Georgia Tech

# Course Introduction

CREATING THE NEXT®

# Course Outline & Logistics

# Motivation

A **Database Management System** (DBMS) is a software that allow applications to store and electronically analyze an organized collection of data.

DBMSs are super important and deployed all over the place

- core component of many applications (*e.g.*, Airlines)
- very large data sets (*e.g.*, IoT data)
- valuable data (*e.g.*, healthcare)

# Motivation

Key challenges:

- scalability to huge data sets
- reliability
- concurrency

Results in very complex software.

# Why you should take this course?

- You want to learn how to make database systems **scalable**, for example, to support web or mobile applications with millions of users.
- You want to make applications that are highly **available** (*i.e.*, minimizing downtime) and operationally robust.
- You have a natural curiosity for the way things work and want to know what goes on inside major websites and online services.
- You are looking for ways of making systems easier to maintain in the long run,even as they grow and as requirements and technologies change.
- If you are good enough to write code for a database system, then you can write code on almost anything else.

# Why you should take this course?

*You will not find a broader set of Computer Science problems inside one piece of software than by working on a cloud database, especially general-purpose databases that attempt to solve a lot of different use cases. You get to work on all kinds of things from memory management, scheduling algorithms, low-level optimizations like SIMD and efficient operations on compressed data, query optimization, etc. And then there's the whole cloud-native set of challenges. There's cloud computing and figuring out how to best use things like blob stores/S3, and security and all the rest of it. – Adam Prout, CTO of Single-Store*

# Course Objectives

- Learn about internals of existing DBMSs and how to build a modern DBMS
- Understanding the impact of hardware trends on software design
- Students will become proficient in:
  - Writing correct + performant code
  - Proper documentation + testing
  - Working on a systems programming project

# Course Topics

The internals of single node systems for disk-oriented and in-memory databases.

Topics include:

- Relational Databases
- Storage
- Access Methods
- Query Execution

# Next Course

In a follow-up course offered in the Spring semester (8803-DSI), we will focus on:

- Logging and Recovery
- Concurrency Control
- Query Optimization
- Potpourri

This course will be a pre-requisite for the next course.

# Textbook

- Silberschatz, Korth, & Sudarshan: *Database System Concepts*. McGraw Hill, 2020.
- Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom: *Database Systems: The Complete Book*. Prentice-Hall, 2008.

Caveat

- These textbooks mostly focus on traditional disk-oriented database systems
- Not modern in-memory database systems

# Background

- You should have taken an introductory course on database systems (*e.g.*, GT 4400).
- All programming assignments will be in C++ or Python.
  - Will train you to develop and test a multi-threaded program.
  - Programming Assignment #1 will help get you caught up with C++.
  - If you have not encountered C++ before, you will need to put in extra effort! Use ChatGPT :)
  - Here a few helpful references: C to C++ Crash Course, Java to C++ Crash Course.
  - I will briefly cover relevant parts of C++ in this course.

# Course Logistics

- Course Web Page
  - Schedule: https://www.cc.gatech.edu/ jarulraj/courses/4420-f23/
  - Links on Canvas
- Discussion Tool: **Piazza**
  - For all technical questions, please use Piazza
  - Don't email me directly
  - All non-technical questions should be sent to me
- Grading Tool: **Gradescope**
  - You will get immediate feedback on your assignment
  - You can **iteratively improve** your score over time
- Hybrid office hours
  - Must sign up for an one-on-one slot
  - Sign-up sheet link posted on Canvas

# Course Rubric

- BuzzDB Programming Assignments (**20**%)
  - Four assignments based on the BuzzDB academic DBMS.
  - You will need to upload the solutions via Gradescope.
- EvaDB Programming Assignments (**25**%)
  - Two open-ended assignments based on the EvaDB AI-SQL DBMS.
  - You will need to share your solutions via Github.
- Exams (**40**%)
  - Two in-person, pen-and-paper exams.
- Class Participation (**15**%)
  - In-class quizzes (two to three questions per lecture) via TurningPoint.
  - Goal is to encourage participation and learning in class.

# Course Rubric

- Emphasis on learning rather than testing you.
- Students enrolled in the 4420 part may skip attending the advanced lectures (marked with a star) in the schedule.
- They will not be expected to answer questions related to these advanced lectures in the exam.

# Course Logistics

- Course Policies
  - The programming assignments and exercise sheets must be your own work.
  - They are **not** group assignments.
  - You may **not** copy source code from other people or the web.
  - Plagiarism will **not** be tolerated.
- Academic Honesty
  - Refer to Georgia Tech Academic Honor Code.
  - If you are not sure, ask me.

# Late Policy

- You are allowed **four** slip days for either programming assignments or exercise sheets.
- You lose 25% of an assignment's points for every 24 hrs it is late.
- Mark on your submission (1) how many days you are late and (2) how many late days you have left.

# Exercise Sheet #1

- Hand in one page (PDF) with the following information:
  - Digital picture (ideally 2x2 inches of face)
  - Name, interests, and other details posted on Gradescope
- The purpose of this sheet is to help me:
  - know more about your background for tailoring the course, and
  - recognize you in class

# Teaching Assistants

- 
  - Ishwarya Sivakumar
  - Shashank Suman
  - Kaushik Ravichandran
  - Aryan Rajoria
  - Chitti Reddy
  - Sayan Sinha
- If you are acing through the structured BuzzDB assignments, you might want to focus on the open-ended EvaDB assignments.
- Drop me a note if you are interested!

# Motivating Example

# Motivating Example

Why is a DBMS different from most other programs?

- many difficult requirements (reliability, concurrency, etc.)
- but a key challenge is **scalability**

Motivating example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Looks simple...

$L_1 = \{1, 2, 3, 5\}$

$L_2 = \{1, 5, 3, 4, 7\}$

$L_1 \cap L_2 = \{1, 3, 5\}$

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Simple if both fit in main memory
Don't need more than a few lines of code

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Simple if both fit in main memory
Don't need more than a few lines of code

- sort both lists and intersect $L_1 = \{1, 2, 3, 5\}$; $L_2 = \{1, 3, 4, 5, 7\}$
- or load one list in an **unordered hash table** [?] and probe
- or load one list in an **ordered tree** structure [?]
- or ...

Note: pairwise comparison is not an option! $O(n^2)$
We will discuss about hash tables and B+trees later in this course.

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Slightly more complex if **only one list** fits in main memory

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Slightly more complex if **only one list** fits in main memory

- load the smaller list into memory
- build tree structure/sort/hash table/...
- scan the larger list one **chunk** (*e.g.*, 10 numbers) at a time
- search for matches in main memory

Code still similar to the pure main-memory case.

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Difficult if neither list fits into main memory

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Difficult if neither list fits into main memory

- no direct interaction possible
- Option 1: Sorting works, but already a difficult problem
  - **external** merge sort (*i.e.*, database does not fit in memory))
- Option 2: Partitioning scheme (*e.g.*, numbers in [1, 100], [101, 200],...)
  - break the problem into smaller problems
  - ensure that each partition fits in memory

Code significantly more involved.

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Hard if we make no assumptions about $L_1$ and $L_2$.

# Motivating Example

*Given two lists $L_1$ and $L_2$, find all entries that occur on both lists.*

Hard if we make no assumptions about $L_1$ and $L_2$.

- tons of corner cases
- a list can contain duplicates
- a single duplicate value might exceed the size of main memory!
- breaks “simple” external memory logic
- multiple ways to solve this, but all of them are somewhat involved
- and a DBMS must not make assumptions about its data!

Code complexity is very high.

# Motivating Example

Designing a robust, scalable algorithm is hard

- must cope with very large instances
- hard even when the database fits in main memory
- billions of data items
- rules out the possibility of using $O(n^2)$ algorithms
- external algorithms (*i.e.*, database does not fit in memory) are even harder

This is why a DBMS is a complex software system.

# Shift in Hardware Trends

# Traditional Assumptions

Historically, a DBMS is designed based on these assumptions:

- database is much larger than main memory
- I/O cost dominates everything with **Hard Disk Drives** (HDD)
- random I/O operations to "mechanical" HDD are very expensive

This led to a very **conservative**, but also very **scalable** design.

# Hardware Trends

Hardware has evolved over the decades (invalidating these assumptions):

- main memory size is increasing
- servers with 1 TB main memory are affordable
- "electromagnetic" **Solid State Drives** (SSD) have lower random I/O cost
- ...

# Hardware Trends

This affects the design of a DBMS

- CPU costs are now more important
- I/O operations are eliminated or greatly reduced
- the classical architecture (**disk-oriented database systems**) has become sub-optimal

But this is more of an evolution as opposed to a revolution. Many of the old techniques are still relevant for scalability.

# Goals

Ideally, a DBMS

- efficiently handles arbitrarily-large databases
- never loses data
- offers a high-level API to manipulate and retrieve data
- this API is the **declarative Structured Query Language** (SQL)
- shields the application from the complexity of data management
- offers excellent performance for all kinds of queries and all kinds of data

This is a very ambitious goal!
This has been accomplished, but comes with inherent complexity.

# Relational Model: Motivation

# Digital Music Store Application

Consider an application that models a digital music store to keep track of artists and albums.

Things we need store:

- Information about **Artists**
- What **Albums** those Artists released

# Flat File Strawman (1)

Store our database as **comma-separated value** (CSV) files that we manage in our own code.

- Use a separate file per entity
- The application has to parse the files each time they want to read/update records

# Flat File Strawman (2)

**Artists.csv**

| Artist | Year | City |
|---|---|---|
| Mozart | 1756 | Salzburg |
| Beethoven | 1770 | Bonn |
| Chopin | 1810 | Warsaw |

**Albums.csv**

| Album | Artist | Year |
|---|---|---|
| The Marriage of Figaro | Mozart | 1786 |
| Requiem Mass In D minor | Mozart | 1791 |
| Für Elise | Beethoven | 1867 |

# Flat File Strawman (3)

Example: Get the Albums composed by Beethoven.

```
for line in file:
  record = parse(line)
  if "Beethoven" == record[1]:
    print record[0]
```

**Albums.csv**

| Album | Artist | Year |
|---|---|---|
| The Marriage of Figaro | Mozart | 1786 |
| Requiem Mass In D minor | Mozart | 1791 |
| Für Elise | Beethoven | 1867 |

# Flat File Strawman (4)

## Data Integrity

- How do we ensure that the artist is the same for each album entry?
- What if somebody overwrites the album year with an invalid string?
- How do we store that there are multiple artists on an album?

## Implementation

- How do you find a particular record?
- What if we now want to create a new application that uses the same database?
- What if two threads try to write to the same file at the same time?

## Durability

- What if the machine crashes while our program is updating a record?
- What if we want to replicate the database on multiple machines for high availability?

# Early DBMSs

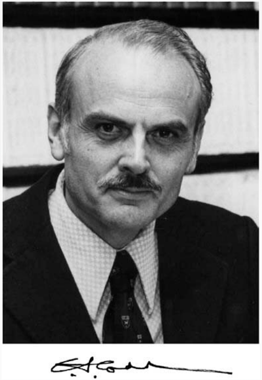
Limitations of early DBMSs (*e.g.*, IBM IMS FastPath in 1966)

- Database applications were difficult to build and maintain.
- Tight coupling between **logical** and **physical** layers.
- You have to (roughly) know what queries your app would execute before you deployed the database.

# Relational Model

# Relational Model

Proposed in 1970 by Ted Codd (IBM Almaden).
Data model to avoid this maintenance.



- Store database in simple data structures
- Access data through high-level language
- Physical storage left up to implementation

# Data Models

A **data model** is collection of concepts for describing the data in a database.
A **schema** is a description of a particular collection of data, using a given data model.

List of data models

- Relational (SQL-based, most DBMSs, focus of this course)
- Non-Relational (*a.k.a.*, NoSQL) models
  - Key/Value, Graph, Document
  - Column-family
- Array/Matrix (Machine learning)
  - Hierarchical/Tree

# Relation

A **relation** is an unordered **set** of **tuples**. Each tuple represents an entity.
A tuple is a set of **attribute** values.
Values are (normally) atomic/scalar.

| **Artist** | **Year** | **City** |
|---|---|---|
| Mozart | 1756 | Salzburg |
| Beethoven | 1770 | Bonn |
| Chopin | 1810 | Warsaw |

# Jargon

- Relations are also referred to as **tables**.
- Tuples are also referred to as **records** or **rows**.
- Attributes are also referred to as **columns**.

# BuzzDB

# BuzzDB

- BuzzDB – version 1
- BuzzDB – version 2
- BuzzDB – version 3

# Machine Setup

- Instructions
- Operating System (OS): Ubuntu 22.04 (Linux Distribution)
- **Build System**: cmake
- Testing Library: Google Testing Library (gtest)
- **Continuous Integration (CI)** System: Gradescope
- Memory Error Detector: valgrind memcheck

# C++: Tuple

```
#include <iostream>
#include <map>
#include <vector>

// A "class" in C++ is a user-defined data type.
// It is a blueprint for creating objects of a particular type,
// providing initial values for state (member variables or fields),
// and implementations of behavior (member functions or methods)
class Tuple {
public:
    int key;
    int value;
};
```

# C++: Database

```
class BuzzDB {
private:
    // a map is an ordered key-value container
    std::map<int, std::vector<int>> index;

public:
    // a vector of Tuple structs acting as a table
    std::vector<Tuple> table;
    ...
};
```

# C++: Loading into Database

```
BuzzDB db;

db.insert(1, 100);
db.insert(1, 200);
db.insert(2, 50);
db.insert(3, 200);
db.insert(3, 200);
db.insert(3, 100);
db.insert(4, 500);

db.selectGroupBySum();
```

# C++: Inserting into Database

```
class BuzzDB {

public:
  // insert function
  void insert(int key, int value) {
      Tuple newTuple = {key, value};
      table.push_back(newTuple);
      index[key].push_back(value);
  }

};
```

# C++: Aggregation Query

```
class BuzzDB {

public:
  // perform a SELECT ... GROUP BY ... SUM query
  void selectGroupBySum() {
    for (auto const& pair : index) { // for each unique key
      int sum = 0;
      for (auto const& value : pair.second) {
        sum += value; // sum all values for the key
      }
      std::cout << "key: " << pair.first << ", sum: " << sum << '\n';
    }
  }

};
```

# C++ Topics

- File I/O
- Threading (later assignments)
- Smart Pointers (later assignments)

# Conclusion

- Complexity of a database system arises from the need for robust, scalable algorithms, better hardware resource management, supporting for different data types, *e.t.c*.
- A database system must satisfy many requirements: reliability, scalability, concurrency *e.t.c*.
- Enroll in Piazza, Gradescope, and TurningPoint.
- In the next lecture, we will learn about relational database systems.