Course Introduction

CREATING THE NEXT®

Today's Agenda

Course Overview

Welcome!

- This course is on the design and implementation of database management systems (DBMSs).

Why you might want to take this course?

- DBMS developers are in demand.
- There are many challenging unsolved problems in data management and processing.
- If you are good enough to write code for a DBMS, then you can write code on almost anything else.

Welcome!

Why you might **not** want to take this course?

- This is not a course on how to use a database to build applications or how to administer a database.

DBA

SQL

Course Objectives

- Learn about modern practices in database internals and systems programming.
- Students will become proficient in:
  - Writing correct + performant code
  - Proper documentation + testing
  - Working on a large systems programming project

C++17

Course Topics

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

Topics include:

- Relational Databases
- Storage
- Access Methods
- Query Execution

Background

- You should have taken an introductory course on data systems (*e.g.*, GT 4400).
- All programming assignments will be written in C++17.
  - 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!
  - Here a couple of helpful references: Java to C++ Transition Tutorial, C++ Language
  - I will briefly cover relevant parts of C++ in this course.

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.

Course Logistics

- Course Web Page
  - Schedule: https://www.cc.gatech.edu/ jarulraj/courses/4420-f21/
- 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
- Virtual Office Hours via BlueJeans
  - Will need to sign up for an one-on-one slot
  - Sign-up sheet will be posted on Canvas

Course Rubric

- Programming Assignments (**35**%)
  - Four assignments based on the BuzzDB academic DBMS.
  - You will need to upload the solutions to Gradescope.
- Exercise Sheets (**15**%)
  - Three pencil-and-paper tasks.
  - You will need to upload the sheets to Gradescope.
- Exams (**20**%)
  - One remote exam.
  - Based on programming assignments and problem sheets.
  - We are planning to use the online proctoring service provided by the university.
- Project (**30**%)
  - Students will organize into groups and choose to implement a project that is relevant to the topics discussed in class.

Course Rubric

- Emphasis on learning rather than testing you.
- Students enrolled in the 4420 section 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 exercise sheets or the exam.

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.

Teaching Assistants

- Pramod Chundhuri
  - Ph.D. (Computer Science)
  - Research Topic: Video analytics using deep learning.
- Jiashen Cao
  - Ph.D. (Computer Science)
  - B.S./M.S. program @ Georgia Tech
  - Research Topic: Accelerating data systems using GPUs.
- If you are acing through the assignments, you might want to hack on the video analytics system (codenamed EVA) that we are building.
- Drop me a note if you are interested!

Motivation

Motivation (1)

A **Database Management System** (DBMS) is a software that allows applications to store and analyze information in a database.

A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases.

DBMSs are super important

- core component of most computer applications
- very large data sets
- valuable data

Google Search

SQL

declarative

Motivation (2)

Key challenges:

- scalability to huge data sets
- reliability
- concurrency

Results in very complex software.

About This Course

Goals of this course

- learning how to build a modern DBMS
- understanding the internals of existing DBMSs
- understanding the impact of hardware properties

Rough structure of the course

1. Relational Databases
2. Storage
3. Indexing
4. Query Execution

Next Course

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

1. Query Compilation
2. Concurrency Control
3. Recovery
4. Query Optimization
5. 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

Motivational Example

Why is a DBMS different from most other programs?

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

Motivational example

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

Looks simple...

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

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

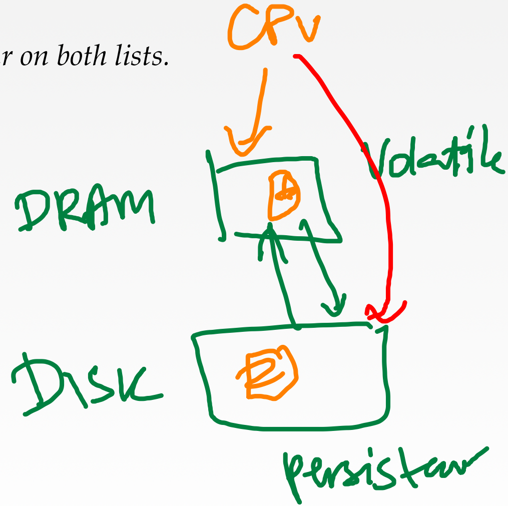
$L_1 \cap L_2 = \{1, 3, 5\}$

Motivational Example (2)

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

$O(n^2)$



Motivational Example (2)

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

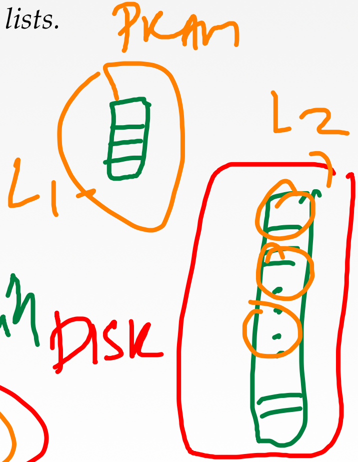
- sort both lists and intersect $L_1 = \{1, 2, 3, 5\}$; $L_2 = \{1, 3, 4, 5, 7\}$
- or load one list in an **unordered hash table** [2] and probe
- or load one list in an **ordered tree** structure [1]
- or ...

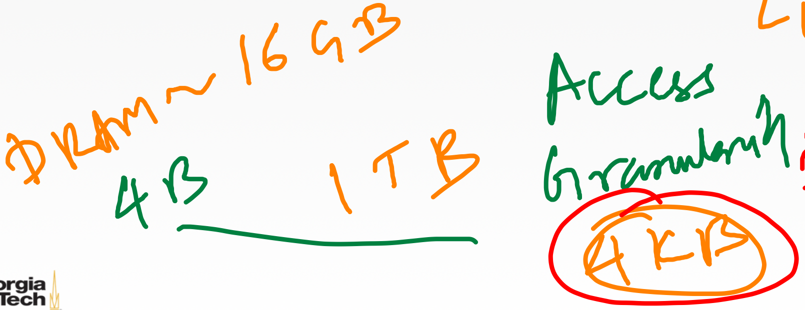
Note: pairwise comparison is not an option! $O(n^2)$
We will discuss about hash tables and B+trees in Access Paths.

Motivational Example (3)

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





Motivational Example (3)

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.

Motivational Example (4)

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

Difficult if neither list fits into main memory

internal

external

Divide-and-Conquer

Motivational Example (4)

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
  - Programming Assignment 1: **external** merge sort
  - We will cover this in External Hash Join.
- 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.

Georgia Tech

Motivational Example (5)

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.

Motivational Example (5)

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.

Motivational Example (6)

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

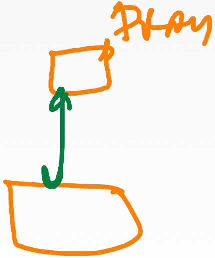
This is why a DBMS is a complex software system.

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

~1TB $10K

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
- . . .

page size=64KB

Hardware Trends

hot data

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 suboptimal

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.

Course Organization

1. Storage Management
2. Access Paths
3. Query Execution (algebraic operators)

In each topic, we will cover aspects of both disk-oriented and modern in-memory DBMSs.

External Sorting

Machine Setup

- Operating System (OS): Ubuntu 18.04
- **Build System**: cmake
- Testing Library: Google Testing Library (gtest)
- **Continuous Integration (CI)** System: Gradescope
- Memory Error Detector: valgrind memcheck

Lab Tutorial

C++ Topics

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

Lab Tutorial Page

Problem Statement

- Sorting an arbitrary amount of data, stored on disk
- Accessing data on disk is slow – so we do not want to access each value individually
- Sorting in main memory is fast – but main memory size is limited

block

Solution

- Load pieces (called **runs**) of the data into main memory
- and sort them
- Use std::sort as the internal sorting algorithm.
- With **m** values fitting into main memory and **d** values that should be sorted:
- number of runs (**k**) = $\lceil \frac{d}{m} \rceil$ runs

d = 9
m = 3

STL

Sort k runs (1)

$O(\log_2 k)$

3

Memory

–	–	–

Disk

8	5	1	4	7	3	2	9	6

9

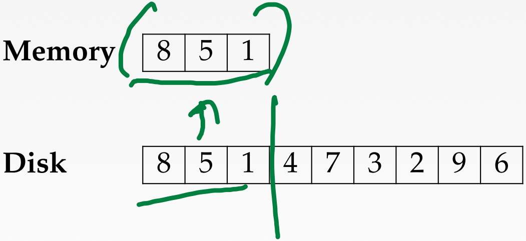
$k = \lceil \frac{9}{3} \rceil$

$k = 3$

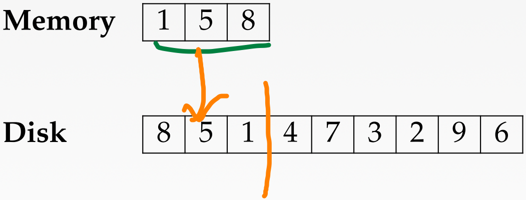
Runs

2-way merge

Sort k runs (2)



Sort k runs (3)



Sort k runs (4)

Memory

–	–	–

Disk

1	5	8	4	7	3	2	9	6

Sort k runs (5)

Memory

–	–	–

Disk

1	5	8	3	4	7	2	6	9

Iterative 2-Way Merge (1)

Memory

–	–

Disk

1	5	8	3	4	7	2	6	9

–	–	–	–	–	–	–	–	–

Iterative 2-Way Merge (2)

Memory

1	3

Disk

1	5	8	3	4	7	2	6	9

–	–	–	–	–	–	–	–	–

Iterative 2-Way Merge (3)

Memory

–	3

Disk

1	5	8	3	4	7	2	6	9

1	–	–	–	–	–	–	–	–

Iterative 2-Way Merge (4)

Memory

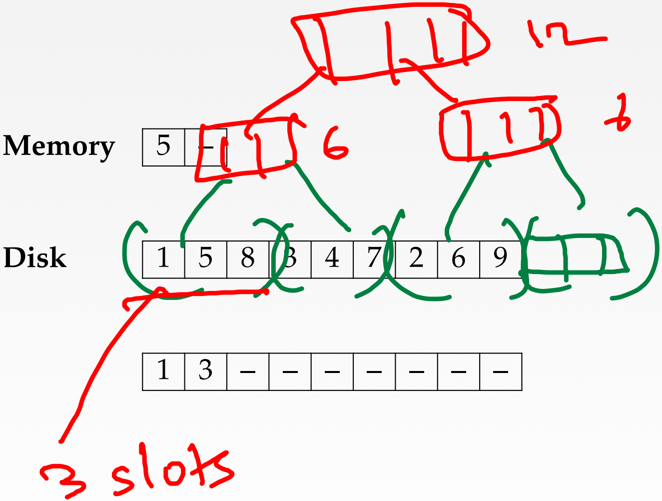
5	3

Disk

1	5	8	3	4	7	2	6	9

1	–	–	–	–	–	–	–	–

Iterative 2-Way Merge (5)



Iterative 2-Way Merge (4)

Memory

–	–

Disk

1	5	8	3	4	7	2	6	9

1	3	4	5	7	8	–	–	–

Iterative 2-Way Merge (5)

- Iteratively merging the first run with the second, the third with the fourth, and so on.
- As the number of runs (k) is halved in each iteration, there are only **Θ(log k) iterations**.
- In each iteration every element is moved exactly once
- So in each iteration, we read the whole input data once from disk
- The running time per iteration is therefore in Θ(n)
- The total running time is therefore in Θ(n log k)
- Still expensive

$k = \lceil \frac{d}{m} \rceil$

n lg k

Conclusion

- Complexity of a database system arises from the need for robust, scalable algorithms
- A database systems must satisfy many requirements: reliability, scalability, *e.t.c.*
- External sorting allows us to sort larger-than-memory datasets
- In the next lecture, we will learn about relational database systems.

References I

[1] CPPReference.
std::map.
https://en.cppreference.com/w/cpp/container/map.

[2] CPPReference.
std::unordered_map.
https://en.cppreference.com/w/cpp/container/unordered_map.