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 not take this course?

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

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 written in C++.
  - 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 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-f22/
  - 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

- Programming Assignments (**20**%)
  - Four assignments based on the BuzzDB academic DBMS.
  - You will need to upload the solutions via Gradescope.
- Exercise Sheets (**15**%)
  - Four pencil-and-paper tasks.
  - You will need to upload the sheets via Gradescope.
- Exams (**25**%)
  - Two in-person exams.

Course Rubric

- Group Project (**25**%)
  - Students will organize into groups and implement a project that is relevant to the topics discussed in class.
- Class Participation (**15**%)
  - Real-time 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 exercise sheets or 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

- Gaurav Kakkar
  - Ph.D. candidate (Computer Science)
  - Worked at Adobe (2 years), Interned at Google and Snowflake
  - Research Topic: Video Database Management System
- Jaeho Bang
  - Ph.D. candidate (Computer Science)
  - Interned at Adobe Research
  - Research Topic: Video Database Management System
- If you are acing through the assignments, you might want to hack on the video database management system (code-named EVA) that we are building.
- 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
  - 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.

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.

External Sorting

Machine Setup

- 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++ Topics

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

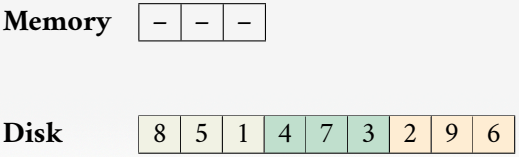
Problem Statement

- 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

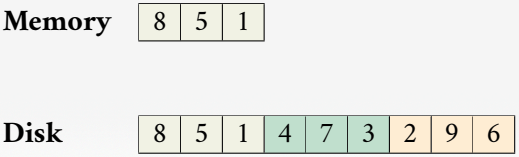
Solution

- Partition the list into a set of smaller-sized **chunks** that fit in main memory
- and sort all the **chunks**
- Use std::sort as the internal sorting algorithm.
- With **b** values fitting into main memory and **n** values that should be sorted:
- number of runs (**r**) = $\lceil \frac{n}{b} \rceil$ runs

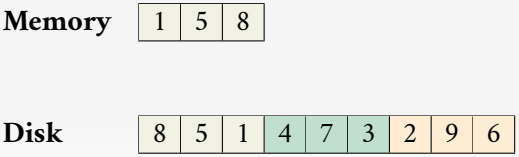
Sort each run



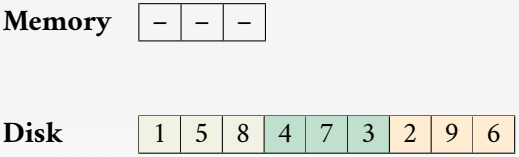
Sort each run



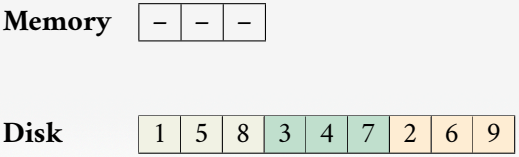
Sort each run



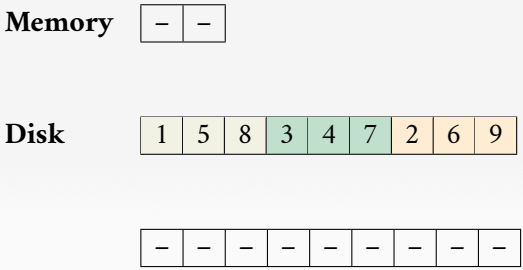
Sort each run



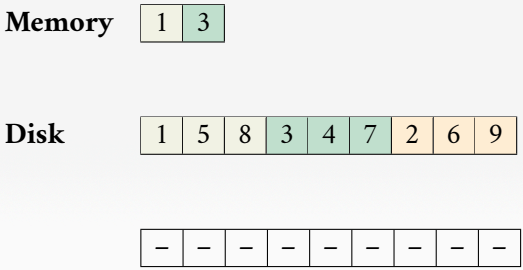
Sort each run



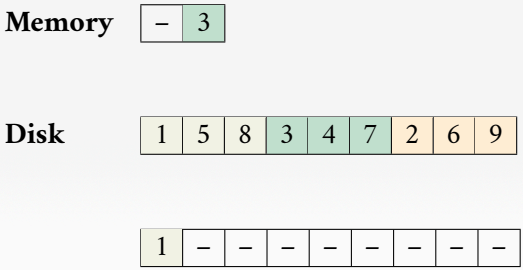
Iterative 2-Way Merge (1)



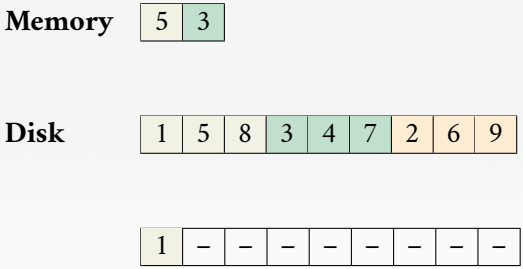
Iterative 2-Way Merge (2)



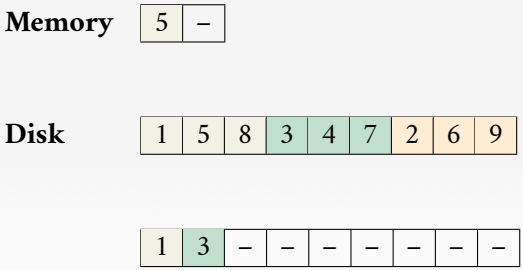
Iterative 2-Way Merge (3)



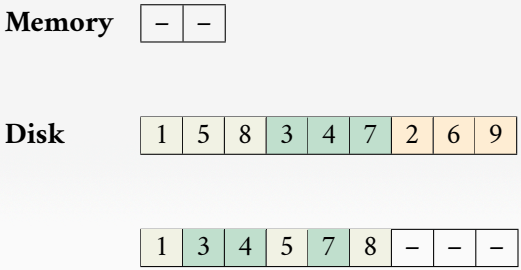
Iterative 2-Way Merge (4)



Iterative 2-Way Merge (5)



Iterative 2-Way Merge (4)



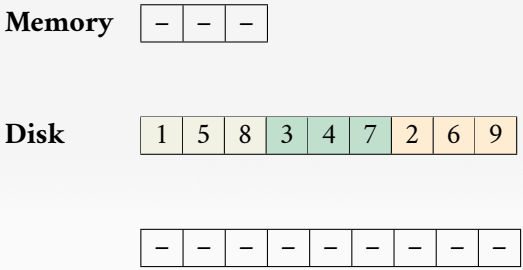
Iterative 2-Way Merge (5)

- Iteratively merging the first run with the second, the third with the fourth, and so on.
- As number of runs (**r**) is halved in each iteration, there are only $\Theta(\log_2 r)$ **iterations**.
- In each iteration every element is moved exactly once
- So in each iteration, we read and write out all the input data
- The running time per iteration is therefore in $\Theta(n)$
- The total I/O cost is therefore in $\Theta(n \log_2 r)$

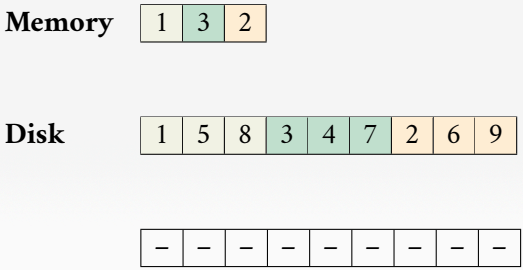
K-Way Merge

- 2-way merge algorithm only uses 3 buffer slots (two input slots and one output slot).
- But we have more available memory – **b** slots!
- **Key idea:** Use as much of the available memory as possible in every pass
- Reducing the number of passes reduces I/O cost
- Increase the length of the initial runs to **b** slots
- Merge **b** - 1 runs at a time (let **k** = **b** - 1)
- As number of runs (**r**) is reduced by **k** times in each iteration, there are only $\Theta(\log_k r)$ **iterations**

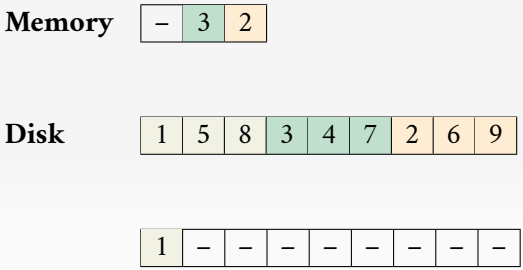
K-Way Merge (1)



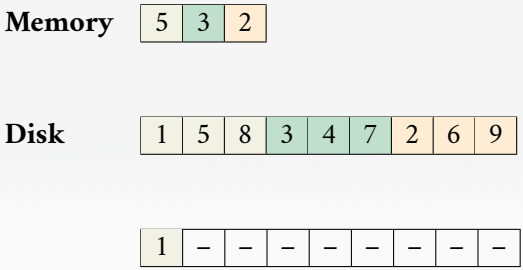
K-Way Merge (2)



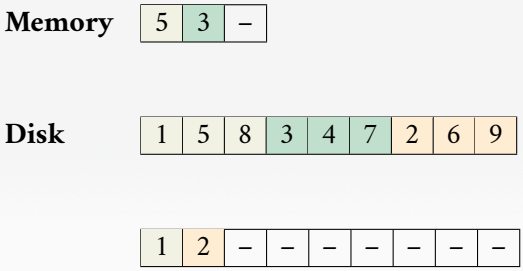
K-Way Merge (3)



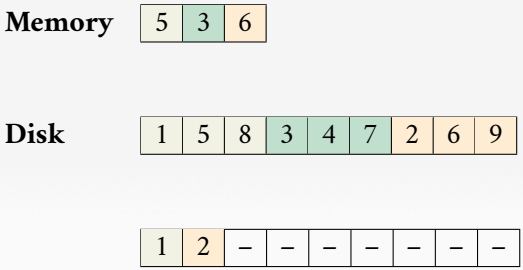
K-Way Merge (4)



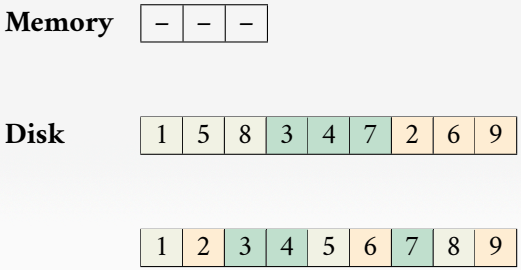
K-Way Merge (5)



K-Way Merge (6)



K-Way Merge (7)



K-Way Merge

Fewer disk reads

- A straightforward implementation would scan all **k** runs to determine the minimum.
- Although it would work, it is not efficient.
- We can improve upon this by computing the smallest element faster.
- By using a heap, the smallest element can be determined in O(log **k**) time.
- Use std::priority_queue (implemented as a heap)

K-way merge might not fit in memory

- Fall back to regular 2-way merge for a few iterations

Conclusion

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