Week 7 Lecture 1

Class	BSCCS2003
Created	@October 18, 2021 10:50 AM
Materials	
Module #	38
Type	Lecture
Week #	7

Memory Hierarchy

Types of Storage Elements

- On-chip registers: 10s-100s of bytes
- SRAM (Cache): 0.1-1MB # Static RAM
- DRAM (Memory/RAM of a PC): 0.1-16GB
- Solid-State Disk (SSD) - Flash: 1-128GB
- Magnetic Disk (HDD-Hard Disk Drive): 0.1-10TB

Storage Parameters

- **Latency:** Time to read the first value from a storage location (lower is better)
  - Register < SRAM < DRAM < SSD < HDD
  - So, the turnaround time when trying to read something from a ...
    - Register: in order of nanoseconds
    - SRAM: tens to hundreds of nanoseconds
    - DRAM: several microseconds
    - SSDs: hundreds of microseconds
    - HDDS: order of milliseconds
- **Throughput:** number of bytes/second that can be read (higher is better)
  - DRAM > SSD > HDD (registers, SRAM have limited capacity)

- **Density:** Number of bits stored per unit area/cost (higher is better)
  - Volume manufacture important
  - HDD > SSD > DRAM > SRAM > Registers

Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)
- Backed by several Gigabytes (GBs) 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 & 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 the 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 the choices and what kind of DB to choose for a given application

Week 7 Lecture 2

Class	BSCCS2003
Created	@October 18, 2021 2:13 PM
Materials	
Module #	39
Type	Lecture
Week #	7

Data Search

O() notation - The Big O notation

- Used in study of algorithmic complexity
- Rough approximation: "order of magnitude", "approximately", etc.
- Main concepts:
  - $O(1)$ - constant time, independent of input size: Excellent!
  - $O(log\ N)$ - logarithmic in input size - grows very 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 the input size grows
  - $O(k^N)$ - exponential - **VERY BAD**: Won't even work for reasonably small inputs

Searching for an element in the memory

Unsorted data in a linked list (or an array)

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

$O(N)$

Sorted data in an array

- Look at the middle element in an array:
  - Greater than the target?
    - Search in the lower half
  - Lesser than the target?
    - Search in the upper half
- Switch focus to the new array: Half the original size
  - Repeat

$O(log\ N)$: $Binary\ Search$

Problems with arrays

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

Alternatives

- Binary Search Tree
  - Maintaining sorted order is easier: growth of the 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 is doable in many cases

Week 7 Lecture 3

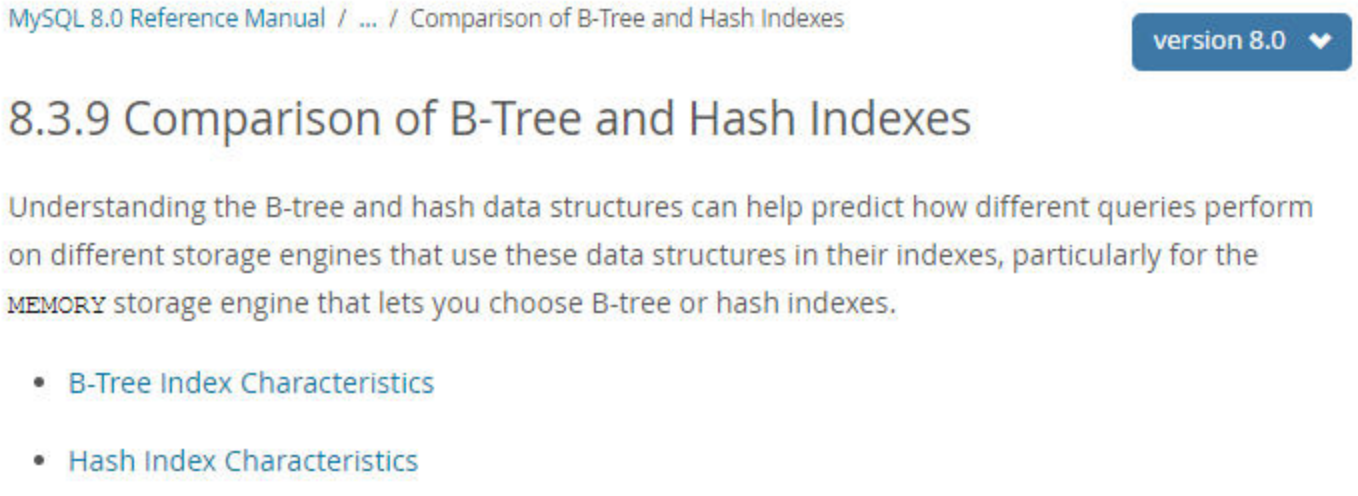
Class	BSCCS2003
Created	@October 18, 2021 2:42 PM
Materials	
Module #	40
Type	Lecture
Week #	7

Database Search

Databases (Tabular)

- Tabular 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 as well

Example: MySQL



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

Index-friendly Query

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

In the first statement, only rows with `'Patrick' <= key_col < 'Patricl'` are considered

In the second statement, only rows with `'Pat' <= key_col < 'Pau'` are considered

Index-unfriendly Query

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

In the first statement, the `LIKE` value begins with a wildcard character

In the second statement, the `LIKE` value is not a constant

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
- Example: (date-of-birth, city-of-birth, name)
  - can query for all the people born on the same date in the same city with the same name easily

Multi-index friendly

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

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

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

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

Multi-index unfriendly

```
/* 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 ranges
- Does not help with "ORDER BY"
- Partial key prefix cannot be used
- But VERY fast where applicable

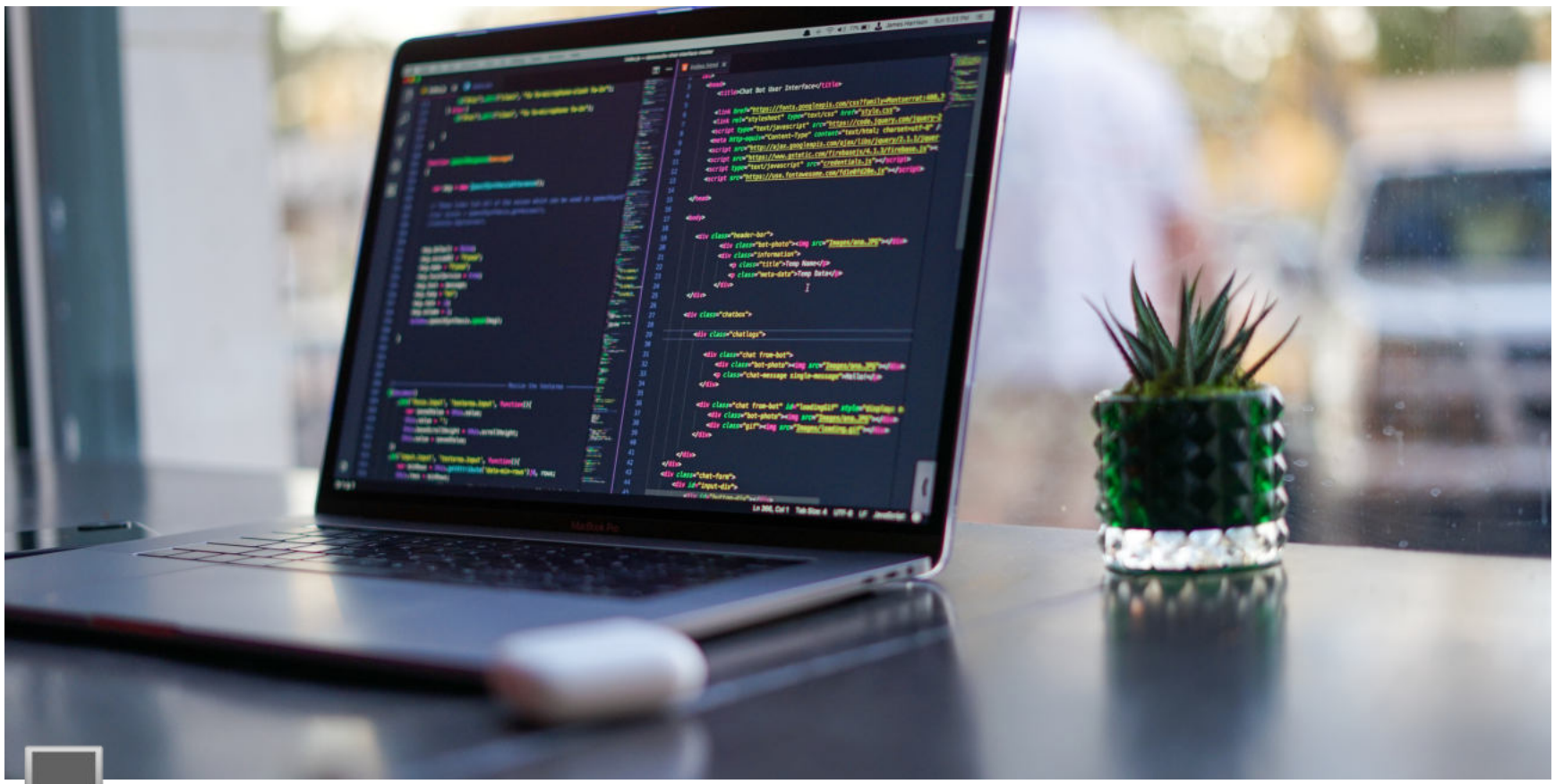
Query Optimization

Database specific:

- https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html
- https://www.sqlite.org/optoverview.html
- https://www.postgresql.org/docs/12/geqo.html

Summary

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



Week 7 Lecture 4

Class	BSCCS2003
Created	@October 18, 2021 3:42 PM
Materials	
Module #	41
Type	Lecture
Week #	7

SQL v/s. NoSQL

SQL

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

Problems with tabular DBs

- Structure (good? bad?)
- All rows in a 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 DBs

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

```
[
  {
    "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/09ERWAU7FS797AJ7LU8HN",
      "plot": "A rock band plays their music at high volumes, annoying the neighbours",
      "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
- **Example:**
  - Redis
  - BerkeleyDB
  - memcached
- Often used alongside other DBs for "in-memory" fast queries

Alternate ways to store: Column store

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

Alternate ways to store: Graphs

- Friend-of-a-friend, social networks, maps: graph oriented relationships
- Different degrees (number of out-going edges), weights of edges, nodes, etc.

- Path-finding more important than just search
  - Connections, knowledge discovery
- Examples:
  - Neo4j
  - Amazon Neptune

Alternate ways to store: Time Series DBs

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

NoSQL

- Started out as an "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

ACID

- **Transaction:** core principle of DB
- **ACID:**
  - Atomic
  - Consistent
  - Isolated
  - Durable
- Many NoSQL DBs sacrifice some part of ACID
  - **Example:** Eventual consistency instead of consistency, for performance
- But, there can be ACID compliant NoSQL DB 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 USA) 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
  - Does not scale across machines
- Disk
  - Different data structures, organization needed

Week 7 Lecture 5

Class	BSCCS2003
Created	@October 18, 2021 6:50 PM
Materials	
Module #	42
Type	Lecture
Week #	7

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, lol
- Eventual consistency instead of Consistency
  - Replicas can take time to reach a consistent state
- Stress on high availability of data

Replication in traditional DBs

- RDBMS replication is possible
- Usually server cluster in the 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 for NoSQL/non-ACID
  - Better suited to cloud model: Google, AWS etc. to 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 is OK
  - e-Commerce: only the financial part needs to go to ACID

Week 7 Lecture 6

Class	BSCCS2003
Created	@October 18, 2021 7:05 PM
Materials	
Module #	43
Type	Lecture
Week #	7

Security of Databases

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 is dangerous about queries?

Typical HTML form

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

Username:

Password:

Code

```
name = form.request['name']
pswd = form.request['password']

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

Example Input vs SQL

Username: *abcd*

Password: *pass*

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

Username: *" or ""="*

Password: *" or ""=""*

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

→ *The above query checks for, basically, nothing and returns everything from the table* `Users`

`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 the 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 before the DB 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 3rd party
  - difficult to spoof
  - based on Mathematical properties - ensure very low probability of mistakes match

- However:
  - Only secures the 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 on other levels of stack.