# IIT Madras
## BSc Degree

# Backend Systems

# Memory Hierarchy

# Types of storage elements

- On-chip registers: 10s-100s of bytes

# Types of storage elements

- On-chip registers: 10s-100s of bytes
- SRAM (cache): 0.1 - 1 MB

# Types of storage elements

- On-chip registers: 10s-100s of bytes
- SRAM (cache): 0.1 - 1 MB
- DRAM: 0.1 - 10 GB

# Types of storage elements

- On-chip registers: 10s-100s of bytes
- SRAM (cache): 0.1 - 1 MB
- DRAM: 0.1 - 10 GB
- Solid-state disk (SSD) - Flash: 1-100 GB

# Types of storage elements

- On-chip registers: 10s-100s of bytes
- SRAM (cache): 0.1 - 1 MB
- DRAM: 0.1 - 10 GB
- Solid-state disk (SSD) - Flash: 1-100 GB
- Magnetic disk (HDD - hard disk drive?): 0.1 - 10 TB

# Types of storage elements

- On-chip registers: 10s-100s of bytes
- SRAM (cache): 0.1 - 1 MB
- DRAM: 0.1 - 10 GB
- Solid-state disk (SSD) - Flash: 1-100 GB
- Magnetic disk (HDD - hard disk drive?): 0.1 - 10 TB
- Optical, magnetic, holographic, . . .

# Storage Parameters

- Latency: time to read first value from a storage location (lower is better)
    - Register < SRAM < DRAM < SSD < HDD

# Storage Parameters

- Latency: time to read first value from a storage location (lower is better)
  - Register < SRAM < DRAM < SSD < HDD
- Throughput: number of bytes/second that can be read (higher is better)
  - DRAM > SSD > HDD (regs, SRAM limited capacity)

# Storage Parameters

- Latency: time to read first value from a storage location (lower is better)
  - Register < SRAM < DRAM < SSD < HDD
- Throughput: number of bytes/second that can be read (higher is better)
  - DRAM > SSD > HDD (regs, SRAM limited capacity)
- Density: number of bits stored per unit area / cost (higher is better)
  - Volume manufacture important
  - HDD > SSD > DRAM > SRAM > Regs

# Computer Organization

- CPU has as many registers as possible

# Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)

# Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)
- Backed by several GB of DRAM working memory

# Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)
- Backed by several GB of DRAM working memory
- Backed by SSD for high throughput

# Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)
- Backed by several GB of DRAM working memory
- Backed by SSD for high throughput
- Backed by HDD for high capacity

# Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)
- Backed by several GB of DRAM working memory
- Backed by SSD for high throughput
- Backed by HDD for high capacity
- Backed by long-term storage, backup

# Cold storage (?)

- Backups and archives:
  - Huge amounts of data
  - Not read very often
  - Can tolerate high read latency
- Amazon Glacier, Google, Azure Cold/Archive storage classes
- High latency of retrieval: up to 48 hours
- Very high durability
- Very low cost

# Impact on application development

- Plan the storage needs based on application growth
- Speed of app determined by types of data stored, how stored
- Some data stores are more efficient for some types of read/write operations

Developer must be aware of choices and what kind of database to choose for a given application

# Data Search

# O() notation

- Used in study of algorithmic complexity: beyond scope of this course
- Rough approximation: “order of magnitude”, “approximately” etc.
- Main concepts here:
    - O(1) - constant time independent of input size - excellent!
    - O(log N) - logarithmic in input size - grows slowly with input - very good
    - O(N) - linear in input size - often the baseline - would like to do better
    - $O(N^k)$ - polynomial (quadratic, cubic etc.) - not good as input size grows
    - $O(k^N)$ - exponential - VERY bad: won’t work even for reasonably small inputs

# Searching for element in memory

Unsorted data in a linked list

- Start from beginning
- Proceed stepwise, comparing each element
- Stop if found and return LOCATION
- If end-of-list, return NOTFOUND

O(N)

# Searching for element in memory

Unsorted data in array

- Start from beginning
- Proceed stepwise, comparing each element
- Stop if found and return LOCATION
- If end-of-list, return NOTFOUND

O(N)

# Searching for element in memory

**Sorted** data in array

- Start from beginning
- Proceed stepwise, comparing each element
- Stop if found and return LOCATION
- If end-of-list, return NOTFOUND

O(N) but...

# Searching for element in memory

**Sorted** data in array

- Look at middle element in array:
    - greater than target - search in lower half
    - lesser than target - search in upper half
- Switch focus to new array: half the size of original
    - Repeat

O(log N)

# Problems with arrays

- Size must be fixed ahead of time
- Adding new entries requires resizing - can try oversize, but eventually ...
- Maintaining sorted order O(N):
  - find location to insert
  - move all further elements by 1 to create a gap
  - insert
- Deleting
  - find location, delete
  - move all entries down by 1 step

# Alternatives

- Binary search tree
  - Maintaining sorted order is easier: growth of tree

# Alternatives

- Binary search tree
  - Maintaining sorted order is easier: growth of tree
- Self-Balancing
  - BST can easily tilt to one side and grow downwards
  - Red-black, AVL, B-tree… more complex, but still reasonable

# Alternatives

- Binary search tree
  - Maintaining sorted order is easier: growth of tree
- Self-Balancing
  - BST can easily tilt to one side and grow downwards
  - Red-black, AVL, B-tree… more complex, but still reasonable
- Hash tables
  - Compute an index for an element: O(1)
  - Hope the index for each element is unique!
    - Difficult but doable in many cases

# Database Search

# Databases (tabular)

- Tables with many columns
- Want to search quickly on some columns
- Maintain “INDEX” of columns to search on
  - Store a sorted version of column
  - Needs column to be “comparable”: integer, short string, date/time etc.
    - Long text fields are not good for index
    - Binary data not good

# Example: MySQL

https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html

MySQL 8.0 Reference Manual / ... / Comparison of B-Tree and Hash Indexes

version 8.0

## 8.3.9 Comparison of B-Tree and Hash Indexes

Understanding the B-tree and hash data structures can help predict how different queries perform on different storage engines that use these data structures in their indexes, particularly for the `MEMORY` storage engine that lets you choose B-tree or hash indexes.

- B-Tree Index Characteristics
- Hash Index Characteristics

# Index-friendly query

```sql
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
```

```sql
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

# Index-**un**friendly query

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
```

```
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

# Multi-column index

- (index_1, index_2, index_3): compound index on 3 columns:
    - first sorted on index_1, then on index_2, then on index_3
    - all values with same index_1 will be sorted on index_2,
    - all values with same index_1 and index_2 will be sorted on index_3
    - etc.
- eg. (date-of-birth, city-of-birth, name)
    - can query for all people born on same date in same city with same name easily
    - but...

# Multi-index friendly

```sql
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
```

```sql
    /* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2
```

```sql
    /* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5
```

```sql
    /* use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

# Multi-index **un**friendly

```
   /* index_part1 is not used */
... WHERE index_part2=1 AND index_part3=2

    /*  Index not used in both parts of the WHERE clause  */
... WHERE index=1 OR A=10

    /* No index spans all rows  */
... WHERE index_part1=1 OR index_part2=10
```

# Hash-index

- Only used in in-memory tables
- Only for equality comparisons - cannot handle “range”
- Does not help with “ORDER BY”
- Partial key prefix cannot be used
- But VERY fast where applicable...

# Query Optimization

Database specific

- https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html
- https://www.sqlite.org/optoverview.html
- Postgres:

## Chapter 59. Genetic Query Optimizer

**Table of Contents**

# Summary

- Setting up queries properly impacts application performance
- Building proper indexes crucial to good search
- Compound indexes, multiple indexes etc. possible
    - Too many can be waste of space
- Make use of structure in data to organize it properly

# SQL vs NoSQL

# SQL

- Structured **Query** Language
  - Used to query databases that have structure
  - Could also be used for CSV files, spreadsheets etc.
- Closely tied to RDBMS - relational databases
  - Columns / Fields
  - Tables of data hold relationships
  - All entries in a table **must** have same set of columns
- Tabular databases
  - Efficient indexing possible - use specified columns
  - Storage efficiency: prior knowledge of data size

# Problem with tabular databases

- Structure (good? bad?)
- All rows in table must have same set of columns

Example

- Student - hostel => mess
- Student - day-scholar => gate pass for vehicle
- Table? Column for mess, column for gate pass???

# Alternate ways to store: Document databases

- Free-form (unstructured) documents
    - Typically JSON encoded
    - Still structured, but each document has own structure
- Examples:
    - MongoDB
    - Amazon DocumentDB


```json
[
    {
        "year" : 2013,
        "title" : "Turn It Down, Or Else!",
        "info" : {
            "directors" : [ "Alice Smith", "Bob Jones"],
            "release_date" : "2013-01-18T00:00:00Z",
            "rating" : 6.2,
            "genres" : ["Comedy", "Drama"],
            "image_url" : "http://ia.media-imdb.com/images/N/O9ERWAU7FS797AJ7LU8HN09AMUP
            "plot" : "A rock band plays their music at high volumes, annoying the neighb
            "actors" : ["David Matthewman", "Jonathan G. Neff"]
        }
    },
    {
        "year": 2015,
        "title": "The Big New Movie",
        "info": {
            "plot": "Nothing happens at all.",
            "rating": 0
        }
    }
]
```

# Alternate ways to store: Key-Value

- Python dictionary, C++ OrderedMap etc.: dictionary/hash table
- Map a key to a value
- Store using search trees or hash tables
- Very efficient key lookup, not good for range type queries
- Examples:
  - Redis
  - BerkeleyDB
  - memcached …
- Often used alongside other databases for “in-memory” fast queries

# Alternate ways to store: Column stores

- Traditional relational DBs store all values of a row together on disk
  - Retrieving all entries of a given row very fast
- Instead store all entries in a column together
  - Retrieve all values of a given attribute (age, place of birth, …) very fast
- Examples:
  - Cassandra
  - HBase...

# Alternate ways to store: Graphs

- Friend-of-a-friend, social networks, maps: graph oriented relationships
- Different degrees (number of outgoing edges), weights of edges, nodes etc.
- Path-finding more important than just search
  - Connections, knowledge discovery
- Examples:
  - Neo4J
  - Amazon Neptune

# Alternative ways to store: Time Series Databases

- Very application specific: store some metric or values as function of time
- Used for log analysis, performance analysis, monitoring
- Queries:
  - How many hits between T1 and T2?
  - Average number of requests per second?
  - Country from where maximum requests came in past 7 days?
- Typical RDBMS completely unsuitable - same for most alternatives
- Examples:
  - RRDTool
  - InfluxDB
  - Prometheus
- Search: elasticsearch, grafana,...

# NoSQL?

- Started out as “alternative” to SQL
- But SQL is just a query language - can be adapted for any kind of query, including from a document store or graph!
- “Not-only-SQL”
- Additional query patterns for other types of data stores

# A word on ACID

- Transaction: core principle of database
- ACID:
    - Atomic
    - Consistent
    - Isolated
    - Durable
- Many NoSQL databases sacrifice some part of ACID (example: eventual consistency instead of consistency) for performance
- But there can be ACID compliant NoSQL databases as well...

# Why not ACID?

- Consistency hard to meet: especially when scaling / distributing
- Eventual consistency easier to meet
- Example:
  - A (located in India) and B (located in the US) both add C as a friend on Facebook
  - Order of adding does not matter!
  - Temporarily seeing C in A’s list but not B, or B’s list but not A - not a catastrophe (?)
- Financial transactions **absolutely require** ACID
  - Consistency is paramount - even a split second of inconsistent data can cause problems

# A word on storage

- In-memory:
  - Fast
  - Doesn’t scale across machines
- Disk
  - Different data structures, organization needed

# Scaling

# Replication and Redundancy

- Redundancy:
  - Multiple copies of same data
  - Often used in connection with backups - even if one fails, others survive
  - One copy is still the master
- Replication:
  - Usually in context of performance
  - May not be for purpose of backup
  - Multiple sources of same data - less chance of server overload
  - Live replication requires careful design

# BASE vs ACID

- “Basically Available”, “Soft state”, “Eventually consistent”
  - Winner of worst acronym award
- Eventual consistency instead of Consistency
  - Replicas can take time to reach consistent state
- Stress on high availability of data

# Replication in traditional DBs

- RDBMS replication possible
- Usually server cluster in same data center
  - Load balancing
- Geographically distributed replication harder
  - Latency constraints for consistency

# Scale-up vs Scale-out

- Scale-up: traditional approach
  - Larger machine
  - More RAM
  - Faster network, processor
  - requires machine restart with each scale change
- Scale-out:
  - Multiple servers
  - Harder to enforce consistency etc. - better suited to NoSQL / non-ACID
  - Better suited to cloud model: Google, AWS etc provide automatic scale-out, cannot do auto-scale-up

# Application Specific

- Financial transactions:
  - cannot afford even slightest inconsistency
  - Only scale-up possible
- Typical web-application
  - Social networks, media: eventual consistency OK
  - e-commerce: only the financial part needs to go to ACID DB

# Security

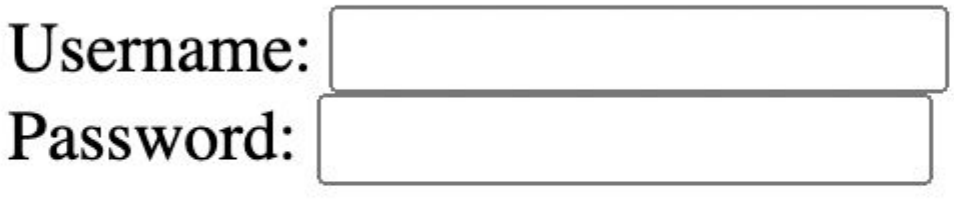
# SQL in context of an application

- Non-MVC app: can have direct SQL queries anywhere
- MVC: only in controller, but any controller can trigger a DB query

So what's dangerous about queries?

# Typical HTML form

```html
<form>
   Username:
   <input type="text" name="name">
   <br />
   Password:
   <input type="password" name="password">
   <br />
</form>
```



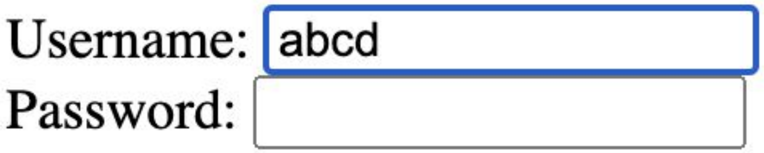
# Code

```
name = form.request["name"]

pswd = form.request["name"]


sql = 'SELECT * FROM Users WHERE Name ="' +
       name + '" AND Pass ="' + pswd + '"'
```

# Example input vs SQL




```
SELECT * FROM Users WHERE Name ="abcd" AND Pass = "pass"
```

# Example input vs SQL

Username: " or ""="

Password: " or ""="

```
SELECT * FROM Users WHERE Name ="" or ""
    AND Pass = "" or ""
```

Result???

# Example input vs SQL

```
sql = "SELECT * FROM Users WHERE Name = " + name
```

```
Input:
a; DROP TABLE Users;
Query:
SELECT * FROM Users WHERE Name = a; DROP TABLE Users;
```

# Problem

- Parameters from HTML taken without validation
- Validation:
    - Are they valid text data (no special characters, other symbols)
    - No punctuation or other invalid input
    - Are they the right kind of input (text, numbers, email, date)?
- Validation **MUST** be done just before the database query - even if you have validation in the HTML or Javascript - not good enough
    - Direct HTTP requests can be made with junk data

# Web Application Security

- SQL injection
  - Use known frameworks, best practices, validation
- Buffer overflows, input overflows
  - Length of inputs, queries
- Server level issues - protocol implementation?
  - Use known servers with good track record of security
  - Update all patches
- Possible outcomes:
  - loss of data - deletion
  - exposure of data - sensitive information leak
  - manipulation of data - change

# HTTPS ?

- Secure sockets: secure communication between client and server
- Server certificate:
  - based on DNS: has been verified by some trusted third party
  - difficult to spoof
  - based on Mathematical properties - ensure very low probability of mistakes match
- However:
  - Only secures link for data transfer - does not perform validation or safety checks
  - Negative impact on “caching” of resources like static files
  - Some overhead on performance

# Summary

- Internet and Web security are complex: enough for a course in themselves
- Generally recommended to use known frameworks with trusted track records
- Code audits
- Patch updates on OS, server, network stack etc. essential

App developers should be very careful of their code, but also aware of problems at other levels of the stack