13")

```
db.movies.find({rated: {$in: [ "PG", "PG-13" ]}})
```

mongosh - find()

- Return movies which were released in Mexico and have an IMDB rating of at least 7

```
db.movies.find( {  
    "countries": "Mexico",  
    "imdb.rating": { $gte: 7 }  
} )
```

mongosh - find()

- Return movies from the **movies** collection which were released in 2010 and either won at least 5 awards or have a genre of Drama

```
db.movies.find( {  
    "year": 2010,  
    $or: [  
        { "awards.wins": { $gte: 5 } },  
        { "genres": "Drama" }  
    ]  
})
```

Comparison Operators

Name	Description
\$eq	Matches values that are equal to a specified value.
\$gt	Matches values that are greater than a specified value.
\$gte	Matches values that are greater than or equal to a specified value.
\$in	Matches any of the values specified in an array.
\$lt	Matches values that are less than a specified value.
\$lte	Matches values that are less than or equal to a specified value.
\$ne	Matches all values that are not equal to a specified value.
\$nin	Matches none of the values specified in an array.

mongosh - countDocuments()

- How many movies from the **movies** collection were released in 2010 and either won at least 5 awards or have a genre of Drama

```
db.movies.countDocuments( {  
    "year": 2010,  
    $or: [  
        { "awards.wins": { $gte: 5 } },  
        { "genres": "Drama" }  
    ]  
})
```

mongosh - project

- Return the names of all movies from the **movies** collection that were released in 2010 and either won at least 5 awards or have a genre of Drama

```
db.movies.countDocuments( {  
    "year": 2010,  
    $or: [  
        { "awards.wins": { $gte: 5 } },  
        { "genres": "Drama" }  
    ]  
}, {"name": 1, "_id": 0} )
```



1 = return; 0 = don't return

PyMongo

PyMongo

- PyMongo is a Python library for interfacing with MongoDB instances

```
from pymongo import MongoClient  
  
client = MongoClient(  
    'mongodb://user_name:pw@localhost:27017'  
)
```

Getting a Database and Collection

```
from pymongo import MongoClient
client = MongoClient(
    'mongodb://user_name:pw@localhost:27017'
)

db = client['ds4300']
collection = db['myCollection']
```

Inserting a Single Document

```
db = client['ds4300']
collection = db['myCollection']

post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

post_id = collection.insert_one(post).inserted_id
print(post_id)
```

Count Documents in Collection

- SELECT count(*) FROM collection

```
demodb.collection.count_documents({})
```

??
• •

DS 4300

MongoDB + PyMongo

Mark Fontenot, PhD
Northeastern University

PyMongo

- PyMongo is a Python library for interfacing with MongoDB instances

```
from pymongo import MongoClient  
  
client = MongoClient(  
    'mongodb://user_name:pw@localhost:27017'  
)
```

Getting a Database and Collection

```
from pymongo import MongoClient

client = MongoClient(
    'mongodb://user_name:pw@localhost:27017'
)

db = client['ds4300'] # or client.ds4300
collection = db['myCollection'] #or db.myCollection
```

Inserting a Single Document

```
db = client['ds4300']
collection = db['myCollection']

post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

post_id = collection.insert_one(post).inserted_id
print(post_id)
```

Find all Movies from 2000

```
from bson.json_util import dumps  
  
# Find all movies released in 2000  
movies_2000 = db.movies.find({"year": 2000})  
  
# Print results  
print(dumps(movies_2000, indent = 2))
```

Jupyter Time

- Activate your DS4300 conda or venv python environment
- Install pymongo with **pip install pymongo**
- Install **Jupyter Lab** in you python environment
 - **pip install jupyterlab**
- Download and unzip > [this](#) < zip file - contains 2 Jupyter Notebooks
- In terminal, navigate to the folder where you unzipped the files, and run **jupyter lab**

??
• •

DS 4300

Introduction to the Graph Data Model

Mark Fontenot, PhD
Northeastern University

What is a Graph Database

- Data model based on the graph data structure
- Composed of nodes and edges
 - edges connect nodes
 - each is uniquely identified
 - each can contain properties (e.g. name, occupation, etc)
 - supports queries based on graph-oriented operations
 - traversals
 - shortest path
 - *lots of others*

Where do Graphs Show up?

- Social Networks
 - yes... things like Instagram,
 - but also... modeling social interactions in fields like psychology and sociology
- The Web
 - it is just a big graph of “pages” (nodes) connected by hyperlinks (edges)
- Chemical and biological data
 - systems biology, genetics, etc.
 - interaction relationships in chemistry

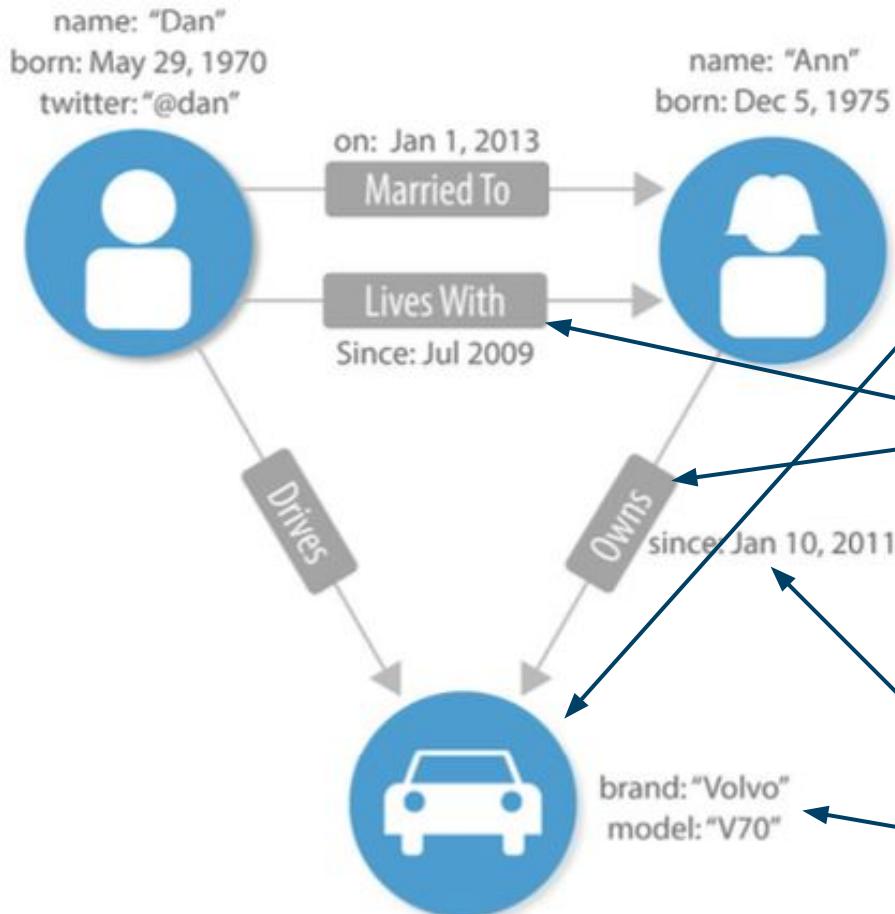
Basics of Graphs and Graph Theory

What is a graph?

Labeled Property Graph

- Composed of a set of node (vertex) objects and relationship (edge) objects
- Labels are used to mark a node as part of a group
- Properties are attributes (think KV pairs) and can exist on nodes and relationships
- Nodes with no associated relationships are OK.
Edges not connected to nodes are not permitted.

Example



2 Labels:

- person
- car

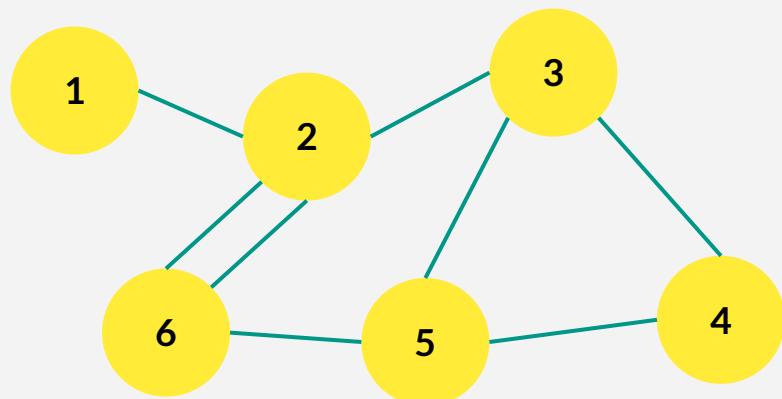
4 relationship types:

- Drives
- Owns
- Lives_with
- Married_to

Properties

Paths

A *path* is an ordered sequence of nodes connected by edges in which no nodes or edges are repeated.



Ex: $1 \rightarrow 2 \rightarrow 6 \rightarrow 5$

Not a path:
 $1 \rightarrow 2 \rightarrow 6 \rightarrow 2 \rightarrow 3$

Flavors of Graphs

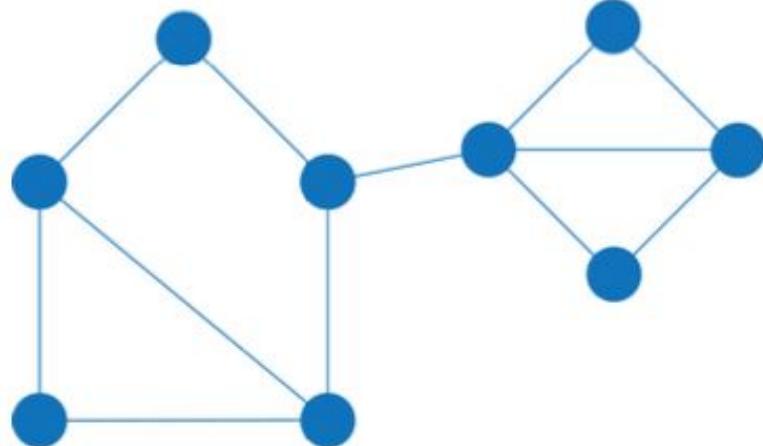
Connected (vs. Disconnected) – there is a path between any two nodes in the graph

Weighted (vs. Unweighted) – edge has a weight property (important for some algorithms)

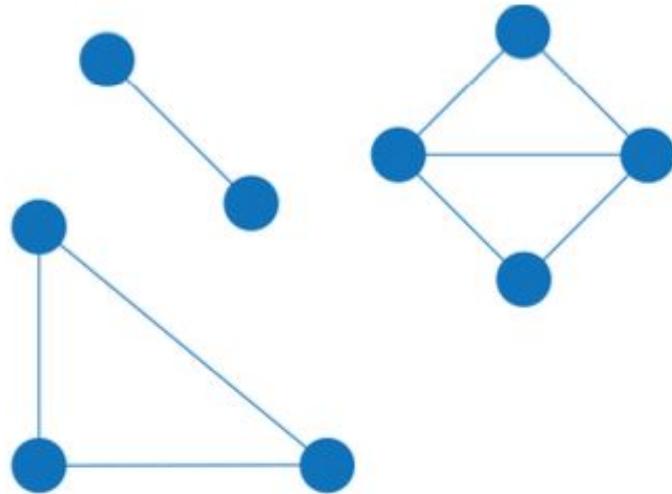
Directed (vs. Undirected) – relationships (edges) define a start and end node

Acyclic (vs. Cyclic) – Graph contains no cycles

Connected vs. Disconnected

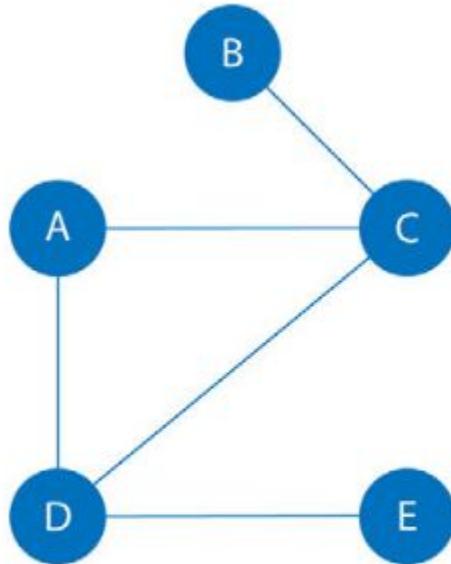


Connected Graph

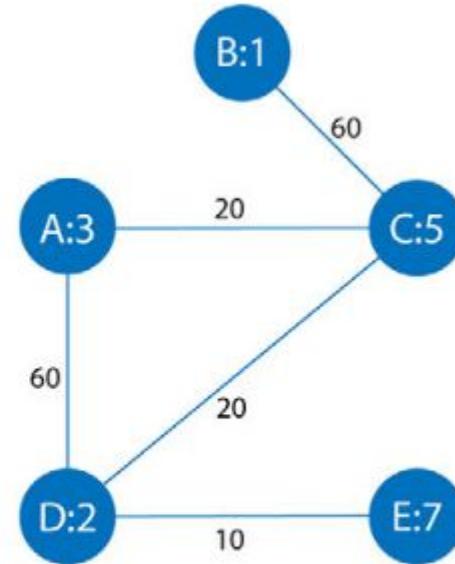


Disconnected Graph
Includes 3 components.

Weighted vs. Unweighted

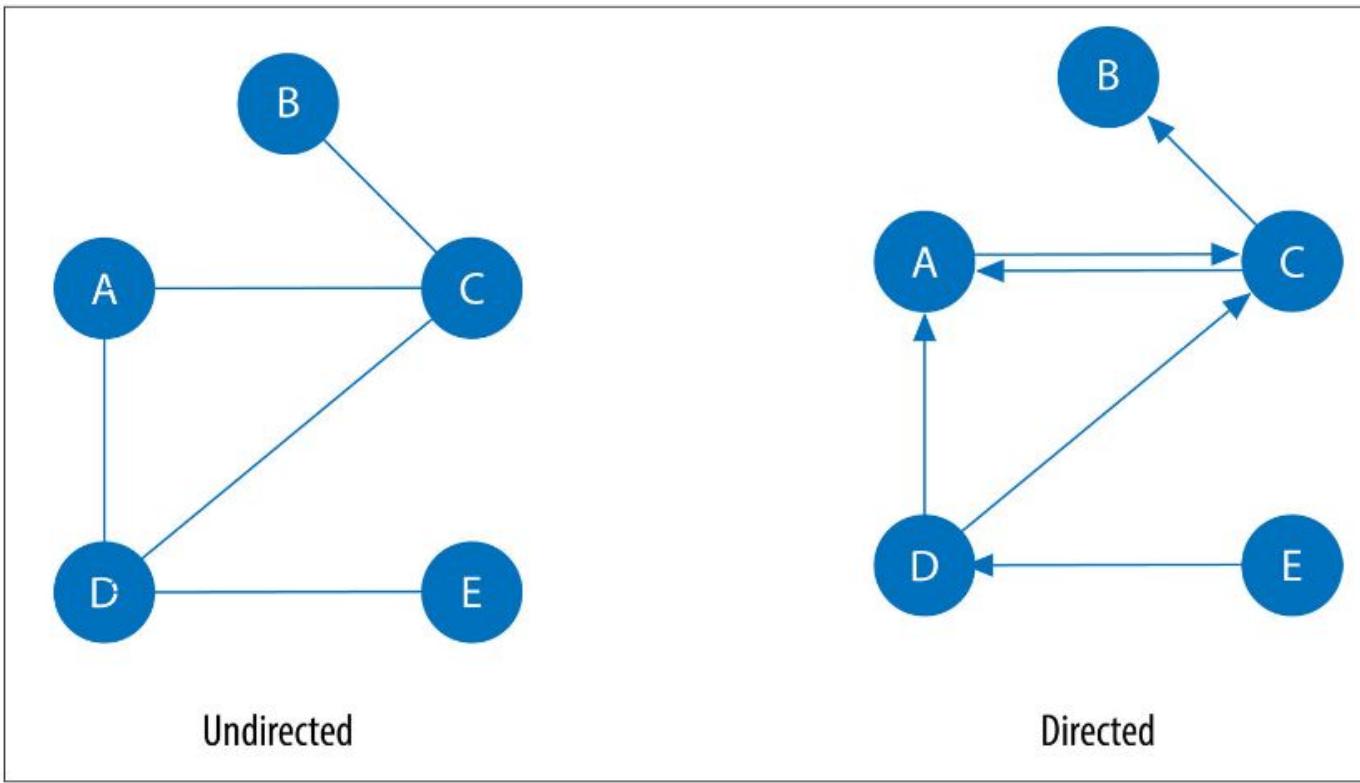


Unweighted

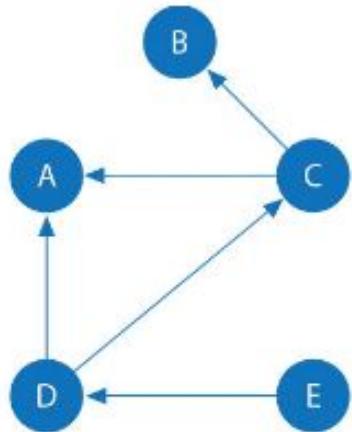


Weighted

Directed vs. Undirected

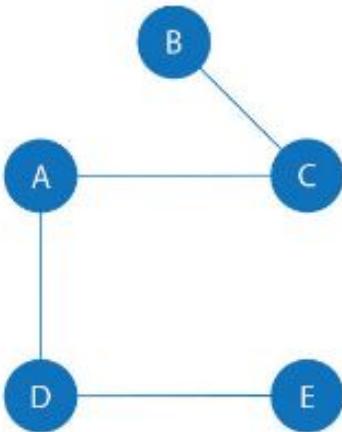


Cyclic vs Acyclic

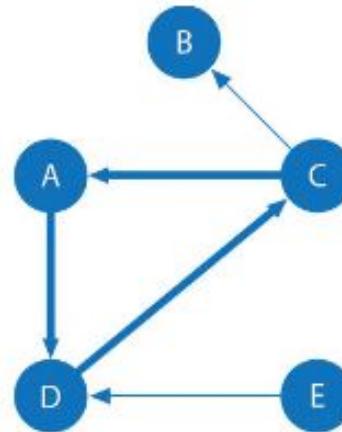


Graph 1

Acyclic

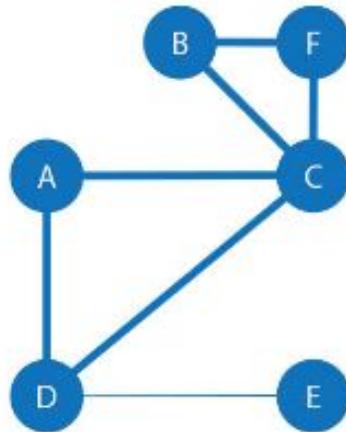


Graph 2



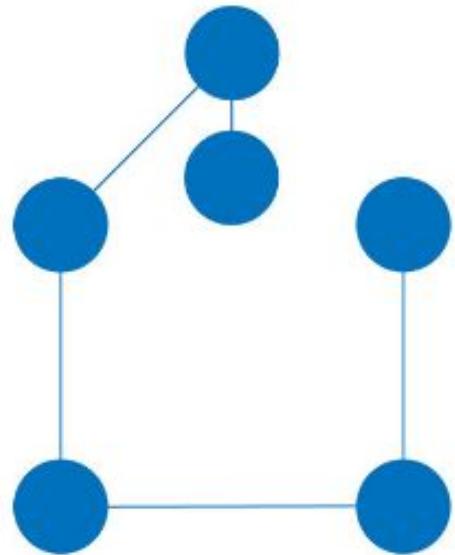
Graph 3

Cyclic

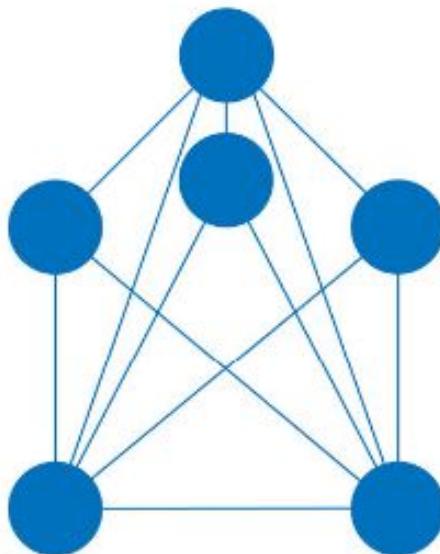


Graph 4

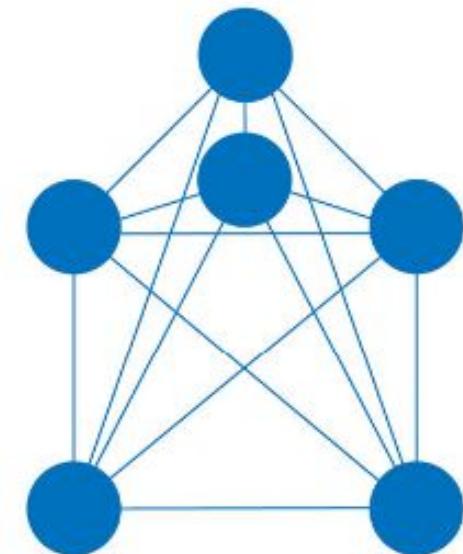
Sparse vs. Dense



Sparse

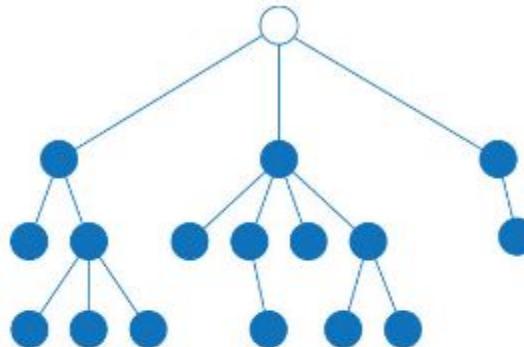


Dense

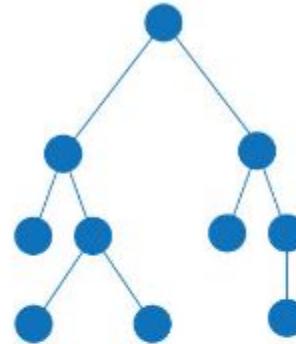


Complete (Clique)

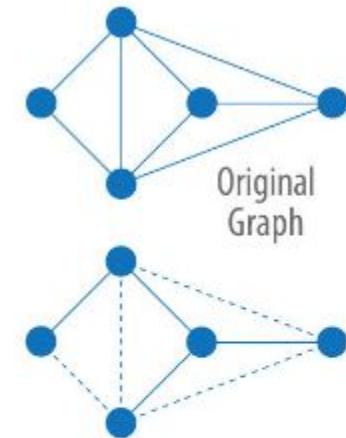
Trees



Rooted Tree
Root node
and no cycles



Binary Tree
Up to 2 child nodes
and no cycles

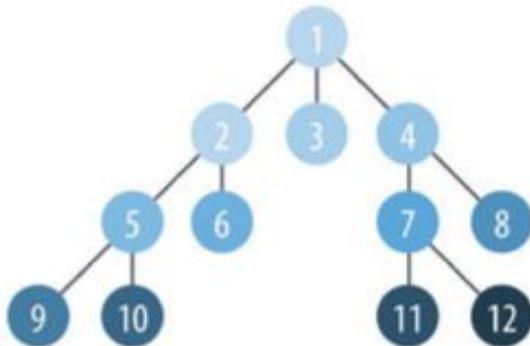


Spanning Tree
Subgraph of all nodes
but not all relationships
and no cycles

Types of Graph Algorithms - Pathfinding

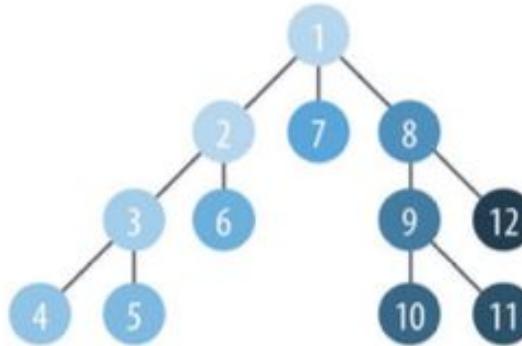
- **Pathfinding**
 - finding the shortest path between two nodes, if one exists, is probably the most common operation
 - “shortest” means fewest edges or lowest weight
 - Average Shortest Path can be used to monitor efficiency and resiliency of networks.
 - Minimum spanning tree, cycle detection, max/min flow... are other types of pathfinding

BFS vs DFS



Breadth First Search

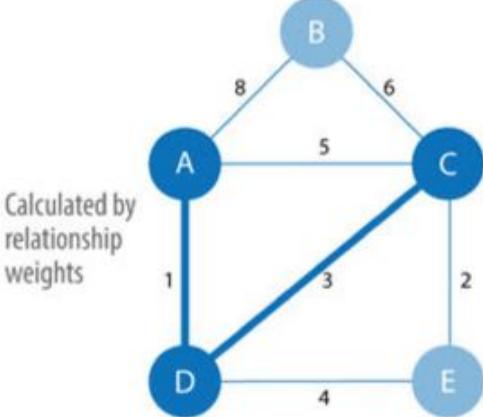
Visits nearest neighbors first



Depth First Search

Walks down each branch first

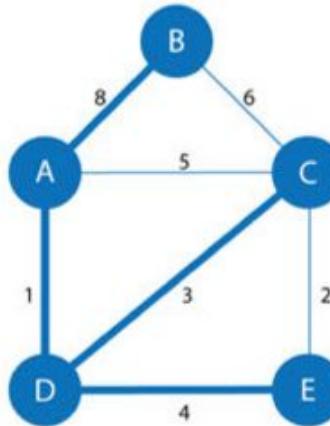
Shortest Path



Shortest Path

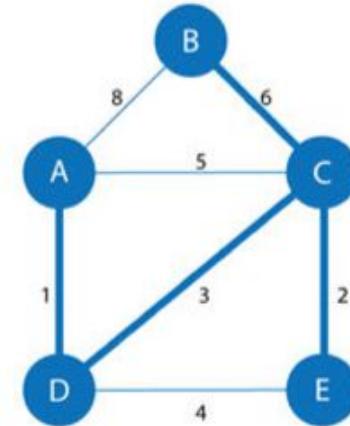
Shortest path between 2 nodes (A to C shown)

$(A, B) = 8$
 $(A, C) = 4$ via D
 $(A, D) = 1$
 $(A, E) = 5$ via D
 $(B, C) = 6$
 $(B, D) = 9$ via A or C
And so on...



All-Pairs Shortest Paths

Optimized calculations for shortest paths from all nodes to all other nodes



Single Source Shortest Path

Shortest path from a root node (A shown) to all other nodes

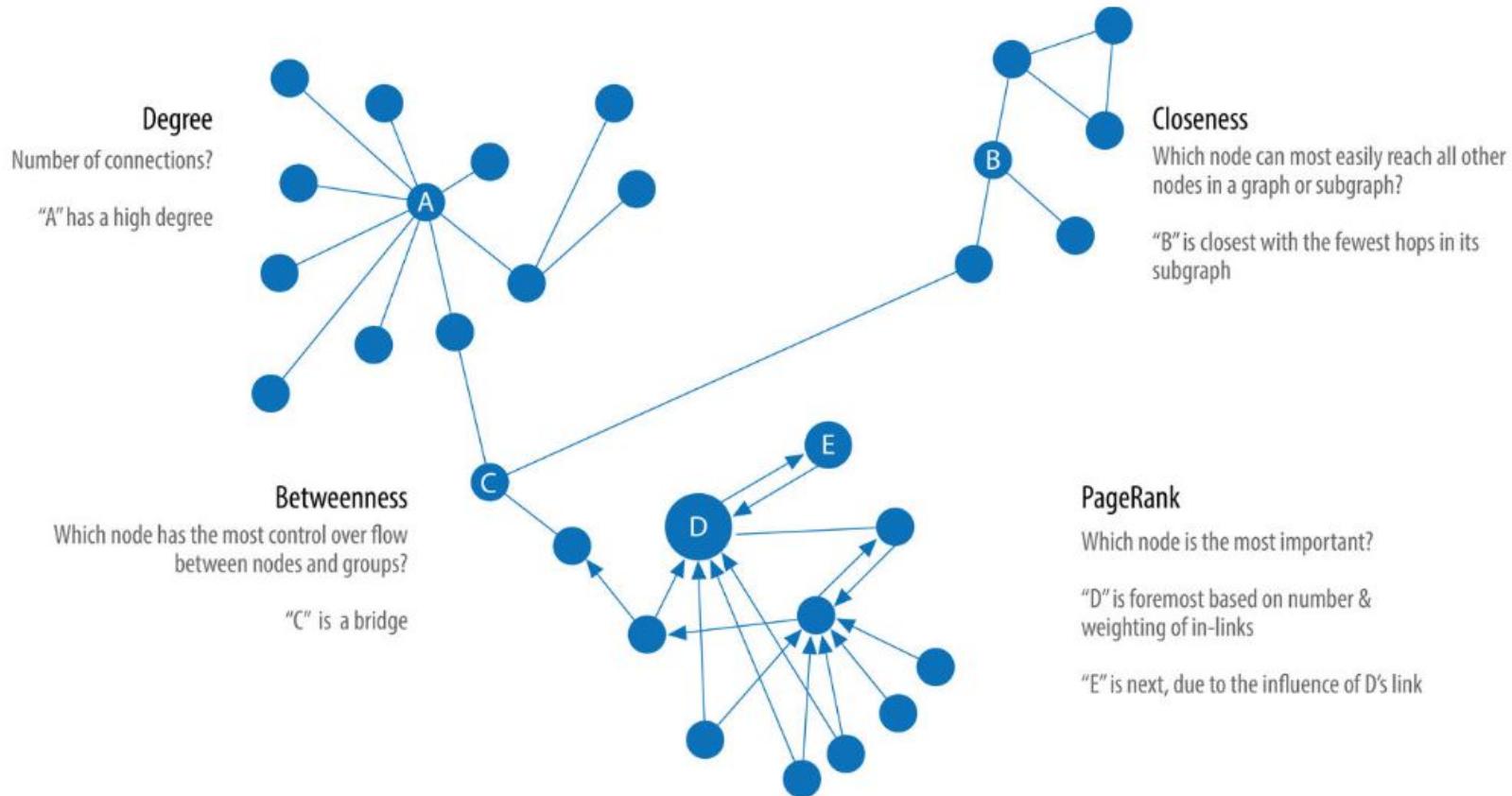
Minimum Spanning Tree

Shortest path connecting all nodes (A start shown)

Types of Graph Algorithms - Centrality & Community Detection

- **Centrality**
 - determining which nodes are “more important” in a network compared to other nodes
 - EX: Social Network Influencers?
- **Community Detection**
 - evaluate clustering or partitioning of nodes of a graph and tendency to strengthen or break apart

Centrality



Some Famous Graph Algorithms

- **Dijkstra's Algorithm** - single-source shortest path algo for positively weighted graphs
- **A* Algorithm** - Similar to Dijkstra's with added feature of using a heuristic to guide traversal
- **PageRank** - measures the importance of each node within a graph based on the number of incoming relationships and the importance of the nodes from those incoming relationships

- A Graph Database System that supports both transactional and analytical processing of graph-based data
- Relatively new class of no-sql DBs
- Considered schema optional (one can be imposed)
- Supports various types of indexing
- ACID compliant
- Supports distributed computing
- Similar: Microsoft CosmoDB, Amazon Neptune

??
..

DS 4300

Neo4j

Mark Fontenot, PhD
Northeastern University

- A Graph Database System that supports both transactional and analytical processing of graph-based data
 - Relatively new class of no-sql DBs
- Considered schema optional (one can be imposed)
- Supports various types of indexing
- ACID compliant
- Supports distributed computing
- Similar: Microsoft CosmoDB, Amazon Neptune

Neo4j - Query Language and Plugins

- Cypher
 - Neo4j's graph query language created in 2011
 - Goal: SQL-equivalent language for graph databases
 - Provides a visual way of matching patterns and relationships
(nodes)-[:CONNECT_TO]->(otherNodes)
- APOC Plugin
 - Awesome Procedures on Cypher
 - Add-on library that provides hundreds of procedures and functions
- Graph Data Science Plugin
 - provides efficient implementations of common graph algorithms (like the ones we talked about yesterday)

Neo4j in Docker Compose

Docker Compose

- Supports **multi-container management**.
- Set-up is declarative - using YAML docker-compose.yaml file
 - services
 - volumes
 - networks, etc.
- 1 command can be used to start, stop, or scale a number of services at one time.
- Provides a consistent method for producing an identical environment (no more “well... it works on my machine!)
- Interaction is mostly via command line

docker-compose.yaml

```
services:  
  neo4j:  
    container_name: neo4j  
    image: neo4j:latest  
    ports:  
      - 7474:7474  
      - 7687:7687  
    environment:  
      - NEO4J_AUTH=neo4j/${NEO4J_PASSWORD}  
      - NEO4J_apoc_export_file_enabled=true  
      - NEO4J_apoc_import_file_enabled=true  
      - NEO4J_apoc_import_file_use_neo4j__config=true  
      - NEO4J_PLUGINS=["apoc", "graph-data-science"]  
    volumes:  
      - ./neo4j_db/data:/data  
      - ./neo4j_db/logs:/logs  
      - ./neo4j_db/import:/var/lib/neo4j/import  
      - ./neo4j_db/plugins:/plugins
```

Never put “secrets” in a docker compose file. Use .env files.



.env Files

- .env files - stores a collection of environment variables
- good way to keep environment variables for different platforms separate
 - .env.local
 - .env.dev
 - .env.prod

.env file

```
NE04J_PASSWORD=abc123!!!
```

Docker Compose Commands

- To test if you have Docker CLI properly installed, run: `docker --version`
- Major Docker Commands
 - `docker compose up`
 - `docker compose up -d`
 - `docker compose down`
 - `docker compose start`
 - `docker compose stop`
 - `docker compose build`
 - `docker compose build --no-cache`

The screenshot shows the Neo4j browser interface at the URL `localhost:7474`. The top navigation bar has a single item: '\$'. A blue header bar displays the message: "Database access not available. Please use `:server connect` to establish connection. There's a graph waiting for you." Below this, a command line interface shows the command `$:server connect`. The main content area is titled "Connect to Neo4j" and contains the following fields:

- Connect URL**: Shows "bolt:// localhost:7687". The "localhost:7687" part is highlighted with a red border.
- Authentication type**: Set to "Username / Password".
- Username**: The value "neo4j" is entered.
- Password**: The value "****" is entered, with a small info icon next to the field.
- Connect**: A blue button at the bottom of the form.

Neo4j Browser

localhost:7474 Then login.

The screenshot illustrates the Neo4j Browser interface with various components labeled:

- Sidebar:** Contains links for Database, Favorites, Guides, Sidebar, Help & resources, Browser sync, Browser settings, and About Neo4j.
- Cypher editor:** Shows the command: `neo4j$ MATCH (p:Person {name:'Tom Hanks'})-[]-(m:Movie) RETURN p,m`.
- Reusable result frame:** Displays a graph of nodes (Person and Movie) and relationships (ACTED_IN, DIRECTED_BY).
- Run query:** A button to execute the current Cypher query.
- Full screen editor:** A button to switch to full-screen mode.
- Result frame views:** Options for Graph, Table, Text, and Code.
- Overview:** Summary statistics: Node labels (Person 13, Movie 12), Relationship types (ACTED_IN 21, DIRECTED_BY 4), and Nodes/Relationships (19 nodes, 19 relationships).
- Node Properties display:** A panel showing properties for selected nodes, with controls for Zoom in, Zoom out, and Fit to screen.
- UI Elements (right side):** Buttons for Full screen result frame, Export, Collapse, Save as Favorite, Pin at top, and Rerun a query.

Inserting Data by Creating Nodes

```
CREATE (:User {name: "Alice", birthPlace: "Paris"})  
CREATE (:User {name: "Bob", birthPlace: "London"})  
CREATE (:User {name: "Carol", birthPlace: "London"})  
CREATE (:User {name: "Dave", birthPlace: "London"})  
CREATE (:User {name: "Eve", birthPlace: "Rome"})
```

```
CREATE (john:User {name: "John"})
```



Variable name
(used to reference this node later in the query)

Node label
(differentiate different types of notes)

Properties
(attached to the node)

Adding an Edge with No Variable Names

```
CREATE (:User {name: "Alice", birthPlace: "Paris"})
```

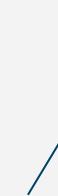
```
CREATE (:User {name: "Bob", birthPlace: "London"})
```

```
MATCH (alice:User {name:"Alice"})
```

```
MATCH (bob:User {name: "Bob"})
```

```
CREATE (alice)-[:KNOWS {since: "2022-12-01"}]-(bob)
```

Note: Relationships are directed in neo4j.



Matching

Which users were born in London?

```
MATCH (usr:User {birthPlace: "London"})  
RETURN usr.name, usr.birthPlace
```

usr.name	usr.birthPlace
"Bob"	"London"
"Carol"	"London"
"Dave"	"London"

Download Dataset and Move to Import Folder

Clone this repo:

<https://github.com/PacktPublishing/Graph-Data-Science-with-Neo4j>

In Chapter02/data of data repo, unzip the **netflix.zip** file

Copy **netflix_titles.csv** into the following folder where
you put your docker compose file

neo4j_db/neo4j_db/import

Importing Data

Basic Data Importing

Type the following into the Cypher Editor in Neo4j Browser

```
LOAD CSV WITH HEADERS
FROM 'file:///netflix_titles.csv' AS line
CREATE (:Movie {
    id: line.show_id,
    title: line.title,
    releaseYear: line.release_year
})
}
```

Loading CSVs - General Syntax

```
LOAD CSV  
[WITH HEADERS]  
FROM 'file:///file_in_import_folder.csv'  
AS line  
[FIELDTERMINATOR ',']  
// do stuffs with 'line'
```

Importing with Directors this Time

```
LOAD CSV WITH HEADERS  
FROM 'file:///netflix_titles.csv' AS line  
WITH split(line.director, ",") as directors_list  
UNWIND directors_list AS director_name  
CREATE (:Person {name: trim(director_name)}))
```

But this generates duplicate Person nodes (a director can direct more than 1 movie)

Importing with Directors Merged

```
MATCH (p:Person) DELETE p
```

```
LOAD CSV WITH HEADERS  
FROM 'file:///netflix_titles.csv' AS line  
WITH split(line.director, ",") as directors_list  
UNWIND directors_list AS director_name  
MERGE (:Person {name: director_name})
```

Adding Edges

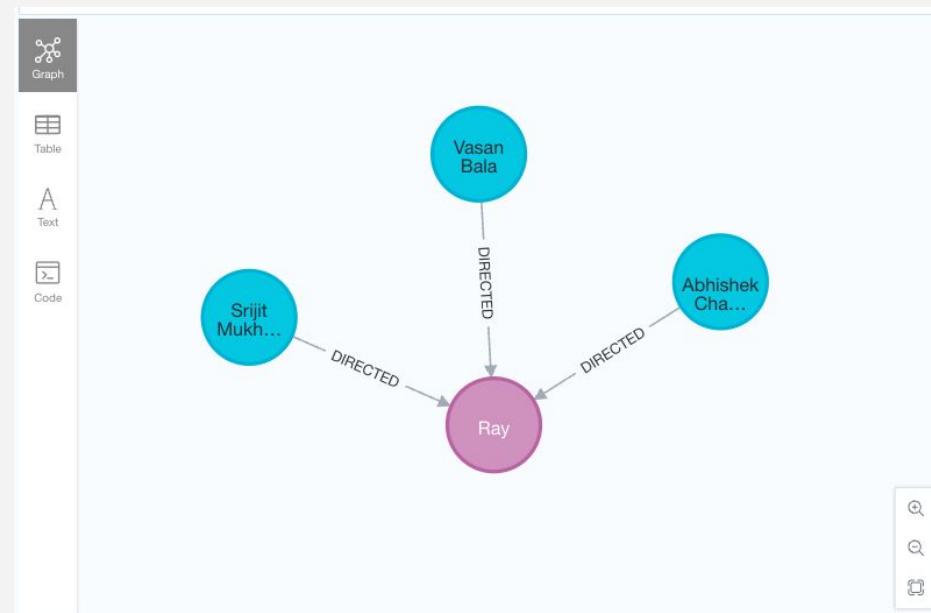
LOAD CSV WITH HEADERS

```
FROM 'file:///netflix_titles.csv' AS line
MATCH (m:Movie {id: line.show_id})
WITH m, split(line.director, ",") as directors_list
UNWIND directors_list AS director_name
MATCH (p:Person {name: director_name})
CREATE (p)-[:DIRECTED]->(m)
```

Gut Check

Let's check the movie titled Ray:

```
MATCH (m:Movie {title: "Ray"})->[:DIRECTED]-(p:Person)  
RETURN m, p
```



??
??

DS 4300

AWS Introduction

Mark Fontenot, PhD
Northeastern University

Amazon Web Services

- Leading Cloud Platform with over 200 different services available
- Globally available via its massive networks of regions and availability zones with their massive data centers
- Based on a pay-as-you-use cost model.
 - Theoretically cheaper than renting rackspace/servers in a data center... *Theoretically.*

History of AWS

- Originally launched in 2006 with only 2 services: S3 & EC2.
- By 2010, services had expanded to include SimpleDB, Elastic Block Store, Relational Database Service, DynamoDB, CloudWatch, Simple Workflow, CloudFront, Availability Zones, and others.
- Amazon had competitions with big prizes to spur the adoption of AWS in its early days
- They've continuously innovated, always introducing new services for ops, dev, analytics, etc... (200+ services now)

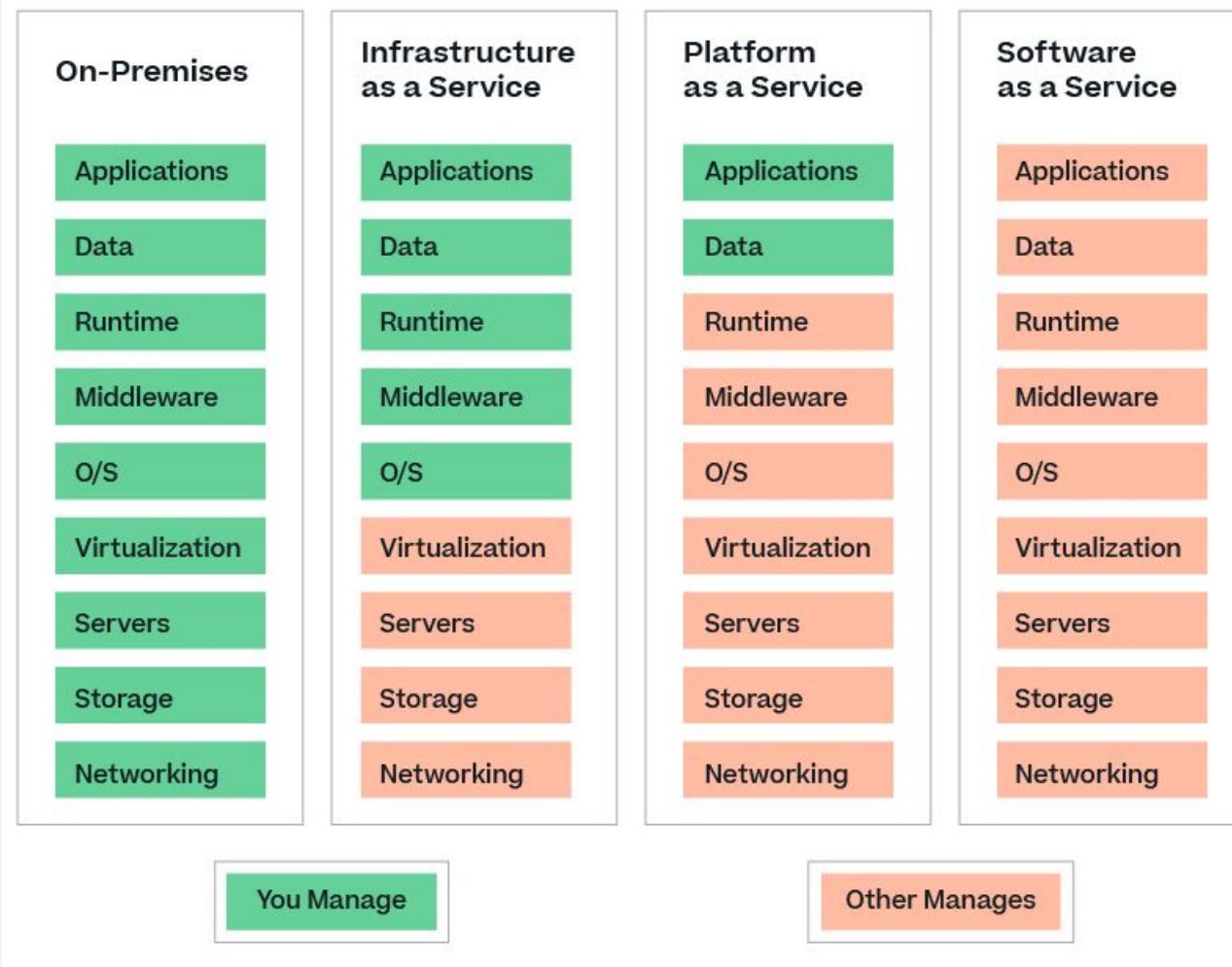
AWS Service Categories

					
Analytics	Application integration	Blockchain	Business applications	Cloud Financial Management	Compute
					
Customer enablement	Containers	Databases	Developer tools	End user computing	Front-end web and mobile
					
Game tech	Internet of Things (IoT)	Machine Learning (ML) and Artificial Intelligence (AI)	Management and governance	Media	Migration and transfer
					
Networking and content delivery	Quantum technologies	Robotics	Satellite	Security, identity, and compliance	Storage

Cloud Models

- IaaS ([more](#)) - Infrastructure as a Service
 - Contains the basic services that are needed to build an IT infrastructure
- PaaS ([more](#)) - Platform as a Service
 - Remove the need for having to manage infrastructure
 - You can get right to deploying your app
- SaaS ([more](#)) - Software as a Service
 - Provide full software apps that are run and managed by another party/vendor

Cloud Models



The Shared Responsibility Model - AWS

- AWS Responsibilities (Security OF the cloud):
 - Security of physical infrastructure (*infra*) and network
 - keep the data centers secure, control access to them
 - maintain power availability, HVAC, etc.
 - monitor and maintain physical networking equipment and global infra/connectivity
 - Hypervisor & Host OSs
 - manage the virtualization layer used in AWS compute services
 - maintaining underlying host OSs for other services
 - Maintaining managed services
 - keep infra up to date and functional
 - maintain server software (patching, etc)

The Shared Responsibility Model - Client

- Client Responsibilities (Security IN the cloud):
 - Control of Data/Content
 - client controls how its data is classified, encrypted, and shared
 - implement and enforce appropriate data-handling policies
 - Access Management & IAM
 - properly configure IAM users, roles, and policies.
 - enforce the *Principle of Least Privilege*
 - Manage self-hosted Apps and associated OSs
 - Ensure network security to its VPC
 - Handle compliance and governance policies and procedures

The AWS Global Infrastructure

- Regions - distinct geographical areas
 - us-east-1, us-west 1, etc
- Availability Zones (AZs)
 - each region has multiple AZs
 - roughly equiv to isolated data centers
- Edge Locations
 - locations for CDN and other types of caching services
 - allows content to be closer to end user.



36 launched Regions
each with multiple Availability Zones

114 Availability Zones

700+ CloudFront POPs
and 13 Regional edge caches

Compute Services

- VM-based:
 - EC2 & EC2 Spot - Elastic Cloud Compute
- Container-based:
 - ECS - Elastic Container Service
 - ECR - Elastic Container Registry
 - EKS - Elastic Kubernetes Service
 - *Fargate* - Serverless container service
- Serverless: AWS Lambda



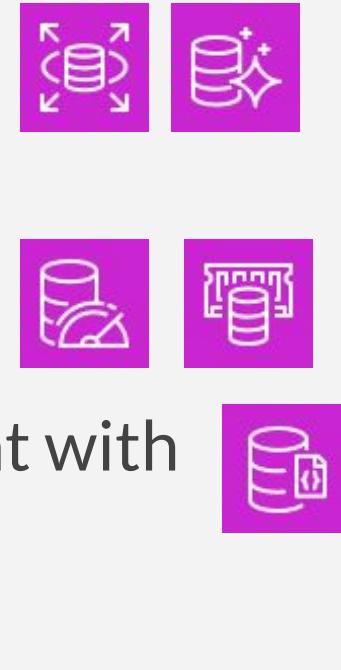
Storage Services

- Amazon S3 - Simple Storage Service
 - Object storage in buckets; highly scalable; different storage classes
- Amazon EFS - Elastic File System
 - Simple, serverless, elastic, “set-and-forget” file system
- Amazon EBS - Elastic Block Storage
 - High-Performance block storage service
- Amazon File Cache
 - High-speed cache for datasets stored anywhere
- AWS Backup
 - Fully managed, policy-based service to automate data protection and compliance of apps on AWS

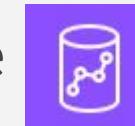
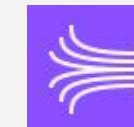
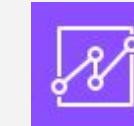


Database Services

- Relational - Amazon RDS, Amazon Aurora
- Key-Value - Amazon DynamoDB
- In-Memory - Amazon MemoryDB, Amazon ElastiCache
- Document - Amazon DocumentDB (Compat with MongoDB)
- Graph - Amazon Neptune



Analytics Services

- Amazon Athena - Analyze petabyte scale data where it lives (S3, for example) 
- Amazon EMR - Elastic MapReduce - Access Apache Spark, Hive, Presto, etc. 
- AWS Glue - Discover, prepare, and integrate all your data 
- Amazon Redshift - Data warehousing service 
- Amazon Kinesis - real-time data streaming 
- Amazon QuickSight - cloud-native BI/reporting tool 

ML and AI Services

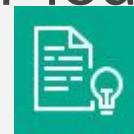
- Amazon SageMaker

- fully-managed ML platform, including Jupyter NBs
- build, train, deploy ML models



- AWS AI Services w/ Pre-trained Models

- Amazon Comprehend - NLP



- Amazon Rekognition - Image/Video analysis



- Amazon Textract - Text extraction



- Amazon Translate - Machine translation



Important Services for Data Analytics/Engineering

- EC2 and Lambda
- Amazon S3
- Amazon RDS and DynamoDB
- AWS Glue
- Amazon Athena
- Amazon EMR
- Amazon Redshift

AWS Free Tier

- Allows you to gain hands-on experience with a subset of the services for 12 months (service limitations apply as well)
 - Amazon EC2 - 750 hours/month (specific OSs and Instance Sizes)
 - Amazon S3 - 5GB (20K GETs, 2K Puts)
 - Amazon RDS - 750 hours/month of DB use (within certain limits)
 - So many free services

??
?

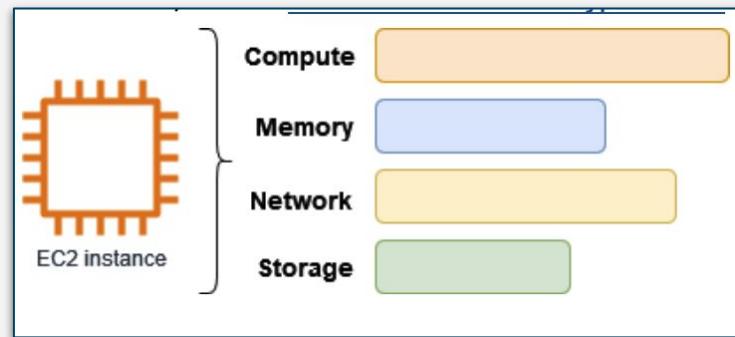
DS 4300

Amazon EC2 & Lambda

Mark Fontenot, PhD
Northeastern University

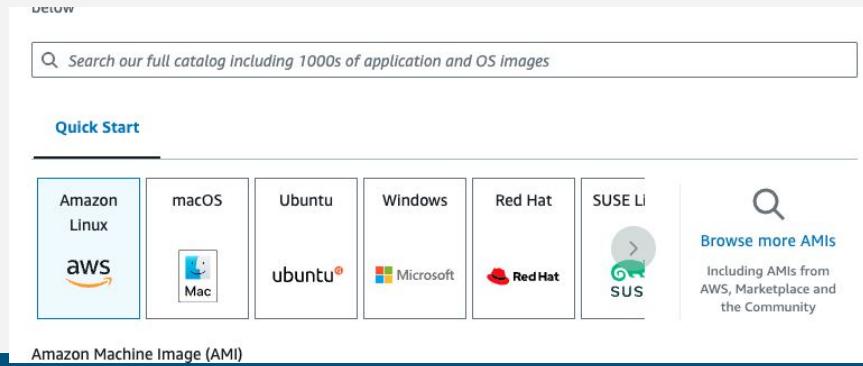
EC2

- EC2 → Elastic Cloud Compute
- Scalable Virtual Computing in the Cloud
- Many (Many!!) instance types available
- Pay-as-you-go model for pricing
- Multiple different Operating Systems



Features of EC2

- **Elasticity** - easily (and programmatically) scale instances up or down as needed
- You can use one of the standard AMIs OR provide your own AMI if pre-config is needed
- Easily integrates with many other services such as S3, RDS, etc.



AMI = Amazon Machine Image

EC2 Lifecycle

- **Launch** - when starting an instance for the first time with a chosen configuration
- **Start/Stop** - Temporarily suspend usage without deleting the instance
- **Terminate** - Permanently delete the instance
- **Reboot** - Restart an instance without sling the data on the root volume

Where Can You Store Data?

- **Instance Store:** Temporary, high-speed storage tied to the instance lifecycle
- **EFS (Elastic File System) Support** - Shared file storage
- **EBS (Elastic Block Storage)** - Persistent block-level storage
- **S3** - large data set storage or EC2 backups even

Common EC2 Use Cases

- Web Hosting - Run a website/web server and associated apps
- Data Processing - It's a VM... you can do anything to data possible with a programming language.
- Machine Learning - Train models using GPU instances
- Disaster Recovery - Backup critical workloads or infrastructure in the cloud

Let's Spin Up an EC2 Instance

Search results for 'ec2'

Services Show more ▾

-  **EC2** ☆
Virtual Servers in the Cloud
-  **EC2 Image Builder** ☆
A managed service to automate build, customize and deploy OS images

[View alarms in CloudWatch \(opens new tab\)](#)

To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.

Launch instance ▾ **Migrate a server** 

Note: Your instances will launch in the US East (N. Virginia) Region

Launch an instance Info

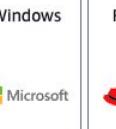
Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags Info

Name

[Add additional tags](#)

Quick Start

 Amazon Linux  macOS  **Ubuntu**  Windows  Red Hat  SUSE Li

 [Browse more AMIs](#)
Including AMIs from AWS, Marketplace and the Community

Amazon Machine Image (AMI)

Ubuntu Server 24.04 LTS (HVM), SSD Volume Type Free tier eligible

ami-0866a3c8686eaeba (64-bit (x86)) / ami-0325498274077fac5 (64-bit (Arm))
Virtualization: hvm ENA enabled: true Root device type: ebs

Let's Spin Up an EC2 Instance

▼ Instance type [Info](#) | [Get advice](#)

Instance type

t2.micro

Family: t2 1 vCPU 1 GiB Memory Current generation: true
On-Demand Windows base pricing: 0.0162 USD per Hour
On-Demand Ubuntu Pro base pricing: 0.0134 USD per Hour
On-Demand SUSE base pricing: 0.0116 USD per Hour
On-Demand RHEL base pricing: 0.026 USD per Hour
On-Demand Linux base pricing: 0.0116 USD per Hour

Free tier eligible

All generations

Compare instance types

Additional costs apply for AMIs with pre-installed software

▼ Key pair (login) [Info](#)

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - required

Select

Create new key pair

Create key pair

Key pair name
Key pairs allow you to connect to your instance securely.
 AWS-4300-ec2

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type

RSA RSA encrypted private and public key pair
 ED25519 ED25519 encrypted private and public key pair

Private key file format

.pem For use with OpenSSH
 .ppk For use with PuTTY

⚠️ When prompted, store the private key in a secure and accessible location on your computer. You will need it later to connect to your instance. [Learn more](#)

Cancel Create key pair

Let's Spin Up an EC2 Instance

▼ Network settings [Info](#)

[Edit](#)[Network](#) | [Info](#)

vpc-025c749a0c68d5aba

[Subnet](#) | [Info](#)

No preference (Default subnet in any availability zone)

[Auto-assign public IP](#) | [Info](#)

Enable

[Additional charges apply](#) when outside of [free tier allowance](#)[Firewall \(security groups\)](#) | [Info](#)

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

 [Create security group](#) [Select existing security group](#)

We'll create a new security group called '[launch-wizard-1](#)' with the following rules:

 [Allow SSH traffic from](#)

Helps you connect to your instance

Anywhere



0.0.0.0/0

 [Allow HTTPS traffic from the internet](#)

To set up an endpoint, for example when creating a web server

 [Allow HTTP traffic from the internet](#)

To set up an endpoint, for example when creating a web server

⚠️ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

[Cancel](#)[Launch instance](#) [Preview code](#)

Ubuntu VM Commands

- Initial user is **ubuntu**
- Access super user commands with **sudo**
- Package manager is **apt**
 - kind of like Homebrew or Choco
- Update the packages installed
 - `sudo apt update; sudo apt upgrade`

MiniConda on EC2

Make sure you're logged in to your EC2 instance

- Let's install MiniConda

- curl -O https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
- bash ./Miniconda3-latest-Linux-x86_64.sh

```
Do you wish to update your shell profile to automatically initialize conda?  
This will activate conda on startup and change the command prompt when activated.  
If you'd prefer that conda's base environment not be activated on startup,  
run the following command when conda is activated:  
  
conda config --set auto_activate_base false  
  
You can undo this by running `conda init --reverse $SHELL`? [yes|no]  
[no] >>> yes
```

Installing & Using Streamlit

- Log out of your EC2 instance and log back in
- Make sure pip is now available:
 - pip --version
- Install Streamlit and sklearn
 - pip install streamlit scikit-learn
- Make a directory for a small web app
 - mkdir web
 - cd web

Basic Streamlit App

```
import streamlit as st

def main():
    st.title("Welcome to my Streamlit App")
    st.write("## Data Sets")
    st.write("""
        - data set 01
        - data set 02
        - data set 03
    """)
    st.write("\n")
    st.write("## Goodbye!")

if __name__ == "__main__":
    main()
```

- nano test.py
- Add code on left
- **ctrl-x** to save and exit
- **streamlit run test.py**

Opening Up The Streamlit Port

Dashboard X

EC2 Global View

Events

▶ Instances

▶ Images

▶ Elastic Block Store

▼ Network & Security

- Security Groups
- Elastic IPs
- Placement Groups
- Key Pairs
- Network Interfaces

sg-0d58f56d20a97916c launch-wizard-1

Inbound rules Outbound rules Sharing - new VPC associations - new Tags

Inbound rules (2)

Name	Security group rule...	IP version	Type	Protocol	Port range	Source	Description
-	sgr-072f78d9d4ee328fd	IPv6	SSH	TCP	22	::/0	-
-	sgr-071d2c2eac0af9f63	IPv4	SSH	TCP	22	0.0.0.0/0	-

Manage tags Edit inbound rules < 1 > ⚙

Edit inbound rules Info

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type <small>Info</small>	Protocol <small>Info</small>	Port range <small>Info</small>	Source <small>Info</small>	Description - optional <small>Info</small>
sgr-072f78d9d4ee328fd	SSH	TCP	22	Custom	<input type="text"/> ::/0 X
sgr-071d2c2eac0af9f63	SSH	TCP	22	Custom	<input type="text"/> 0.0.0.0/0 X
-	Custom TCP	TCP	8501	Anywh...	<input type="text"/> 0.0.0.0/0 X

Add rule

Cancel Preview changes Save rules

In a Browser

```
(base) ubuntu@ip-172-31-91-161:~/web$ streamlit run test.py
```

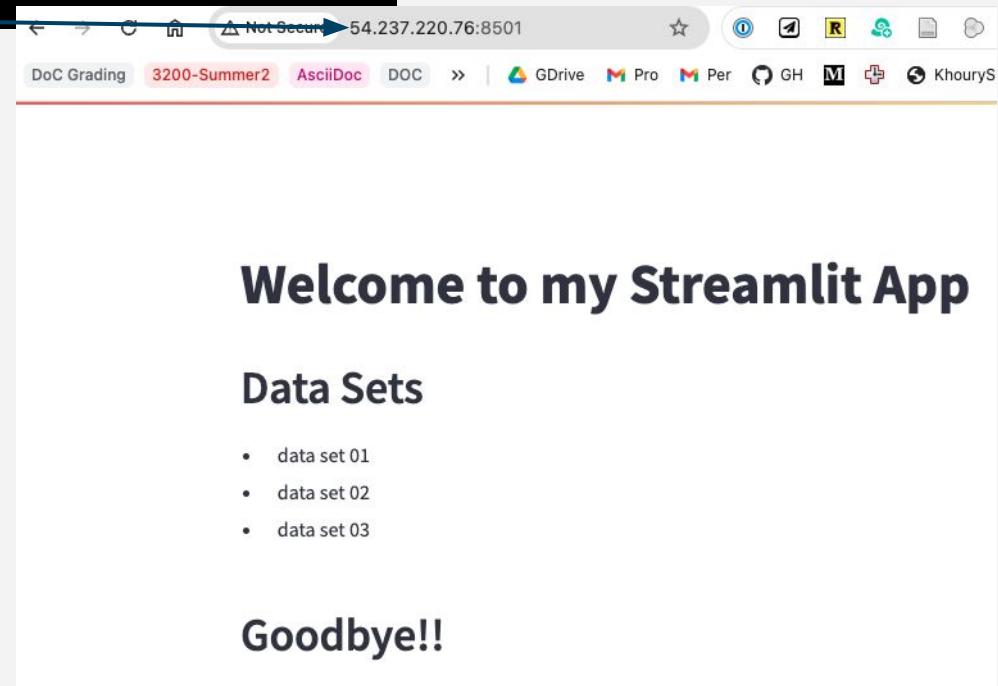
```
Collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.
```

```
You can now view your Streamlit app in your browser.
```

```
Local URL: http://localhost:8501
```

```
Network URL: http://172.31.91.161:8501
```

```
External URL: http://54.237.220.76:8501
```



AWS Lambda

Lambdas

- Lambdas provide *serverless computing*
- Automatically run code in response to **events**.
- Relieves you from having to manage servers - only worry about the code
- You only pay for **execution time**, not for idle compute time (different from EC2)

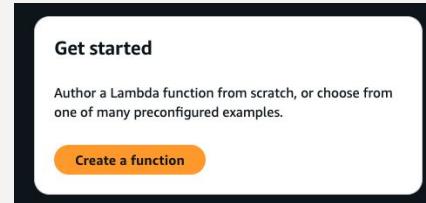
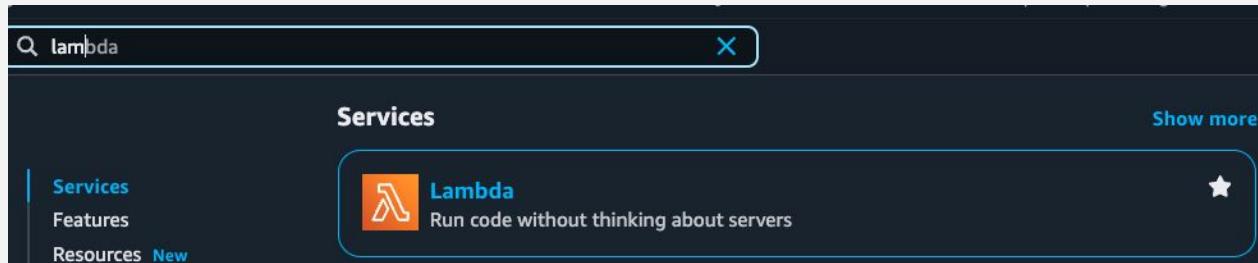
Lambda Features

- **Event-driven execution** - can be triggered by many different events in AWS
- Supports a large number of runtimes... Python, Java, Node.js, etc
- HIGHLY integrated with other AWS services
- Extremely scalable and can rapidly adjust to demands

How it Works

- Add/upload your code through AWS MGMT console
- Configure event source(s)
- Watch your Lambda run when one of the event sources fires an event

Let's Make One



Making a Lambda

Create function Info

Choose one of the following options to create your function.

- Author from scratch
Start with a simple Hello World example.
- Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.
- Container image
Select a container image to deploy for your function.

Basic information Info

Blueprint name

Hello world function	python3.10
A starter AWS Lambda function.	<small>▲</small>

Search by blueprint name, runtime or workload category

Learn about Lambda by creating an HTTP endpoint that outputs a server-side rendered web page.

Hello world function	nodejs18.x
A starter AWS Lambda function.	<small>valid characters are a-z, A-Z, 0-9, hyphens (-), and</small>

Hello world function	python3.10
A starter AWS Lambda function.	<small>✓</small>

Make an HTTPS request	nodejs18.x
Demonstrates using a built-in Node.js module to make an HTTPS request.	<small>valid characters are a-z, A-Z, 0-9, hyphens (-), and</small>

Creating a Function

Function name

Enter a name that describes the purpose of your function.

SimpleTestFunction

Function name must be 1 to 64 characters, must be unique

Lambda function code

Code is preconfigured by the chosen blueprint. You can configure it after you create the function. [Learn more](#) about deploying Lambda functions.

This function contains external libraries.

```
1 import json
2
3 print('Loading function')
4
5
6 def lambda_handler(event, context):
7     #print("Received event: " + json.dumps(event, indent=2))
8     print("value1 = " + event['key1'])
9     print("value2 = " + event['key2'])
10    print("value3 = " + event['key3'])
11    return event['key1'] # Echo back the first key value
12    #raise Exception('Something went wrong')
13
```

1:1 Python Spaces: 4

Cancel

Create function

Sample Code

- Edit the code
- Deploy the code!

The screenshot shows the AWS Lambda Code Source interface. On the left, there's an Explorer sidebar with icons for file operations like Open, Find, and Deploy. The main area shows a file tree for a project named 'FIRSTLAMBDA'. Inside, there are folders for 'EXPLORER', '__pycache__', and 'lambda_function.py'. The 'lambda_function.py' file is selected and its content is displayed in the editor:

```
lambda_function.py x
lambda_function.py
1 import json
2
3 print('Loading function')
4
5
6 def lambda_handler(event, context):
7     #print("Received event: " + json.dumps(event, indent=2))
8     print("Name = " + event['name'])
9     print("Year = " + event['year'])
10    print("Major = " + event['major'])
11    return event['name'] # Echo back the first key value
12
13
```

At the bottom of the interface, there are two buttons: 'Deploy (F5)' and 'Test (Shift+F5)'. The 'Deploy (F5)' button is highlighted with a red box.

Test event Info

CloudWatch Logs Live Tail

Save

Test

To invoke your function without saving an event, configure the JSON event, then choose Test.

Test event action

 Create new event Edit saved event

Event name

LambdaTest2

Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

Event sharing settings

 Private

This event is only available in the Lambda console and to the event creator. You can configure a total of 10. [Learn more](#)

 Shareable

This event is available to IAM users within the same account who have permissions to access and use shareable events. [Learn more](#)

Template - optional

LambdaTest

Event JSON

```
1 {  
2   "name": "mark",  
3   "year": "14th",  
4   "major": "CS"  
5 }
```

Format JSON

??
..

B-Trees

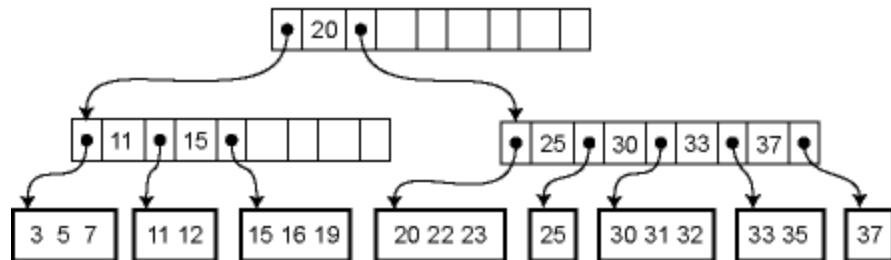
The idea we saw earlier of putting multiple set (list, hash table) elements together into large chunks that exploit locality can also be applied to trees. Binary search trees are not good for locality because a given node of the binary tree probably occupies only a fraction of any cache line. **B-trees** are a way to get better locality by putting multiple elements into each tree node.

B-trees were originally invented for storing data structures on disk, where locality is even more crucial than with memory. Accessing a disk location takes about 5ms = 5,000,000ns. Therefore, if you are storing a tree on disk, you want to make sure that a given disk read is as effective as possible. B-trees have a high branching factor, much larger than 2, which ensures that few disk reads are needed to navigate to the place where data is stored. B-trees may also be useful for in-memory data structures because these days main memory is almost as slow relative to the processor as disk drives were to main memory when B-trees were first introduced!

A B-tree of order m is a search tree in which each nonleaf node has up to m children. The actual elements of the collection are stored in the leaves of the tree, and the nonleaf nodes contain only keys. Each leaf stores some number of elements; the maximum number may be greater or (typically) less than m . The data structure satisfies several invariants:

1. Every path from the root to a leaf has the same length
2. If a node has n children, it contains $n-1$ keys.
3. Every node (except the root) is at least half full
4. The elements stored in a given subtree all have keys that are between the keys in the parent node on either side of the subtree pointer. (This generalizes the BST invariant.)
5. The root has at least two children if it is not a leaf.

For example, the following is an order-5 B-tree ($m=5$) where the leaves have enough space to store up to 3 data records:



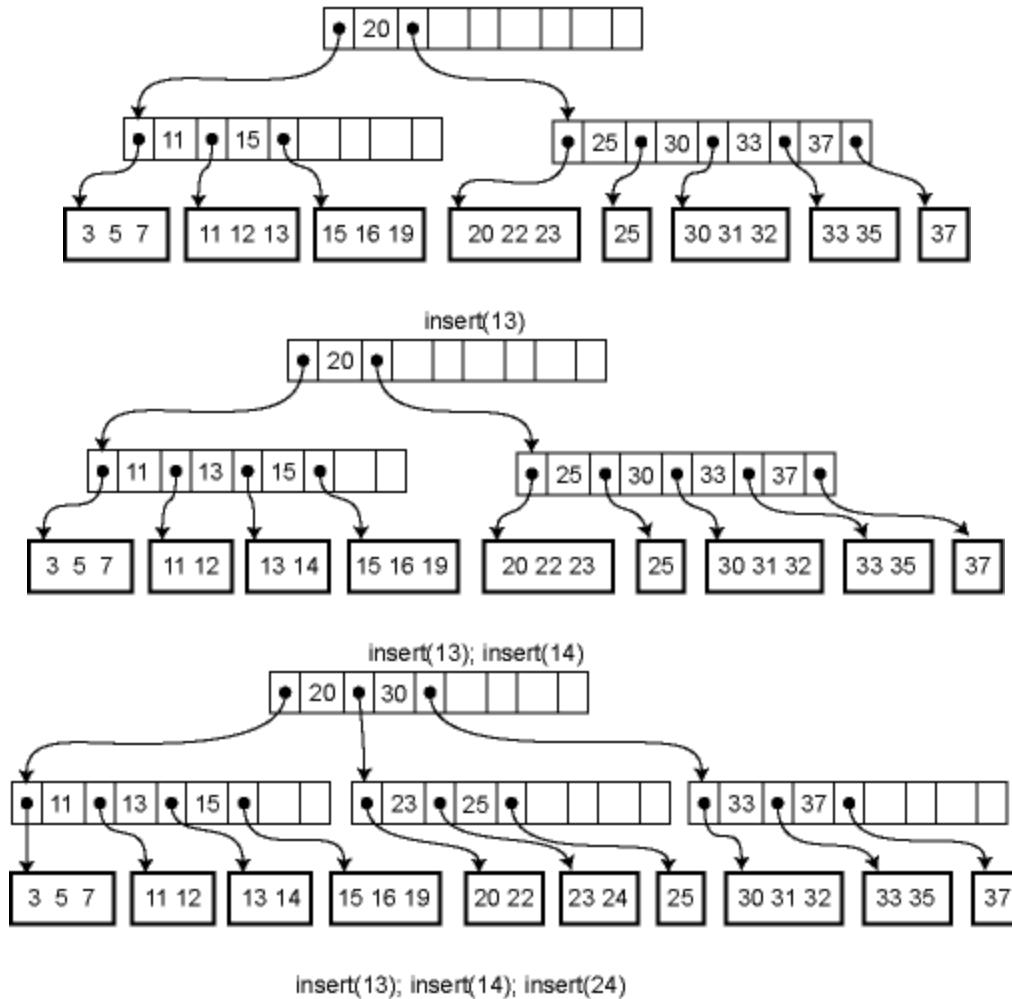
Because the height of the tree is uniformly the same and every node is at least half full, we are guaranteed that the asymptotic performance is $O(\lg n)$ where n is the size of the collection. The real win is in the constant factors, of course. We can choose m so that the pointers to the m children plus the $m-1$ elements fill out a cache line at the highest level of the memory hierarchy where we can expect to get cache hits. For example, if we are accessing a large disk database then our "cache lines" are memory blocks of the size that is read from disk.

Lookup in a B-tree is straightforward. Given a node to start from, we use a simple linear or binary search to find whether the desired element is in the node, or if not, which child pointer

to follow from the current node.

Insertion and deletion from a B-tree are more complicated; in fact, they are notoriously difficult to implement correctly. For insertion, we first find the appropriate leaf node into which the inserted element falls (assuming it is not already in the tree). If there is already room in the node, the new element can be inserted simply. Otherwise the current leaf is already full and must be split into two leaves, one of which acquires the new element. The parent is then updated to contain a new key and child pointer. If the parent is already full, the process ripples upwards, eventually possibly reaching the root. If the root is split into two, then a new root is created with just two children, increasing the height of the tree by one.

For example, here is the effect of a series of insertions. The first insertion (13) merely affects a leaf. The second insertion (14) overflows the leaf and adds a key to an internal node. The third insertion propagates all the way to the root.



Deletion works in the opposite way: the element is removed from the leaf. If the leaf becomes empty, a key is removed from the parent node. If that breaks invariant 3, the keys of the parent node and its immediate right (or left) sibling are reapportioned among them so that invariant 3 is satisfied. If this is not possible, the parent node can be combined with that sibling, removing a key another level up in the tree and possibly causing a ripple all the way to the root. If the root has just two children, and they are combined, then the root is deleted and the new combined node becomes the root of the tree, reducing the height of the tree by one.

Further reading: Aho, Hopcroft, and Ullman, *Data Structures and Algorithms*, Chapter 11.

12.6. B-Trees

12.6.1. B-Trees

This module presents the B-tree. B-trees are usually attributed to R. Bayer and E. McCreight who described the B-tree in a 1972 paper. By 1979, B-trees had replaced virtually all large-file access methods other than hashing. B-trees, or some variant of B-trees, are the standard file organization for applications requiring insertion, deletion, and key range searches. They are used to implement most modern file systems. B-trees address effectively all of the major problems encountered when implementing disk-based search trees:

1. The B-tree is shallow, in part because the tree is always height balanced (all leaf nodes are at the same level), and in part because the branching factor is quite high. So only a small number of disk blocks are accessed to reach a given record.
2. Update and search operations affect only those disk blocks on the path from the root to the leaf node containing the query record. The fewer the number of disk blocks affected during an operation, the less disk I/O is required.
3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on range searches.
4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

A B-tree of order m is defined to have the following shape properties:

The root is either a leaf or has at least two children.

Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.

All leaves are at the same level in the tree, so the tree is always height balanced.

The B-tree is a generalization of the 2-3 tree. Put another way, a 2-3 tree is a B-tree of order three. Normally, the size of a node in the B-tree is chosen to fill a disk block. A B-tree node implementation typically allows 100 or more children. Thus, a B-tree node is equivalent to a disk block, and a “pointer” value stored in the tree is actually the number of the block containing the child node (usually interpreted as an offset from the beginning of the corresponding disk file). In a typical application, the B-tree’s access to the disk file will be managed using a **buffer pool** and a block-replacement scheme such as **LRU**.

Figure 12.6.1 shows a B-tree of order four. Each node contains up to three keys, and internal nodes have up to four children.

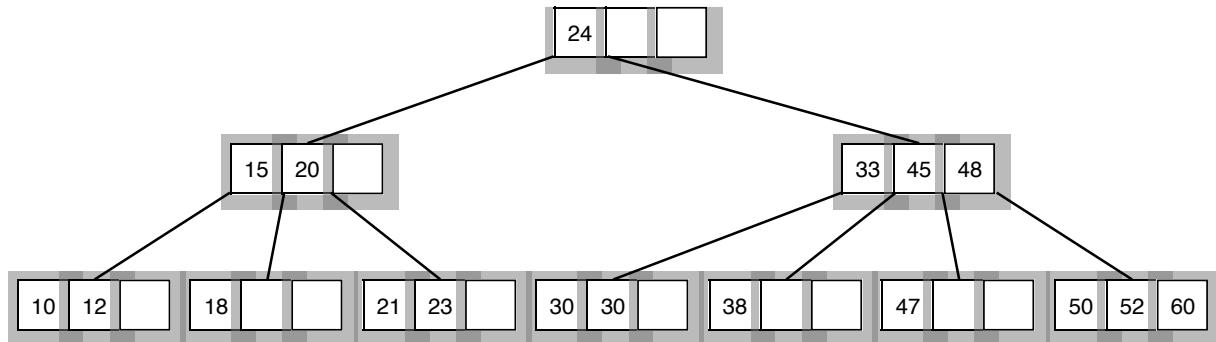


Figure 12.6.1: A B-tree of order four.

Search in a B-tree is a generalization of search in a 2-3 tree. It is an alternating two-step process, beginning with the root node of the B-tree.

1. Perform a binary search on the records in the current node. If a record with the search key is found, then return that record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.

2. Otherwise, follow the proper branch and repeat the process.

For example, consider a search for the record with key value 47 in the tree of Figure 12.6.1. The root node is examined and the second (right) branch taken. After examining the node at level 1, the third branch is taken to the next level to arrive at the leaf node containing a record with key value 47.

B-tree insertion is a generalization of 2-3 tree insertion. The first step is to find the leaf node that should contain the key to be inserted, space permitting. If there is room in this node, then insert the key. If there is not, then split the node into two and promote the middle key to the parent. If the parent becomes full, then it is split in turn, and its middle key promoted.

Note that this insertion process is guaranteed to keep all nodes at least half full. For example, when we attempt to insert into a full internal node of a B-tree of order four, there will now be five children that must be dealt with. The node is split into two nodes containing two keys each, thus retaining the B-tree property. The middle of the five children is promoted to its parent.

12.6.1.1. B+ Trees

The previous section mentioned that B-trees are universally used to implement large-scale disk-based systems. Actually, the B-tree as described in the previous section is almost never implemented. What is most commonly implemented is a variant of the B-tree, called the B⁺ tree. When greater efficiency is required, a more complicated variant known as the B* tree is used.

Consider again the **linear index**. When the collection of records will not change, a linear index provides an extremely efficient way to search. The problem is how to handle those pesky inserts and deletes. We could try to keep the core idea of storing a sorted array-based list, but make it more flexible by breaking the list into manageable chunks that are more easily updated. How might we do that? First, we need to decide how big the chunks should be. Since the data are on disk, it seems reasonable to store a chunk that is the size of a disk block, or a small multiple of the disk block size. If the next record to be inserted belongs to a chunk that hasn't filled its block then we can just insert it there. The fact that this might cause other records in that chunk to move a little bit in the array is not important, since this does not cause any extra disk accesses so long as we move data within that chunk. But what if the chunk fills up the entire block that contains it? We could just split it in half. What if we want to delete a record? We could just take the deleted record out of the chunk, but we might not want a lot of near-empty chunks. So we could put adjacent chunks together if they have only a small amount of data between them. Or we could shuffle data between adjacent chunks that together contain more data. The big problem would be how to find the desired chunk when processing a record with a given key. Perhaps some sort of tree-like structure could be used to locate the appropriate chunk. These ideas are exactly what motivate the B⁺ tree. The B⁺ tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.

The most significant difference between the B⁺ tree and the BST or the standard B-tree is that the B⁺ tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search. This means that internal nodes are significantly different in structure from leaf nodes. Internal nodes store keys to guide the search, associating each key with a pointer to a child B⁺ tree node. Leaf nodes store actual records, or else keys and pointers to actual records in a separate disk file if the B⁺ tree is being used purely as an index. Depending on the size of a record as compared to the size of a key, a leaf node in a B⁺ tree of order m might have enough room to store more or less than m records. The requirement is simply that the leaf nodes store enough records to remain at least half full. The leaf nodes of a B⁺ tree are normally linked together to form a doubly linked list. Thus, the entire collection of records can be traversed in sorted order by visiting all the leaf nodes on the linked list. Here is a Java-like pseudocode representation for the B⁺ tree node interface. Leaf node and internal node subclasses would implement this interface.

```
/** Interface for B+ Tree nodes */
public interface BPNode<Key, E> {
    public boolean isLeaf();
    public int numrecs();
    public Key[] keys();
}
```

An important implementation detail to note is that while Figure 12.6.1 shows internal nodes containing three keys and four pointers, class BPNode is slightly different in that it stores key/pointer pairs. Figure 12.6.1 shows the B⁺ tree as it is traditionally drawn. To simplify implementation in practice, nodes really do associate a key with each pointer. Each internal node should be assumed to hold in the

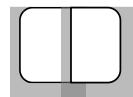
leftmost position an additional key that is less than or equal to any possible key value in the node's leftmost subtree. B⁺ tree implementations typically store an additional dummy record in the leftmost leaf node whose key value is less than any legal key value.

Let's see in some detail how the simplest B⁺ tree works. This would be the "2 – 3⁺ tree", or a B⁺ tree of order 3.

1 / 28



Example 2-3+ Tree Visualization: Insert

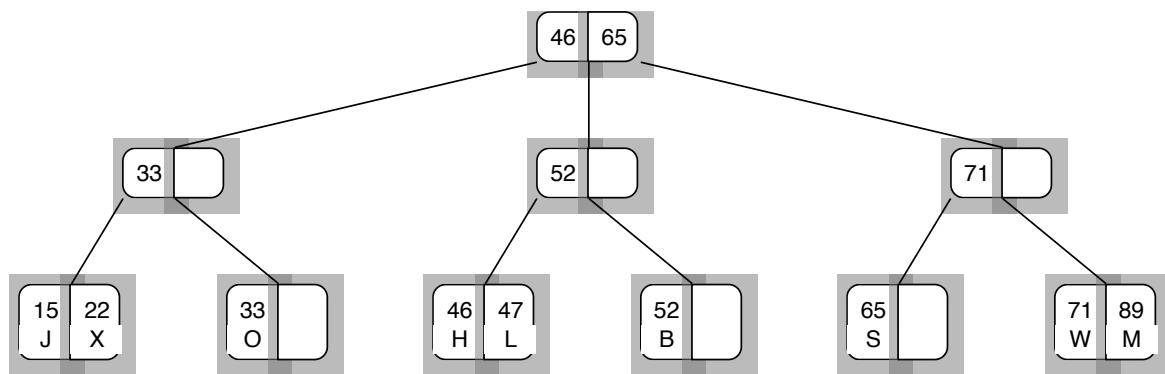
Figure 12.6.2: An example of building a 2 – 3⁺ tree

Next, let's see how to search.

1 / 10



Example 2-3+ Tree Visualization: Search

Figure 12.6.3: An example of searching a 2 – 3⁺ tree

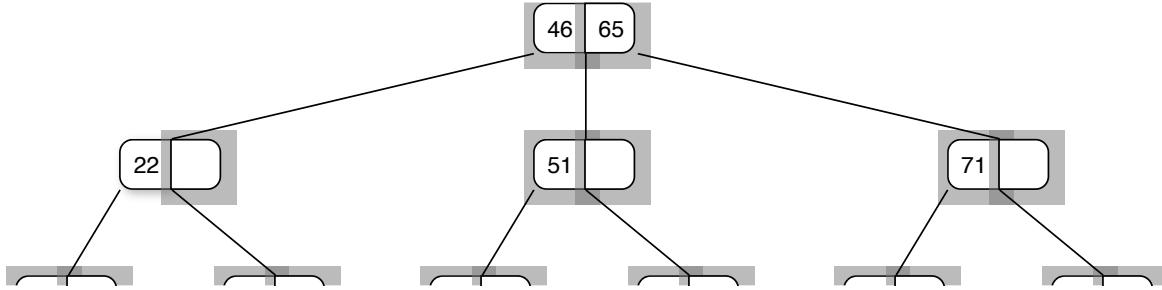
Finally, let's see an example of deleting from the 2 – 3⁺ tree

1 / 33





Example 2-3+ Tree Visualization: Delete

Figure 12.6.4: An example of deleting from a 2 – 3⁺ tree

Now, let's extend these ideas to a B⁺ tree of higher order.

B⁺ trees are exceptionally good for range queries. Once the first record in the range has been found, the rest of the records with keys in the range can be accessed by sequential processing of the remaining records in the first node, and then continuing down the linked list of leaf nodes as far as necessary. Figure illustrates the B⁺ tree.



Example B+ Tree Visualization: Search in a tree of degree 4

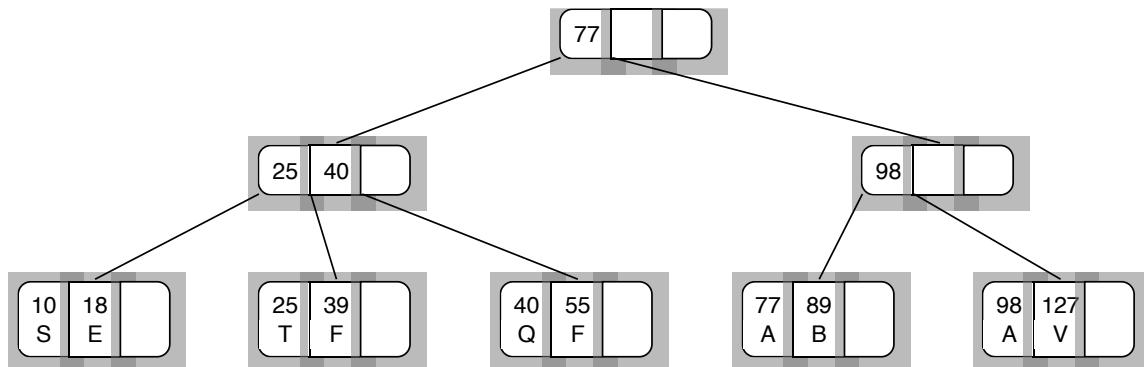


Figure 12.6.5: An example of search in a B+ tree of order four. Internal nodes must store between two and four children.

Search in a B⁺ tree is nearly identical to search in a regular B-tree, except that the search must always continue to the proper leaf node. Even if the search-key value is found in an internal node, this is only a placeholder and does not provide access to the actual record. Here is a pseudocode sketch of the B⁺ tree search algorithm.

```
private E findhelp(BPNode<Key, E> rt, Key k) {
    int currec = binaryle(rt.keys(), rt.numrecs(), k);
    if (rt.isLeaf()) {
        if (((BPLeaf<Key, E>)rt).keys())[currec] == k) {
```

```

        return ((BPLeaf<Key,E>)rt).recs(currec);
    }
    else { return null; }
}
else{
    return findhelp(((BPInternal<Key,E>)rt).pointers(currec), k);
}
}
}

```

B^+ tree insertion is similar to B-tree insertion. First, the leaf L that should contain the record is found. If L is not full, then the new record is added, and no other B^+ tree nodes are affected. If L is already full, split it in two (dividing the records evenly among the two nodes) and promote a copy of the least-valued key in the newly formed right node. As with the 2-3 tree, promotion might cause the parent to split in turn, perhaps eventually leading to splitting the root and causing the B^+ tree to gain a new level. B^+ tree insertion keeps all leaf nodes at equal depth. Figure illustrates the insertion process through several examples.

1 / 42



Example B+ Tree Visualization: Insert into a tree of degree 4

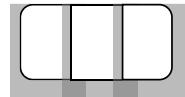


Figure 12.6.6: An example of building a B+ tree of order four.

Here is a Java-like pseudocode sketch of the B^+ tree insert algorithm.

```

private BPNode<Key,E> inserthelp(BPNode<Key,E> rt,
                                    Key k, E e) {
    BPNode<Key,E> retval;
    if (rt.isLeaf()) { // At leaf node: insert here
        return ((BPLeaf<Key,E>)rt).add(k, e);
    }
    // Add to internal node
    int currec = binaryle(rt.keys(), rt.numrecs(), k);
    BPNode<Key,E> temp = inserthelp(
        ((BPInternal<Key,E>)root).pointers(currec), k, e);
    if (temp != ((BPInternal<Key,E>)rt).pointers(currec)) {
        return ((BPInternal<Key,E>)rt).
            add((BPInternal<Key,E>)temp);
    }
    else{
        return rt;
    }
}

```

Here is an exercise to see if you get the basic idea of B⁺ tree insertion.

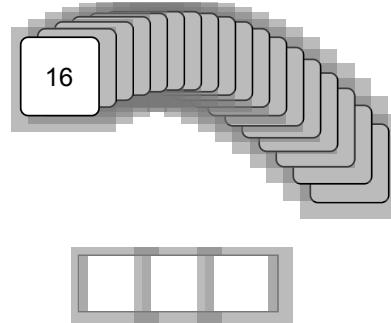
B⁺ Tree Insertion

Instructions:

In this exercise your job is to insert the values from the stack to the B⁺ tree.

Search for the leaf node where the topmost value of the stack should be inserted, and click on that node. The exercise will take care of the rest. Continue this procedure until you have inserted all the values in the stack.

[Undo](#) [Reset](#) [Model Answer](#) [Grade](#)

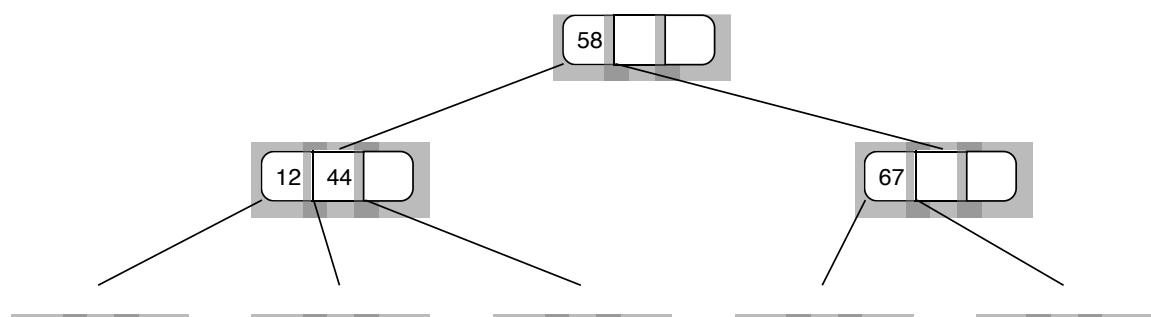


To delete record R from the B⁺ tree, first locate the leaf L that contains R. If L is more than half full, then we need only remove R, leaving L still at least half full. This is demonstrated by Figure .

1 / 23



Example B+ Tree Visualization: Delete from a tree of degree 4



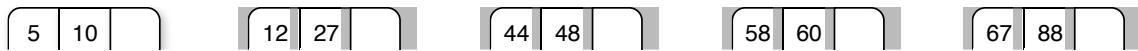


Figure 12.6.7: An example of deletion in a B+ tree of order four.

If deleting a record reduces the number of records in the node below the minimum threshold (called an **underflow**), then we must do something to keep the node sufficiently full. The first choice is to look at the node's adjacent siblings to determine if they have a spare record that can be used to fill the gap. If so, then enough records are transferred from the sibling so that both nodes have about the same number of records. This is done so as to delay as long as possible the next time when a delete causes this node to underflow again. This process might require that the parent node has its placeholder key value revised to reflect the true first key value in each node.

If neither sibling can lend a record to the under-full node (call it N), then N must give its records to a sibling and be removed from the tree. There is certainly room to do this, because the sibling is at most half full (remember that it had no records to contribute to the current node), and N has become less than half full because it is under-flowing. This merge process combines two subtrees of the parent, which might cause it to underflow in turn. If the last two children of the root merge together, then the tree loses a level.

Here is a Java-like pseudocode for the B⁺ tree delete algorithm.

```
/** Delete a record with the given key value, and
   return true if the root underflows */
private boolean removehelp(BPNode<Key, E> rt, Key k) {
    int currec = binaryle(rt.keys(), rt.numrecs(), k);
    if (rt.isLeaf()) {
        if (((BPLear<Key, E>)rt).keys()[currec] == k) {
            return ((BPLear<Key, E>)rt).delete(currec);
        }
        else { return false; }
    }
    else{ // Process internal node
        if (removehelp(((BPInternal<Key, E>)rt).pointers(currec),
                      k)) {
            // Child will merge if necessary
            return ((BPInternal<Key, E>)rt).underflow(currec);
        }
        else { return false; }
    }
}
```

The B⁺ tree requires that all nodes be at least half full (except for the root). Thus, the storage utilization must be at least 50%. This is satisfactory for many implementations, but note that keeping nodes fuller will result both in less space required (because there is less empty space in the disk file) and in more efficient processing (fewer blocks on average will be read into memory because the amount of information in each block is greater). Because B-trees have become so popular, many algorithm designers have tried to improve B-tree performance. One method for doing so is to use the B⁺ tree variant known as the B^{*} tree. The B^{*} tree is identical to the B⁺ tree, except for the rules used to split and merge nodes. Instead of splitting a node in half when it overflows, the B^{*} tree gives some records to its neighboring sibling, if possible. If the sibling is also full, then these two nodes split into three. Similarly, when a node underflows, it is combined with its two siblings, and the total reduced to two nodes. Thus, the nodes are always at least two thirds full. [1]

Finally, here is an example of building a B+ Tree of order five. You can compare this to the example above of building a tree of order four with the same records.



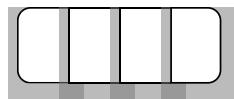


Figure 12.6.8: An example of building a B+ tree of degree 5

[Click here](#) for a visualization that will let you construct and interact with a B⁺ tree. This visualization was written by David Galles of the University of San Francisco as part of his [Data Structure Visualizations](#) package.

[1]

This concept can be extended further if higher space utilization is required. However, the update routines become much more complicated. I once worked on a project where we implemented 3-for-4 node split and merge routines. This gave better performance than the 2-for-3 node split and merge routines of the B^{*} tree. However, the splitting and merging routines were so complicated that even their author could no longer understand them once they were completed!

12.6.1.2. B-Tree Analysis

The asymptotic cost of search, insertion, and deletion of records from B-trees, B⁺ trees, and B^{*} trees is $\Theta(\log n)$ where n is the total number of records in the tree. However, the base of the log is the (average) branching factor of the tree. Typical database applications use extremely high branching factors, perhaps 100 or more. Thus, in practice the B-tree and its variants are extremely shallow.

As an illustration, consider a B⁺ tree of order 100 and leaf nodes that contain up to 100 records. A B-B⁺ tree with height one (that is, just a single leaf node) can have at most 100 records. A B⁺ tree with height two (a root internal node whose children are leaves) must have at least 100 records (2 leaves with 50 records each). It has at most 10,000 records (100 leaves with 100 records each). A B⁺ tree with height three must have at least 5000 records (two second-level nodes with 50 children containing 50 records each) and at most one million records (100 second-level nodes with 100 full children each). A B⁺ tree with height four must have at least 250,000 records and at most 100 million records. Thus, it would require an *extremely* large database to generate a B⁺ tree of more than height four.

The B⁺ tree split and insert rules guarantee that every node (except perhaps the root) is at least half full. So they are on average about 3/4 full. But the internal nodes are purely overhead, since the keys stored there are used only by the tree to direct search, rather than store actual data. Does this overhead amount to a significant use of space? No, because once again the high fan-out rate of the tree structure means that the vast majority of nodes are leaf nodes. A **K-ary tree** has approximately $1/K$ of its nodes as internal nodes. This means that while half of a full binary tree's nodes are internal nodes, in a B⁺ tree of order 100 probably only about 1/75 of its nodes are internal nodes. This means that the overhead associated with internal nodes is very low.

We can reduce the number of disk fetches required for the B-tree even more by using the following methods. First, the upper levels of the tree can be stored in main memory at all times. Because the tree branches so quickly, the top two levels (levels 0 and 1) require relatively little space. If the B-tree is only height four, then at most two disk fetches (internal nodes at level two and leaves at level three) are required to reach the pointer to any given record.

A buffer pool could be used to manage nodes of the B-tree. Several nodes of the tree would typically be in main memory at one time. The most straightforward approach is to use a standard method such as LRU to do node replacement. However, sometimes it might be desirable to “lock” certain nodes such as the root into the buffer pool. In general, if the buffer pool is even of modest size (say at least twice the depth of the tree), no special techniques for node replacement will be required because the upper-level nodes will naturally be accessed frequently.



Chapter 12: Binary Search Trees

A **binary search tree** is a binary tree with a special property called the **BST-property**, which is given as follows:

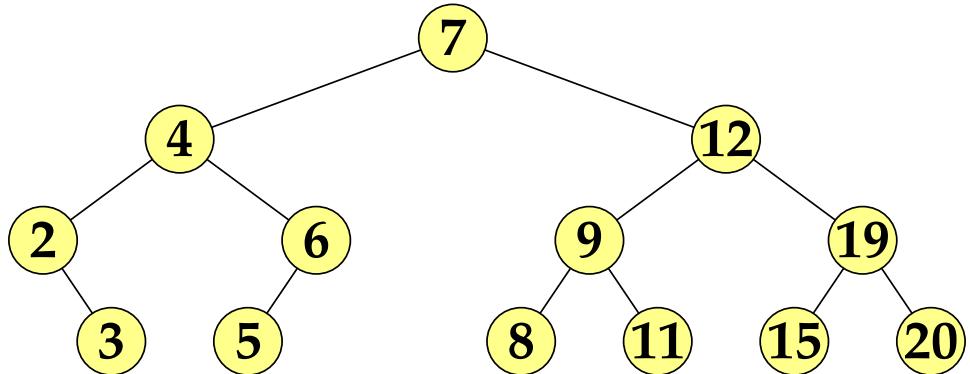
- * *For all nodes x and y , if y belongs to the left subtree of x , then the key at y is less than the key at x , and if y belongs to the right subtree of x , then the key at y is greater than the key at x .*

We will assume that the keys of a BST are pairwise distinct.

Each node has the following attributes:

- p , $left$, and $right$, which are pointers to the parent, the left child, and the right child, respectively, and
- key , which is key stored at the node.

An example



Traversal of the Nodes in a BST

By “traversal” we mean visiting all the nodes in a graph. Traversal strategies can be specified by the ordering of the three objects to visit: the current node, the left subtree, and the right subtree. We assume the the left subtree always comes before the right subtree. Then there are three strategies.

1. **Inorder**. The ordering is: the left subtree, the current node, the right subtree.
2. **Preorder**. The ordering is: the current node, the left subtree, the right subtree.
3. **Postorder**. The ordering is: the left subtree, the right subtree, the current node.

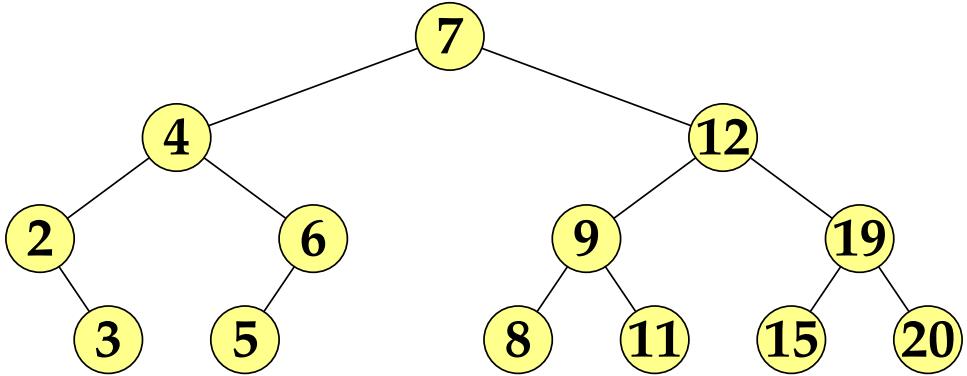
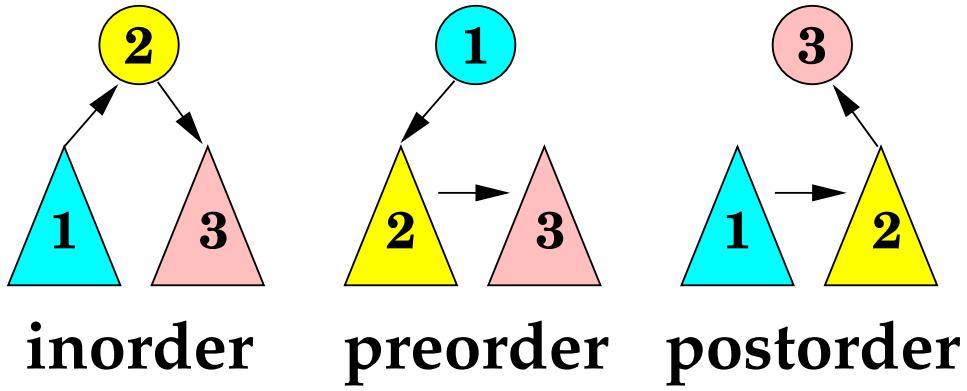
Inorder Traversal Pseudocode

This recursive algorithm takes as the input a pointer to a tree and executed inorder traversal on the tree. While doing traversal it prints out the key of each node that is visited.

Inorder-Walk(x)

- 1: **if** $x = \text{nil}$ **then return**
- 2: **Inorder-Walk($\text{left}[x]$)**
- 3: Print $\text{key}[x]$
- 4: **Inorder-Walk($\text{right}[x]$)**

We can write a similar pseudocode for preorder and postorder.



*What is the outcome of
inorder traversal on this BST?
How about postorder traversal
and preorder traversal?*

Inorder traversal gives: 2, 3,
4, 5, 6, 7, 8 , 9, 11, 12, 15,
19, 20.

Preorder traversal gives: 7, 4,
2, 3, 6, 5, 12, 9, 8, 11, 19,
15, 20.

Postorder traversal gives: 3,
2, 5, 6, 4, 8, 11, 9, 15, 20,
19, 12, 7.

So, inorder travel on a BST
finds the keys in
nondecreasing order!

Operations on BST

1. Searching for a key

We assume that a key and the subtree in which the key is searched for are given as an input. We'll take the full advantage of the BST-property.

Suppose we are at a node. If the node has the key that is being searched for, then the search is over. Otherwise, the key at the current node is either strictly smaller than the key that is searched for or strictly greater than the key that is searched for. If the former is the case, then by the BST property, all the keys in the left subtree are strictly less than the key that is searched for. That means that we do not need to search in the left subtree. Thus, we will examine only the right subtree. If the latter is the case, by symmetry we will examine only the right subtree.

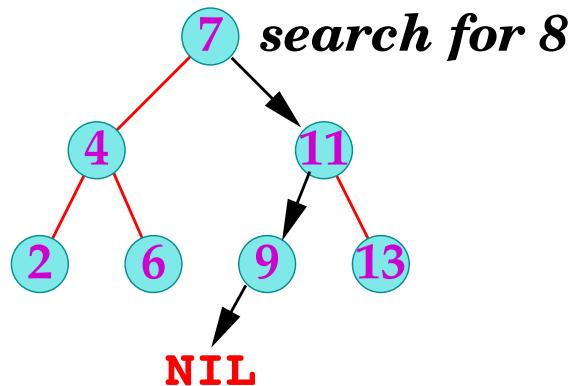
Algorithm

Here k is the key that is searched for and x is the start node.

BST-Search(x, k)

```
1:  $y \leftarrow x$ 
2: while  $y \neq \text{nil}$  do
3:   if  $\text{key}[y] = k$  then return  $y$ 
4:   else if  $\text{key}[y] < k$  then  $y \leftarrow \text{right}[y]$ 
5:   else  $y \leftarrow \text{left}[y]$ 
6: return (“NOT FOUND”)
```

An Example



*What is the running time of
search?*

2. The Maximum and the Minimum

To find the minimum identify the leftmost node, i.e. the farthest node you can reach by following only left branches.

To find the maximum identify the rightmost node, i.e. the farthest node you can reach by following only right branches.

BST-Minimum(x)

- 1: **if** $x = \text{nil}$ **then return** (“Empty Tree”)
- 2: $y \leftarrow x$
- 3: **while** $\text{left}[y] \neq \text{nil}$ **do** $y \leftarrow \text{left}[y]$
- 4: **return** ($\text{key}[y]$)

BST-Maximum(x)

- 1: **if** $x = \text{nil}$ **then return** (“Empty Tree”)
- 2: $y \leftarrow x$
- 3: **while** $\text{right}[y] \neq \text{nil}$ **do** $y \leftarrow \text{right}[y]$
- 4: **return** ($\text{key}[y]$)

3. Insertion

Suppose that we need to insert a node z such that $k = \text{key}[z]$. Using binary search we find a nil such that replacing it by z does not break the BST-property.

BST-Insert(x, z, k)

```
1: if  $x = \text{nil}$  then return “Error”
2:  $y \leftarrow x$ 
3: while true do {
4:   if  $\text{key}[y] < k$ 
5:     then  $z \leftarrow \text{left}[y]$ 
6:     else  $z \leftarrow \text{right}[y]$ 
7:   if  $z = \text{nil}$  break
8: }
9: if  $\text{key}[y] > k$  then  $\text{left}[y] \leftarrow z$ 
10: else  $\text{right}[p[y]] \leftarrow z$ 
```

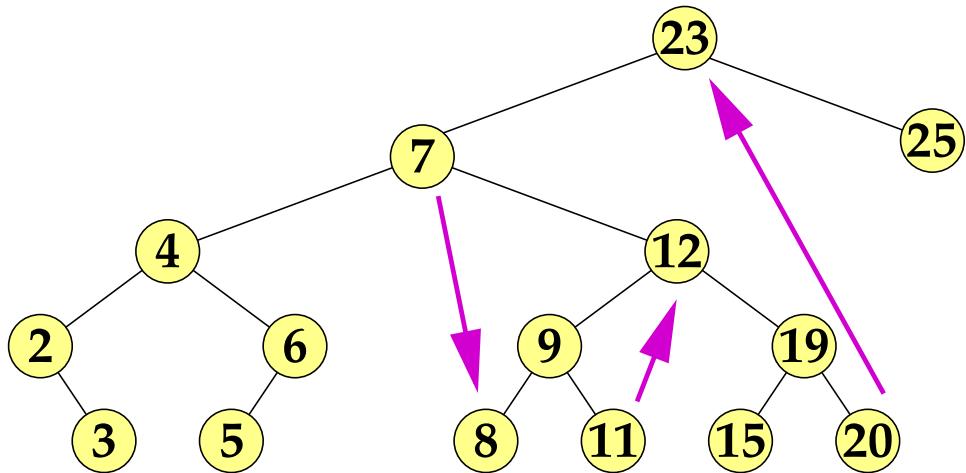
4. The Successor and The Predecessor

The successor (respectively, the predecessor) of a key k in a search tree is the smallest (respectively, the largest) key that belongs to the tree and that is strictly greater than (respectively, less than) k .

The idea for finding the successor of a given node x .

- If x has the right child, then the successor is the minimum in the right subtree of x .
- Otherwise, the successor is the parent of the farthest node that can be reached from x by following only right branches backward.

An Example



Algorithm

BST-Successor(x)

```
1: if  $right[x] \neq \text{nil}$  then
2: {    $y \leftarrow right[x]$ 
3:   while  $left[y] \neq \text{nil}$  do  $y \leftarrow left[y]$ 
4:   return ( $y$ ) }
5: else
6: {    $y \leftarrow x$ 
7:   while  $right[p[x]] = x$  do  $y \leftarrow p[x]$ 
8:   if  $p[x] \neq \text{nil}$  then return ( $p[x]$ )
9:   else return ("NO SUCCESSOR") }
```

The predecessor can be found similarly with the roles of left and right exchanged and with the roles of maximum and minimum exchanged.

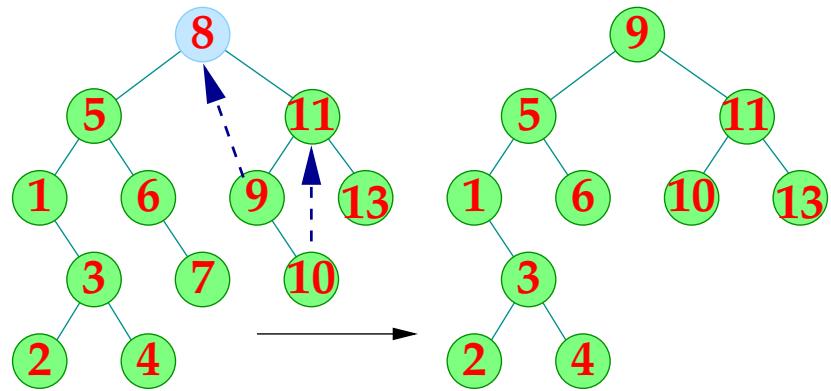
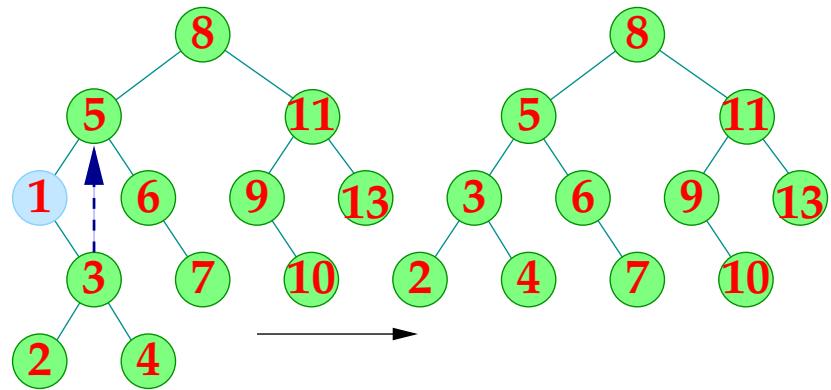
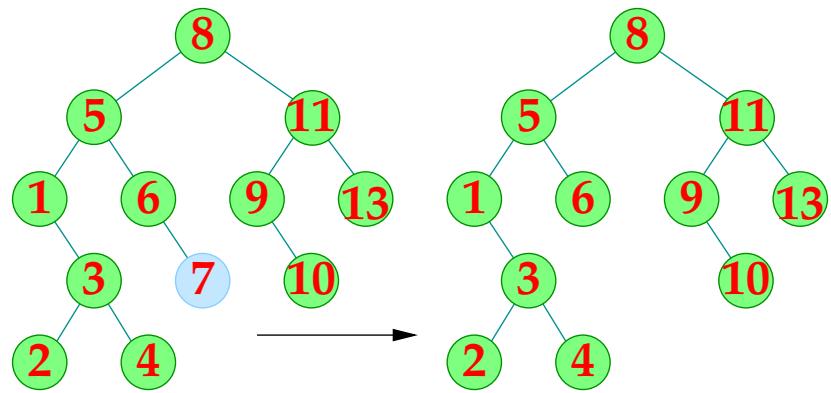
For which node is the successor undefined?

What is the running time of the successor algorithm?

5. Deletion

Suppose we want to delete a node z .

1. If z has no children, then we will just replace z by **nil**.
2. If z has only one child, then we will promote the unique child to z 's place.
3. If z has two children, then we will identify z 's successor. Call it y . The successor y either is a leaf or has only the right child. Promote y to z 's place. Treat the loss of y using one of the above two solutions.



Algorithm

This algorithm deletes z from BST T .

BST-Delete(T, z)

- 1: **if** $left[z] = \text{nil}$ **or** $right[z] = \text{nil}$
- 2: **then** $y \leftarrow z$
- 3: **else** $y \leftarrow \text{BST-Successor}(z)$
- 4: $\triangleright y$ is the node that's actually removed.
- 5: \triangleright Here y does not have two children.
- 6: **if** $left[y] \neq \text{nil}$
- 7: **then** $x \leftarrow left[y]$
- 8: **else** $x \leftarrow right[y]$
- 9: $\triangleright x$ is the node that's moving to y 's position.
- 10: **if** $x \neq \text{nil}$ **then** $p[x] \leftarrow p[y]$
- 11: $\triangleright p[x]$ is reset If x isn't NIL.
- 12: \triangleright Resetting is unnecessary if x is NIL.

Algorithm (cont'd)

```
13: if  $p[y] = \text{nil}$  then  $\text{root}[T] \leftarrow x$ 
14:  $\triangleright$  If  $y$  is the root, then  $x$  becomes the root.
15:  $\triangleright$  Otherwise, do the following.
16: else if  $y = \text{left}[p[y]]$ 
17:   then  $\text{left}[p[y]] \leftarrow x$ 
18:  $\triangleright$  If  $y$  is the left child of its parent, then
19:  $\triangleright$  Set the parent's left child to  $x$ .
20:   else  $\text{right}[p[y]] \leftarrow x$ 
21:  $\triangleright$  If  $y$  is the right child of its parent, then
22:  $\triangleright$  Set the parent's right child to  $x$ .
23: if  $y \neq z$  then
24:   {  $\text{key}[z] \leftarrow \text{key}[y]$ 
25:     Move other data from  $y$  to  $z$  }
27: return ( $y$ )
```

Summary of Efficiency Analysis

Theorem A On a binary search tree of height h , Search, Minimum, Maximum, Successor, Predecessor, Insert, and Delete can be made to run in $O(h)$ time.

Randomly built BST

Suppose that we insert n distinct keys into an initially empty tree. Assuming that the $n!$ permutations are equally likely to occur, what is the average height of the tree?

To study this question we consider the process of constructing a tree T by **inserting in order randomly selected n distinct keys** to an initially empty tree. Here the actually values of the keys do not matter. What matters is the position of the inserted key in the n keys.

The Process of Construction

So, we will view the process as follows:

A key x from the keys is selected uniformly at random and is inserted to the tree. Then all the other keys are inserted. Here all the keys greater than x go into the right subtree of x and all the keys smaller than x go into the left subtree. Thus, the height of the tree thus constructed is one plus the larger of the height of the left subtree and the height of the right subtree.

Random Variables

n = number of keys

X_n = height of the tree of n keys

$$Y_n = 2^{X_n}.$$

We want an upper bound on $E[Y_n]$.

For $n \geq 2$, we have

$$E[Y_n] = \frac{1}{n} \left(\sum_{i=1}^n 2E[\max\{Y_{i-1}, Y_{n-i}\}] \right).$$

$$\begin{aligned} E[\max\{Y_{i-1}, Y_{n-i}\}] &\leq E[Y_{i-1} + Y_{n-i}] \\ &\leq E[Y_{i-1}] + E[Y_{n-i}] \end{aligned}$$

Collecting terms:

$$E[Y_n] \leq \frac{4}{n} \sum_{i=1}^{n-1} E[Y_i].$$

Analysis

We claim that for all $n \geq 1$ $E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}$.
We prove this by induction on n .

Base case: $E[Y_1] = 2^0 = 1$.

Induction step: We have

$$E[Y_n] \leq \frac{4}{n} \sum_{i=1}^{n-1} E[Y_i]$$

Using the fact that

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$$

$$E[Y_n] \leq \frac{4}{n} \cdot \frac{1}{4} \cdot \binom{n+3}{4}$$

$$E[Y_n] \leq \frac{1}{4} \cdot \binom{n+3}{3}$$

Jensen's inequality

A function f is **convex** if for all x and y , $x < y$, and for all λ , $0 \leq \lambda \leq 1$,

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

Jensen's inequality states that for all random variables X and for all convex function f

$$f(E[X]) \leq E[f(X)].$$

Let this X be X_n and $f(x) = 2^x$. Then $E[f(X)] = E[Y_n]$. So, we have

$$2^{E[X_n]} \leq \frac{1}{4} \binom{n+3}{3}.$$

The right-hand side is at most $(n+3)^3$. By taking the log of both sides, we have

$$E[X_n] = O(\log n).$$

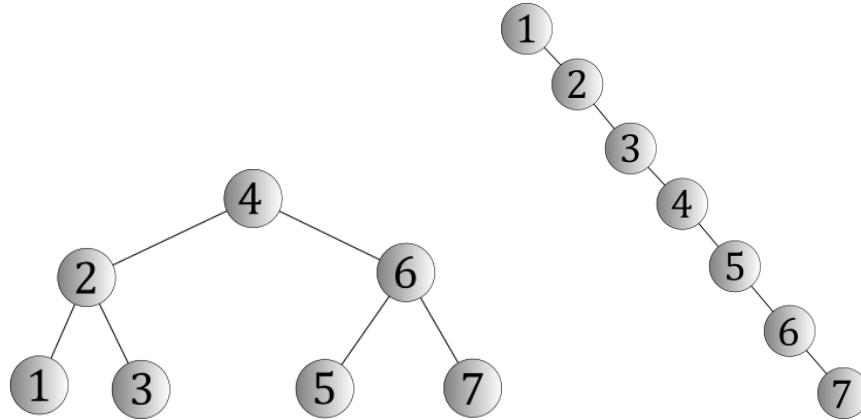
Thus the average height of a randomly build BST is $O(\log n)$.

ICS 46 Spring 2022

Notes and Examples: AVL Trees

Why we must care about binary search tree balancing

We've seen previously that the performance characteristics of [binary search trees](#) can vary rather wildly, and that they're mainly dependent on the shape of the tree, with the height of the tree being the key determining factor. By definition, binary search trees restrict what keys are allowed to present in which nodes — smaller keys have to be in left subtrees and larger keys in right subtrees — but they specify no restriction on the tree's shape, meaning that both of these are perfectly legal binary search trees containing the keys 1, 2, 3, 4, 5, 6, and 7.



Yet, while both of these are legal, one is better than the other, because the height of the first tree (called a *perfect binary tree*) is smaller than the height of the second (called a *degenerate tree*). These two shapes represent the two extremes — the best and worst possible shapes for a binary search tree containing seven keys.

Of course, when all you have is a very small number of keys like this, any shape will do. But as the number of keys grows, the distinction between these two tree shapes becomes increasingly vital. What's more, the degenerate shape isn't even necessarily a rare edge case: It's what you get when you start with an empty tree and add keys that are already in order, which is a surprisingly common scenario in real-world programs. For example, one very obvious algorithm for generating unique integer keys — when all you care about is that they're unique — is to generate them sequentially.

What's so bad about a degenerate tree, anyway?

Just looking at a picture of a degenerate tree, your intuition should already be telling you that something is amiss. In particular, if you tilt your head 45 degrees to the right, they look just like linked lists; that perception is no accident, as they behave like them, too (except that they're more complicated, to boot!).

From a more analytical perspective, there are three results that should give us pause:

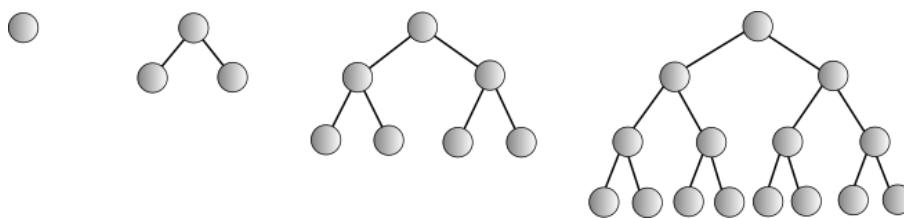
- Every time you perform a lookup in a degenerate binary search tree, it will take $O(n)$ time, because it's possible that you'll have to reach every node in the tree before you're done. As n grows, this is a heavy burden to bear.
- If you implement your lookup recursively, you might also be using $O(n)$ memory, too, as you might end up with as many as n frames on your run-time stack — one for every recursive call. There are ways to mitigate this — for example, some kinds of carefully-written recursion (in some programming languages, including C++) can avoid run-time stack growth as you recurse — but it's still a sign of potential trouble.
- The time it will take you to build the degenerate tree will also be prohibitive. If you start with an empty binary search tree and add keys to it in order, how long does it take to do it?
 - The first key you add will go directly to the root. You could think of this as taking a single step: creating the node.
 - The second key you add will require you to look at the root node, then take one step to the right. You could think of this as taking two steps.
 - Each subsequent key you add will require one more step than the one before it.
 - The total number of steps it would take to add n keys would be determined by the sum $1 + 2 + 3 + \dots + n$. This sum, which we'll see several times throughout this course, is equal to $n(n + 1) / 2$.
 - So, the total number of steps to build the entire tree would be $\Theta(n^2)$.

Overall, when n gets large, the tree would be hideously expensive to build, and then every subsequent search would be painful, as well. So this, in general, is a situation we need to be sure to avoid, or else we should probably consider a data structure other than a binary search tree; the worst case is simply too much of a burden to bear if n might get large. But if we can find a way to control the tree's shape more carefully, to force it to remain more *balanced*, we'll be fine. The question, of course, is how to do it, and, as importantly, whether we can do it while keeping the cost low enough that it doesn't outweigh the benefit.

Aiming for perfection

The best goal for us to shoot for would be to maintain perfection. In other words, every time we insert a key into our binary search tree, it would ideally still be a perfect binary tree, in which case we'd know that the height of the tree would always be $\Theta(\log n)$, with a commensurate effect on performance.

However, when we consider this goal, a problem emerges almost immediately. The following are all perfect binary trees, by definition:



The perfect binary trees pictured above have 1, 3, 7, and 15 nodes respectively, and are the only possible perfect shapes for binary trees with that number of nodes. The problem, though, lies in the fact that there is no valid perfect binary tree with 2 nodes, or with 4, 5, 6, 8, 9, 10, 11, 12, 13, or 14 nodes. So, generally, it's impossible for us to guarantee that a binary search tree will always be "perfect," by our definition, because there's simply no way to represent most numbers of keys.

So, first thing's first: We'll need to relax our definition of "perfection" to accommodate every possible number of keys we might want to store.

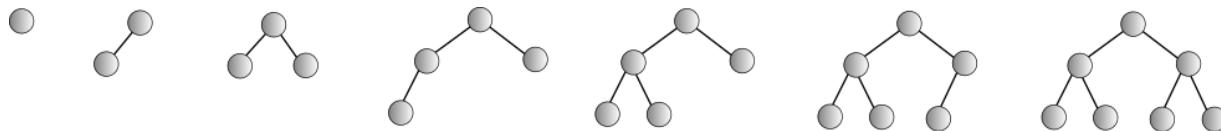
Complete binary trees

A somewhat more relaxed notion of "perfection" is something called a *complete binary tree*, which is defined as follows.

A *complete binary tree* of height h is a binary tree where:

- If $h = 0$, its left and right subtrees are empty.
- If $h > 0$, one of two things is true:
 - The left subtree is a perfect binary tree of height $h - 1$ and the right subtree is a complete binary tree of height $h - 1$
 - The left subtree is a complete binary tree of height $h - 1$ and the right subtree is a perfect binary tree of height $h - 2$

That can be a bit of a mind-bending definition, but it actually leads to a conceptually simple result: On every level of a complete binary tree, every node that could possibly be present will be, *except* the last level might be missing nodes, but if it is missing nodes, the nodes that are there will be as far to the left as possible. The following are all complete binary trees:



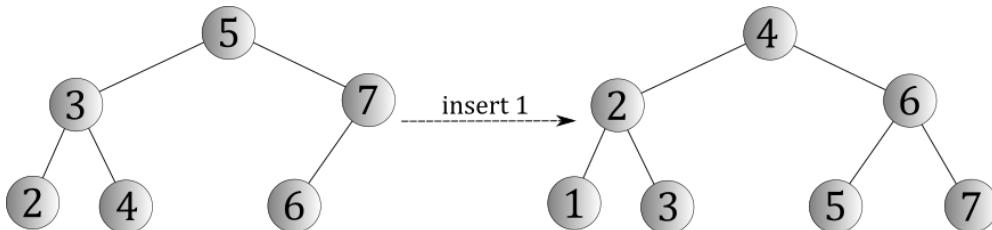
Furthermore, these are the only possible complete binary trees with these numbers of nodes in them; any other arrangement of, say, 6 keys besides the one shown above would violate the definition.

We've seen that the height of a perfect binary tree is $\Theta(\log n)$. It's not a stretch to see that the height of a complete binary tree will be $\Theta(\log n)$, as well, and we'll accept that via our intuition for now and proceed. All in all, a complete binary tree would be a great goal for us to attain: If we could keep the shape of our binary search trees complete, we would always have binary search trees with height $\Theta(\log n)$.

The cost of maintaining completeness

The trouble, of course, is that we need an algorithm for maintaining completeness. And before we go to the trouble of trying to figure one out, we should consider whether it's even worth our time. What can we deduce about the cost of maintaining completeness, even if we haven't figured out an algorithm yet?

One example demonstrates a very big problem. Suppose we had the binary search tree on the left — which is complete, by our definition — and we wanted to insert the key 1 into it. If so, we would need an algorithm that would transform the tree on the left into the tree on the right.



The tree on the right is certainly complete, so this would be the outcome we'd want. But consider what it would take to do it. *Every key in the tree had to move!* So, no matter what algorithm we used, we would still have to move every key. If there are n keys in the tree, that would take $\Omega(n)$ time — moving n keys takes at least linear time, even if you have the best possible algorithm for moving them; the work still has to get done.

So, in the worst case, maintaining completeness after a single insertion requires $\Omega(n)$ time. Unfortunately, this is more time than we ought to be spending on maintaining balance. This means we'll need to come up with a compromise; as is often the case when we learn or design algorithms, our willingness to tolerate an imperfect result that's still "good enough" for our uses will often lead to an algorithm that is much faster than one that achieves a perfect result. So what would a "good enough" result be?

What is a "good" balance condition

Our overall goal is for lookups, insertions, and removals from a binary search tree to require $O(\log n)$ time in every case, rather than letting them degrade to a worst-case behavior of $O(n)$. To do that, we need to decide on a *balance condition*, which is to say that we need to understand what shape is considered well-

enough balanced for our purposes, even if not perfect.

A "good" balance condition has two properties:

- The height of a binary search tree meeting the condition is $\Theta(\log n)$.
- It takes $O(\log n)$ time to re-balance the tree on insertions and removals.

In other words, it guarantees that the height of the tree is still logarithmic, which will give us logarithmic-time lookups, and the time spent re-balancing won't exceed the logarithmic time we would otherwise spend on an insertion or removal when the tree has logarithmic height. The cost won't outweigh the benefit.

Coming up with a balance condition like this on our own is a tall task, but we can stand on the shoulders of the giants who came before us, with the definition above helping to guide us toward an understanding of whether we've found what we're looking for.

A compromise: AVL trees

There are a few well-known approaches for maintaining binary search trees in a state of near-balance that meets our notion of a "good" balance condition. One of them is called an *AVL tree*, which we'll explore here. Others, which are outside the scope of this course, include red-black trees (which meet our definition of "good") and splay trees (which don't always meet our definition of "good", but do meet it on an amortized basis), but we'll stick with the one solution to the problem for now.

AVL trees

AVL trees are what you might call "nearly balanced" binary search trees. While they certainly aren't as perfectly-balanced as possible, they nonetheless achieve the goals we've decided on: maintaining logarithmic height at no more than logarithmic cost.

So, what makes a binary search tree "nearly balanced" enough to be considered an AVL tree? The core concept is embodied by something called the *AVL property*.

We say that a node in a binary search tree has the *AVL property* if the heights of its left and right subtrees differ by no more than 1.

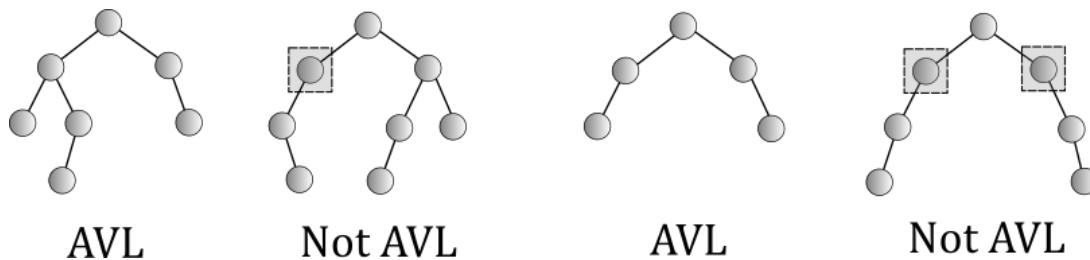
In other words, we tolerate a certain amount of imbalance — heights of subtrees can be slightly different, but no more than that — in hopes that we can more efficiently maintain it.

Since we're going to be comparing heights of subtrees, there's one piece of background we need to consider. Recall that the *height of a tree* is the length of its longest path. By definition, the height of a tree with just a root node (and empty subtrees) would then be zero. But what about a tree that's totally empty? To maintain a clear pattern, relative to other tree heights, we'll say that the *height of an empty tree* is -1. This means that a node with, say, a childless left child and no right child would still be considered balanced.

This leads us, finally, to the definition of an AVL tree:

An AVL tree is a binary search tree in which all nodes have the AVL property.

Below are a few binary trees, two of which are AVL and two of which are not.



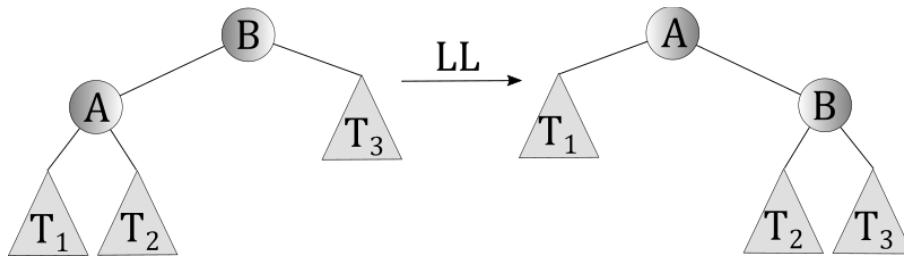
The thing to keep in mind about AVL is that it's not a matter of squinting at a tree and deciding whether it "looks" balanced. There's a precise definition, and the two trees above that don't meet that definition fail to meet it because they each have at least one node (marked in the diagrams by a dashed square) that doesn't have the AVL property.

AVL trees, by definition, are required to meet the balance condition after every operation; every time you insert or remove a key, every node in the tree should have the AVL property. To meet that requirement, we need to restructure the tree periodically, essentially detecting and correcting imbalance whenever and wherever it happens. To do that, we need to rearrange the tree in ways that improve its shape without losing the essential ordering property of a binary search tree: smaller keys toward the left, larger ones toward the right.

Rotations

Re-balancing of AVL trees is achieved using what are called *rotations*, which, when used at the proper times, efficiently improve the shape of the tree by altering a handful of pointers. There are a few kinds of rotations; we should first understand how they work, then focus our attention on when to use them.

The first kind of rotation is called an *LL rotation*, which takes the tree on the left and turns it into the tree on the right. The circle with A and B written in them are each a single node containing a single key; the triangles with T_1 , T_2 , and T_3 written in them are arbitrary subtrees, which may be empty or may contain any number of nodes (but which are, themselves, binary search trees).



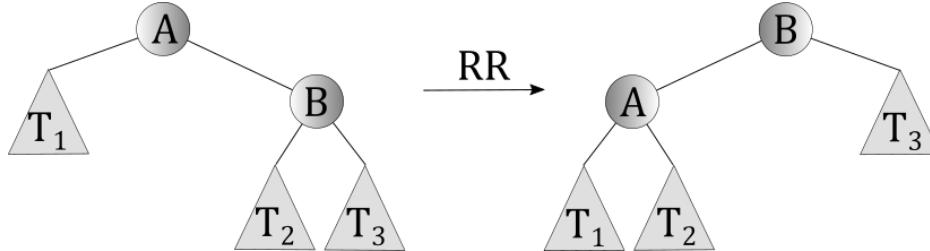
It's important to remember that both of these trees — before and after — are binary search trees; the rotation doesn't harm the ordering of the keys in nodes, because the subtrees T_1 , T_2 , and T_3 maintain the appropriate positions relative to the keys A and B:

- All keys in T_1 are smaller than A.
- All keys in T_2 are larger than A and smaller than B.
- All keys in T_3 are larger than B.

Performing this rotation would be a simple matter of adjusting a few pointers — notably, a constant number of pointers, no matter how many nodes are in the tree, which means that this rotation would run in $\Theta(1)$ time:

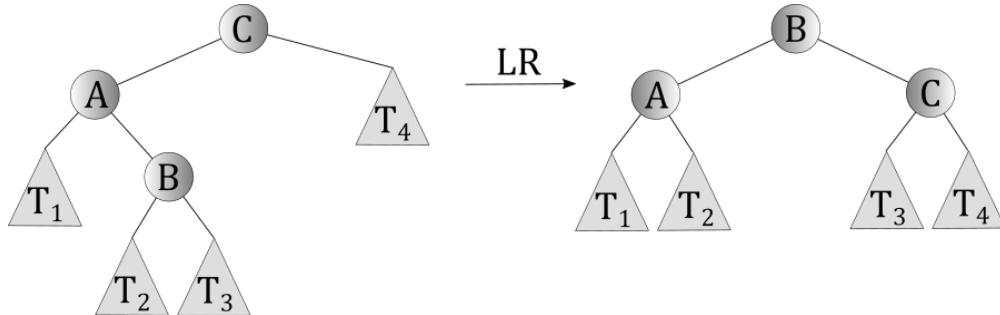
- B's parent would now point to A where it used to point to B
- A's right child would now be B instead of the root of T_2
- B's left child would now be the root of T_2 instead of A

A second kind of rotation is an *RR rotation*, which makes a similar adjustment.



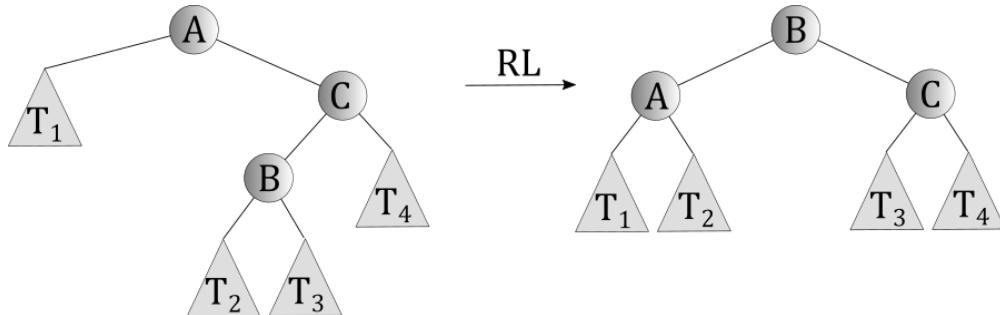
Note that an RR rotation is the mirror image of an LL rotation.

A third kind of rotation is an *LR rotation*, which makes an adjustment that's slightly more complicated.



An LR rotation requires five pointer updates instead of three, but this is still a constant number of changes and runs in $\Theta(1)$ time.

Finally, there is an *RL rotation*, which is the mirror image of an LR rotation.



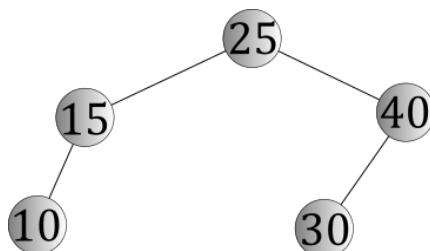
Once we understand the mechanics of how rotations work, we're one step closer to understanding AVL trees. But these rotations aren't arbitrary; they're used specifically to correct imbalances that are detected after insertions or removals.

An insertion algorithm

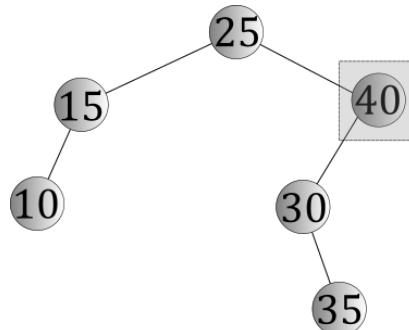
Inserting a key into an AVL tree starts out the same way as insertion into a binary search tree:

- Perform a lookup. If you find the key already in the tree, you're done, because keys in a binary search tree must be unique.
- When the lookup terminates without the key being found, add a new node in the appropriate leaf position where the lookup ended.

The problem is that adding the new node introduced the possibility of an imbalance. For example, suppose we started with this AVL tree:



and then we inserted the key 35 into it. A binary search tree insertion would give us this as a result:



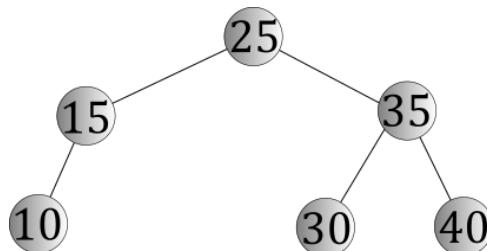
But this resulting tree is not an AVL tree, because the node containing the key 40 does not have the AVL property, because the difference in the heights of its subtrees is 2. (Its left subtree has height 1, its right subtree — which is empty — has height -1.) What can we do about it?

The answer lies in the following algorithm, which we perform after the normal insertion process:

- Work your way back up the tree from the position where you just added a node. (This could be quite simple if the insertion was done recursively.) Compare the heights of the left and right subtrees of each node. When they differ by more than 1, choose a rotation that will fix the imbalance.
 - Note that comparing the heights of the left and right subtrees would be quite expensive if you didn't already know what they were. The solution to this problem is for each node to store its height (i.e., the height of the subtree rooted there). This can be cheaply updated after every insertion or removal as you unwind the recursion.
- The rotation is chosen considering the two links along the path *below* the node where the imbalance is, heading back down toward where you inserted a node. (If you were wondering where the names LL, RR, LR, and RL come from, this is the answer to that mystery.)
 - If the two links are both to the left, perform an LL rotation rooted where the imbalance is.
 - If the two links are both to the right, perform an RR rotation rooted where the imbalance is.
 - If the first link is to the left and the second is to the right, perform an LR rotation rooted where the imbalance is.
 - If the first link is to the right and the second is to the left, perform an RL rotation rooted where the imbalance is.

It can be shown that any one of these rotations — LL, RR, LR, or RL — will correct any imbalance brought on by inserting a key.

In this case, we'd perform an LR rotation — the first two links leading from 40 down toward 35 are a **Left** and a **Right** — rooted at 40, which would correct the imbalance, and the tree would be rearranged to look like this:



Compare this to the diagram describing an LR rotation:

- The node containing 40 is C
- The node containing 30 is A
- The node containing 35 is B
- The (empty) left subtree of the node containing 30 is T_1
- The (empty) left subtree of the node containing 35 is T_2
- The (empty) right subtree of the node containing 35 is T_3
- The (empty) right subtree of the node containing 40 is T_4

After the rotation, we see what we'd expect:

- The node B, which in our example contained 35, is now the root of the newly-rotated subtree
- The node A, which in our example contained 30, is now the left child of the root of the newly-rotated subtree
- The node C, which in our example contained 40, is now the right child of the root of the newly-rotated subtree
- The four subtrees T_1 , T_2 , T_3 , and T_4 were all empty, so they are still empty.

Note, too, that the tree is more balanced after the rotation than it was before. This is no accident; a single rotation (LL, RR, LR, or RL) is all that's necessary to correct an imbalance introduced by the insertion algorithm.

A removal algorithm

Removals are somewhat similar to insertions, in the sense that you would start with the usual binary search tree removal algorithm, then find and correct imbalances while the recursion unwinds. The key difference is that removals can require more than one rotation to correct imbalances, but will still only require rotations on the path back up to the root from where the removal occurred — so, generally, $O(\log n)$ rotations.

Asymptotic analysis

The key question here is *What is the height of an AVL tree with n nodes?* If the answer is $\Theta(\log n)$, then we can be certain that lookups, insertions, and removals will take $O(\log n)$ time. How can we be so sure?

Lookups would be $O(\log n)$ because they're the same as they are in a binary search tree that doesn't have the AVL property. If the height of the tree is $\Theta(\log n)$, lookups will run in $O(\log n)$ time. Insertions and removals, despite being slightly more complicated in an AVL tree, do their work by traversing a single path in the tree — potentially all the way down to a leaf position, then all the way back up. If the length of the longest path — that's what the height of a tree is! — is $\Theta(\log n)$, then we know that none of these paths is longer than that, so insertions and removals will take $O(\log n)$ time.

So we're left with that key question. What is the height of an AVL tree with n nodes? (If you're not curious, you can feel free to just assume this; if you want to know more, keep reading.)

What is the height of an AVL tree with n nodes? (Optional)

The answer revolves around noting how many nodes, at minimum, could be in a binary search tree of height n and still have it be an AVL tree. It turns out AVL trees of height $n \geq 2$ that have the minimum number of nodes in them all share a similar property:

The AVL tree with height $h \geq 2$ with the minimum number of nodes consists of a root node with two subtrees, one of which is an AVL tree with height $h - 1$ with the minimum number of nodes, the other of which is an AVL tree with height $h - 2$ with the minimum number of nodes.

Given that observation, we can write a recurrence that describes the number of nodes, at minimum, in an AVL tree of height h .

$$\begin{aligned} M(0) &= 1 && \text{When height is 0, minimum number of nodes is 1 (a root node with no children)} \\ M(1) &= 2 && \text{When height is 1, minimum number of nodes is 2 (a root node with one child and not the other)} \\ M(h) &= 1 + M(h - 1) + M(h - 2) \end{aligned}$$

While the repeated substitution technique we learned previously isn't a good way to try to solve this particular recurrence, we can prove something interesting quite easily. We know for sure that AVL trees with larger heights have a bigger minimum number of nodes than AVL trees with smaller heights — that's fairly self-explanatory — which means that we can be sure that $1 + M(h - 1) \geq M(h - 2)$. Given that, we can conclude the following:

$$M(h) \geq 2M(h - 2)$$

We can then use the repeated substitution technique to determine a lower bound for this recurrence:

$$\begin{aligned} M(h) &\geq 2M(h - 2) \\ &\geq 2(2M(h - 4)) \\ &\geq 4M(h - 4) \\ &\geq 4(2M(h - 6)) \\ &\geq 8M(h - 6) \\ &\dots \\ &\geq 2^j M(h - 2j) && \text{We could prove this by induction on } j, \text{ but we'll accept it on faith} \\ \text{let } j &= h/2 \\ &\geq 2^{h/2} M(h - h) \\ &\geq 2^{h/2} M(0) \\ M(h) &\geq 2^{h/2} \end{aligned}$$

So, we've shown that the minimum number of nodes that can be present in an AVL tree of height h is at least $2^{h/2}$. In reality, it's actually more than that, but this gives us something useful to work with; we can use this result to figure out what we're really interested in, which is the opposite: what is the height of an AVL tree with n nodes?

$$\begin{aligned} M(h) &\geq 2^{h/2} \\ \log_2 M(h) &\geq h/2 \\ 2 \log_2 M(h) &\geq h \end{aligned}$$

Finally, we see that, for AVL trees of height h with the minimum number of nodes, the height is no more than $2 \log_2 n$, where n is the number of nodes in the tree. For AVL trees with more than the minimum number of nodes, the relationship between the number of nodes and the height is even better, though, for

reasons we've seen previously, we know that the relationship between the number of nodes and the height of a binary tree can never be better than logarithmic. So, ultimately, we see that the height of an AVL tree with n nodes is $\Theta(\log n)$.

(In reality, it turns out that the bound is lower than $2 \log_2 n$; it's something more akin to about $1.44 \log_2 n$, even for AVL trees with the minimum number of nodes, though the proof of that is more involved and doesn't change the asymptotic result.)