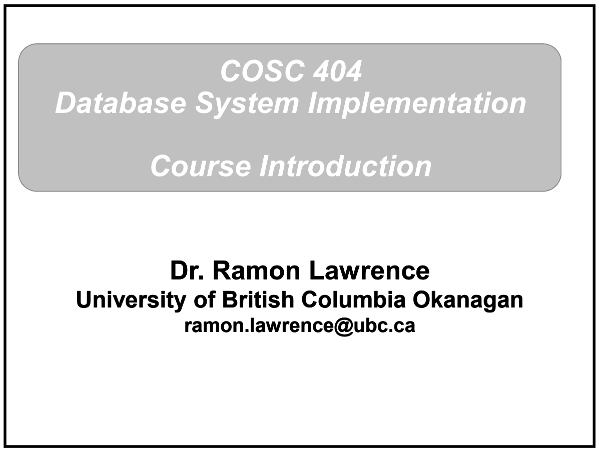
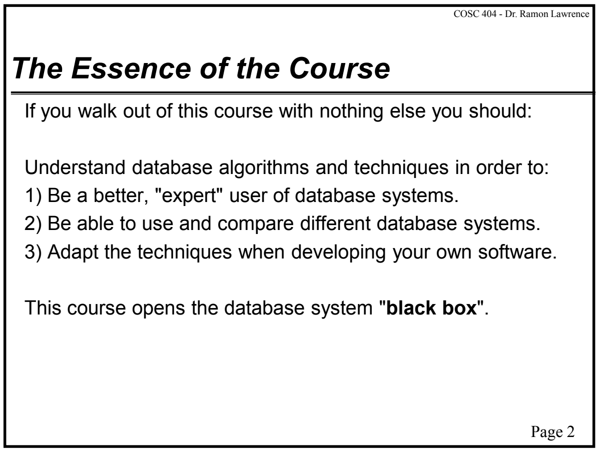


COSC 404
Database System Implementation

Course Introduction

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

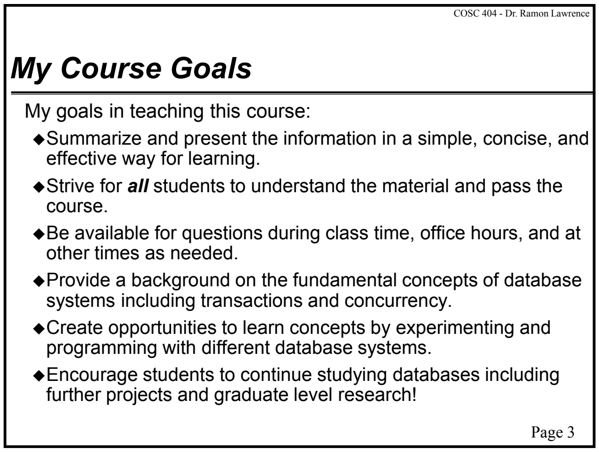


The Essence of the Course

If you walk out of this course with nothing else you should:

Understand database algorithms and techniques in order to:
1) Be a better, "expert" user of database systems.
2) Be able to use and compare different database systems.
3) Adapt the techniques when developing your own software.

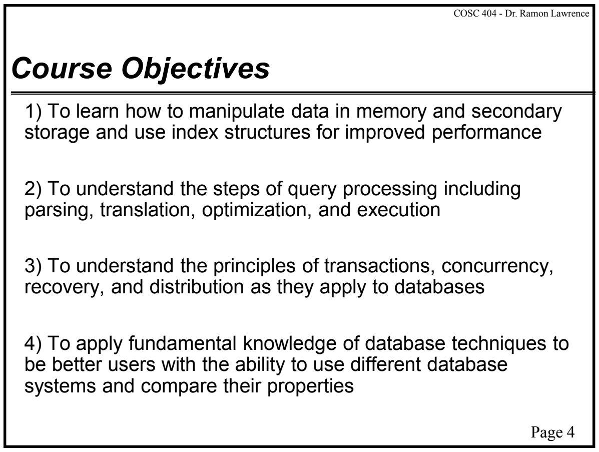
This course opens the database system "**black box**".



My Course Goals

My goals in teaching this course:

- Summarize and present the information in a simple, concise, and effective way for learning.
- Strive for ***all*** students to understand the material and pass the course.
- Be available for questions during class time, office hours, and at other times as needed.
- Provide a background on the fundamental concepts of database systems including transactions and concurrency.
- Create opportunities to learn concepts by experimenting and programming with different database systems.
- Encourage students to continue studying databases including further projects and graduate level research!



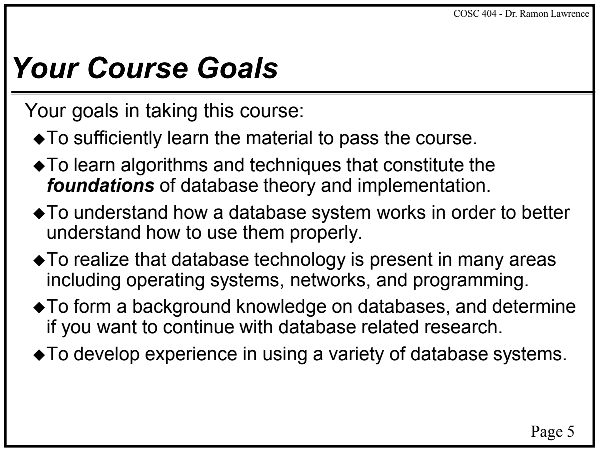
Course Objectives

1) To learn how to manipulate data in memory and secondary storage and use index structures for improved performance

2) To understand the steps of query processing including parsing, translation, optimization, and execution

3) To understand the principles of transactions, concurrency, recovery, and distribution as they apply to databases

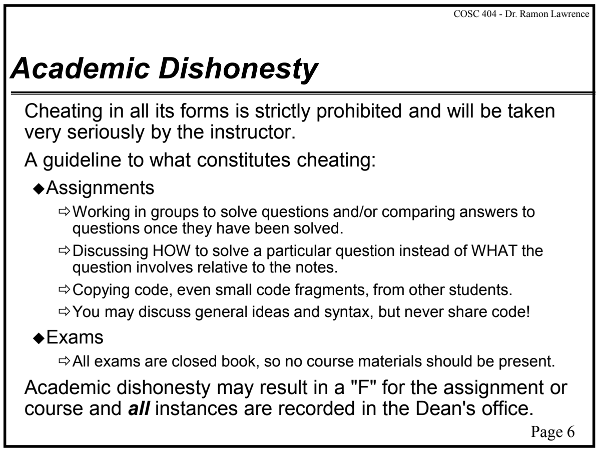
4) To apply fundamental knowledge of database techniques to be better users with the ability to use different database systems and compare their properties



Your Course Goals

Your goals in taking this course:

- To sufficiently learn the material to pass the course.
- To learn algorithms and techniques that constitute the ***foundations*** of database theory and implementation.
- To understand how a database system works in order to better understand how to use them properly.
- To realize that database technology is present in many areas including operating systems, networks, and programming.
- To form a background knowledge on databases, and determine if you want to continue with database related research.
- To develop experience in using a variety of database systems.



Academic Dishonesty

Cheating in all its forms is strictly prohibited and will be taken very seriously by the instructor.

A guideline to what constitutes cheating:

- Assignments
  - ⇨ Working in groups to solve questions and/or comparing answers to questions once they have been solved.
  - ⇨ Discussing HOW to solve a particular question instead of WHAT the question involves relative to the notes.
  - ⇨ Copying code, even small code fragments, from other students.
  - ⇨ You may discuss general ideas and syntax, but never share code!
- Exams
  - ⇨ All exams are closed book, so no course materials should be present.

Academic dishonesty may result in a "F" for the assignment or course and ***all*** instances are recorded in the Dean's office.

Assignments

There will be weekly written and programming assignments.

Written Assignments (15% of overall grade):

- ◆Practice questions similar to midterm/final exams.
- ◆Will have some time in class but mostly as homework.

Programming Assignments (20% of overall grade):

- ◆Experience applying concepts to a variety of database systems.
- ◆Will be mostly done in lab but may take more than 2 hours.

Both written and programming assignments can be done individually or in pairs.

The assignments are critical to learning the material and are designed to prepare you for the exams!

The In-Class Clicker Questions

To encourage attendance and effort, ***5%*** of your overall grade is allocated to answering in-class questions using a clicker.

- ◆The clicker can be purchased at the bookstore and sold back to the bookstore like a used textbook.
- ◆The clicker is personalized to you with your student number.
- ◆At different times during the lectures, questions reviewing material will be asked. Reponses are given using the clickers.

There will be at least 60 questions throughout the semester. Each question is worth 1 mark, and you need at least 50 right answers to get the full 5%.

- ◆That is, if you answer 40 questions right, you get 40/50 or 80%.
- ◆No make-ups for forgetting clicker or missing class.

Database Implementation Project

For graduate students only:

20% of your mark is for a major database development project.

Goal of the project is to experiment with new database systems or experiment with novel techniques expanding on class material.

This is ***not*** implementing a web site with a relational database like COSC 304.

How to Pass This Course

The most important things to do to pass this course:

- ◆Attend class
  - ⇨Read notes *before* class as preparation.
- ◆Do the written assignments
  - ⇨Important practice to learn the material for the exams!
- ◆Spend time doing the programming assignments
  - ⇨Programming with databases is a valuable, employable skill.

To get an “A” in this course do all the above plus:

- ◆Do additional practice questions.
  - ⇨Practice questions are especially helpful to re-enforce concepts.
- ◆Spend additional time programming
  - ⇨Programming assignments may take longer than a lab time. Extra time invested will payoff significantly in grades and future jobs.

Systems and Tools

Connect is used for a discussion board, for posting marks, and for anonymous feedback.

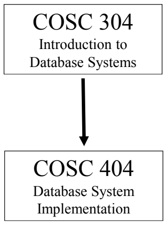
- ◆Please use the discussion board and feedback survey.

All software is available in the laboratory at SCI 234.

Access to database systems will be provided as needed.

These systems will have separate user ids and passwords.

COSC 304 vs. COSC 404


COSC 304
Introduction to Database Systems
COSC 404
Database System Implementation

Database Design and Programming

- • Data models - ER, relational, XML, JSON
- • Query languages - SQL, relational algebra
- • Design project
- • Database skills and techniques as a user
- • How to use a DBMS ; how to build a database

Database System Implementation

- • Storage and index structures
- • Transaction management, concurrency control
- • Query processing, recovery and reliability
- • How to build a DBMS
- • Non-relational systems and architectures
- • How to select a DBMS

Why are you here? Reasons Why People Take This Course

A) I need an upper-year Computer Science elective, and this course was all there was…

B) I liked COSC 304 (Intro. Databases) and thought this course may be okay too.

C) I am curious about what is in the database "black box".

D) I want to be a better developer and database user to improve my skills for future jobs.

E) I am interested in database research and advanced studies.

What to Learn What Topic are You Most Interested In?

A) Accessing data on hard drives and solid state drives

B) Learning how SQL queries get processed inside a database system

C) Learning how a database handles multiple users and recovers from failures

D) Experimenting with different databases like PostgreSQL, MongoDB, and MySQL

E) None of the above

What do you expect? What Grade are You Expecting to Get?

A) A

B) B

C) C

D) D

E) F

My Expectations

My goal is for you to learn the material and walk out of this course confident in your abilities:

- ◆To understand how a DBMS is constructed
- ◆To make intelligent decisions on data allocation, indexing, and physical designs
- ◆To describe how a DBMS supports concurrent users, transactions, and recovers from failure

I have high standards on the amount and difficulty of material that we cover. I expect a strong, continual effort in keeping up with readings and doing assignments.

The course will be very straightforward – If you do the work, you will do well.

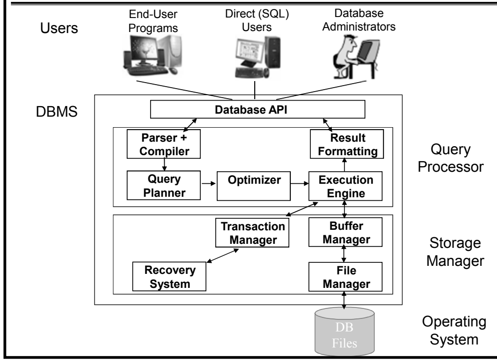
Your mark is 60% perspiration and 40% inspiration.

Database System Implementation Motivation

Key requirements of a database system:

- ◆1) Data Storage and Persistence:
  - ⇨How is data organized? Where is it located?
- ◆2) Query Processing:
  - ⇨How does the user query the data? How efficient is it?
- ◆3) Transactions, Consistency, and Reliability:
  - ⇨What happens if the computer crashes while the user is updating data?
- ◆4) Concurrency:
  - ⇨Can multiple users access the data at the same time? What happens if multiple users update the same data item?
- ◆5) Security:
  - ⇨How do you verify the user has access to the data?
- ◆6) Scalability:
  - ⇨How do you handle Big Data and lots of users?

Traditional Database System Architecture



Databases Architectures Not "One Size Fits All"

Relational databases (RDBMS) are still the dominant database architecture and apply to many data management problems.

- Over $20 billion annual market in 2015.

However, recent research and commercial systems have demonstrated that "one size fits all" is not true. There are better architectures for classes of data management problems:

- Transactional systems: In-memory architectures
- Data warehousing: Column stores, parallel query processing
- Big Data: Massive scale-out with fault tolerance
- "NoSQL": simplified query languages/structures for high performance, consistency relaxation

COSC 304 Review Question

Question: What was the acronym used to describe transactional processing systems?

A) TP

B) OLAP

C) OLTP

D) DBMS

Research Question

Question: What company is the largest database software vendor by ***sales volume***?

A) Microsoft

B) Oracle

C) IBM

D) Google

Database Architectures: NoSQL vs Relational

"NoSQL" databases are useful for several problems not well-suited for relational databases with some typical features:

- **Variable data:** semi-structured, evolving, or has no schema
- **Massive data:** terabytes or petabytes of data from new applications (web analysis, sensors, social graphs)
- **Parallelism:** large data requires architectures to handle massive parallelism, scalability, and reliability
- **Simpler queries:** may not need full SQL expressiveness
- **Relaxed consistency:** more tolerant of errors, delays, or inconsistent results ("eventual consistency")
- **Easier/cheaper:** less initial cost to get started

NoSQL is not really about SQL but instead developing data management architectures designed for scale.

- NoSQL – "Not Only SQL"

Example NoSQL Systems

MapReduce – useful for large scale, fault-tolerant analysis

- Hadoop, Pig, Hive

Key-value stores – ideal for retrieving specific items from a large set of data (architecture like a distributed hash table)

- high-scalability, availability, and performance but weaker consistency and simpler query interfaces
- Cassandra, Amazon Dynamo, Google BigTable, HBase

Document stores – similar to key-value stores except value is a document in some form (e.g. JSON)

- MongoDB, CouchDB

Graph databases – represent data as graphs

- Neo4J

Survey Question

Question: Have you used any database system besides MySQL and Microsoft SQL Server used in COSC 304?

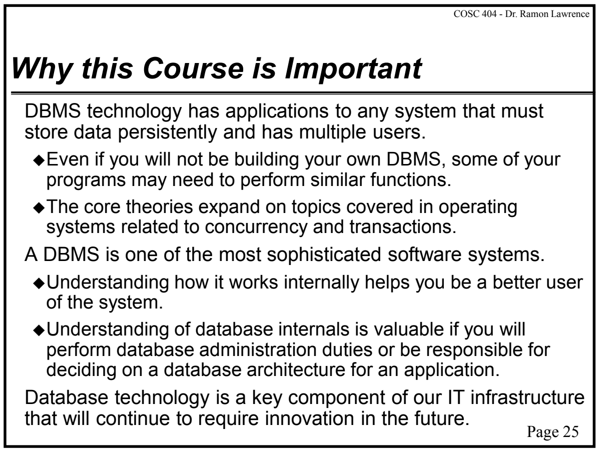
A) Oracle

B) MongoDB

C) PostgreSQL

D) More than two different databases used

E) No other databases used



Why this Course is Important

DBMS technology has applications to any system that must store data persistently and has multiple users.

- ◆Even if you will not be building your own DBMS, some of your programs may need to perform similar functions.
- ◆The core theories expand on topics covered in operating systems related to concurrency and transactions.

A DBMS is one of the most sophisticated software systems.

- ◆Understanding how it works internally helps you be a better user of the system.
- ◆Understanding of database internals is valuable if you will perform database administration duties or be responsible for deciding on a database architecture for an application.

Database technology is a key component of our IT infrastructure that will continue to require innovation in the future.

COSC 404
Database System Implementation

Data Storage and Organization

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Storage and Organization Overview

The first task in building a database system is determining how to represent and store the data.

Since a database is an application that is running on an operating system, the database must use the file system provided by the operating system to store its information.

- ◆However, many database systems implement their own file security and organization on top of the operating system file structure.

We will study techniques for storing and representing data.

Representing Data on Devices

Physical storage of data is dependent on the computer system and its associated devices on which the data is stored.

How we represent and manipulate the data is affected by the physical media and its properties.

- ◆sequential versus random access
- ◆read and write costs
- ◆temporary versus permanent memory

Review: Memory Definitions

Temporary memory retains data only while the power is on.

- ◆Also referred to as ***volatile storage***.
- ◆e.g. dynamic random-access memory (DRAM) (main memory)

Permanent memory stores data even after the power is off.

- ◆Also referred to as ***non-volatile storage***.
- ◆e.g. flash memory, hard drive, SSD, DVD, tape drives
- ◆Most permanent memory is **secondary storage** because the memory is stored in a separate device such as a hard drive.

Cache is faster memory used to store a subset of a larger, slower memory for performance.

- ◆processor cache (Level 1 & 2), disk cache, network cache

Research Question In-Memory Database

Question: Does an in-memory database need a secondary storage device for persistence?

A) Yes

B) No

Review: Sequential vs. Random Access

RAM, hard drives, and flash memory allow random access. ***Random access*** allows retrieval of any data location in any order.

Tape drives allow sequential access. ***Sequential access*** requires visiting all previous locations in sequential order to retrieve a given location.

- ◆That is, you cannot skip ahead, but must go through the tape in order until you reach the desired location.

Review: Memory Sizes

Memory size is a measure of memory storage capacity.

- ◆Memory size is measured in ***bytes***.
  - ⇨Each byte contains 8 **bits** - a bit is either a 0 or a 1.
  - ⇨A byte can store one character of text.
- ◆Large memory sizes are measured in:

⇨kilobytes (KBs)	= 10^3	= 1,000 bytes
⇨kibibyte (KiB)	= 2^{10}	= 1,024 bytes
⇨megabytes (MBs)	= 10^6	= 1,000,000 bytes
⇨mebibyte (MiBs)	= 2^{20}	= 1,048,576 bytes
⇨gigabytes (GBs)	= 10^9	= 1,000,000,000 bytes
⇨gibibytes (GiBs)	= 2^{30}	= 1,073,741,824 bytes
⇨terabytes (TBs)	= 10^{12}	= 1,000,000,000,000 bytes
⇨tebibytes (TiBs)	= 2^{40}	= 1,099,511,627,776 bytes

Transfer Size, Latency, and Bandwidth

Transfer size is the unit of memory that can be individually accessed, read and written.

- ◆DRAM, EEPROM – byte addressable
- ◆Hard drive, flash – block addressable (must read/write blocks)

Latency is the time it takes for information to be delivered after the initial request is made.

Bandwidth is the rate at which information can be delivered.

- ◆***Raw device bandwidth*** is the maximum sustained transfer rate of the device to the interface controller.
- ◆***Interface bandwidth*** is the maximum sustained transfer rate of the interface device onto the system bus.

Memory Devices Dynamic Random Access Memory

Dynamic random access memory (DRAM) is general purpose, volatile memory currently used in computers.

- ◆DRAM uses only one transistor and one capacitor per bit.
- ◆DRAM needs periodic refreshing of the capacitor.

DRAM properties:

- ◆low cost, high capacity
- ◆volatile
- ◆byte addressable
- ◆latency ~ 10 ns
- ◆bandwidth = 5 to 20 GB/s

Memory Devices Processor Cache

Processor cache is faster memory storing recently used data that reduces the average memory access time.

- ◆Cache is organized into lines/blocks of size from 64-512 bytes.
- ◆Various levels of cache with different performance.

Cache properties:

- ◆higher cost, very low capacity
- ◆cache operation is hardware controlled
- ◆byte addressable
- ◆latency – a few clock cycles
- ◆bandwidth – very high, limited by processor bus

Memory Devices Flash Memory

Flash memory is used in many portable devices (cell phones, music/video players) and also solid-state drives.

NAND Flash Memory properties:

- ◆non-volatile
- ◆low cost, high capacity
- ◆block addressable
- ◆asymmetric read/write performance: reads are fast, writes (which involve an erase) are slow
- ◆erase limit of 1,000,000 cycles
- ◆bandwidth (per chip): 40 MB/s (read), 20 MB/s (write)

Memory Devices EEPROM

EEPROM (**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory) is non-volatile and stores small amounts of data.

- ◆Often available on small microprocessors.

EEPROM properties:

- ◆non-volatile
- ◆high cost, low capacity
- ◆byte addressable
- ◆erase limit of 1,000,000 cycles
- ◆latency: 250 ns

Memory Devices
Magnetic Tapes

Tape storage is non-volatile and is used primarily for backup and archiving data.

- Tapes are ***sequential access*** devices, so they are much slower than disks.

Since most databases can be stored in hard drives and RAID systems that support direct access, tape drives are now relegated to secondary roles as backup devices.

- Database systems no longer worry about optimizing queries for data stored on tapes.

"**Tape is Dead. Disk is Tape. Flash is Disk. RAM Locality is King**." – Jim Gray (2006), Microsoft/IBM, Turing Award Winner 1998 - For seminal contributions to database and transaction processing research and technical leadership in system implementation.

Memory Devices
Solid State Drives

A ***solid state drive*** uses flash memory for storage.

Solid state drives have many benefits over hard drives:

- Increased performance (especially random reads)
- Better power utilization
- Higher reliability (no moving parts)

The performance of the solid state drive depends as much on the drive organization/controller as the underlying flash chips.

- Write performance is an issue and there is a large erase cost.

Solid state drives are non-volatile and block addressable like hard drives. The major difference is random reads are much faster (no seek time). This has a dramatic affect on the database algorithms used, and it is an active research topic.

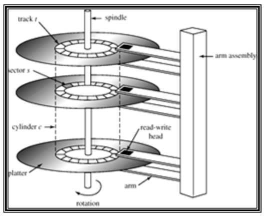
Memory Devices
Hard Drives

Data is stored on a ***hard drive*** on the surface of ***platters***. Each ***platter*** is divided into circular tracks, and each ***track*** is divided into sectors. A ***sector*** is the smallest unit of data that can be read or written. A ***cylinder i*** consists of the ***i***-th track of all the platters (surfaces).

The ***read-write head*** is positioned close to the platter surface where it reads/writes magnetically encoded data.

To read a sector, the head is moved over the correct track by the ***arm assembly***. Since the platter spins continuously, the head reads the data when the sector rotates under the head.

Head-disk assemblies allow multiple disk platters on a single spindle with multiple heads (one per platter) mounted on a common arm.



Disk Controller and Interface

The ***disk controller*** interfaces between the computer system and the disk drive hardware.

- Accepts high-level commands to read or write a sector.
- Initiates actions such as moving the disk arm to the right track and actually reading or writing the data.
- Uses a data buffer and will re-order requests for increased performance.

The disk controller has the interface to the computer.

- E.g. 3.0 Gbit/s SATA can transfer from disk buffer to computer at 300 MB/s. Note that 7200 RPM disk has a sustained disk-to-buffer transfer rate of only about 70 MB/sec.

Device Performance Calculations

We will use simple models of devices to help understand the performance benefits and trade-offs.

These models are simplistic yet provide metrics to help determine when to use particular devices and their performance.

Memory Performance Calculations

Memory model will consider only transfer rate (determined from bus and memory speed). We will assume sequential and random transfer rates are the same.

Limitations:

- There is an advantage to sequential access compared to completely random access, especially with caching. Cache locality has a major impact as can avoid accessing memory.
- Memory alignment (4 byte/8 byte) matters.
- Memory and bus is shared by multiple processes.

Memory Performance Calculations Example

A system has 8 GB DDR4 memory with 20 GB/sec. bandwidth.

Question 1: How long does it take to transfer 1 contiguous block of 100 MB memory?

transfer time = 100 MB / 20,000 MB/sec. = 0.005 sec = **5 ms**

Question 2: How long does it take to transfer 1000 contiguous blocks of 100 KB memory?

transfer time = 1000 * (100 KB / 20,000,000 KB/sec.)
= 0.005 sec = **5 ms**

Disk Performance Measures

Disk capacity is the size of the hard drive.

- = #cylinders * #tracks/cylinder * #sectors/track * #bytes/sector

Disk access time is the time required to transfer data.

- = seek time + rotational latency + transfer time
- ***Seek time*** – time to reposition the arm over the correct track.
  - Average is 1/3rd the worst. (depends on arm position and target track)
- ***Rotational latency*** – time for first sector to appear under head.
  - Average latency is 1/2 of worst case. (one half rotation of disk)
- ***Transfer time*** – time to transfer data to memory.

Data-transfer rate – the rate at which data can be retrieved from disk which is directly related to the rotational speed.

Mean time to failure (MTTF) – the average time the disk is expected to run continuously without any failure.

Disk Performance Example

Given a hard drive with 10,000 cylinders, 10 tracks/cylinder, 60 sectors/track, and 500 bytes/sector, calculate its capacity.

Answer:

capacity = 10000 * 10 * 60 * 500 = 3,000,000,000 bytes
= 3,000,000,000 bytes / 1,048,576 bytes/MiB
= **2,861 MiB** = **2.8 GiB**
= **3,000 MB** = **3 GB**

Disk Performance Example (2)

If the hard drive spins at 7,200 rpm and has an average seek time of 10 ms, how long does a 2,000 byte transfer take?

Answer:

transfer size = 2,000 bytes / 500 bytes/sector = 4 sectors
revolution time = 1 / (7200 rpm / 60 rpm/sec) = 8.33 ms
latency = 1/2 revolution time on average = 4.17 ms
transfer time = revolution time * #sectorsTransfered / #sectors/track
= 8.33 ms * 4 / 60 = 0.56 ms
total transfer time = seek time + latency + transfer time
= 10 ms + 4.17 ms + 0.56 ms = **14.73 ms**

Sequential versus Random Disk Performance Example

A hard drive spins at 7,200 rpm, has an average seek time of 10 ms, and a track-to-track seek time of 2 ms. How long does a 1 MiB transfer take under the following conditions?

- Assume 512 bytes/sector, 64 sectors/track, and 1 track/cyl.

1) The data is stored randomly on the disk.

transfer size = 1,048,576 bytes / 512 bytes/sector = 2048 sectors
revolution time = 1 / (7200 rpm / 60 rpm/sec) = 8.33 ms
latency = 1/2 revolution time on average = 4.17 ms
transfer time = revolution time / #sectors/track
= 8.33 ms / 64 = 0.13 ms per sector
total transfer time = (seek time + latency + transfer time) * #sectors
= (10 ms + 4.17 ms + 0.13 ms)*2048
= **29,286.4 ms = 29.3 seconds**

Sequential versus Random Disk Performance Example (2)

2) The data is stored sequentially on the disk .

transfer size = 1,048,576 bytes / 512 bytes/sector = 2048 sectors
= 2048 sectors / 64 sectors/track = 32 tracks
latency = 1/2 revolution time on average = 4.17 ms
transfer time = revolution time / #sectors/track
= 8.33 ms / 64 = 0.13 ms per sector
total transfer time = seek time + latency + transfer time * #sectors + track-to-track seek time * (#tracks-1)
= 10 ms + 4.17 ms + 0.13 ms*2048 + 2 ms * 31
= **342.41 ms = 0.34 seconds**

3) What would be the optimal configuration of data if the hard drive had 4 heads? What is the time in this case?

Disk Performance Practice Questions

A Seagate Cheetah 15K 3.5" hard drive has 8 heads, 50,000 cylinders, 3,000 sectors/track, and 512 bytes/sector. Its average seek time is 3.4 ms with a speed of 15,000 rpm, and a reported data transfer rate of 600 MB/sec on a 6-Gb/S SAS interface.

1) What is the capacity of the drive?

2) What is the latency of the drive?

3) What is the maximum sustained transfer rate?

4) What is the total access time to transfer 400KiB?

Disk Performance Practice Questions Older Drive

The Maxtor DiamondMax 80 has 34,741 cylinders, 4 platters, each with 2 heads, 576 sectors/track, and 512 bytes/sector. Its average seek time is 9 ms with a speed of 5,400 rpm, and a reported maximum interface data transfer rate of 100 MB/sec.

1) What is the capacity of the Maxtor Drive?

2) What is the latency of the drive?

3) What is the actual maximum sustained transfer rate?

4) What is the total access time to transfer 4KB?

Hard Drive Model Limitations and Notes

- 1) Disk sizes are quoted after formatting.
  - ***Formatting*** is done by the OS to divide the disk into blocks.
  - A sector is a physical unit of the disk while a block is a logical OS unit.
- 2) Blocks are non-continuous. ***Interblock gaps*** store control information and are used to find the correct block on a track.
  - Since these gaps do not contain user data, the actual transfer rate is less than the theoretical transfer rate based on the rotation of the disk.
  - Manufactures quote bulk transfer rates (BTR) that measure the performance of reading multiple adjacent blocks when taking gaps into account. BTR = B/(B+G) * TR (B-block size, G-gap size)
- 3) Although the bit density on the media is relatively consistent, the number of sectors per track is not.
  - More sectors/track for tracks near outer edge of platter.
  - Faster transfer speed when reading outer tracks.
- 4) Buffering and read-ahead at controller and re-ordering requests (elevator algorithm) used to increase performance.

SSD Performance Calculations

SSD model will consider:

- **IOPS** – Input/Output Operations per Second (of given data size)
- latency
- bandwidth or transfer rate
- Different performance for read and write operations.

Limitations:

- Write bandwidth is not constant. It depends on request ordering and volume, space left in hard drive, and SSD controller implementation.

SSD Performance Calculations Examples

Question 1: A SSD has read bandwidth of 500 MB/sec. How long does it take to read 100 MB of data?

read time = 100 MB / 500 MB/sec. = **0.2 sec**

Question 2: The SSD IOPS for 4 KB write requests is 25,000. What is its effective write bandwidth?

write bandwidth = 25,000 IOPS * 4 KB requests
= 100,000 KB/sec. = **100 MB/sec.**

Device Performance

Question: What device would be the fastest to read 1 MB of data?

A) DRAM with bandwidth of 20 MB/sec.

B) SSD with read 400 IOPS for 100 KB data chunks.

C) 7200 rpm hard drive with seek time of 8 ms. Assume all data is on one track.

Summary of Memory Devices

Memory Type	Volatile?	Capacity	Latency	Bandwidth	Transfer Size	Notes
DRAM	yes	High	Small	High	Byte	Best price/speed.
Cache	Yes	Low	Lowest	Very high	Byte	Large reduction in memory latency.
NAND Flash	No	Very high	Small	High	Block	Asymmetric read/write costs.
EEPROM	No	Very low	Very small	High	Byte	High cost per bit. On small CPUs.
Tape Drive	No	Very high	Very high	Medium	Block	Sequential access: Even lost backup?
Solid State Drive	No	Very high	High	Medium	Block	Great random I/O. Issue in write costs.
Hard drive	No	Very high	High	Medium	block	Beats SSDs by cost/bit but not by performance/cost.

RAID

Redundant Arrays of Independent Disks is a disk organization technique that utilizes a large number of inexpensive, mass-market disks to provide increased reliability, performance, and storage.

- Originally, the "I" stood for inexpensive as RAID systems were a cost-effective alternative to large, expensive disks. However, now performance and reliability are the two major factors.

Improvement of Reliability via Redundancy

RAID systems improve reliability by introducing ***redundancy*** to the system as they store extra information that can be used to rebuild information lost due to a disk failure.

- Redundancy occurs by duplicating data across multiple disks.
- ***Mirroring*** or ***shadowing*** duplicates an entire disk on another. Every write is performed on both disks, and if either disk fails, the other contains all the data.

By introducing more disks to the system the chance that some disk out of a set of ***N*** disks will fail is much higher than the chance that a specific single disk will fail.

- E.g., A system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days).

Review: Parity

Parity is used for error checking. A parity bit is an extra bit added to the data. A single parity bit can detect one bit error.

In ***odd parity*** the number of `1` bits in the data plus the parity bit must be odd. In ***even parity***, the number of `1` bits is even.

Example: What is the parity bit with *even parity* and the bit string: `01010010`?

- Answer: The parity bit must be a `1`, so that the # of `1`'s is even.

Parity Question

Question: What is the parity bit with *odd parity* and the bit string: `11111110`?

A) 0
B) 1
C) 2

Improvement in Performance via Parallelism

The other advantage of RAID systems is increased ***parallelism***. With multiple disks, two types of parallelism are possible:

- 1. Load balance multiple small accesses to increase throughput.
- 2. Parallelize large accesses to reduce response time.

Maximum transfer rates can be increased by allocating (***striping***) data across multiple disks then retrieving the data in parallel from the disks.

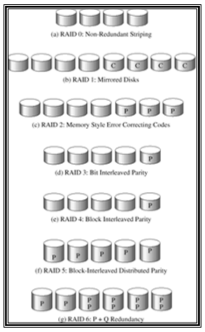
- ***Bit-level striping*** – split the bits of each byte across the disks
  - In an array of eight disks, write bit *i* of each byte to disk *i*.
  - Each access can read data at eight times the rate of a single disk.
  - But seek/access time worse than for a single disk.
- ***Block-level striping*** – with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1

RAID Levels

There are different RAID organizations, or ***RAID levels***, that have differing cost, performance and reliability characteristics:

- ◆**Level 0:** Striping at the block level (non-redundant).
- ◆**Level 1:** Mirrored disks (redundancy)
- ◆**Level 2:** Memory-Style Error-Correcting-Codes with bit striping.
- ◆**Level 3:** Bit-Interleaved Parity - a single parity bit used for error correction. Subsumes Level 2 (same benefits at a lower cost).
- ◆**Level 4:** Block-Interleaved Parity - uses block-level striping, and keeps all parity blocks on a single disk (for all other disks).
- ◆**Level 5:** Block-Interleaved Distributed Parity - partitions data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk. Subsumes Level 4.
- ◆**Level 6**: P+Q Redundancy scheme - similar to Level 5, but stores extra info to guard against multiple disk failures.

RAID Levels Discussion


(a) RAID 0: Non-Redundant Striping
C
(b) RAID 1: Mirrored Disks
P
(c) RAID 2: Memory Style Error Correcting Codes
(d) RAID 3: Bit Interleaved Parity
(e) RAID 4: Block Interleaved Parity
(f) RAID 5: Block-Interleaved Distributed Parity
(g) RAID 6: P + Q Redundancy

Level 0 is used for high-performance where data loss is not critical (parallelism).

Level 1 is for applications that require redundancy (protection from disk failures) with minimum cost.

- ◆ Level 1 requires at least two disks.

Level 5 is a common because it offers both reliability and increased performance.

- ◆ With 3 disks, the parity block for *n*th block is stored on disk (*n mod* 3) + 1. Do not have single disk bottleneck like Level 4.

Level 6 offers extra redundancy compared to Level 5 and is used to deal with multiple drive failures.

RAID Question

Question: What RAID level offers the high performance but no redundancy?

A) RAID 0
B) RAID 1
C) RAID 5
D) RAID 6

RAID Practice Question

Question: The capacity of a hard drive is 800 GB. Determine the capacity of the following RAID configurations:

i) 8 drives in RAID 0 configuration
ii) 8 drives in RAID 1 configuration
iii) 8 drives in RAID 5 configuration

A) i) 6400 GB ii) 3200 GB iii) 5600 GB
B) i) 3200 GB ii) 6400 GB iii) 5600 GB
C) i) 6400 GB ii) 3200 GB iii) 6400 GB
D) i) 3200 GB ii) 3200 GB iii) 6400 GB

RAID Summary

Level	Performance	Protection	Capacity (for N disks)
0	Best (parallel read/write)	Poor (lose all on 1 failure)	N
1	Good (write slower as 2x)	Good (have drive mirror)	N / 2
5	Good (must write parity block)	Good (one drive can fail)	N - 1
6	Good (must write multiple parity blocks)	Better (can have as many drives fail as dedicated to parity)	N – X (where X is # of parity drives such as 2)

File Interfaces

Besides the physical characteristics of the media and device, how the data is allocated on the media affects performance (***file organization***).

The physical device is controlled by the operating system. The operating system provides one or more interfaces to accessing the device.

Block-Level Interface

A ***block-level interface*** allows a program to read and write a chunk of memory called a ***block*** (or ***page***) from the device.

The page size is determined by the operating system. A page may be a multiple of the physical device's block or sector size.

The OS maintains a mapping from logical page numbers (starting at 0) to physical sectors/blocks on the device.

Block-Level Interface Operations

The block level operations at the OS level include:

- ◆read(n,p) – read block *n* on disk into memory page *p*
- ◆write(n,p) – write memory page *p* to block *n* on disk
- ◆allocate(k,n) – allocate space for *k* contiguous blocks on device as close to block *n* as possible and return first block
- ◆free(k,n) – marks *k* contiguous blocks starting at *n* as unused

The OS must maintain information on which blocks on the device are used and which are free.

Byte-Level Interface

A ***byte-level interface*** allows a program to read and write individually addressable bytes from the device.

A device will only directly support a byte-level interface if it is byte-addressable. However, the OS may provide a file-level byte interface to a device even if it is only block addressable.

File-Level Interface

A ***file-level interface*** abstracts away the device addressable characteristics and provides a standard byte-level interface for files to programs running on the OS.

A file is treated as a sequence of bytes starting from 0. File level commands allow for randomly navigating in the file and reading/writing at any location at the byte level.

Since a device may not support such access, the OS is responsible for mapping the logical byte address space in a file to physical device sectors/blocks. The OS performs buffering to hide I/O latency costs.

- ◆Although beneficial, this level of abstraction may cause poor performance for I/O intensive operations.

Databases and File Interfaces

A database optimizes performance using device characteristics, so the file interface provided on the device is critical.

General rules:

- ◆The database system needs to know block boundaries if the device is block addressable. It should not use the OS file interface mapping bytes to blocks.
  - ⇨Full block I/Os should be used. Transferring groups of blocks is ideal.
- ◆If the device has different performance for random versus sequential I/O and reads/writes, it should exploit this knowledge.
- ◆If placement of blocks on the device matters, the database should control this not the OS.
- ◆The database needs to perform its own buffering separate from the OS. Cannot use the OS virtual memory!

Databases and File Interfaces (2)

Two options:

- ◆1) Use a RAW block level interface to the device and manage everything. Very powerful but also a lot of complexity.
- ◆2) Use the OS file-level interface for data. Not suitable in general as OS hides buffering and block boundaries.

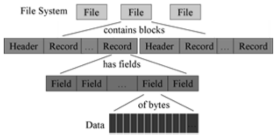
Compromise: Allocate data in OS files but treat files as raw disks. That is, do not read/write bytes but read/write to the file at the block level.

- ◆The OS stills maps from logical blocks to physical blocks on the device and manages the device.
- ◆BUT many performance issues with crossing block boundaries or reading/writing at the byte-level are avoided.
- ◆Many systems make this compromise.

Representing Data in Databases Overview

A ***database*** is made up of one or more files.

- Each ***file*** contains one or more blocks.
- Each ***block*** has a header and contains one or more records.
- Each ***record*** contains one or more fields.
- Each ***field*** is a representation of a data item in a record.



Representing Data in Memory

Consider an employee database where each employee record contains the following fields:

- name : string
- age : integer
- salary : double
- startDate : Date
- picture : BLOB

Each field is data that is represented as a sequence of bytes.

How would we store each field in memory or on disk?

Representing Data in Memory Integers and Doubles

Integers are represented in two's complement format. The amount of space used depends on the machine architecture.

- e.g. `byte`, `short`, `int`, `long`

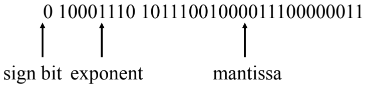
Double values are stored using a ***mantissa*** and an ***exponent***:

- Represent numbers in scientific format: $N = m * 2^e$
  - ⇨ *m* - mantissa, *e* - exponent, 2 - radix
  - ⇨ Note that converting from base 10 to base 2 is not always precise, since real numbers cannot be represented precisely in a fixed number of bits.
- The most common standard is **IEEE 754 Format**:
  - ⇨ 32 bit float - 1-bit sign; 8-bit exponent; 23-bit mantissa
  - ⇨ 64 bit double - 1-bit sign; 11-bit exponent; 52-bit mantissa

Representing Data in Memory Doubles Example

The salary $56,455.01 stored as 4 consecutive bytes is:

- Hexadecimal value is: 475C8703 Stored value is: 56455.012


0 10001110 10111001000011100000011

- Divided into bytes looks like this:

Memory Address	F001	F002	F003	F004
	01000111	01011100	10000111	00000011

Representing Data in Memory Strings and Characters

A ***character*** is represented by mapping the character symbol to a particular number.

- **ASCII** - maps characters/symbols to a number from 0 to 255.
- **UNICODE** - maps characters to a two-byte number (0 to 32,767) which allows for the encoding of larger alphabets.

A ***string*** is a sequence of characters allocated in consecutive memory bytes. A pointer indicates the location of the first byte.

- ***Null-terminated string*** - last byte value of 0 indicates end
- ***Byte-length string*** - length of string in bytes is specified (usually in the first few bytes before string starts).
- ***Fixed-length string*** - always the same size.

Representing Data in Memory Dates

A ***date*** value can be represented in multiple ways:

- Integer representation - number of days past since a given date
  - ⇨ Example: # days since Jan 1, 1900
- String representation - represent a date's components (year, month, day) as individual characters of a string
  - ⇨ Example: YYYYMMDD or YYYYDDD
  - ⇨ Please do not reinvent Y2K by using YYMMDD!!

A ***time*** value can also be represented in similar ways:

- Integer representation - number of seconds since a given time
  - ⇨ Example: # of seconds since midnight
- String representation - hours, minutes, seconds, fractions
  - ⇨ Example: HHMMSSFF

Representing Data in Memory BLOBs and Large Objects

A ***BLOB (Binary Large Object)*** type is represented as a sequence of consecutive bytes with the size of the object stored in the first few bytes.

All variable length types and objects will store a size as the first few bytes of the object.

Fixed length objects do not require a size, but may require a type identifier.

Storing Records in Memory

Now that we can allocate space for each field in memory, we must determine a way of allocating an entire record.

A ***record*** consists of one or more fields grouped together.

- Each tuple of a relation in the relational model is a record.

Two main types of records:

- ***Variable-length records*** - the size of the record varies.
- ***Fixed-length records*** - all records have the same size.

Separating Fields of a Record

The fields of a record can be separated in multiple ways:

- 1) ***No separator*** - store length of each field, so do not need a separate separator (fixed length field).
  - ⇨Simple but wastes space within a field.
- 2) ***Length indicator*** - store a length indicator at the start of the record (for the entire record) and a size in front of each field.
  - ⇨Wastes space for each length field and need to know length beforehand.
- 3) ***Use offsets*** – at start of record store offset to each field
- 4) ***Use delimiters*** - separate fields with delimiters such as a comma (comma-separated files).
  - ⇨Must make sure that delimiter character is not a valid character for field.
- 5) ***Use keywords*** - self-describing field names before field value (XML and JSON).
  - ⇨Wastes space by using field names.

Schemas

A ***schema*** is a description of the record layout.

A schema typically contains the following information:

- names and number of fields
- size and type of each field
- field ordering in record
- description or meaning of each field

Schemas Fixed versus Variable Formats

If every record has the same fields with the same types, the schema defines a ***fixed record format***.

- Relational schemas generally define a fixed format structure.

It is also possible to have no schema (or a limited schema) such that not all records have the same fields or organization.

- Since each record may have its own format, the record data itself must be ***self-describing*** to indicate its contents.
- XML and JSON documents are considered self-describing with variable schemas (***variable record formats***).

Schemas Fixed Format Example

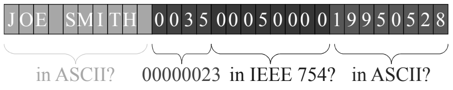
Employee record is a fixed relational schema format:

Field Name	Type	Size in Bytes
name	char(10)	10
age	integer	4
salary	double	8
startDate	Date	8 (YYYYMMDD)

Example record:

- Joe Smith, 35, $50,000, 1995/05/28

Memory allocation:



Schemas Fixed Format with Variable fields

It is possible to have a fixed format (schema), yet have variable sized records.

- ◆In the Employee example, the picture field is a BLOB which will vary in size depending on the type and quality of the image.

It is not efficient to allocate a set memory size for large objects, so the fixed record stores a pointer to the object and the size of the object which have fixed sizes.

The object itself is stored in a separate file or location from the rest of the records.

Variable Formats XML and JSON

XML:

```
<employees>
<employee>
   <name>Joe Smith</name> <age>35</age>
   <salary>50000</salary> <hired>1995/05/28</hired>
</employee>
<employee>
   <name>CEO</name><age>55</age><hired>1994/06/23</hired>
</employee>
</employees>
```

JSON:

```
{ "employees": [ { "name":"Joe Smith", "age":35,
                   "salary":50000, "hired":"1995/05/28"},
                 { "name":"CEO", "age":55,
                   "hired":"1994/06/23"} ] }
```

Variable Format Discussion

Variable record formats are useful when:

- ◆The data does not have a regular structure in most cases.
- ◆The data values are sparse in the records.
- ◆There are repeating fields in the records.
- ◆The data evolves quickly so schema evolution is challenging.

Disadvantages of variable formats:

- ◆Waste space by repeating schema information for every record.
- ◆Allocating variable-sized records efficiently is challenging.
- ◆Query processing is more difficult and less efficient when the structure of the data varies.

Format and Size Question

Question: JSON and XML are best described as:

A) fixed format, fixed size

B) fixed format, variable size

C) variable format, fixed size

D) variable format, variable size

Relational Format and Size Question

Question: A relational table uses a VARCHAR field for a person's name. It can be best described as:

A) fixed format, fixed size

B) fixed format, variable size

C) variable format, fixed size

D) variable format, variable size

Fixed vs. Variable Formats Discussion

There are also many variations that have properties of both fixed and variable format records:

- ◆Can have a record type code at the beginning of each record to denote what fixed schema it belongs to.
  - ⇨Allows the advantage of fixed schemas with the ability to define and store multiple record types per file.
- ◆Define custom record headers within the data that is only used once.
  - ⇨Do not need separate schema information, and do not repeat the schema information for every record.
- ◆It is also possible to have a record with a fixed portion and a variable portion. The fixed portion is always present, while the variable portion lists only the fields that the record contains.

Fixed versus Variable Formats Discussion (2)

We have seen fixed length/fixed format records, and variable length/variable format records.

1) Do fixed format and variable length records make sense?

Yes, you can have a fixed format schema where certain types have differing sizes. BLOBs are one example.

2) Do variable format and fixed length records make sense?

Surprisingly, Yes. Allocate a fixed size record then put as many fields with different sizes as you want and pad the rest.

|320587 | Joe Smith | SC | 95 | 3 | ←——Padding→
|184923 | Kathy Li | EN | 92 | 3 | ←—— Padding→
| 249793 | Albert Chan | SC | 94 | 3 |←—Padding→

Research Question CHAR versus VARCHAR

Question: We can represent a person's name in MySQL using either CHAR(50) or VARCHAR(50). Assume that the person's name is 'Joe'. How much space is actually used?

A) CHAR = 3 ; VARCHAR = 3

B) CHAR = 50 ; VARCHAR = 3

C) CHAR = 50 ; VARCHAR = 4

D) CHAR = 50 ; VARCHAR = 50

Storing Records in Blocks

Now that we know how to represent entire records, we must determine how to store sets of records in blocks.

There are several issues related to storing records in blocks:

- 1) ***Separation*** - how do we separate adjacent records?
- 2) ***Spanning*** - can a record cross a block boundary?
- 3) ***Clustering*** - can a block store multiple record types?
- 4) ***Splitting*** - are records allocated in multiple blocks?
- 5) ***Ordering*** - are the records sorted in any way?
- 6) ***Addressing*** - how do we reference a given record?

Storing Records in Blocks Separation

If multiple records are allocated per block, we need to know when one record ends and another begins.

Record ***separation*** is easy if the records are a fixed size because we can calculate the end of the record from its start.

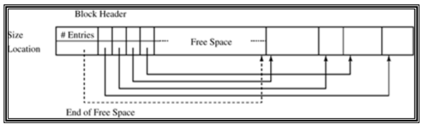
Variable length records can be separated by:

- 1) Using a special separator marker in the block.
- 2) Storing the size of the record at the start of each record.
- 3) Store the length or offset of each record in the block header.

Variable Length Records Separation and Addressing

A ***block header*** contains the number of records, the location and size of each record, and a pointer to block free space.

Records can be moved around within a block to keep them contiguous with no empty space between them and the header is updated accordingly.



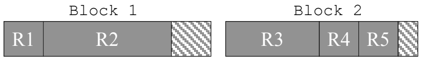
Storing Records in Blocks Spanning

If records do not exactly fit in a block, we have two choices:

- 1) Waste the space at the end of each block.
- 2) Start a record at the end of a block and continue on the next.

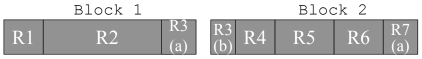
Choice #1 is the ***unspanned*** option.

- Simple because do not have to allocate records across blocks.



Choice #2 is the ***spanned*** option.

- Each piece must have a pointer to its other part.
  - ⇨Spanning is required if the record size is larger than the block size.



Storing Records in Blocks
Spanning Example

If the block size is 4096 bytes, the record size is 2050 bytes, and we have 1,000,000 records:

- How many blocks are needed for spanned/unspanned records?
- What is the block (space) utilization in both cases?

Answer:

- Unspanned
  - ⇨put one record per block implies 1,000,000 blocks
  - ⇨each block is only 2050/4096 * 100% = 50% full (utilization = 50%)
- Spanned
  - ⇨all blocks are completely full except the last one
  - ⇨# of blocks required = 1,000,000 * 2050 / 4096 = 500,049 blocks
  - ⇨utilization is almost 100%

Storing Records in Blocks
Clustering

Clustering is allocating records of different types together on the same block (or same file) because they are frequently accessed together.

Example:

- Consider creating a block where a department record is allocated together with all employees in the department:

Block 1

DPT1	EMP1	EMP2	DEPT2	EMP3	EMP4

Storing Records in Blocks
Clustering (2)

If the database commonly processes queries such as:

```
select * from employee, department
where employee.deptId = department.Id
```

then the clustering is beneficial because the information about the employee and department are adjacent in the same block.

However, for queries such as:

```
select * from employee

select * from department
```

clustering is harmful because the system must read in more blocks, as each block read contains information that is not needed to answer the query.

Storing Records in Blocks
Split Records

A ***split record*** is a record where portions of the record are allocated on multiple blocks for reasons other than spanning.

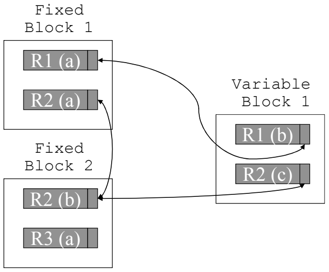
- ⇨Record splitting may be used with or without spanning.

Typically, hybrid records are allocated as split records:

- The ***fixed portion*** of the record is allocated on one block (with other fixed record portions).
- The ***variable portion*** of the record is allocated on another block (with other variable record portions).

Splitting a record is done for efficiency and simplifying allocation. The fixed portion of a record is easier to allocate and optimize for access than the variable portion.

Storing Records in Blocks
Split Records with Spanning Example



Storing Records in Blocks
Ordering Records

Ordering (or sequencing) records is when the records in a file (block) are sorted based on the value of one or more fields.

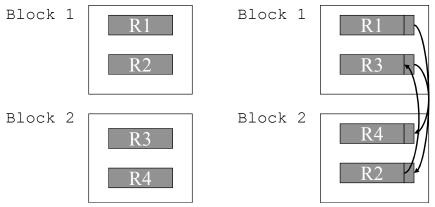
Sorting records allows some query operations to be performed faster including searching for keys and performing joins.

Records can either be:

- 1) ***physically ordered*** - the records are allocated in blocks in sorted order.
- 2) ***logically ordered*** - the records are not physical sorted, but each record contains a pointer to the next record in the sorted order.

Storing Records in Blocks
Ordering Records Example

Physical ordering **Logical Ordering**



What are the tradeoffs between the two approaches?
What are the tradeoffs of any ordering versus unordered?

Storing Records in Blocks
Addressing Records

Addressing records is a method for defining a unique value or address to reference a particular record.

Records can either be:

- ◆1) ***physically addressed*** - a record has a physical address based on the device where it is stored.
  - ⇨A physical disk address may use a sector # or a physical address range exposed by the device.
- ◆2) ***logically addressed*** - a record that is logically addressed has a key value or some other identifier that can be used to lookup its physical address in a table.
  - ⇨Logical addresses are indirect addresses because they provide a mechanism for looking up the actual physical addresses. They do not provide a method for locating the record directly on the device.
  - ⇨E.g. OS provides logical block to physical sector mapping for files.

Storing Records in Blocks
Addressing Records Tradeoff

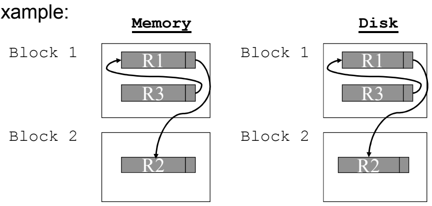
There is a tradeoff between physical and logical addressing:

- ◆Physical addresses have better ***performance*** because the record can be accessed directly (no lookup cost).
- ◆Logical addresses provide more ***flexibility*** because records can be moved on the physical device and only the mapping table needs to be updated.
  - ⇨The actual records or fields that use the logical address do not have to be changed.
  - ⇨Easier to move, update, and change records with logical addresses.

Pointer Swizzling

When transferring blocks between the disk and memory, we must be careful when handling pointers in the blocks.

For example:



Pointer swizzling is the process for converting disk pointers to memory pointers and vice versa when blocks move between memory and disk.

Operations on Files

Once data has been stored to a file consisting of blocks of records, the database system will perform operations such as update and delete to the stored records.

How records are allocated and addressed affects the performance for update and delete operations.

Operations on Files
Record Deletion

When a record is deleted from a block, we have several options:

- ◆1) Reclaim deleted space
  - ⇨Move another record to the location or compress file.
- ◆2) Mark deleted space as available for future use

Tradeoffs:

- ◆Reclaiming space guarantees smaller files, but may be expensive especially if the file is ordered.
- ◆Marking space as deleted wastes space and introduces complexities in maintaining a record of the free space available.

Operations on Files
Issues with Record Deletion

We must also be careful on how to handle references to a record that has been deleted.

- ◆If we re-use the space by storing another record in the same location, how do we know that the correct record is returned or indicate the record has been deleted?

Solutions:

- ◆1) Track down and update all references to the record.
- ◆2) Leave a "tombstone" marker at the original address indicating record deletion and not overwrite that space.
  - ⇨Tombstone is in the block for physical addressing, in the lookup table for logical addressing.
- ◆3) Allocate a unique record id to every record and every pointer or reference to a record must indicate the record id desired.
  - ⇨Compare record id of pointer to record id of record at address to verify correct record is returned.

Research Question
PostgreSQL `VACUUM`

Question: What does the **`VACUUM`** command do in PostgreSQL?

A) Cleans up your dirty house for you

B) Deletes records from a given table

C) Reclaims space used by records marked as deleted

D) Removes tables no longer used

Operations on Files
Record Insertion

Inserting a record into a file is simple if the file is not ordered.

- ◆The record is ***appended*** to the end of the file.

If the file is physically ordered, then all records must be shifted down to perform insert.

- ◆Extremely costly operation!

Inserting into a logically ordered file is simpler because the record can be inserted anywhere there is free space and linked appropriately.

- ◆However, a logically ordered file should be periodically re-organized to ensure that records with similar key values are in nearby blocks.

Memory and Buffer Management

Memory management involves utilizing buffers, cache, and various levels of memory in the memory hierarchy to achieve the best performance.

- ◆A database system seeks to minimize the number of block transfers between the disk and memory.

A ***buffer*** is a portion of main memory available to store copies of disk blocks.

A ***buffer manager*** is a subsystem responsible for allocating buffer space in main memory.

Buffer Manager Operations

All read and write operations in the database go through the buffer manager. It performs the following operations:

- ◆**read block *B*** – if block *B* is currently in buffer, return pointer to it, otherwise allocate space in buffer and read block from disk.
- ◆**write block *B*** – update block *B* in buffer with new data.
- ◆**pin block *B*** – request that *B* cannot be flushed from buffer
- ◆**unpin block *B*** – remove pin on block *B*
- ◆**output block *B*** – save block *B* to disk (can either be requested or done by buffer manager to save space)

Key challenge: How to decide which block to remove from the buffer if space needs to be found for a new block?

Buffer Management
Replacement Strategy

A ***buffer replacement strategy*** determine which block should be removed from the buffer when space is required.

- ◆Note: When a block is removed from the buffer, it must be written to disk if it was modified. and replaced with a new block.

Some common strategies:

- ◆Random replacement
- ◆Least recently used (LRU)
- ◆Most recently used (MRU)

Buffer Replacement Strategies and Database Performance

Operating systems typically use least recently used for buffer replacement with the idea that the past pattern of block references is a good predictor of future references.

However, database queries have well-defined access patterns (such as sequential scans), and a database system can use the information to better predict future references.

- ◆LRU can be a bad strategy for certain access patterns involving repeated scans of data!

Buffer manager can use statistical information regarding the probability that a request will reference a particular relation.

- ◆E.g., The schema is frequently accessed, so it makes sense to keep schema blocks in the buffer.

Research Question MySQL Buffer Management

Question: What buffer replacement policy does MySQL InnoDB use?

A) LRU

B) MRU

C) 2Q

Column Storage

The previous discussion on storage formats assumed records were allocated on blocks. For large data warehouses, it is more efficient to allocate data at the column level.

Each file represents all the data for a column. A file entry contains the column value and a record id. Records are rebuilt by combining columns using the record id.

The column format reduces the amount of data retrieved from disk (as most queries do not need all columns) and allows for better compression.

Research Question PostgreSQL Column Layout

Question: Does PostgreSQL support column layout?

A) Yes

B) No

Issues in Disk Organizations

There are many ways to organize information on a disk.

- ◆There is no one correct way.

The "best" disk organization will be determined by a variety of factors such as: ***flexibility***, ***complexity***, ***space utilization***, and ***performance***.

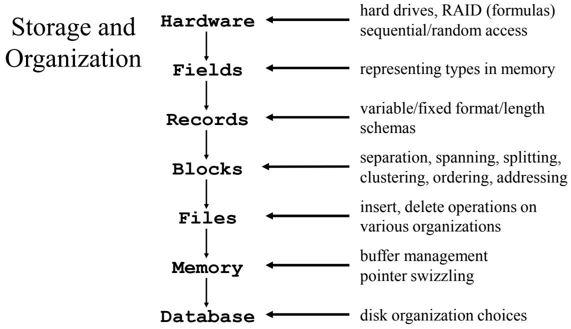
Performance measures to evaluate a given strategy include:

- ◆space utilization
- ◆expected times to search for a record given a key, search for the next record, insert/append/delete/update records, reorganize the file, read the entire file.

Key terms:

- ◆***Storage structure*** is a particular organization of data.
- ◆***Access mechanism*** is an algorithm for manipulating the data in a storage structure.

Summary



Storage and Organization

Hardware ← hard drives, RAID (formulas)
sequential/random access

Fields ← representing types in memory

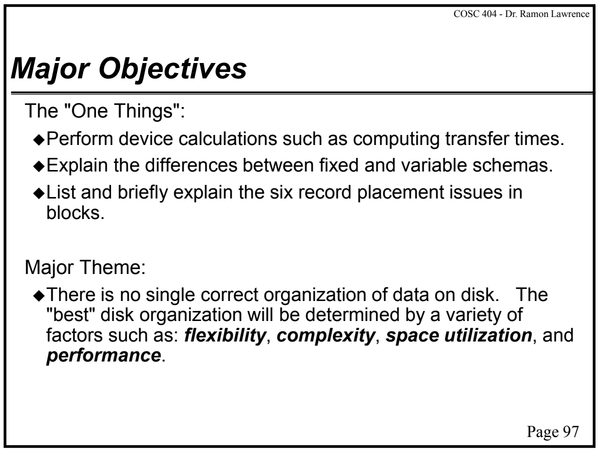
Records ← variable/fixed format/length schemas

Blocks ← separation, spanning, splitting, clustering, ordering, addressing

Files ← insert, delete operations on various organizations

Memory ← buffer management
pointer swizzling

Database ← disk organization choices



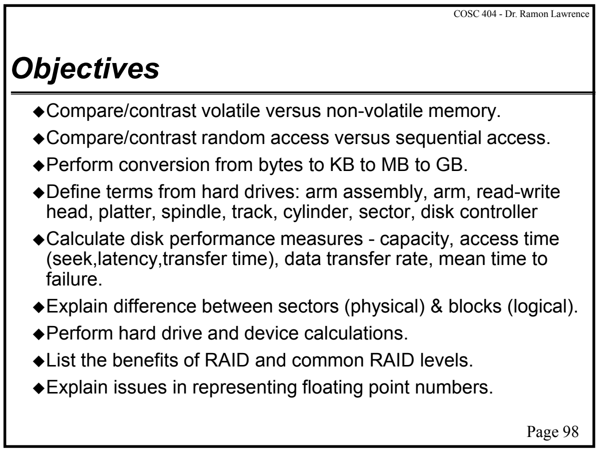
Major Objectives

The "One Things":

- Perform device calculations such as computing transfer times.
- Explain the differences between fixed and variable schemas.
- List and briefly explain the six record placement issues in blocks.

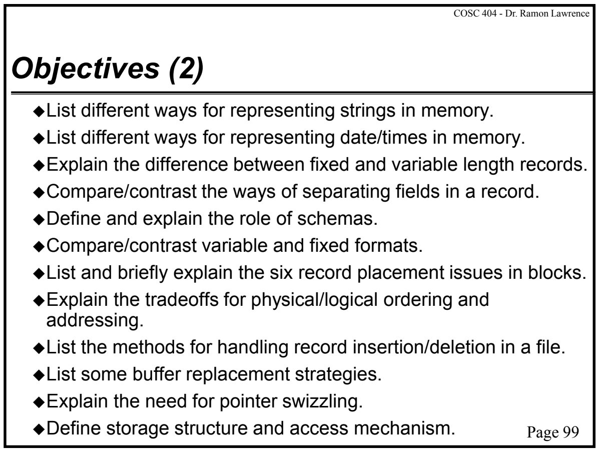
Major Theme:

- There is no single correct organization of data on disk. The "best" disk organization will be determined by a variety of factors such as: ***flexibility***, ***complexity***, ***space utilization***, and ***performance***.



Objectives

- Compare/contrast volatile versus non-volatile memory.
- Compare/contrast random access versus sequential access.
- Perform conversion from bytes to KB to MB to GB.
- Define terms from hard drives: arm assembly, arm, read-write head, platter, spindle, track, cylinder, sector, disk controller
- Calculate disk performance measures - capacity, access time (seek,latency,transfer time), data transfer rate, mean time to failure.
- Explain difference between sectors (physical) & blocks (logical).
- Perform hard drive and device calculations.
- List the benefits of RAID and common RAID levels.
- Explain issues in representing floating point numbers.



Objectives (2)

- List different ways for representing strings in memory.
- List different ways for representing date/times in memory.
- Explain the difference between fixed and variable length records.
- Compare/contrast the ways of separating fields in a record.
- Define and explain the role of schemas.
- Compare/contrast variable and fixed formats.
- List and briefly explain the six record placement issues in blocks.
- Explain the tradeoffs for physical/logical ordering and addressing.
- List the methods for handling record insertion/deletion in a file.
- List some buffer replacement strategies.
- Explain the need for pointer swizzling.
- Define storage structure and access mechanism.

COSC 404
Database System Implementation

Indexing

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Indexing Overview

An ***index*** is a data structure that allows for fast lookup of records in a file.

An index may also allow records to be retrieved in sorted order.

Indexing is important for file systems and databases as many queries require only a small set of the data in a file.

Index Terminology

The ***data file*** is the file that actually contains the records.

The ***index file*** is the file that stores the index information.

The ***search key*** is the set of attributes stored by the index to find the records in the data file.

- Note that the search key does not have to be unique - more than one record may have the same search key value.

An ***index entry*** is one index record that contains a search key value and a pointer to the location of the record with that value.

Evaluating Index Methods

Index methods can be evaluated for functionality, efficiency, and performance.

The ***functionality*** of an index can be measured by the types of queries it supports. Two query types are common:

- exact match on search key
- query on a range of search key values

The ***performance*** of an index can be measured by the time required to execute queries and update the index.

- Access time, update, insert, delete time

The ***efficiency*** of an index is measured by the amount of space required to maintain the index structure.

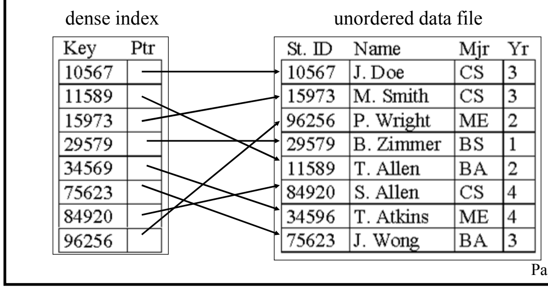
Types of Indexes

There are several different types of indexes:

- Indexes on ***ordered*** versus ***unordered*** files
  - ⇨ An ordered file is sorted on the search key. Unordered file is not.
- ***Dense*** versus ***sparse*** indexes
  - ⇨ A dense index has an index entry for every record in the data file.
  - ⇨ A sparse index has index entries for only some of the data file records (often indexes by blocks).
- ***Primary*** (clustering) indexes versus ***secondary*** indexes
  - ⇨ A primary index sorts the data file by its search key. The search key **DOES NOT** have to be the same as the primary key.
  - ⇨ A secondary index does not determine the organization of the data file.
- ***Single-level*** versus ***multi-level*** indexes
  - ⇨ A single-level index has only one index level.
  - ⇨ A multi-level index has several levels of indexes on the same file.

(Secondary) Index on Unordered File

Dense, single-level index on an unordered file.



dense index

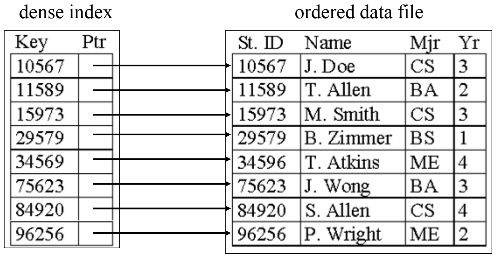
Key	Ptr
10567	
11589	
15973	
29579	
34569	
75623	
84920	
96256	

unordered data file

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
15973	M. Smith	CS	3
96256	P. Wright	ME	2
29579	B. Zimmer	BS	1
11589	T. Allen	BA	2
84920	S. Allen	CS	4
34596	T. Atkins	ME	4
75623	J. Wong	BA	3

Primary Index on Ordered File

Dense, primary, single-level index on an ordered file.



dense index

Key	Ptr
10567	
11589	
15973	
29579	
34569	
75623	
84920	
96256	

ordered data file

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

Index on Unordered/Ordered Files

An index on an unordered file makes immediate sense as it allows us to access the file in sorted order without maintaining the records in sorted order.

- ◆Insertion/deletion are more efficient for unordered files.
  - ⇨Append record at end of file or move record from end for delete.
- ◆Must only update index after data file is updated.
- ◆Searching for a search key can be done using binary search on the index.

What advantage is there for a primary index on an ordered file?

- ◆Less efficient to maintain an ordered file PLUS we must now also maintain an ordered index!

Answer: The index will be smaller than the data file as it does not store entire records. Thus, it may be able to fit entirely in memory.

Index Performance Example

We will calculate the increased performance of a dense index on an unordered/ordered file with the following parameters:

- ◆Each disk block stores 4000 bytes.
- ◆Each index entry occupies 20 bytes.
  - ⇨10 bytes for search key, 10 bytes for record pointer
  - ⇨Assume 200 index records fit in a disk block.
- ◆Each record has size 1000 bytes.
  - ⇨Assume 4 data records fit in a disk block.
- ◆The data file contains 100,000 records.

How long does it take to retrieve a record based on its key?

How much faster is this compared to having no index?

Index Performance Example (2)

Answer:

#indexBlocks = 100,000 records / 200 entries/block = 500 blocks

#diskBlocks = 100,000 records / 4 records/block = 25,000 blocks

Search index using a binary search = $\log_2 N = \log_2(500) = 8.97$ blocks

# of blocks retrieved = 9 index blocks + 1 data block = **10 blocks**

Time to find record using linear search (unordered file) = N/2

= 25,000 blocks/2 = **12,500 blocks** retrieved on average

Time to find record using binary search (ordered file) = $\log_2 N$

= $\log_2(25000) = 14.60$ blocks = **15 blocks**

Index Performance

Question: What statement is true for a non-empty, indexed table when searching for a single record?

A) Using an index is always faster than scanning the file if the data is on a hard drive

B) Using an index is always faster than scanning the file if the data is on a SSD

C) Binary searching an index is more suited to a HDD than a SSD.

D) None of the above.

Sparse Index on Ordered Files

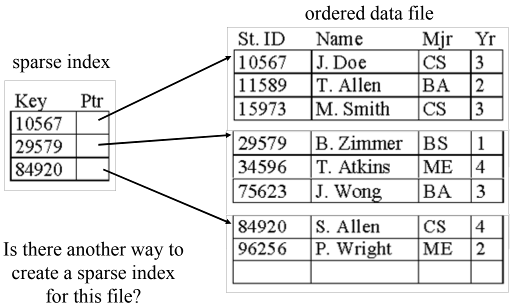
A ***sparse*** index only contains a subset of the search keys that are in the data file.

A better index for an ordered file is a sparse index since we can take advantage of the fact that the data file is already sorted.

- ◆The index will be smaller as not all keys are stored.
  - ⇨Fewer index entries than records in the file.
  - ⇨Binary search over index can be faster as fewer index blocks to read than unordered file approach.

For an ordered file, we will store ***one search key per block*** of the data file.

Sparse Index on an Ordered File



sparse index

Key	Ptr
10567	
29579	
84920	

ordered data file

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

Is there another way to create a sparse index for this file?

Sparse Index versus Dense Index

A sparse index is much more space efficient than a dense index because it only stores one search key per block.

- If a block can store 10 data records, then a sparse index will be 10 times smaller than a dense index!
- This allows more (or all) of the index to be stored in main memory and reduces disk accesses if the index is on disk.

A dense index has an advantage over a sparse index because it can answer queries like: does search key ***K*** exist? without accessing the data file (by using only the index).

- Finding a record using a dense index is easier as the index entry points directly to the record. For a sparse index, the block that may contain the data value must be loaded into memory and then searched for the correct key.

Index Performance Question

Calculate the performance of a sparse index on an **ordered** file with the following parameters:

- Each disk block stores 2000 data bytes.
- Each index entry occupies 8 bytes.
- Each record has size 100 bytes.
- The data file contains 1,000,000 records.

How long does it take to retrieve a record based on its key?

How much faster is this compared to having no index?

How much faster is this compared to a dense index?

Multi-level Index

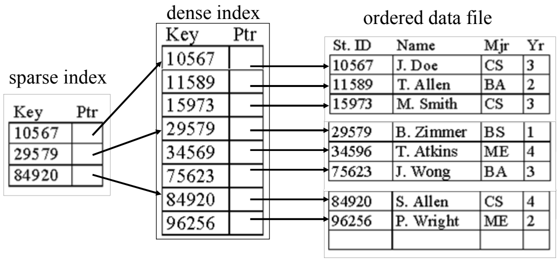
A ***multi-level index*** has more than one index level for the same data file.

- Each level of the multi-level index is smaller, so that it can be processed more efficiently.
- The first level of a multi-level index may be either sparse or dense, but all higher levels must be sparse. Why?

Having multiple levels of index increases the level of indirection, but is often quicker because the upper levels of the index may be stored entirely in memory.

- However, index maintenance time increases with each level.

Multi-level Index on an Ordered File



sparse index

Key	Ptr
10567	
29579	
84920	

dense index

Key	Ptr
10567	
11589	
15973	
29579	
34569	
75623	
84920	
96256	

ordered data file

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

Multi-level Index Performance Question

Calculate the performance of a multi-level index on an **ordered** file with the following parameters:

- Each disk block stores 2000 data bytes.
- Each index entry occupies 8 bytes.
- Each record has size 100 bytes.
- The data file contains 10,000,000 records.
- There are 3 levels of multi-level index.
  - First level is a sparse index - one entry per block.

How long does it take to retrieve a record based on its key?

Compare this to a single level sparse index.

Indexes with Duplicate Search Keys

What happens if the search key for our index is not unique?

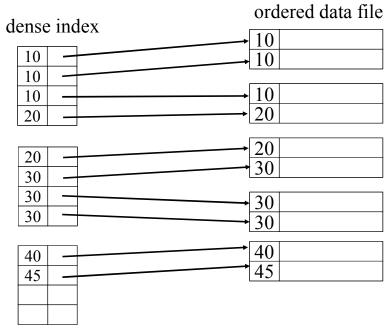
- The data file contains many records with the same search key. This is possible because we may index a field that is not a primary key of the relation.

Both sparse and dense indexes still apply:

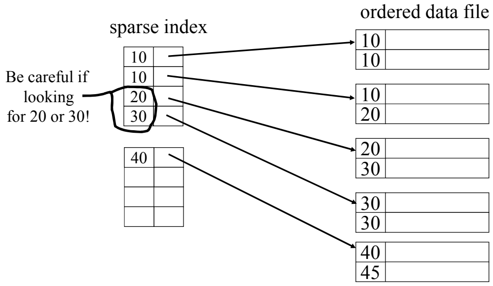
- 1) Dense index with entry for every record
- 2) Sparse index containing one entry per block

Note: Search strategy changes if have many records with same search key.

Handling Duplicate Keys
Dense Index - One Entry per Record



Handling Duplicate Keys
Sparse Index - One Entry per Block



Secondary Indexes

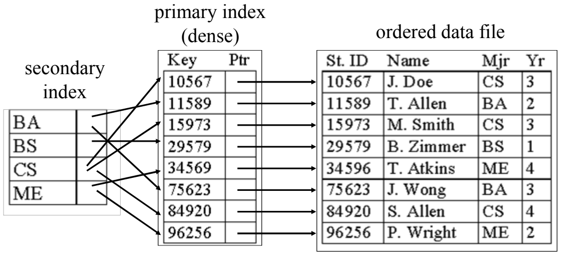
A ***secondary index*** is an index whose search key does not determine the ordering of the data file.

- A data file can have only one primary index but many secondary indexes.

Secondary index entries often refer to the primary index instead of the data records directly.

- Advantage - simpler maintenance of secondary index.
  - ⇨ Secondary index changes only when primary index changes not when the data file changes.
- Disadvantage - less efficient due to indirection.
  - ⇨ Multiple levels of indirection as must use secondary index, then go to primary index, then access record in data file.

Secondary Index Example



St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

Secondary Indexes
Handling Duplicate Search Keys

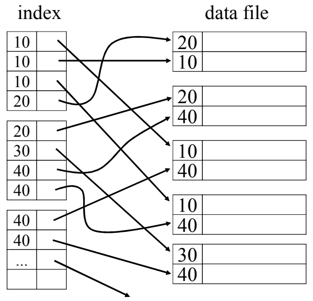
A secondary index may have duplicate search keys.

Techniques for handling duplicates:

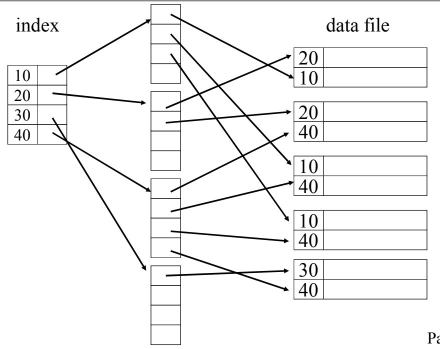
- 1) Create an index entry for each record (dense)
  - ⇨ Wastes space as key value repeated for each record
- 2) Use buckets (blocks) to store records with same key
  - ⇨ The index entry points to the first record in the bucket.
  - ⇨ All other matching records are retrieved from the bucket.

Handling Duplicates
Secondary Index - One Entry per Record

Problem:
Excess overhead!
- disk space
- search time



Handling Duplicates
Secondary Index - Buckets (as blocks)



Secondary Indexes
Discussion

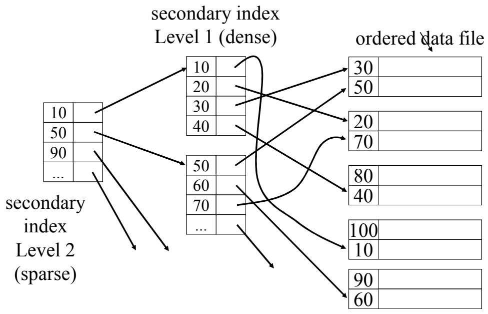
It is not possible to have a sparse secondary index. There must be an entry in the secondary index for **EACH KEY VALUE**.

- ◆However, it is possible to have a multi-level secondary index with upper levels sparse and the lowest level dense.

Secondary indexes are especially useful for indexing foreign key attributes.

The bucket method for handling duplicates is preferred as the index size is smaller.

Multi-level Secondary Index



Secondary Indexes
Buckets in Query Processing

Consider the query:

```
        select * from student
        where Major = "CS" and Year = "3"
```

If there were secondary indexes on both `Major` and `Year`, then we could retrieve the buckets for `Major="CS"` and `Year="3"` and compare the records that are in both.

- ◆We then retrieve only the records that are in both buckets.

Question: How would answering the query change if:

- ◆a) There were no secondary indexes?
- ◆b) There was only one secondary index?

Secondary Index Example

We will calculate the increased performance of a secondary index on a data file with the following parameters:

- ◆Each disk block stores 4000 bytes.
- ◆Each index entry occupies 20 bytes.
  - ⇨10 bytes for search key, 10 bytes for record pointer
  - ⇨Assume 200 index records fit in a disk block.
  - ⇨**Assume one index entry per record.**
- ◆Each record has size 1000 bytes.
  - ⇨Assume 4 data records fit in a disk block.
- ◆The data file contains 1,000,000 records.

How long does it take to retrieve a record based on its key?

How much faster is this compared to having no index?

Secondary Index Example (2)

Answer:

#indexBlocks = 1,000,000 records / 200 entries/block = 5,000 blocks

#diskBlocks = 1,000,000 records / 4 records/block = 250,000 blocks

Search index using a binary search

= $\log_2 N = \log_2(5000) = 12.28$ blocks

# of blocks retrieved

= 13 blocks + 1 primary index block + 1 data block = **15 blocks**

Time to find record using linear scan (unordered file) = N/2

= 250,000 /2 = **125,000 blocks** retrieved on average

Note that need to do full table scan (250,000 blocks) ALWAYS if want to find all records with a given key value (not just one).

Lesson: Secondary indexes allow significant speed-up because the alternative is a linear search of the data file!

Secondary Index

Question: A secondary index is constructed that refers to the primary index to locate its records. What is the minimum number of blocks that must be processed to retrieve a record using the secondary index?

A) 0

B) 1

C) 2

D) 3

E) 4

Index Maintenance

As the data file changes, the index must be updated as well. The two operations are ***insert*** and ***delete***.

Maintenance of an index is similar to maintenance of an ordered file. The only difference is the index file is smaller.

Techniques for managing the data file include:

- 1) Using overflow blocks
- 2) Re-organizing blocks by shifting records
- 3) Adding or deleting new blocks in the file

These same techniques may be applied to both sparse and dense indexes.

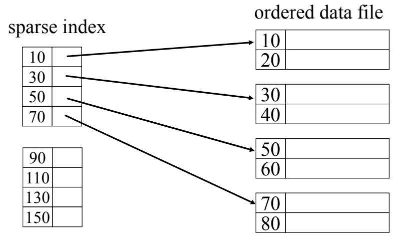
Index Maintenance Summary

In the process of handling inserts and deletes in the data file, any of the previous 3 techniques may be used on the data file.

The effect of these techniques on the index file are as follows:

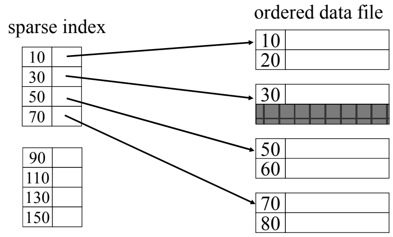
- Create/delete overflow block for data file
  - No effect on both sparse/dense index (overflow block not indexed).
- Create/delete new sequential block for data file
  - Dense index unaffected, sparse index needs new index entry for block.
- Insert/Delete/Move record
  - Dense index must either insert/delete/update entry.
  - Sparse index may only have to update entry if the smallest key value in the block is changed by the operation.

Index Maintenance Record Deletion with a Sparse Index



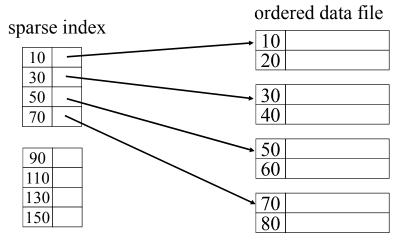
Delete record with key 40 from data file.

Index Maintenance Record Deletion with a Sparse Index (2)



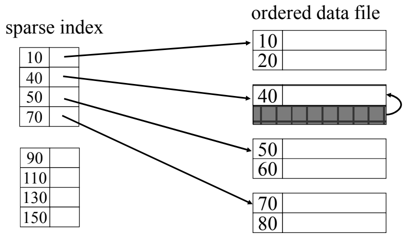
Record deleted.
No change to index.

Index Maintenance
Record Deletion with a Sparse Index (3)



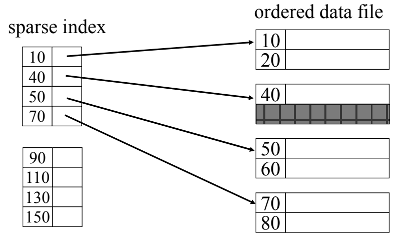
Delete record with key
30 from data file.

Index Maintenance
Record Deletion with a Sparse Index (4)



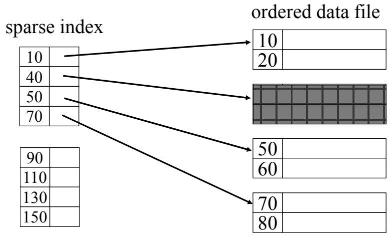
Record 30 deleted.
Shift record up in data block.
Update index entry to 40.

Index Maintenance
Record Deletion with a Sparse Index (5)



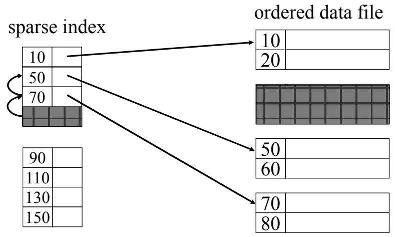
Delete record with key 40.

Index Maintenance
Record Deletion with a Sparse Index (6)



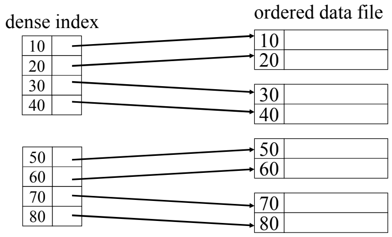
Delete record. Delete block.

Index Maintenance
Record Deletion with a Sparse Index (7)



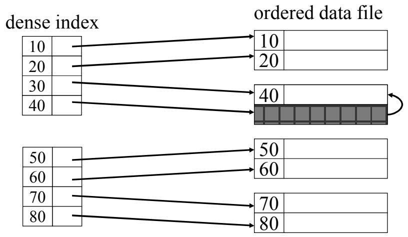
Delete index entry.
Shift index entries in block up.

Index Maintenance
Record Deletion with a Dense Index



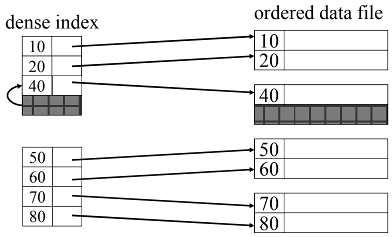
Delete record with key 30.

Index Maintenance
Record Deletion with a Dense Index (2)



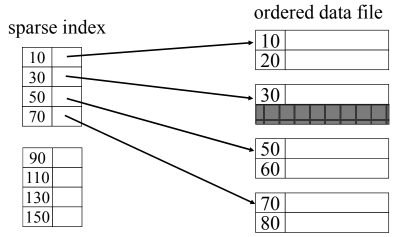
Delete record. Shift 40 up.

Index Maintenance
Record Deletion with a Dense Index (3)



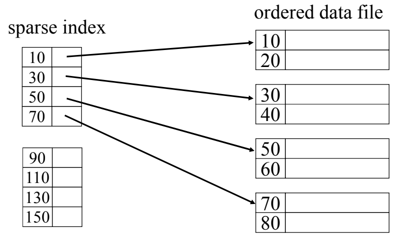
Delete index entry.
Shift index entry for 40 up.

Index Maintenance
Record Insertion with a Sparse Index



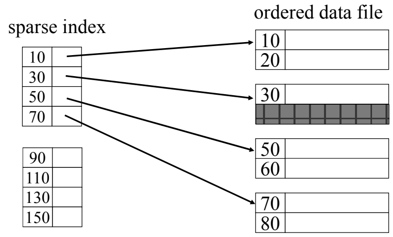
Insert record with key 40.

Index Maintenance
Record Insertion with a Sparse Index (2)



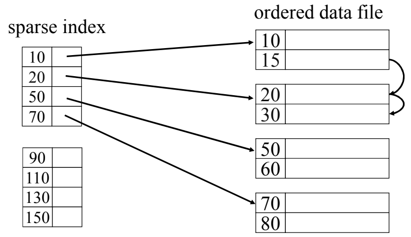
Record inserted in free
space in second block.
No updates to index.

Index Maintenance
Record Insertion with a Sparse Index (3)



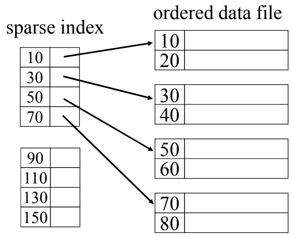
Insert record with key 15.
Use immediate re-organization.

Index Maintenance
Record Insertion with a Sparse Index (4)



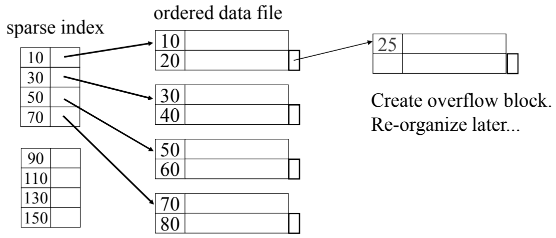
Shift records down to make room for 15.
Update index pointer for block 2.

Index Maintenance Record Insertion with a Sparse Index (5)



Insert record with key 25.
Use overflow blocks.

Index Maintenance Record Insertion with a Sparse Index (6)



Create overflow block.
Re-organize later...

Handling Data Evolution

Since it is common for both the data file and index file to evolve as the database is used, often blocks storing data records and index records are not filled completely.

By leaving a block 75% full when it is first created, then data evolution can occur without having to create overflow blocks or move records around.

The tradeoff is that with completely filled blocks the file occupies less space and is faster to process.

Conclusion

Indexes are lookup mechanisms to speed access to particular records in the data file.

- ◆An ***index*** consists of an ordered sequence of index entries containing a search key and a pointer.
  - ⇨An index may be either ***dense*** (have one entry per record) or ***sparse*** (have one entry per block).
  - ⇨***Primary*** indexes have the index search key as the same key that is used to physically order the file. ***Secondary*** indexes do not have an affect on the data file ordering.

An index is an ordered data file when inserting/deleting entries.

- ◆When the data file is updated the index may be updated.

Major Objectives

The "One Things":

- ◆Explain the types of indexes: ordered/unordered, sparse/dense, primary/secondary, single/multi-level
- ◆Perform calculations on how fast it takes to retrieve one record or answer a query given a certain data file and index type.

Major Theme:

- ◆Indexing results in a dramatic increase in the performance of many database queries by minimizing the number of blocks accessed. However, indexes must be maintained, so they should not be used indiscriminately.

Objectives

- ◆Define: index file, search key, index entry
- ◆List the index evaluation metrics/criteria.
- ◆Explain the difference between the difference types of indexes: ordered/unordered, dense/sparse, primary/secondary, single/multi level and be able to perform calculations.
- ◆List the techniques for indexing with duplicate search keys.
- ◆Discuss some of the issues in index maintenance.
- ◆Compare/contrast single versus multi-level indexes.
- ◆Explain the benefit of secondary indexes on query performance and be able to perform calculations.

COSC 404
Database System Implementation

B-trees

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

B-Trees and Indexing Overview

We have seen how multi-level indexes can improve search performance.

One of the challenges in creating multi-level indexes is maintaining the index in the presence of inserts and deletes.

We will learn B+-trees which are the most common form of index used in database systems today.

B-trees Introduction

A ***B-tree*** is a search tree where each node has >= ***n*** data values and <= ***2n***, where we chose ***n*** for our particular tree.

- Each key in a node is stored in a sorted array.
  - ⇨ key[0] is the first key, key[1] is the second key,…,key[2*n*-1] is the 2*n*th key
  - ⇨ key[0] < key[1] < key[2] < … < key[2*n*-1]
- There is also an array of pointers to children nodes:
  - ⇨ child[0], child[1], child[2], …, child[2*n*]
  - ⇨ Recursive definition: Each subtree pointed to by child[i] is also a B-tree.
- For any key[i]:
  - ⇨ 1) key[i] > all entries in subtree pointed to by child[i]
  - ⇨ 2) key[i] <= all entries in subtree pointed to by child[i+1]
- A node may not contain all key values.
  - ⇨ # of children = # of keys +1

A B-tree is **balanced** as every leaf has the same depth.

B-trees Order Debate

There is an interesting debate on how to define an ***order*** of a B-tree. The original definition was the one given:

- The ***order n*** is the minimum # of keys in a node. The maximum number is ***2n***.

However, may want to have a B-tree where the maximum # of keys in a node is odd.

- This is not possible by the above definition.

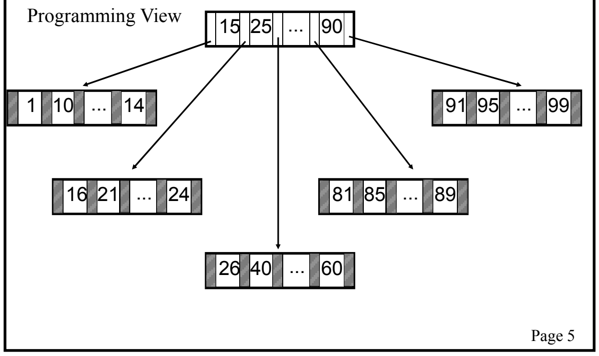
Consequently, can define order as the maximum # of keys in a node (instead of the minimum).

- Further, some use maximum # of pointers instead of keys.

Bottom line: B-trees with an odd maximum # of keys will be avoided in the class.

- The minimum # of nodes for an odd maximum ***n*** will be $\lfloor n/2 \rfloor$.

B-trees Example



B-Trees Performance

Question: A B-tree has a maximum of 10 keys per node. What is the maximum number of children for a given node?

A) 0
B) 1
C) 10
D) 11
E) 20

2-3 Trees Introduction

A ***2-3 tree*** is a B-tree where each node has either ***1*** or ***2*** data values and ***2*** or ***3*** children pointers.

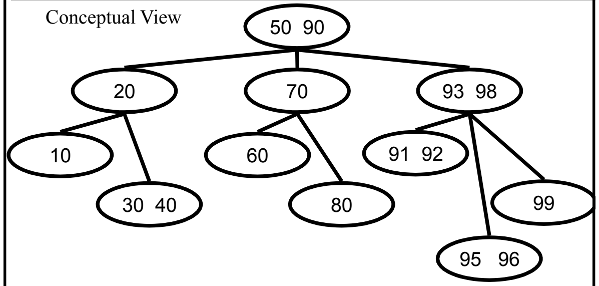
- It is a special case of a B-tree.

Fact:

- A 2-3 tree of height ***h*** always has at least as many nodes as a full binary tree of height ***h***.
  - ⇨That is, a 2-3 tree will always have at least 2^h-1 nodes.

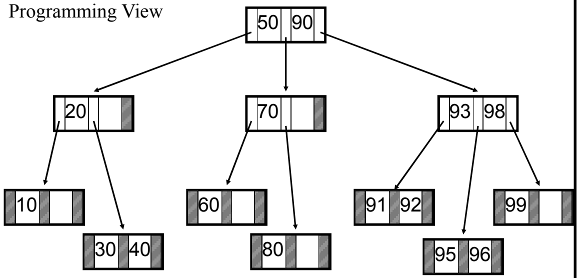
2-3 Search Tree Example

Conceptual View



2-3 Tree Example

Programming View



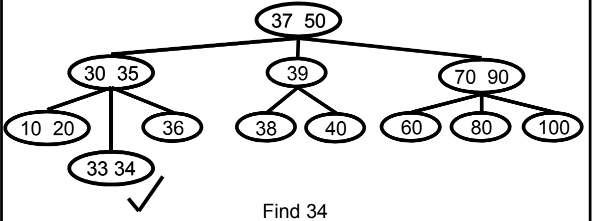
Searching a 2-3 Tree

Searching a 2-3 tree is similar to searching a binary search tree.

Algorithm:

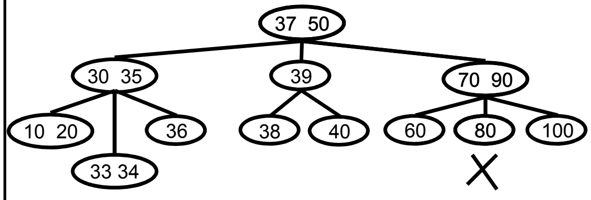
- Start at the root which begins as the curNode.
- If curNode contains the search key we are done, and have found the search key we were looking for.
- A 2-node contains one key:
  - ⇨If search key < key[0], go left (child[0]) otherwise go right (child[1])
- A 3-node contains two key values:
  - ⇨If search key < key[0], go left with first child pointer (child[0])
  - ⇨else if search key < key[1] go down middle child pointer (child[1])
  - ⇨else (search key >= key[1]) go right with last child pointer (child[2])
- If we encounter a NULL pointer, then we are done and the search failed.

Searching a 2-3 Tree Example #1



Find 34

Searching a 2-3 Tree Example #2



Find 82

Insertion into a 2-3 Tree

Algorithm:

- Find the leaf node where the new key belongs.
- This insertion node will contain either a single key or two keys.
- If the node contains 1 key, insert the new key in the node (in the correct sorted order).
- If the node contains 2 keys:
  - Insert the node in the correct sorted order.
  - The node now contains 3 keys (overflow).
  - Take the middle key and promote it to its parent node. (split node)
  - If the parent node now has more than 3 keys, repeat the procedure by promoting the middle node to its parent node.
- This promotion procedure continues until:
  - Some ancestor has only one node, so overflow does not occur.
  - All ancestors are "full" in which case the current root node is split into two nodes and the tree "grows" by one level.

Insertion into a 2-3 Tree Splitting Algorithm

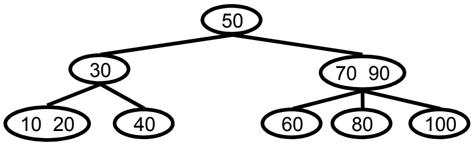
Splitting Algorithm:

- Given a node with overflow (more than 2 keys in this case), we split the node into two nodes each having a single key.
- The middle value (in this case key[1]) is passed up to the parent of the node.
  - This, of course, requires *parent* pointers in the 2-3 tree.
- This process continues until we find a node with sufficient room to accommodate the node that is being percolated up.
- If we reach the root and find it has 2 keys, then we split it and create a new root consisting of the "middle" node.

The splitting process can be done in logarithmic time since we split at most one node per level of the tree and the depth of the tree is logarithmic in the number of nodes in the tree.

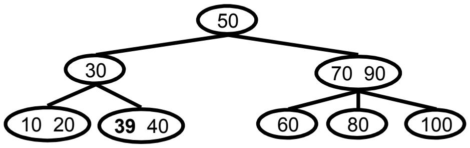
- Thus, 2-3 trees provide an efficient height balanced tree.

Insertion Examples



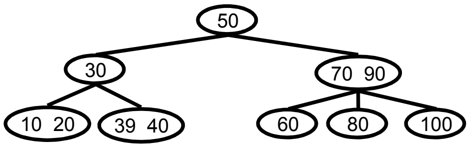
Insert 39

Insertion Examples



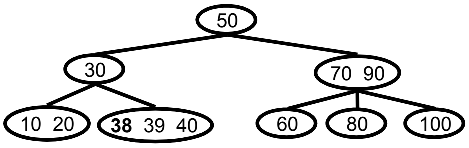
Done!

Insertion Examples



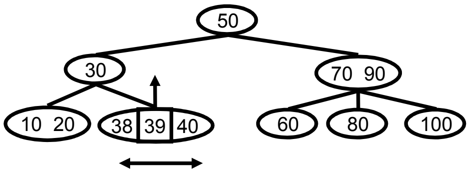
Insert 38

Insertion Examples



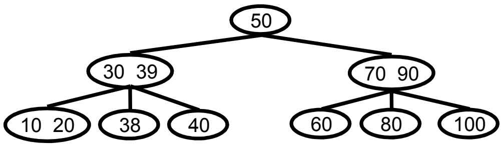
Insert 38

Insertion Examples



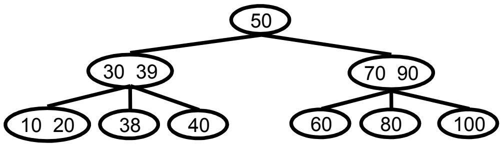
Push up, split apart

Insertion Examples



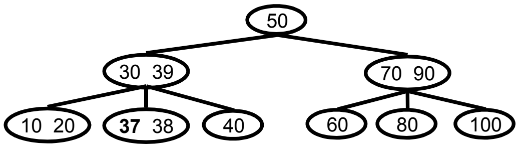
Done!

Insertion Examples



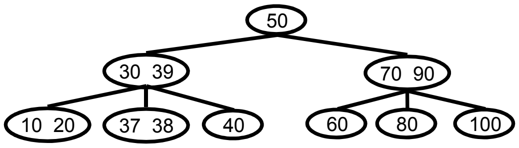
Insert 37

Insertion Examples



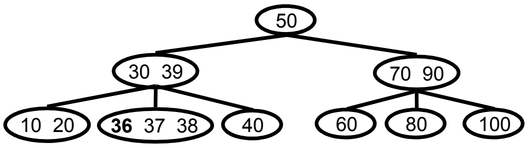
Done!

Insertion Examples



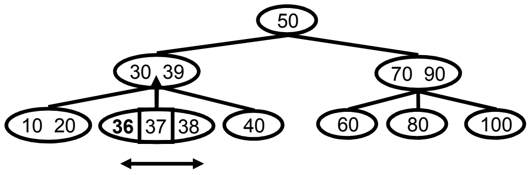
Insert 36

Insertion Examples



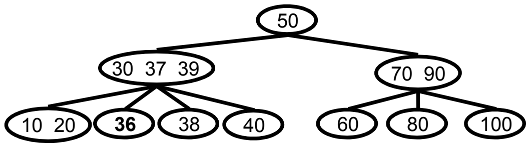
Insert 36

Insertion Examples



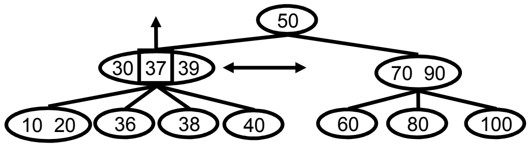
Push up, split apart

Insertion Examples



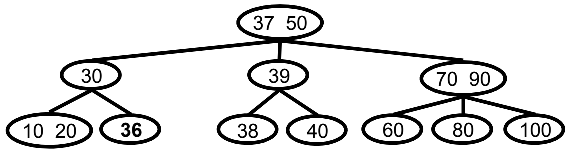
Need to go further up the tree to resolve overcrowding

Insertion Examples



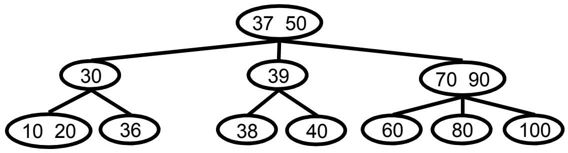
Push up, split apart

Insertion Examples



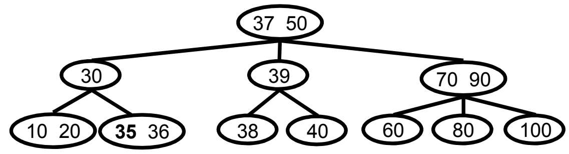
Done!

Insertion Examples



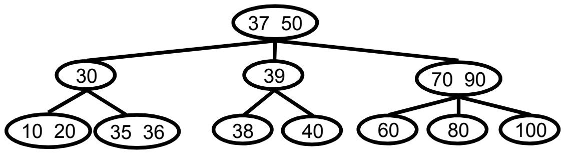
Insert 35

Insertion Examples



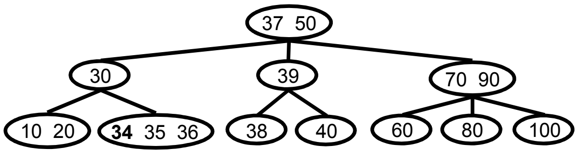
Insert 35

Insertion Examples



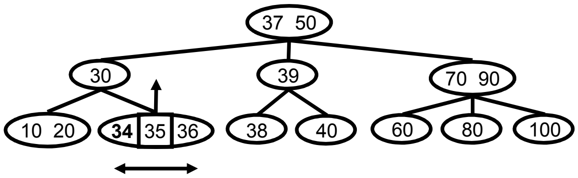
Insert 34

Insertion Examples



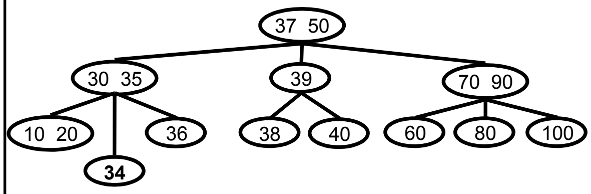
Insert 34

Insertion Examples



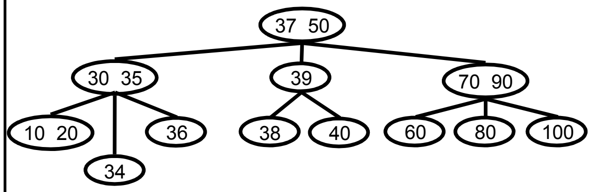
Push up, split apart

Insertion Examples



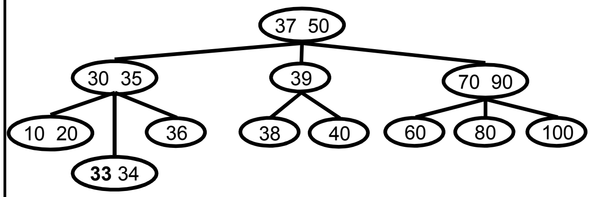
Done!

Insertion Examples



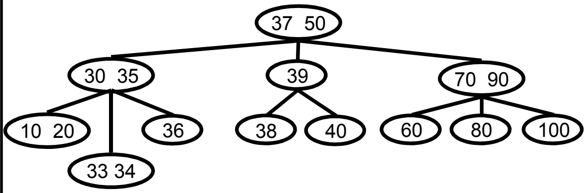
Insert 33

Insertion Examples



Done!

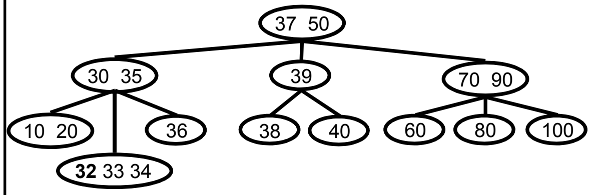
Insertion Examples



Insert 32

Page 37

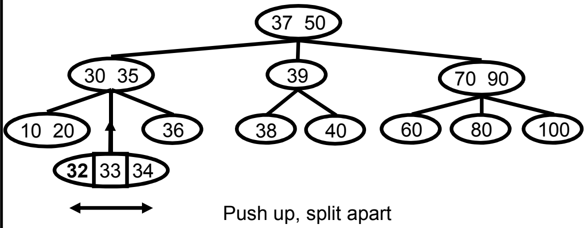
Insertion Examples



Insert 32

Page 38

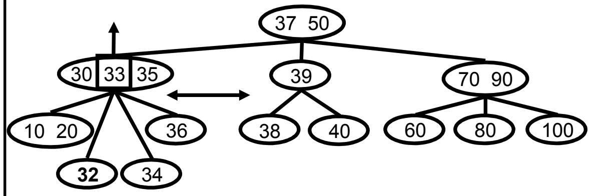
Insertion Examples



Push up, split apart

Page 39

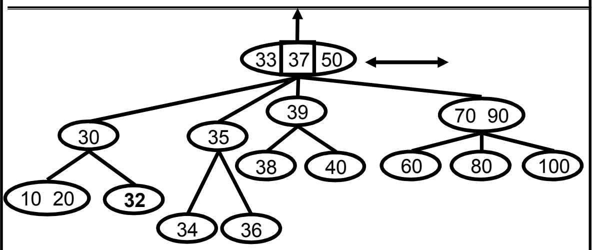
Insertion Examples



Push up, split apart

Page 40

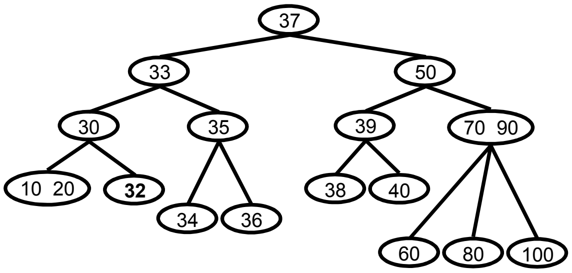
Insertion Examples



Push up, split apart

Page 41

Insertion Examples



A new level is born!

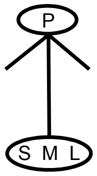
Page 42

Insertion Special Cases

There are 3 cases of splitting for insertion:

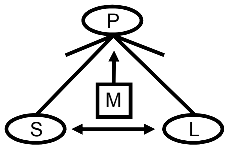
- 1) Splitting a leaf node
  - Promote middle key to parent and create two new nodes containing half the keys.
  - Do not have child pointers to worry about.
- 2) Splitting an interior node
  - Promote middle key to parent and create two new nodes containing half the keys.
  - Make sure child pointers are copied over as well as keys.
- 3) Splitting the root node
  - Similar to splitting an interior node, but now the tree will grow by one level and will have a new root node (must update root pointer).
- Case 2 is **ONLY** possible if a leaf node has been previously split. Case 3 is only possible if all ancestors of the leaf node had to be split.

Special Case: Splitting a Leaf Node



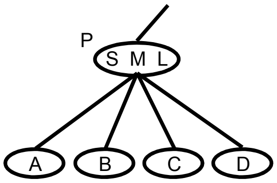
Leaf node overflow

Special Case: Splitting a Leaf Node (2)



Splitting a leaf node

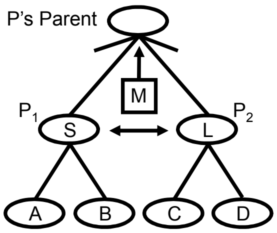
Special Case: Splitting an Interior Node



Interior node overflow

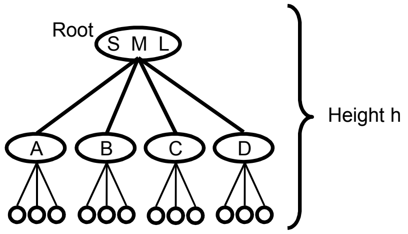
Splitting an internal node

Special Case: Splitting an Interior Node (2)



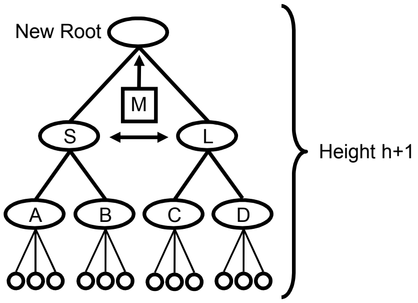
Splitting an internal node

Special Case: Splitting the Root Node



Splitting the root node

Special Case: Splitting the Root Node (2)



B-tree Insertion Practice Question

For a B-tree of ***order 1 (max. keys=2)***, insert the following keys in order:

- 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150

Deletion From a 2-3 Tree

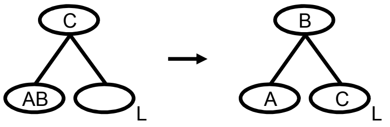
Algorithm:

- To delete a key ***K***, first locate the node ***N*** containing ***K***.
  - If K is not found, then deletion algorithm terminates.
- If ***N*** is an interior node, find ***K***'s in-order successor and swap it with ***K***. As a result, deletion always begins at a leaf node ***L***.
- If leaf node ***L*** contains a value in addition to ***K***, delete ***K*** from ***L***, and we're done. (no underflow)
  - For B-trees, underflow occurs if # of nodes < minimum.
- If underflow occurs (node has less than required # of keys), we merge it with its neighboring nodes.
  - Check siblings of leaf. If sibling has two values, redistribute them.
  - Otherwise, merge L with an adjacent sibling and bring down a value from L's parent.
  - If L's parent has underflow, recursively apply merge procedure.
  - If underflow occurs to the root, the tree may shrink a level.

Deletion
Re-distributing values in Leaf Nodes

If deleting K from L causes L to be empty:

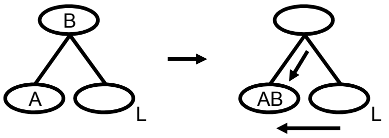
- Check siblings of now empty leaf.
- If sibling has two values, redistribute the values.



Deletion
Merging Leaf Nodes

Merging leaf nodes:

- If no sibling node has extra keys to spare, merge L with an adjacent sibling and bring down a value from L's parent.

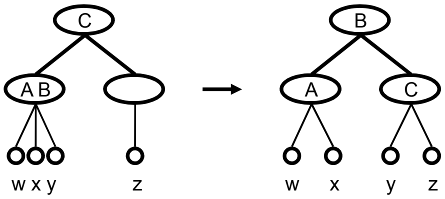


- The merging of L may cause the parent to be left without a value and only one child. If so, recursively apply deletion procedure to the parent.

Deletion
Re-distributing values in Interior Nodes

Re-distributing values in interior nodes:

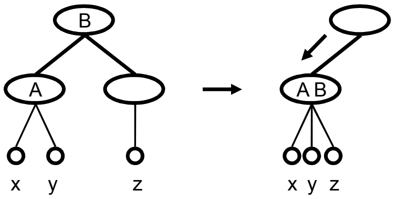
- If the node has a sibling with two values, redistribute the values.



Deletion
Merging Interior Nodes

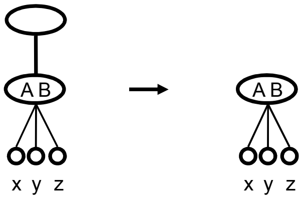
Merging interior nodes:

- ◆If the node has no sibling with two values, merge the node with a sibling, and let the sibling adopt the node's child.

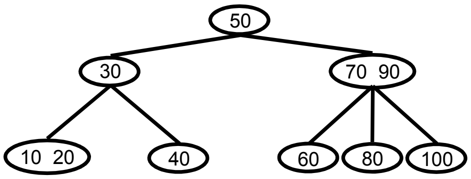


Deletion
Merging on the Root Node

If the merging continues so that the root of the tree is without a value (and has only one child), delete the root. Height will now be h-1.

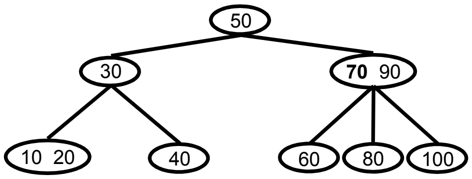


Deletion Examples



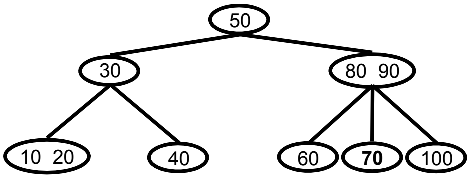
Original tree

Deletion Examples



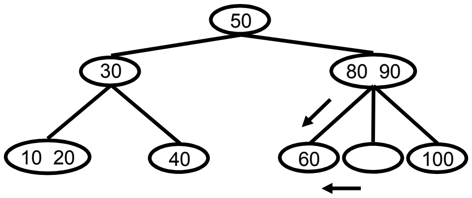
Delete 70

Deletion Examples



Swap with in-order successor

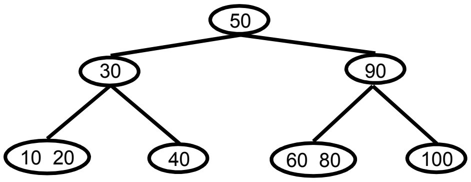
Deletion Examples



Merge and pull down

COSC 404 - Dr. Ramon Lawrence

Deletion Examples

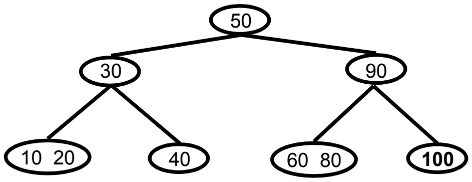


Done!

Page 61

COSC 404 - Dr. Ramon Lawrence

Deletion Examples

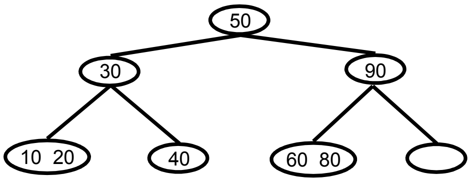


Delete 100

Page 62

COSC 404 - Dr. Ramon Lawrence

Deletion Examples

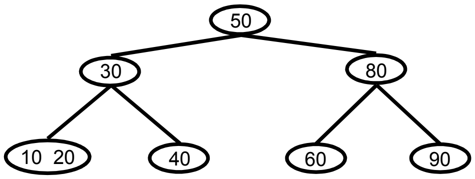


Redistribute

Page 63

COSC 404 - Dr. Ramon Lawrence

Deletion Examples

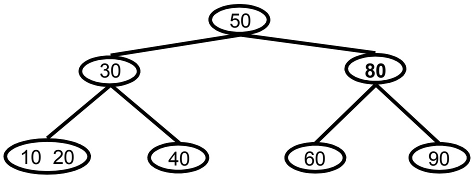


Done!

Page 64

COSC 404 - Dr. Ramon Lawrence

Deletion Examples

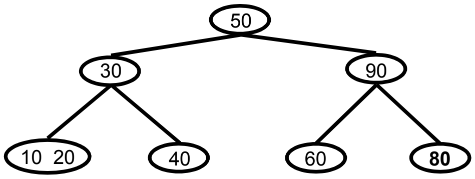


Delete 80

Page 65

COSC 404 - Dr. Ramon Lawrence

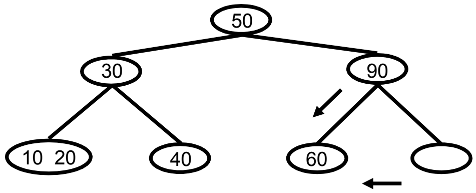
Deletion Examples



Swap with in-order successor

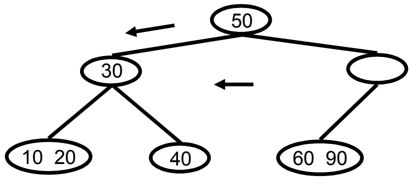
Page 66

Deletion Examples



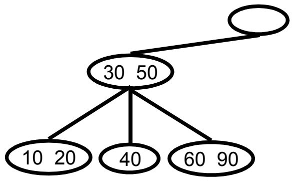
Merge and pull down

Deletion Examples



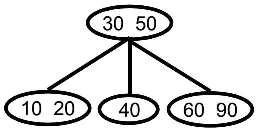
Merge and pull down

Deletion Examples



Merge and pull down

Deletion Examples



Done

B-tree Deletion Practice Question

Using the previous tree constructed by inserting into a B-tree of ***order 1 (max. keys=2)*** the keys:

- 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150

Delete these keys (in order):

- 40
- 70
- 80

B-trees as External Data Structures

Now that we understand how a B-tree works as a data structure, we will investigate how it can be used for an index.

A regular B-tree can be used as an index by:

- Each node in the B-tree stores not only keys, but also a record pointer for each key to the actual data being stored.
  - Could also potentially store the record in the B-tree node itself.
- To find the data you want, search the B-tree using the key, and then use the pointer to retrieve the data.
  - No additional disk access is required if the record is stored in the node.

Given this description, it is natural to wonder how we might calculate the best B-tree ***order***.

- Depends on disk block and record size.
- We want a node to occupy an entire block.

Calculating the Size of a B-tree Node

Given a block of 4096 bytes, calculate the order of a B-tree if the key size is 4 bytes, the pointer to the data record is 8 bytes, and the child pointers are 8 bytes.

Answer:

Assuming no header information is kept in blocks:

node size = keySize*numKeys + dataPtrSize*numKeys
+ childPtrSize*(numKeys+1)

Let k=numKeys.

size of one node = 4*k + 8*k + 8*(k+1) <= 4096

k = 204 keys

Maximum order is 102.

B-tree Question

Question: Given a block of 4096 bytes, calculate the maximum number of keys in a node if the key size is 4 bytes, internal B-tree pointers are 8 bytes, and we store the record itself in the B-tree node instead of a pointer. The record size is 100 bytes.

A) 18

B) 36

C) 340

D) 680

Advantages of B-trees

The advantages of a B-tree are:

- 1) B-trees automatically create or destroy index levels as the data file changes.
- 2) B-trees automatically manage record allocation to blocks, so no overflow blocks are needed.
- 3) A B-tree is always balanced, so the search time is the same for any search key and is logarithmic.

For these reasons, B-trees and B+-trees are the index scheme of choice for commercial databases.

B+-trees

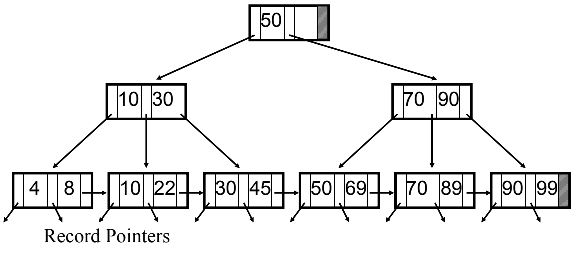
A ***B+-tree*** is a multi-level index structure like a B-tree except that *all* data is stored at the leaf nodes of the resulting tree instead of within the tree itself.

- Each leaf node contains a pointer to the next leaf node which makes it easy to chain together and maintain the data records in “sequential” order for sequential processing.

Thus, a B+-tree has two distinct node types:

- 1) ***interior nodes*** - store pointers to other interior nodes or leaf nodes.
- 2) ***leaf nodes*** - store keys and pointers to the data records (or the data records themselves).

B+-tree Example



Operations on B+-trees

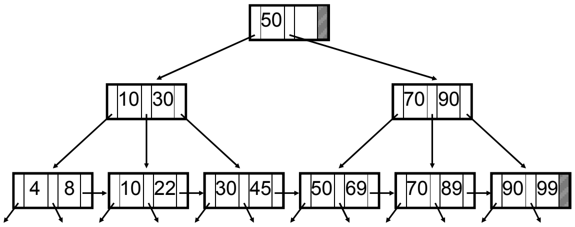
The general algorithms for inserting and deleting from a B+-tree are similar to B-trees except for one important difference:

All key values stay in leaves.

When we must merge nodes for deletion or add nodes during splitting, the key values removed/promoted to the parent nodes from leaves are **copies**.

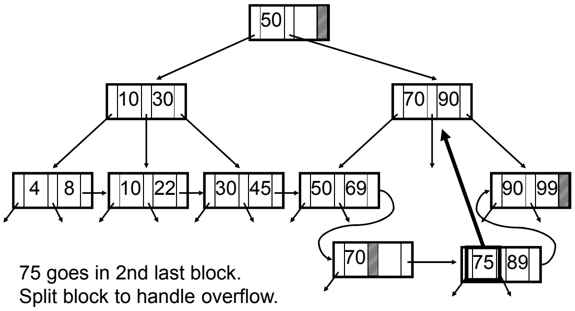
- All non-leaf levels do not store actual data, they are simply a hierarchy of multi-level index to the data.

B+-tree Insert Example



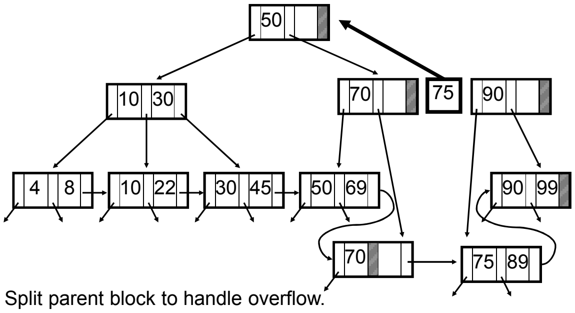
Insert 75

B+-tree Insert Example (2)



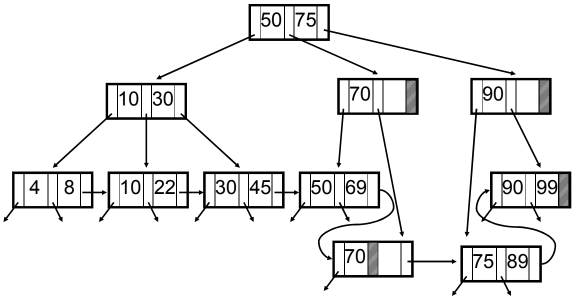
75 goes in 2nd last block.
Split block to handle overflow.
Promote 75. Note that 75 stays in a leaf!

B+-tree Insert Example (3)



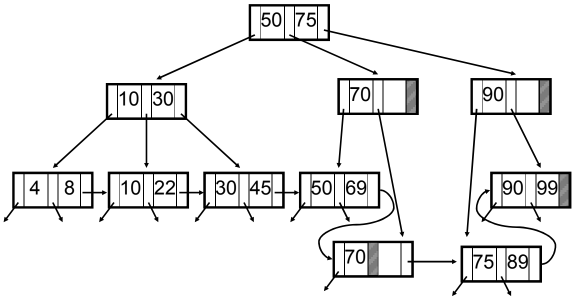
Split parent block to handle overflow.
Promote 75. Note that 75 does not stay!

B+-tree Insert Example (4)



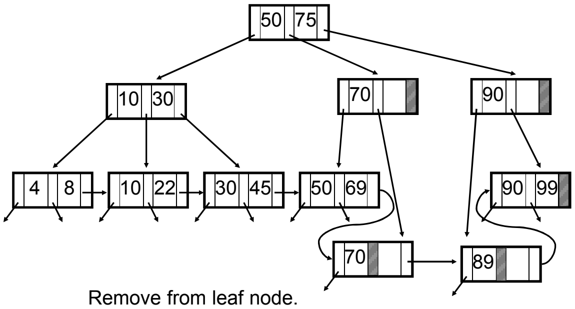
Insertion done!

B+-tree Delete Example



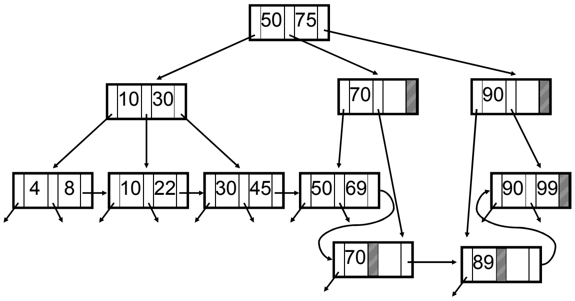
Delete 75.

B+-tree Delete Example (2)



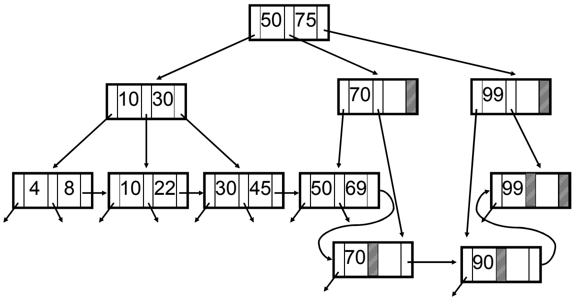
Remove from leaf node.
No other updates.

B+-Tree Delete Example 2



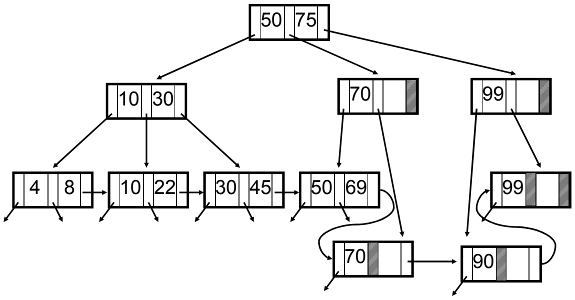
Delete 89.

B+-Tree Delete Example 2 (2)



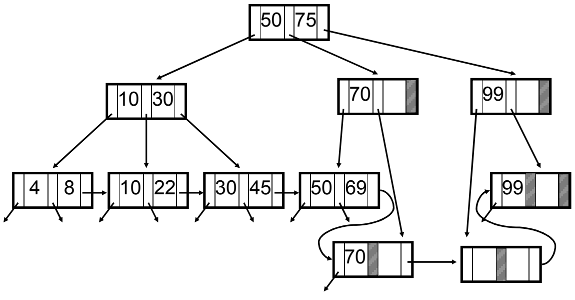
Redistribute keys 90 and 99.

B+-Tree Delete Example 3



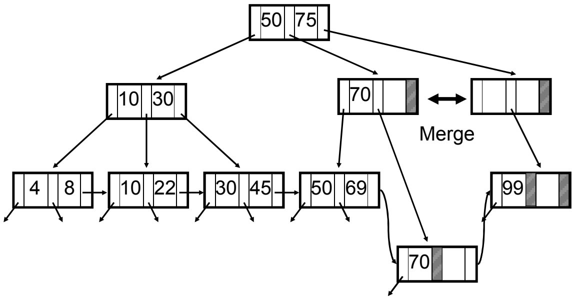
Delete 90.

B+-Tree Delete Example 3 (2)



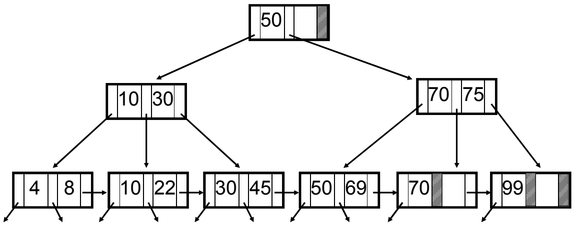
Empty leaf node. Merge with sibling.

B+-Tree Delete Example 3 (2)



Empty interior node. Merge with sibling.

B+-Tree Delete Example 3 (3)



Bring down 75 from parent node. Done.

B+-tree Practice Question

For a B+-tree of ***order 2 (max. keys=4)***, insert the following keys in order:

- 10, 20, 30, 40, 50, 60, 70, 80, 90
- Assuming keys increasing by 10, what is the first key added that causes the B+-tree to grow to height 3?
  - a) 110 b) 120 c) 130 d) 140 e) 150

Show the tree after deleting the following keys:

- a) 70
- b) 90
- c) 10
- Assume you start with the tree after inserting 90 above.

B+-tree Challenge Exercise

For a B+-tree with ***maximum keys=3***, insert the following keys in order:

- 10, 20, 30, 40, 50, 60, 70, 80, 90,100

Show the tree after deleting the following keys:

- a) 70
- b) 90
- c) 10

Try the deletes when the minimum # of keys is 1 and when the minimum # of keys is 2.

Observations about B+-trees

Since the inter-node connections are done by pointers, there is no assumption that in the B+-tree, the “logically” close blocks are “physically” close.

The B+-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches and modifications can be conducted efficiently.

Example:

- If a B+-tree node can store 300 key-pointer pairs at maximum, and on average is 69% full, then 208 (207+1) pointers/block.
- Level 3 B+-tree can index 208^3 records = 8,998,912 records!

B+-trees Discussion

By isolating the data records in the leaves, we also introduce additional implementation complexity because the leaf and interior nodes have different structures.

- Interior nodes contain only pointers to additional index nodes or leaf nodes while leaf nodes contain pointers to data records.

This additional complexity is outweighed by the advantages of B+-trees which include:

- Better sequential access ability.
- Greater overall storage capacity for a given block size since the interior nodes can hold more pointers each of which requires less space.
- Uniform data access times.

B-trees Summary

A ***B-tree*** is a search tree where each node has >= ***n*** data values and <= ***2n***, where we chose ***n*** for our particular tree.

- A 2-3 tree is a special case of a B-tree.
- Common operations: search, insert, delete
  - Insertion may cause node overflow that is handled by promotion.
  - Deletion may cause node underflow that is handled by mergers.
- Handling special cases for insertion and deletion make the code for implementing B-trees complex.
- Note difference between B+-tree and B-tree for insert/delete!

B+-trees are a good index structure because they can be searched/updated in logarithmic time, manage record pointer allocation on blocks, and support sequential access.

Major Objectives

The "One Things":

- Insert and delete from a B-tree and a B+-tree.
- Calculate the maximum order of a B-tree.

Major Theme:

- B-trees are the standard index method due to their time/space efficiency and logarithmic time for insertions/deletions.

Other objectives:

- Calculate query access times using B-trees indexes.
- Compare/contrast B-trees and B+-trees.

COSC 404
Database System Implementation

R-trees

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

R-Trees Introduction

R-trees (or region tree) is a generalized B-tree suitable for processing spatial queries. Unlike B-trees where the keys have only one dimension, R-trees can handle multidimensional data.

The basic R-tree was proposed by Guttman in 1984 and extensions and modifications have been later developed.

- ◆R+-tree (Sellis *et al.* 1987)
- ◆R*-tree (Beckmann *et al.* 1990)

We begin by looking at the properties of spatial data and spatial query processing.

Types of Spatial Data

Spatial data includes multidimensional points, lines, rectangles, and other geometric objects.

A spatial data object occupies a region of space, called its spatial extent, which is defined by its location and boundary.

Point Data - points in multidimensional space

Region Data - objects occupy a region (spatial extent) with a location and a boundary.

Types of Spatial Queries

Spatial Range Queries - query has associated region and asks to find matches within that region

- ◆e.g. Find all cities within 50 miles of Kelowna.
- ◆Answer to query may include *overlapping* or *contained* regions.

Nearest Neighbor Queries - find closest region to a region.

- ◆e.g. Find the 5 closest cities to Kelowna.
- ◆Results are ordered by proximity (distance from given region).

Spatial Join Queries - join two types of regions

- ◆e.g. Find all cities near a lake.
- ◆Expensive to compute as join condition involves regions and proximity.

Spatial Data Applications

Geographic Information Systems (GIS) use spatial data for modeling cities, roads, buildings, and terrain.

Computer-aided design and manufacturing (CAD/CAM) process spatial objects when designing systems.

- ◆Spatial constraints: "There must be at least 6 inches between the light switch and turn signal."

Multimedia databases storing images, text, and video require spatial data management to answer queries like "Return the images similar to this one." Involves use of feature vectors.

- ◆Similarity query converted into nearest neighbor query.

Spatial Queries

Question: What type of spatial query is: "Find the city with the largest population closest to Chicago?"

A) Spatial Range Query

B) Nearest Neighbor Query

C) Spatial Join Query

D) Not a spatial query

Spatial Indexing

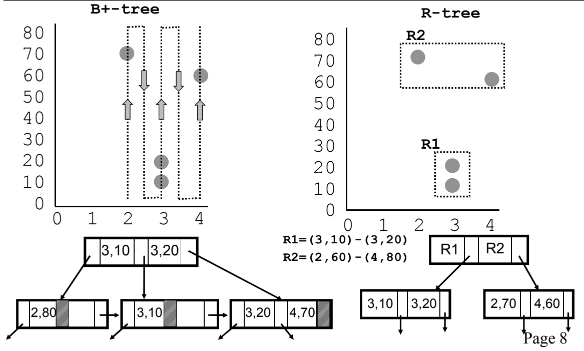
A multidimensional or spatial index utilizes some kind of spatial relationship to organize data entries. Each key value in the index is a point (or region) in *k*-dimensional space, where *k* is the number of fields in the search key.

Although multidimensions (multiple key fields) can be handled in a B+-tree, this is accomplished by imposing a total ordering on the data as B+-trees are single-dimensional indexes.

For instance, B+-tree index on <*x*,*y*> would sort the points by *x* then by *y*.

- ◆I.e. <2,70>, <3,10>, <3,20>, <4,60>

B+-tree versus R-tree



B+-tree versus R-tree Querying

Consider these three queries on *x* and *y*:

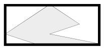
- ◆1) Return all points with *x* < 3.
  - ⇨Works well on B+-tree and R-tree. Most efficient on B+-tree.
- ◆2) Return all points with *y* < 50.
  - ⇨Cannot be efficiently processed with B+-tree as data sorted on *x* first.
  - ⇨Can be efficiently processed on R+-tree.
- ◆3) Return all points with *x* < 3 and *y* < 50.
  - ⇨B+-tree is only useful for selection on *x*. Not very good if many points satisfy this criteria.
  - ⇨Efficient for R-tree as only search regions that may contain points that satisfy both criteria.

R-Tree Structure

R-tree is adaptation of B+-tree to handle spatial data.

The search key for an R tree is a collection of intervals with one interval per dimension. Search keys are referred to as ***bounding boxes*** or ***minimum bounding rectangles*** (***MBR***s).

- ◆Example:



Each entry in a node consists of a pair <*n-dimensional box*, *id*> where the *id* identifies the object and the *box* is its MBR.

Data entries are stored in leaf nodes and non-leaf nodes contain entries consisting of <*n-dimensional box*, *node pointer*>.

The box at a non-leaf node is the smallest box that contains all the boxes associated with the child nodes.

R-Tree Notes

The bounding box for two children of a given node can overlap.

- ◆Thus, more than one leaf node could potentially store a given data region.

A data point (region) is only stored in one leaf node.

R-Tree Searching

Start at the root.

- ◆If current node is non-leaf, for each entry <*E*, *ptr*> if box *E* overlaps *Q*, search subtree identified by *ptr*.
- ◆If current node is leaf, for each entry <*E*, *id*>, if *E* overlaps *Q*, *id* identifies an object that might overlap *Q*.

Note that you may have to search several subtrees at each node. In comparison, a B-tree equality search goes to just one leaf.

R-Tree Searching Improvements

Although it is more convenient to store boxes to represent regions because they can be represented compactly, it is possible to get more precise bounding regions by using convex polygons.

Although testing overlap is more complicated and slower, this is done in main-memory so it can be done quite efficiently. This often leads to an improvement.

R-Tree Insertion Algorithm

Start at root and go down to "best-fit" leaf *L*.

- ◆Go to child whose box needs ***least enlargement*** to cover *B*; resolve ties by going to smallest area child.

If best-fit leaf *L* has space, insert entry and stop.

Otherwise, split *L* into *L1* and *L2*.

- ◆Adjust entry for *L* in its parent so that the box now covers (only) *L1*.
- ◆Add an entry (in the parent node of *L*) for *L2*. (This could cause the parent node to recursively split.)

R-Tree Insertion Algorithm Splitting a Node

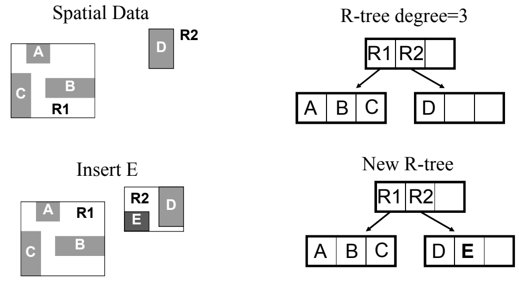
The existing entries in node *L* plus the newly inserted entry must be distributed between *L1* and *L2*.

Goal is to reduce likelihood of both *L1* and *L2* being searched on subsequent queries.

Idea: Redistribute so as to minimize area of *L1* plus area of *L2*.

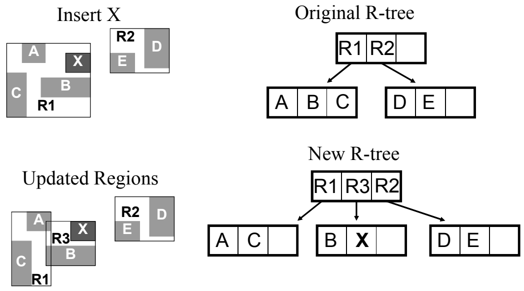
An exhaustive search of possibilities is too slow so quadratic and linear heuristics are used.

Insertion Example



Extended region R2 to hold E.

Insertion Example 2



Split R1 into R1 and R3.

R+-Tree

R+-tree avoids overlap by inserting an object into multiple leaves if necessary.

Reduces search cost as now take a single path to leaf.

R*-Tree

R*-tree uses the concept of forced reinserts to reduce overlap in tree nodes.

When a node overflows, instead of splitting:

- Remove some (say 30%) of the entries and reinsert them into the tree.
- Could result in all reinserted entries fitting on some existing pages, avoiding a split.

R*-trees also use a different heuristic, minimizing box parameters, rather than box areas during insertion.

GiST

The Generalized Search Tree (GiST) abstracts the "tree" nature of a class of index including B+-trees and R-tree variants.

- Striking similarities in insert/delete/search and even concurrency control algorithms make it possible to provide "templates" for these algorithms that can be customized to obtain the many different tree index structures.
- B+ trees are so important (and simple enough to allow further specialization) that they are implemented specifically in all DBMSs.
- GiST provides an alternative for implementing other index types.
- Implemented in PostgreSQL. Make your own index!

R-Tree Variants

Question: Select a true statement.

A) Searching in a R-tree always follows a single path.

B) R-tree variants may have different ways for splitting nodes during insertion.

C) A R+-tree search always follows a single path to a leaf node.

D) None of the above

R-Trees Summary

An R-tree is useful for indexing and searching spatial data.

Variants of R-trees are used in commercial databases.

Major Objectives

The "One Thing":

- Be able to explain the difference between an R-tree and a B+-tree.

Other objectives:

- List some types of spatial data.
- List some types of spatial queries.
- List some applications of spatial data and queries.
- Understand the idea of insertion in a R-tree.

COSC 404
Database System Implementation

Hash Indexes

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Hash Indexes Overview

B-trees reduce the number of block accesses to 3 or 4 even for extremely large data sets. The goal of hash indexes is to make all operations require **only 1 block access.**

Hashing is a technique for mapping key values to locations.

Hashing requires the definition of a hash function f(*x*), that takes the key value *x* and computes *y*=f(*x*) which is the location of where the key should be stored.

A ***collision*** occurs when we attempt to store two different keys in the same location.

- $f(x_1) = y$ and $f(x_2) = y$ for two keys $x_1 != x_2$

Handling Collisions

A ***perfect hash function*** is a function that:

- For any two key values x_1 and x_2, $f(x_1) != f(x_2)$ for all x_1 and x_2, where $x_1 != x_{2.}$
- That is, no two keys map to the same location.
- It is not always possible to find a perfect hash function for a set of keys depending on the situation.
  - ⇨ Recent research on perfect hash functions is useful for databases.

We must determine how to handle collisions where two different keys map to the same location.

One simple way of handling collisions is to make the hash table really large to minimize the probability of collisions.

- This is not practical in general. However, we do want to make our hash table moderately larger than the # of keys.

Open Addressing

Open addressing with linear probing is a method for handling hash collisions.

Open addressing:

- Computes *y=f(x)* and attempts to put key in location *y*.
- If location *y* is occupied, scan the array to find the next open location. Treat the array as circular.

Open Addressing Example

Open addressing on a 11 element array with f(x) = x % 11:

				917						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 917 at location 4.

Open Addressing Example (2)

Open addressing on a 11 element array with f(x) = x % 11:

	254			917						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 254 at location 1.

Open Addressing Example (3)

Open addressing on a 11 element array with f(x) = x % 11:

	254			917		589				
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 589 at location 6.

Open Addressing Example (4)

Open addressing on a 11 element array with f(x) = x % 11:

	254			917		589	457			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 457 at location 6.

- Collision with 589.
- Next open location is 7, so insert there.

Open Addressing Example (5)

Open addressing on a 11 element array with f(x) = x % 11:

	254			917	136	589	457			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 136 at location 4.

- Collision with 917.
- Next open location is 5, so insert there.

Open Addressing Example (6)

Open addressing on a 11 element array with f(x) = x % 11:

	254			917	136	589	457	654		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 654 at location 5.

- Collision with 136.
- Note that a collision occurs with a key that did not even originally hash to location 5.
- Keep going down array until find location to insert which is 8.

Open Addressing Example (7)

Open addressing on a 11 element array with f(x) = x % 11:

	254			917	136	589	457	654	306	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Insert 306 at location 9.

Open Addressing Example Summary

Insert

917	1 probe(s)
589	1
254	1
457	2
136	2
654	4
306	1

Average number of probes = 12 / 7 = 1.7

Open Addressing Insert and Delete

Insert using linear probing creates the potential that a key may be inserted many locations away from its original hash location.

What happens if an element is then deleted in between its proper insert location and the location where it was put?

- ◆How does this affect insert and delete?

Example: Delete 589 (f(589)=6), then search for 654 (f(654)=5).

	254			917	136	??	457	654	306	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

- ◆Problem! Search would normally terminate at empty location 6!

Solution: Have special constants to mark when a location is empty and never used OR empty and was used.

Open Address Hashing

Question: What location is **19** inserted at?

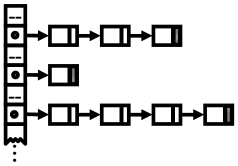
	12			5			18	8	(del.)	21
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

A) 8
B) 9
C) 6
D) 0

Separate Chaining

Separate chaining resolves collisions by having each array location point to a linked list of elements.

- ◆Algorithms for operations such as insert, delete, and search are obvious and straightforward.
- ◆As with open addressing, separate chaining has the potential to degenerate into a linear algorithm if the hash function does not distribute the keys evenly in the array.



Hash Limitations and Analysis

Hashing gives very good performance when the hash function is good and the number of conflicts is low.

- ◆If the # of conflicts is high, then the performance of hashing rapidly degrades. The worse case is O(n).
- ◆Collisions can be reduced by minimizing the: ***load factor*** = # of occupied locations/size of hash table.

However, on average, inserts, searches, and deletes are O(1)!

The limitations of hashing are:

- ◆Ordered traversals are difficult without an additional structure and a sort. (hashing randomizes locations of records)
- ◆Partial key searches are difficult as the hash function must use the entire key.

Hash Limitations and Analysis (2)

The ***hash field space*** is the set of all possible hash field values for records.

- ◆i.e. It is the set of all possible key values that we may use.

The ***hash address space*** is the set of all record slots (or storage locations).

- ◆i.e. Size of array in memory or physical locations in a file.

Tradeoff:

- ◆The larger the address space relative to the hash field space, the easier it is to avoid collisions, BUT
- ◆the larger the address space relative to the number of records stored, the worse the storage utilization.

Hashing Questions How to handle real data?

Determine your own hash function for each of the following set of keys. Assume the hash table size is 100,000.

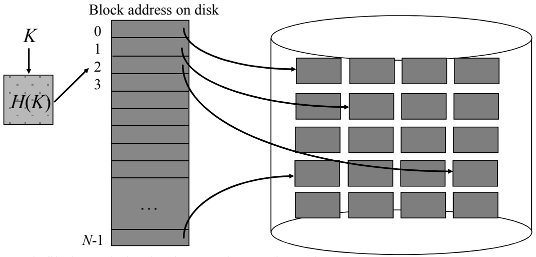
1) The keys are part numbers in the range 9,000,000 to 9,099,999.

2) The keys are people's names.

- ◆E.g. "Joe Smith", "Tiffany Connor", etc.

External Hashing Overview

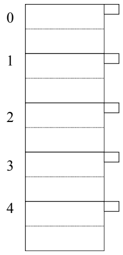
External hashing algorithms allocate records with keys to blocks on disk rather than locations in a memory array.



Hash file has relative bucket numbers 0 through N-1.
Map ***logical*** bucket numbers to ***physical*** disk block addresses.
Disk blocks are buckets that hold several data records each.

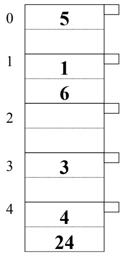
External Hashing Example

<u>External Hash Table</u>
- 5 buckets
- 2 records per bucket
- use overflow blocks
- f(x) = x % 5



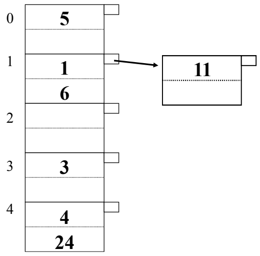
External Hashing Example Insertion

<u>Insert:</u>
1,5,3,6,4,24



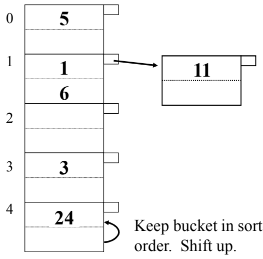
External Hashing Example Insertion with Overflow

<u>Insert:</u> 11



External Hashing Example Deletion

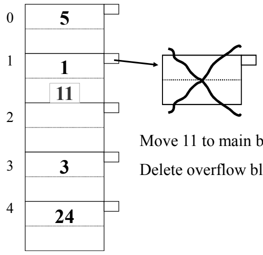
<u>Delete:</u> 4



Keep bucket in sorted order. Shift up.

External Hashing Example Deletion with Overflow

<u>Delete:</u> 6



Move 11 to main bucket.

Delete overflow block.

Deficiencies of Static Hashing

In static hashing, the hash function maps keys to a fixed set of bucket addresses. However, databases grow with time.

- If initial number of buckets is too small, performance will degrade due to too many overflows.
- If file size is made larger to accommodate future needs, a significant amount of space will be wasted initially.
- If database shrinks, again space will be wasted.
- One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- ***Bottom line:*** Must determine optimal utilization of hash table.
  - ⇨ Try to keep utilization between 50% and 80%. Hard when data changes.

These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Linear Hashing

Linear hashing allows a hash file to expand and shrink dynamically.

A linear hash table starts with $\boldsymbol{2^d}$ buckets where ***d*** is the # of bits used from the hash value to determine bucket membership.

- Take the ***last d*** bits of ***H*** where ***d*** is the current # of bits used.

The growth of the hash table can either be triggered:

- 1) Every time there is a bucket overflow.
- 2) When the load factor of the hash table reaches a given point.

We will use the load factor method.

- Since bucket overflows may not always trigger hash table growth, overflow blocks are used.

Linear Hashing Load Factor

The ***load factor lf*** of the hash table is the number of records stored divided by the number of possible storage locations.

- The initial number of blocks ***n*** is a power of 2.
  - ⇨ As the table grows, it may not always be a power of 2.
- The number of storage locations ***s*** = #blocks X #records/block.
- The initial number of records in the table ***r*** is 0 and is increased as records are added.
- Load factor = ***r / s*** = ***r / n*** * #records/block

We will expand the hash table when the load factor > 85%.

Linear Hashing Load Factor

Question: A linear hash table has 5 blocks each with space for 4 records. There are currently 2 records in the hash table. What is its load factor?

A) 10%

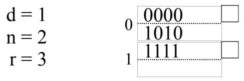
B) 40%

C) 50%

D) 0%

Linear Hashing Example

- Example:
  - ⇨ Assume each hashed key is a sequence of four binary digits.
  - ⇨ Store values 0000, 1010, 1111.



d = 1
n = 2
r = 3

0: 0000, 1010

1: 1111

Linear Hashing Insertions

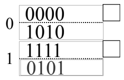
Insertion algorithm:

- Insert a record with key ***K*** by first computing its hash value ***H***.
- Take the ***last d*** bits of ***H*** where ***d*** is the current # of bits used.
- Find the bucket ***m*** where ***K*** would belong using the ***d*** bits.
- If ***m < n***, then bucket exists. Go to that bucket.
  - ⇨ If the bucket has space, then insert ***K***. Otherwise, use an overflow block.
- If ***m >= n***, then put ***K*** in bucket $\boldsymbol{m - 2^{d-1}}$.
- After each insert, check to see if the load factor ***lf*** < threshold.
- If ***lf >=*** threshold perform a split:
  - ⇨ Add new bucket ***n***. (Adding bucket ***n*** may increase the directory size ***d***.)
  - ⇨ Divide the records between the new bucket n=$\boldsymbol{1b_2 \ldots b_d}$ and bucket $\boldsymbol{0b_2..b_d}$.
  - ⇨ **Note that the bucket split may not be the bucket where the record was added!** Update ***n*** and ***d*** to reflect the new bucket.

Linear Hashing Insertion Example

Insert 0101.

d = 1
n = 2
r = 3

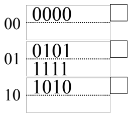


4/4 = 100% full.
Above threshold triggers split.

Linear Hashing Insertion Example (2)

Added new bucket 10. (2 in binary - old n!)
Divide records of bucket 00 and 10.

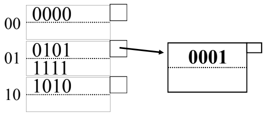
d = 2
n = 3
r = **4**



Linear Hashing Insertion Example (3)

Insert 0001.
Use overflow block.
(May sort records later.)

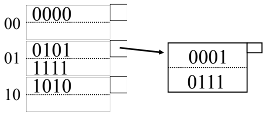
d = 2
n = 3
r = **5**



Linear Hashing Insertion Example (4)

Insert 0111.

d = 2
n = 3
r = **6**

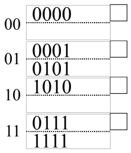


6/6 = 100% full.
Above threshold triggers split.

Linear Hashing Insertion Example (5)

Create bucket 11.
Split records between 01 and 11.

d = 2
n = 4
r = 6



Linear Hashing Question

1) Show the resulting hash directory when hashing the keys: 0, 15, 8, 4, 7, 12, 10, 11 using linear hashing.

- Assume a bucket can hold two records (keys).
- Assume 4 bits of hash key.
- Add a new bucket when utilization is >= 85%.

Clicker: What bucket is 11 in?

A) 000
B) 001
C) 1011
D) 011

B+-trees versus Linear Hashing

B+-trees versus linear hashing: which one is better?

Factors:

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?

Expected type of queries:

- Hashing is generally better at retrieving records having a specified value for the key.
- If range queries are common, B+-trees are preferred.

Real-world result: PostgreSQL implements both B+-trees and linear hashing. Currently, linear hashing is not recommended for use.

Hash Indexes Summary

Hashing is a technique for mapping key values to locations.

- With a good hash function and collision resolution, insert, delete and search operations are O(1).
- Ordered scans and partial key searches however are inefficient.
- Collision resolution mechanisms include:
  - open addressing with linear probing - linear scan for open location.
  - separate chaining - create linked list to hold values and handle collisions at an array location.

Dynamic hashing is required for databases to handle updates.

Linear hashing performs dynamic hashing and grows the hash table one bucket at a time.

Major Objectives

The "One Things":

- Perform open address hashing with linear probing.
- Perform linear hashing.

Major Theme:

- Hash indexes improve average access time but are not suitable for ordered or range searches.

Other objectives:

- Define: hashing, collision, perfect hash function
- Calculate load factor of a hash table.
- Compare/contrast external hashing and main memory hashing.
- Compare/contrast B+-trees and linear hashing.

COSC 404
Database System Implementation

SQL Indexing

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Creating Indexes in SQL

There are two general ways of creating an index:

- 1) By specifying it in your `CREATE TABLE` statement:

```
CREATE TABLE test
(  a int,
   b int,
   c varchar(10)
   PRIMARY KEY (a),
   UNIQUE (b),
   INDEX (c)
);
```

Only one primary key index allowed.

`UNIQUE` index does not allow duplicate keys.

Creates an index that supports duplicates.

- 2) Using a `CREATE INDEX` command after a table is created:

```
CREATE INDEX myIdxName ON test (a,b);
```

`CREATE INDEX` Command

`CREATE INDEX` syntax:

```
CREATE [UNIQUE] INDEX indexName
   ON  tableName (colName [ASC|DESC] [,...])

DROP INDEX indexName;
```

- `UNIQUE` means that each value in the index is unique.
- `ASC`/`DESC` specifies the sorted order of index.
- Note: The syntax varies slightly between systems.

`CREATE INDEX` Command Examples

Examples:

`CREATE UNIQUE INDEX idxStudent ON Student(sid)`

- Creates an index on the field `sid` in the table `Student`
- `idxStudent` is the name of the index.
- The `UNIQUE` keyword ensures the uniqueness of `sid` values in the table (and index).
  - ⇨Uniqueness is enforced even when adding an index to a table with existing data. If the `sid` field is non-unique then the index creation fails.

`CREATE INDEX clMajor ON Student(Major) CLUSTER`

- Creates a clustered (primary) index on the `Major` field of `Student` table.
- Note: Clustered index may or may not be on a key field.

`CREATE INDEX` Command Examples (2)

`CREATE INDEX idxMajorYear ON student(Major,Year)`

- Creates an index with two fields.
- Duplicate search keys are possible.

Creating Indexes in MySQL

MySQL supports both ways of creating indexes. The `CREATE INDEX` command is mapped to an `ALTER TABLE` statement.

Syntax for `CREATE TABLE`:

```
CREATE TABLE tbl_Name
(
    [CONSTRAINT [name]] PRIMARY KEY [index_type] (index_col,...)
  | KEY [index_name] [index_type] (index_col,...)
  | INDEX [index_name] [index_type] (index_col,...)
  | [CONSTRAINT [symbol]] UNIQUE [INDEX]
        [index_name] [index_type] (index_col,...)
  | [FULLTEXT|SPATIAL] [INDEX] [index_name] (index_col,...)
  | [CONSTRAINT [symbol]] FOREIGN KEY
        [index_name] (index_col_name,...)
    ...
)
```

Creating Indexes in MySQL (2)

Notes:

- ◆1) By specifying a primary key, an index is automatically created by MySQL. You do not have to create another one!
- ◆2) The primary key index (and any other type of index) can have more than one attribute.
- ◆3) MySQL assigns default names to indexes if you do not provide them.
- ◆4) MySQL supports B+-tree, Hash, and R-tree indexes but support depends on table type.
- ◆5) Can index only the first few characters of a `CHAR`/`VARCHAR` field by using col_name(length) syntax. (smaller index size)
- ◆6) FULLTEXT indexes allow more powerful natural language searching on text fields (but have a performance penalty).
- ◆7) SPATIAL indexes can index spatial data.

Creating Indexes in SQL Server

Microsoft SQL Server supports defining indexes in the `CREATE TABLE` statement or using a `CREATE INDEX` command.

Notes:

- ◆1) The primary index is a cluster index (rows sorted and stored by indexed column). Unique indexes are non-clustered.
  - ⇨A clustered (primary) index stores the records in the index.
  - ⇨A secondary index stores pointers to the records in the index.
  - ⇨Clustered indexes use B+-trees.
- ◆2) A primary key constraint auto-creates a clustered index.
- ◆2) Also supports full-text and spatial indexing.

Performance Improvement of Indexes

Indexes can improve query performance, especially when indexing foreign keys ***and*** for queries with ***low selectivity***.

Experiment:

- ⇨Use TPC-H database and perform join between `Orders` and `Customer` where the `o_custkey` field in `Orders` table is and is not indexed.
- ⇨select * from orders o, customers c where o.o_custkey = c.c_custkey
  - Result size = 1,500,000 rows in time 40 seconds
- ⇨add condition: `where o_custkey = 10`
  - # of rows = 20, without index = 7 seconds ; with index = less than a second
- ⇨add condition: `where o_custkey < 100`
  - # of rows = 979; without index = 7 seconds; with index = less than a second
- ⇨add condition: `where o_custkey < 1000`
  - What do you think will be faster a) with or b) without an index?

Bottom line: Indexes improve performance but only for queries that have low selectivity (get return rows from index).

Indexing with Multiple Fields

Consider an index with multiple fields:

```
CREATE INDEX idxMajorYear ON student(Major,Year)
```

and a query that could use this index:

```
SELECT * FROM student WHERE Major="CS" and Year="3"
```

Commercial databases use a B+-tree index. Note order is important as the index is sorted on the attributes in order.

There are also other methods for multiple field indexing:

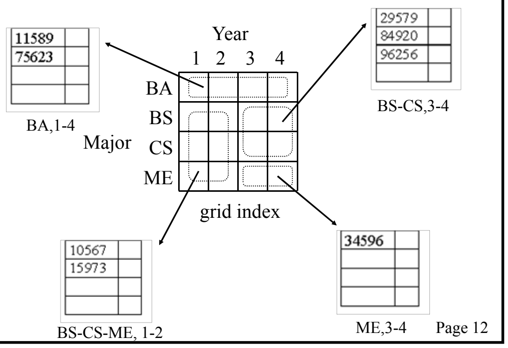
- ◆Partitioned Hashing
- ◆Grid Files

Multiple Key Indexing
Grid Files

A ***grid file*** is designed for multiple search-key queries.

- ◆The grid file has a grid array and a linear scale for each search-key attribute.
- ◆The grid array has a number of dimensions equal to number of search-key attributes.
- ◆Each cell of the grid points to a disk bucket. Multiple cells of the grid array can point to the same bucket.
- ◆To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer.
- ◆If a bucket becomes full, a new bucket can be created if more than one cell points to it. If only one cell points to it, an overflow bucket needs to be created.

Example Grid File for Student Database



Grid Files Querying

A grid file on two attributes A and B can answer queries:

- Exact match queries:
  - ⇨A=value
  - ⇨B=value
  - ⇨A=value AND B=value
- Range queries:
  - ⇨$(a_1 \le A \le a_2)$
  - ⇨$(b_1 \le B \le b_2)$
  - ⇨$(a_1 \le A \le a_2 \wedge b_1 \le B \le b_2)$
- For example, to answer $(a_1 \le A \le a_2 \wedge b_1 \le B \le b_2)$, use linear scales to find candidate grid array cells, and look up all the buckets pointed to from those cells.

Linear scales must be chosen to uniformly distribute records across cells. Otherwise there will be many overflow buckets.

Grid Files Discussion

Using grid cells as bucket pointers allows the grid to be regular, but increases the indirection.

Note that the linear scales are often allocated in a table where each value maps to a number between 0 and ***N***.

This allows easier indexing of the grid, and also permits the linear scales to be ranges. Example:

Salary Linear Scale

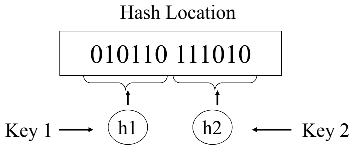
$0-$10,000	0
$10,001-$50,000	1
$50,001-$100,000	2
$100,000+	3

Overall: Grid files are good for multi-key searches but require space overhead and ranges that evenly split keys.

Multiple Key Indexing Partitioned Hashing

The idea behind ***partitioned hashing*** is that the overall hash location is a combination of the hash values from each key.

For example,



The overall hash location *L* is 12 bits long.
The first 6 bits are from *h1*, the second 6 from *h2*.

Partitioned Hashing Example

Hash Table

h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Insert <10567,CS,3>, <11589,BA,2>, <15973,CS,3>, <29579,BS,1>,<34596,ME,4>, <75623,BA,3>, <84920,CS,4>, <96256,ME,2>

Partitioned Hashing Example Searching

Hash Table

h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Find Major="CS" AND Year="3"

Partitioned Hashing Example Searching (2)

Hash Table

h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Find Year="2"

Partitioned Hashing Example Searching (3)

Hash Table
h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Find ⟩ Major="BA"

Partitioned Hashing Question

Hash Table
h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Find ⟩ Major="BS" OR Year="1"

Buckets searched:

A) 2 buckets
B) 4 buckets
C) 5 buckets
D) 6 buckets
E) 8 buckets

Grid Files versus Partitioned Hashing

Both grid files and partitioned hashing have different query performance.

Grid Files:

- Good for all types of queries including range and nearest-neighbor queries.
- However, many buckets will be empty or nearly empty because of attribute correlation. Thus, grid can be space inefficient.

Partitioned Hashing:

- Useless for range and nearest-neighbor queries because physical distance between points is not reflected in closeness of buckets.
- However, hash function will randomize record locations which should more evenly divide records across buckets.
  - ⇨ Partial key searches should be faster than grid files.

Bitmap Indexes

A ***bitmap index*** is useful for indexing attributes that have a small number of values. (e.g. gender)

- For each attribute value, create a bitmap where a 1 indicates that a record at that position has that attribute value.
- Retrieve matching records by id.

student table

Rec#	St. ID	Name	Mjr	Yr
0	10567	J. Doe	CS	3
1	11589	T. Allen	BA	2
2	15973	M. Smith	CS	3
3	29579	B. Zimmer	BS	1
4	34596	T. Atkins	ME	4
5	75623	J. Wong	BA	3
6	84920	S. Allen	CS	4
7	96256	P. Wright	ME	2

bitmap index on Mjr

Mjr	bitmap
BA	01000100
BS	00010000
CS	10100010
ME	00001001

bitmap index on Yr

Yr	bitmap
1	00010000
2	01000001
3	10100100
4	00001010

How could we use bitmap indexes to answer:

```
SELECT count(*) FROM student
WHERE Mjr = 'BA' and Year=2
```

Conclusion

The index structures we have seen, specifically, B+-trees are used for indexing in commercial database systems.

- There are also special indexing structures for text and spatial data.

When tuning a database, examine the types of indexes you can use and the configuration options available.

Grid files and ***partitioned hashing*** are specialized indexing methods for multi-key indexes.

Bitmap indexes allow fast lookups when attributes have few values and can be efficiently combined using logical operations.

Major Objectives

The "One Things":

- Perform searches using grid files.
- Perform insertions and searches using partitioned hashing.

Major Theme:

- Various DBMSs give you control over the types of indexes that you can use and the ability to tune their parameters. Knowledge of the underlying index structures helps performance tuning.

Objectives:

- Understand how bitmap indexes are used for searching and why they provide a space and speed improvement in certain cases.

COSC 404
Database System Implementation

Query Processing

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

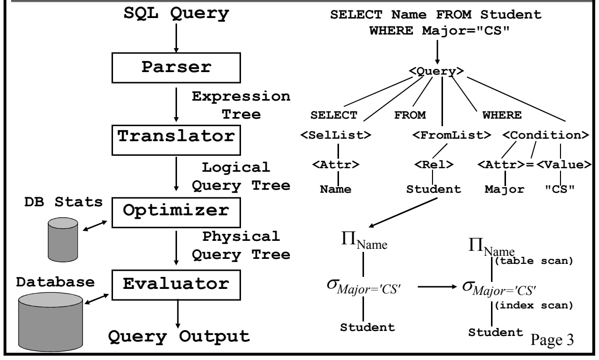
Query Processing Overview

The goal of the query processor is very simple:

Return the answer to a SQL query in the most efficient way possible given the organization of the database.

Achieving this goal is anything but simple:
- Different file organizations and indexing affect performance.
- Different algorithms can be used to perform the relational algebra operations with varying performance based on the DB.
- Estimating the cost of the query itself is hard.
- Determining the best way to answer one query in isolation is challenging. How about many concurrent queries?

Components of a Query Processor



SQL Query

Parser

Expression Tree

Translator

Logical Query Tree

DB Stats

Optimizer

Physical Query Tree

Database

Evaluator

Query Output

SELECT Name FROM Student
WHERE Major="CS"

<Query>

SELECT <SelList> <Attr> Name

FROM <FromList> <Rel> Student

WHERE <Condition> <Attr>=<Value> Major "CS"

Π_{Name} $\sigma_{Major='CS'}$ Student

Π_{Name} (table scan) $\sigma_{Major='CS'}$ (index scan) Student

Review: SQL Query Summary

The general form of the `SELECT` statement is:

SELECT <attribute list>
FROM <table list>
[WHERE (*condition*)]
[GROUP BY <grouping attributes>]
[HAVING <group condition>]
[ORDER BY <attribute list>]

- Clauses in square brackets ([,]) are optional.
- There are often numerous ways to express the same query in SQL.

Review: SQL and Relational Algebra

The `SELECT` statement can be mapped directly to relational algebra.

SELECT $A_1, A_2, \ldots, A_n$
FROM $R_1, R_2, \ldots, R_m$
WHERE P

is equivalent to:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))$$

Review: Relational Algebra Operators

Relational Operators:

◆selection	σ	- return subset of rows
◆projection	π	- return subset of columns
◆Cartesian product	$\times$	- all combinations of two relations
◆join	$\bowtie$	- combines σ and $\times$
◆duplicate elimination	δ	- eliminates duplicates

Set operators:

◆Union	$\cup$	- tuple in output if in either or both
◆Difference	-	- tuple in output if in 1st but not 2nd
◆Intersection	$\cap$	- tuple in output if in both

- Union compatibility means relations must have the same number of columns with compatible domains.

Review: Selection and Projection

The ***selection operation*** returns an output relation that has a subset of the tuples of the input by using a ***predicate***.

The ***projection operation*** returns an output relation that contains a subset of the attributes of the input.

Note: Duplicate tuples are eliminated.

Selection Example

$\sigma_{salary\ >\ 35000\ OR\ title\ =\ 'PR'}$ (Emp)

eno	ename	title	salary
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000
E5	B. Casey	SA	50000
E7	R. Davis	ME	40000
E8	J. Jones	SA	50000

Input Relation

Emp Relation

eno	ename	title	salary
E1	J. Doe	EE	30000
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000
E5	B. Casey	SA	50000
E6	L. Chu	EE	30000
E7	R. Davis	ME	40000
E8	J. Jones	SA	50000

Projection Example

$\Pi_{eno,ename}$ (Emp)

eno	ename
E1	J. Doe
E2	M. Smith
E3	A. Lee
E4	J. Miller
E5	B. Casey
E6	L. Chu
E7	R. Davis
E8	J. Jones

Review: Cartesian Product

The ***Cartesian (or cross) product*** of two relations *R* (of degree k_1) and S (of degree k_2) combines the tuples of *R* and *S* in all possible ways.

The result of *R* × *S* is a relation of degree (k_1 + k_2) and consists of all (k_1 + k_2)-tuples where each tuple is a concatenation of one tuple of *R* with one tuple of *S*. The cardinality of R × S is |*R*| * |*S*|.

Emp Relation

eno	ename	title	salary
E1	J. Doe	EE	30000
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000

Proj Relation

pno	pname	budget
P1	Instruments	150000
P2	DB Develop	135000
P3	CAD/CAM	250000

Emp × *Proj*

eno	ename	title	salary	pno	pname	budget
E1	J. Doe	EE	30000	P1	Instruments	150000
E2	M. Smith	SA	50000	P1	Instruments	150000
E3	A. Lee	ME	40000	P1	Instruments	150000
E4	J. Miller	PR	20000	P1	Instruments	150000
E1	J. Doe	EE	30000	P2	DB Develop	135000
E2	M. Smith	SA	50000	P2	DB Develop	135000
E3	A. Lee	ME	40000	P2	DB Develop	135000
E4	J. Miller	PR	20000	P2	DB Develop	135000
E1	J. Doe	EE	30000	P3	CAD/CAM	250000
E2	M. Smith	SA	50000	P3	CAD/CAM	250000
E3	A. Lee	ME	40000	P3	CAD/CAM	250000
E4	J. Miller	PR	20000	P3	CAD/CAM	250000

Review: Join

Theta (θ) join combines cross product and selection: $R \bowtie_F S = \sigma_F (R \times S)$.

An ***equijoin*** only contains the equality operator (=) in the join predicate.

- e.g. WorksOn $\bowtie_{WorksOn.pno\ =\ Proj.pno}$ Proj

A ***natural join*** *R* ⋈ *S* is the equijoin of *R* and *S* over a set of attributes common to both *R* and *S* that removes duplicate join attributes.

WorksOn Relation

eno	pno	resp	dur
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P4	Engineer	48
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E7	P4	Engineer	23

Proj Relation

pno	pname	budget
P1	Instruments	150000
P2	DB Develop	135000
P3	CAD/CAM	250000
P4	Maintenance	310000
P5	CAD/CAM	500000

WorksOn $\bowtie_{WorksOn.pno\ =\ Proj.pno}$ Proj

eno	pno	resp	dur	P.pno	pname	budget
E1	P1	Manager	12	P1	Instruments	150000
E2	P1	Analyst	24	P1	Instruments	150000
E2	P2	Analyst	6	P2	DB Develop	135000
E3	P4	Engineer	48	P4	Maintenance	310000
E5	P2	Manager	24	P2	DB Develop	135000
E6	P4	Manager	48	P4	Maintenance	310000
E7	P3	Engineer	36	P3	CAD/CAM	250000
E7	P4	Engineer	23	P4	Maintenance	310000

Review Question

Given this table and the query:

```
SELECT eno, salary
FROM   emp
WHERE  salary >= 40000
```

How many rows in the result?

A) 2

B) 3

C) 4

D) 5

Emp Relation

eno	ename	title	salary
E1	J. Doe	EE	30000
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000
E5	B. Casey	SA	50000
E6	L. Chu	EE	30000
E7	R. Davis	ME	40000
E8	J. Jones	SA	50000

Review Question

Given these tables and the query:

$\Pi_{eno,\ ename}\ (\sigma_{title='EE'}(Emp \bowtie_{dno=dno} Dept)\ )$

How many rows in the result?

A) 0

B) 1

C) 2

D) 8

Dept Relation

dno	dname	mgreno
D1	Management	E8
D2	Consulting	E7
D3	Accounting	E5
D4	Development	null

Emp Relation

eno	ename	bdate	title	salary	supereno	dno
E1	J. Doe	01-05-75	EE	30000	E2	null
E2	M. Smith	06-04-66	SA	50000	E5	D3
E3	A. Lee	07-05-66	ME	40000	E7	D2
E4	J. Miller	09-01-50	PR	20000	E6	D3
E5	B. Casey	12-25-71	SA	50000	E8	D3
E6	L. Chu	11-30-65	EE	30000	E7	D2
E7	R. Davis	09-08-77	ME	40000	E8	D1
E8	J. Jones	10-11-72	SA	50000	null	D1

Review Question

Question: What is the symbol for duplicate elimination?

A) σ

B) ×

C) π

D) ⋈

E) δ

Algorithms for Relational Operators

Our initial focus is developing algorithms to implement the relational operators of selection, projection, and join.

The query processor contains these implementations and uses them to answer queries.

We will discuss when the algorithms should be applied when discussing optimization. For now, we will build a ***toolkit*** of potential algorithms that could be used.

Query Processing Classifying Algorithms

Two ways to classify relational algebra algorithms:

1) By the number of times the data is read:

- ***One-Pass*** - selection or projection operators or binary operators where one relation fits entirely in memory.
- ***Two-Pass*** - data does not fit entirely in memory in one pass, but algorithm can process data using only two passes.
- ***Multi-Pass*** - generalization to larger data sets.

2) By the type of relational algebra operator performed:

- ***Tuple-at-a-time, unary operators*** - selection, projection
  - Do not need entire relation to perform operation.
- ***Full-relation, unary operators*** - grouping,duplicate elimination
- ***Full-relation, binary operators*** - join, set operations

Measuring Cost of Algorithms

Algorithms will be compared using number of block I/Os.

- Note: CPU time is important but harder to model.

Assumptions:

- The arguments of any operator are found on disk, but the operator result is left in memory.
  - For example, a select operation on a relation, must read the relation from disk, but after the operation is performed, the result is left in memory (and can be potentially used by the next operator).
  - This is also true for the query result.

Measuring Cost of Algorithms (2)

Some basic statistics will be useful when discussing algorithms:

- 1) The number of buffer blocks available to the algorithm is ***M***.
  - We will assume memory blocks are the same size as disk blocks.
  - The buffers are used to stored input and intermediate results; the buffers do not have to be used to store output which is assumed to go elsewhere.
  - ***M*** is always less than the size of memory, but in practice, may even be much smaller than that as many operators can be executing at once.
- 2) ***B(R)*** or just ***B*** (if ***R*** is assumed) is the # of blocks on disk used to store all the tuples of ***R***.
  - Usually, assume that ***R*** is clustered and that we can only read 1 block at a time. Note that we will ignore free-space in blocks even though in practice blocks are not normally kept completely full.
- 3) ***T(R)*** or just ***T*** (if ***R*** is assumed) is the # of tuples in ***R***.
- 4) ***V(R,a)*** is the # of distinct values of the column ***a*** in ***R***.
  - Note: *V(Student,Id)* = *T(Student)* as *Id* is a key.

Metrics Question

Question: The number of rows in table `Emp` is 50. There are 10 possible values for the `title` attribute. Select a true statement.

A) T(Emp) = 10

B) V(Emp, eno) = 10

C) V(Emp, title) = 10

D) V(Emp, title) = 50

Scans and Sorts

Two basic operations are scanning and sorting an input.

There are two types of scans:

- 1) ***Table scan*** - read the relation ***R*** from disk one block at a time.
- 2) ***Index scan*** - read the relation ***R*** or only the tuples of ***R*** that satisfy a given condition, by using an index on ***R***.

Sorting can be performed in three ways:

- 1) ***Index sort*** - used when the relation ***R*** has a B+-tree index on sort attribute ***a***.
- 2) ***Main-memory sort*** - read the entire relation ***R*** into main memory and use an efficient sorting algorithm.
- 3) ***External-merge sort*** - use the external-merge sort if the entire relation ***R*** is too large to fit in memory.
  - We will discuss this sorting algorithm later.

Measuring Cost of Scan Operators

The cost of a table scan for relation *R* is ***B***.

What would be the cost of an index scan of relation *R* that has ***B*** data blocks and ***I*** index blocks?

- Does it depend on the type of index?

Iterators for Operators

Database operations are implemented as ***iterators***.

- Also called pipelining or producer-consumer.

Instead of completing the entire operation before releasing output, an operator releases output to other operators as it is produced ***one tuple at a time***.

Iterators are combined into a tree of operators. Iterators execute in parallel and query results are produced faster.

Structure of Iterators

Database iterators implement three methods:

- `init()` - initialize the iterator variables and algorithm.
  - Starts the process, but does not retrieve a tuple.
- `next()` - return the next tuple of the result and perform any required processing to be able to return the next tuple of the result the next time `next()` is called.
  - `next()` returns `NULL` if there are no more tuples to return.
- `close()` - destroy iterator variables and terminate the algorithm.

Each algorithm we discuss can be implemented as an iterator.

Iterator Example
Table Scan Iterator

```
init() {
   b = the first block of R;
   t = first tuple of R;
}
next() {
   if (t is past the last tuple on block b) {
      increment b to the next block;
      if (there is no next block)
         return NULL;
      else  /* b is a new block */
         t = first tuple on block b;
   }
   oldt = t;
   increment t to the next tuple of b;
   return oldt;
}
close() {}
```

Iterator Example
Main-Memory Sort Iterator

```
init() {
   Allocate buffer array A
   read entire relation R block-by-block into A;
   sort A using quick sort;
   tLoc = 0;   // First tuple location in A
}

next() {
   if (tLoc >= T)
         return NULL;
   else
   {  tLoc++;
      return A[tLoc-1];
   }
}

close() {}
```

How is this iterator different than the table scan iterator?

Programming Iterators in Java

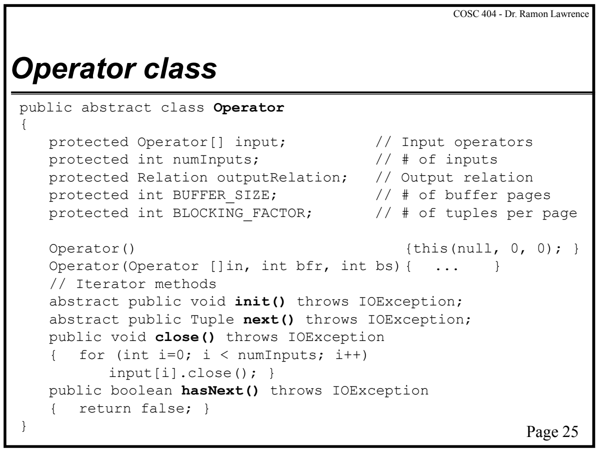
We will implement iterators in Java and combine them to build execution trees.

Iterators are derived from the `Operator` class.

This class has the methods `init()`, `next()`, `hasNext()`, and `close()`.

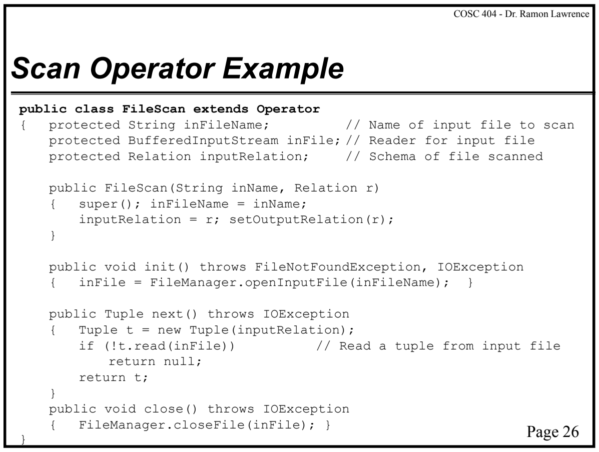
The operator has an array of input operators which may consist of 0, 1, or 2 operators.

- A relation scan has 0 input operators.



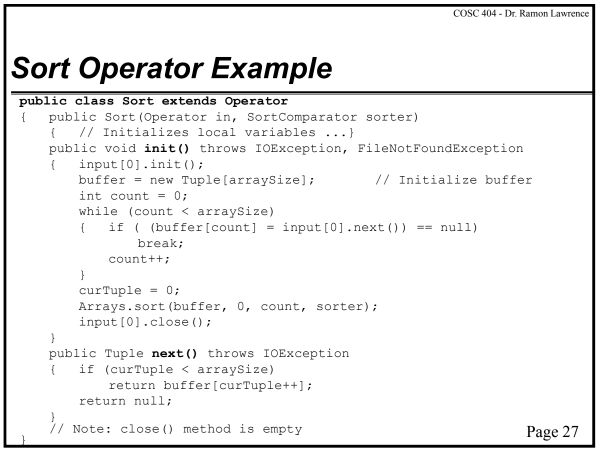
Operator class

```
public abstract class Operator
{
    protected Operator[] input;          // Input operators
    protected int numInputs;             // # of inputs
    protected Relation outputRelation;   // Output relation
    protected int BUFFER_SIZE;           // # of buffer pages
    protected int BLOCKING_FACTOR;       // # of tuples per page

    Operator()                              {this(null, 0, 0); }
    Operator(Operator []in, int bfr, int bs){   ...    }
    // Iterator methods
    abstract public void init() throws IOException;
    abstract public Tuple next() throws IOException;
    public void close() throws IOException
    {   for (int i=0; i < numInputs; i++)
            input[i].close(); }
    public boolean hasNext() throws IOException
    {   return false; }
}
```



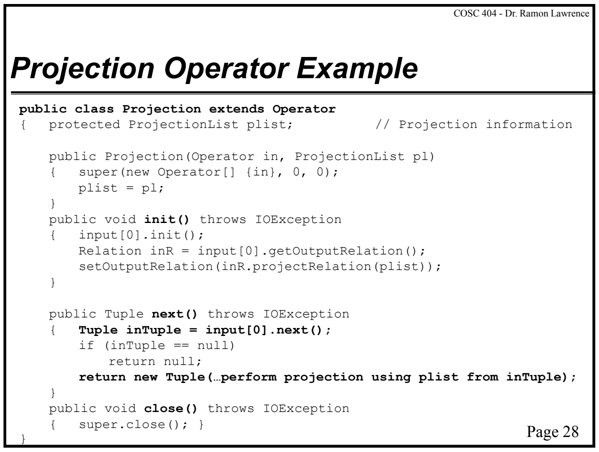
Scan Operator Example

```
public class FileScan extends Operator
{   protected String inFileName;          // Name of input file to scan
    protected BufferedInputStream inFile;// Reader for input file
    protected Relation inputRelation;     // Schema of file scanned

    public FileScan(String inName, Relation r)
    {   super(); inFileName = inName;
        inputRelation = r; setOutputRelation(r);
    }

    public void init() throws FileNotFoundException, IOException
    {   inFile = FileManager.openInputFile(inFileName);  }

    public Tuple next() throws IOException
    {   Tuple t = new Tuple(inputRelation);
        if (!t.read(inFile))           // Read a tuple from input file
            return null;
        return t;
    }
    public void close() throws IOException
    {   FileManager.closeFile(inFile); }
}
```



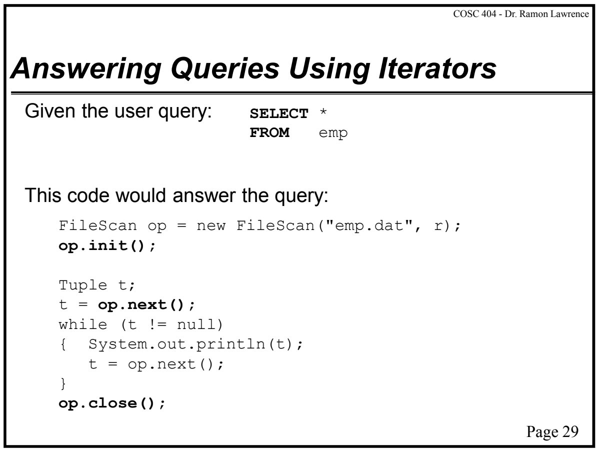
Sort Operator Example

```
public class Sort extends Operator
{   public Sort(Operator in, SortComparator sorter)
    {   // Initializes local variables ...}
    public void init() throws IOException, FileNotFoundException
    {   input[0].init();
        buffer = new Tuple[arraySize];        // Initialize buffer
        int count = 0;
        while (count < arraySize)
        {   if ( (buffer[count] = input[0].next()) == null)
                break;
            count++;
        }
        curTuple = 0;
        Arrays.sort(buffer, 0, count, sorter);
        input[0].close();
    }
    public Tuple next() throws IOException
    {   if (curTuple < arraySize)
            return buffer[curTuple++];
        return null;
    }
    // Note: close() method is empty
}
```



Projection Operator Example

```
public class Projection extends Operator
{   protected ProjectionList plist;          // Projection information

    public Projection(Operator in, ProjectionList pl)
    {   super(new Operator[] {in}, 0, 0);
        plist = pl;
    }
    public void init() throws IOException
    {   input[0].init();
        Relation inR = input[0].getOutputRelation();
        setOutputRelation(inR.projectRelation(plist));
    }

    public Tuple next() throws IOException
    {   Tuple inTuple = input[0].next();
        if (inTuple == null)
            return null;
        return new Tuple(…perform projection using plist from inTuple);
    }
    public void close() throws IOException
    {   super.close(); }
}
```



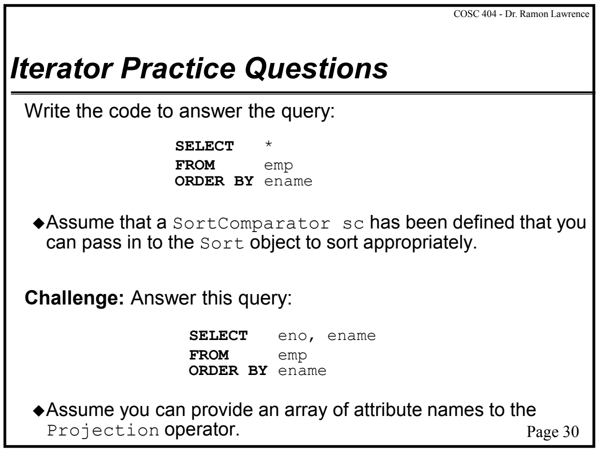
Answering Queries Using Iterators

Given the user query:

```
SELECT *
FROM   emp
```

This code would answer the query:

```
FileScan op = new FileScan("emp.dat", r);
op.init();

Tuple t;
t = op.next();
while (t != null)
{  System.out.println(t);
   t = op.next();
}
op.close();
```



Iterator Practice Questions

Write the code to answer the query:

```
SELECT   *
FROM     emp
ORDER BY ename
```

- ◆Assume that a `SortComparator sc` has been defined that you can pass in to the `Sort` object to sort appropriately.

Challenge: Answer this query:

```
SELECT   eno, ename
FROM     emp
ORDER BY ename
```

- ◆Assume you can provide an array of attribute names to the `Projection` operator.

One-Pass Algorithms

One-pass algorithms read data from the input only once.

Selection and projection are one-pass, tuple-at-a-time operators.

Tuple-at-a-time operators require only one main memory buffer (***M=1***) and cost the same as the scan.

- Note that the CPU cost is the dominant cost of these operators.

One-Pass Algorithms
Grouping and Duplicate Elimination

Duplication elimination (δ) and grouping (γ) require reading the entire relation and remembering tuples previously seen.

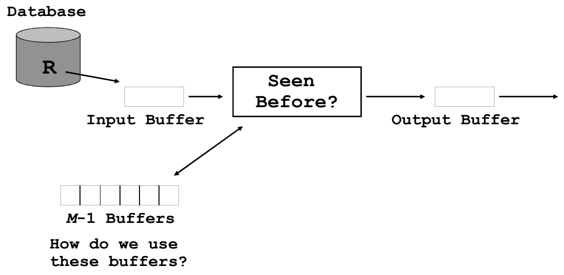
One-pass duplicate elimination algorithm:

- 1) Read each block of relation ***R*** one at a time.
- 2) For each tuple read, determine if:
  - ⇨ This is the first time the tuple has been seen. If so, copy to output.
  - ⇨ Otherwise, discard duplicate tuple.

Challenge: How do we know if a tuple has been seen before?

Answer: We must build a main memory data structure that stores copies of all the tuples that we have already seen.

One-Pass Algorithms
Duplicate Elimination Overview



One-Pass Algorithms
Duplicate Elimination Discussion

The ***M-1*** buffers are used to store a fast lookup structure such that given a tuple, we can determine if we have seen it before.

- Main-memory hashing or balanced binary trees are used.
  - ⇨ Note that an array would be inefficient. Why?
- Space overhead for the data structure is ignored in our calculations.

M-1 buffers allows us to store ***M-1*** blocks of ***R***. Thus, the number of main memory buffers required is approximately:

$$M >= B(\delta(R))$$

One Pass Duplicate Elimination Question

Question: If T(R)=100 and V(R,a)=1 and we perform $\delta(\Pi_a(R))$, select a true statement.

A) The maximum memory size used is 100 tuples (not counting input tuple).

B) The size of the result is 100 tuples.

C) The size of the result is unknown.

D) The maximum memory size used is 1 tuple (not counting input tuple).

One-Pass Algorithms
Grouping

The grouping (γ) operator can be evaluated similar to duplicate elimination except now besides identifying if a particular group already exists, we must also calculate the aggregate values for each group as requested by the user.

How to calculate aggregate values:

- `MIN`(***a***) or `MAX`(***a***) - for each group maintain the minimum or maximum value of attribute ***a*** seen so far. Update as required.
- `COUNT(*)` - add one for each tuple of the group seen.
- `SUM`(***a***) - keep a running sum for ***a*** for each group.
- `AVG`(***a***) - keep running sum and count for ***a*** for each group and return `SUM`(***a***)/`COUNT`(***a***) after all tuples are seen.

One-Pass Algorithms Grouping Example

```
SELECT Major, Count(*), Min(Year),
        Max(Year), AVG(Year)
  FROM Student GROUP BY Major
```

Student Relation

St. ID	Name	Mjr	Yr	
10567	J. Doe	CS	3	✓
11589	T. Allen	BA	2	✓
15973	M. Smith	CS	3	✓
29579	B. Zimmer	BS	1	✓
34596	T. Atkins	ME	4	✓
75623	J. Wong	BA	3	✓
84920	S. Allen	CS	4	✓
96256	P. Wright	ME	2	✓

Memory Buffers

Major	Count	Min	Max	Avg
CS	3	3	4	3.33
BA	2	2	3	2.5
BS	1	1	1	1
ME	2	2	4	3

```
Main memory table copied to output
to answer query.
```

One-Pass Algorithms Grouping Discussion

After all tuples are seen and aggregate values are calculated, write each tuple representing a group to the output.

The cost of the algorithm is ***B(R)***, and the memory requirement ***M*** is almost always less than ***B(R)***, although it can be much smaller depending on the group attributes.

- Question: When would ***M*** ever be larger than ***B(R)***?

Both duplicate elimination and grouping are ***blocking algorithms by nature*** that do not fit well into the iterator model!

One-Pass Algorithms Binary Operations

It is also possible to implement one-pass algorithms for the binary operations of union, intersection, difference, cross-product, and natural join.

For the set operators, we must distinguish between the set and bag versions of the operators:

- **Union** - set union ($\cup_S$) and bag union ($\cup_B$)
- **Intersection** - set intersection ($\cap_S$) and bag intersection ($\cap_B$)
- **Difference** - set difference ($-_S$) and bag difference ($-_B$)

Note that only bag union is a tuple-at-a-time algorithm. All other operators require one of the two operands to fit entirely in main memory in order to support a one-pass algorithm.

- We will assume two operand relations ***R*** and ***S***, with ***S*** being small enough to fit entirely in memory.

One-Pass Algorithms Binary Operations - General Algorithm

The general algorithm is similar for all binary operations:

- 1) Read the smaller relation, ***S***, entirely into main memory and construct an efficient search structure for it.
  - This requires approximately ***B(S)*** main memory blocks.
- 2) Allocate one buffer for reading one block of the larger relation, ***R***, at a time.
- 3) For each block and each tuple of ***R***
  - Compare the tuple of ***R*** with the tuples of ***S*** in memory and perform the specific function required for the operator.

The function performed in step #3 is operator dependent.

All binary one-pass algorithms take ***B(R) + B(S)*** disk operations.

They work as long as ***B(S) <= M-1*** or ***B(S) < M***.

One-Pass Algorithms Binary Operations Algorithms

Function performed on each tuple ***t*** of ***R*** for the operators:

- 1) **Set Union** - If ***t*** is not in ***S***, copy to output, otherwise discard.
  - Note: All tuples of ***S*** were initially copied to output.
- 2) **Set Intersection**-If ***t*** is in ***S***, copy to output, otherwise discard.
  - Note: No tuples of ***S*** were initially copied to output.
- 3) **Set difference**
  - **R $-_S$ S:** If ***t*** is not in ***S***, copy to output, otherwise discard.
  - **S $-_S$ R:** If ***t*** is in ***S***, then delete ***t*** from the copy of ***S*** in main memory. If ***t*** is not in ***S***, do nothing. After seeing all tuples of ***R***, copy to output tuples of ***S*** that remain in memory.
- 4) **Bag Intersection**
  - Read ***S*** into memory and associate a count for each distinct tuple.
  - If ***t*** is found in ***S*** and count is still positive, decrement count by 1 and output ***t***. Otherwise, discard ***t***.

One-Pass Algorithms Binary Operations Algorithms (2)

Function performed on each tuple ***t*** of ***R*** for the operators:

- 5) **Bag difference**
  - **S $-_B$ R:** Similar to bag intersection (using counts), except only output tuples of ***S*** at the end if they have positive counts (and output that many).
  - **R $-_B$ S:** Exercise - try it for yourself.
- 6) **Cross-product** - Concatenate ***t*** with each tuple of ***S*** in main memory. Output each tuple formed.
- 7) **Natural Join**
  - Assume connecting relations ***R(X,Y)*** and ***S(Y,Z)*** on attribute set ***Y***.
  - ***X*** is all attributes of ***R*** not in ***Y***, and ***Z*** is all attributes of ***S*** not in ***Y***.
  - For each tuple ***t*** of ***R***, find all tuples of ***S*** that match on ***Y***.
  - For each match output a joined tuple.

One-Pass Algorithms
Review Questions

1) How many buffers are required to perform a selection operation on a relation that has size 10,000 blocks?

2) Assume the number of buffers M=100. Let B(R)=10,000 and B(S)=90. How many block reads are performed for R ∪ S?

3) If M=100, B(R)=5,000 and B(S)=1,000, how many block reads are performed for R - S using a one-pass algorithm?

Nested-Loop Joins

Nested-loop joins are join algorithms that compute a join using simple `for` loops.

These algorithms are essentially "one-and-a-half-pass" algorithms because one of the relations is read only once, while the other relation is read repeatedly.

There are two variants:

- ◆1) Tuple-based nested-loop join
- ◆2) Block-based nested-loop join

For this discussion, we will assume a natural join is to be computed on relations ***R(X,Y)*** and ***S(Y,Z)***.

Nested-Loop Joins
Tuple-based Nested-Loop Join

In the tuple-based nested-loop join, tuples are matched using two `for` loops. Algorithm:

```
for (each tuple s in S)
      for (each tuple r in R)
            if (r and s join to make a tuple t)
                  output t;
```

Notes:

- ◆Very simple algorithm that can vary widely in performance if:
  - ⇨There is an index on the join attribute of ***R***, so the entire relation ***R*** does not have to be read.
  - ⇨Memory is managed smartly so that tuples are in memory when needed (use buffers intelligently).
  - ⇨Worse case is ***T(R)*T(S)*** if for every tuple we have to read it from disk!

Nested-Loop Joins
Tuple-based Nested-Loop Join Iterator

```
// Initialize relation iterators and read tuple of S
init() {  R.init(); S.init(); s = S.next();  }

next() {
   do {
      r = R.next();
      if (r == NULL){// R is exhausted for current s
         R.close();
         s = S.next();
         if (s == NULL) return NULL;  // Done
         R.open();      // Re-initialize scan of R
         r = R.next();
      }
   } while !(r and s join); // Found one joined tuple
   return (the tuple created by joining r and s);
}
close() { R.close(); S.close();}
```

Nested-Loop Joins
Block-based Nested-Loop Join

Block-based nested-loop join is more efficient because it operates on blocks instead of individual tuples.

Two major improvements:

- ◆1) Access relations by blocks instead of by tuples.
- ◆2) Buffer as many blocks as available of the outer relation ***S***. That is, load chunks of relation ***S*** into the buffer at a time.

The first improvement makes sure that as we read ***R*** in the inner loop, we do it a block at a time to minimize I/O.

The second improvement enables us to join one tuple of ***R*** (inner loop) with as many tuples of ***S*** that fit in memory at one time (outer loop).

- ◆This means that we do not have to continually load a block of ***S*** at time.

Nested-Loop Joins
Nested-Block Join Algorithm

```
for (each chunk of M-1 blocks of S)
   read these blocks into main memory buffers;
   organize these tuples into an efficient search
      structure whose search key is the join attributes;
   for (each block b of R)
      read b into main memory;
      for (each tuple t of b)
         find tuples of S in memory that join with t;
         output the join of t with each of these tuples;
```

Note that this algorithm has 3 for loops, but does the same processing more efficiently than the tuple-based algorithm. Outer loop processes tuples of ***S***, inner loop processes tuples of ***R***.

Nested-Loop Joins Analysis and Discussion

Nested-block join analysis:

Assume *S* is the smaller relation.

# of outer loop iterations = $\lceil B(S)/M\text{-}1 \rceil$

Each iteration reads *M-1* blocks of *S* and *B(R)* (all) blocks of *R*.

Number of disk I/O is:

$$B(S) + B(R) * \left\lceil \frac{B(S)}{M-1} \right\rceil$$

In general, this can be approximated by ***B(S)*B(R)/M***.

Nested-Loop Joins Performance Example

If M=1,000, B(R)=100,000, T(R)=1,000,000, B(S)=5,000, and T(S)=250,000, calculate the performance of tuple-based and block-based nested loop joins.

Tuple-Based Join:

worst case = T(R) * T(S) = 1,000,000 * 250,000

= **25,000,000,000** = 25 billion!

Block-Based Join:

worst case = B(S) + B(R)*ceiling(B(S)/(M-1))

= 5,000 + 100,000 * ceiling(5,000 / 999)

= **605,000**

Question: What is the I/Os if the larger relation R is in the outer loop?

Nested Loop Join Question

Question: Select a true statement.

A) NLJ buffers the smaller relation in memory.

B) NLJ buffers the larger relation in memory.

Sorting-based Two-Pass Algorithms

Two-pass algorithms read the input at most twice.

Sorting-based two-pass algorithms rely on the external sort merge algorithm to accomplish their goals.

The basic process is as follows:

- 1) Create sorted sublists of size ***M*** blocks of the relation ***R***.
- 2) Merge the sorted sublists by continually taking the minimum value in each list.
- 3) Apply the appropriate function to implement the operator.

We will first study the external sort-merge algorithm then demonstrate how its variations can be used to answer queries.

External Sort-Merge Algorithm

1) Create sorted runs as follows:

⇨ Let ***i*** be 0 initially, and ***M*** be the number of main memory blocks.

- Repeat these steps until the end of the relation:
  - ⇨ (a) Read ***M*** blocks of relation into memory.
  - ⇨ (b) Sort the in-memory blocks.
  - ⇨ (c) Write sorted data to run $\boldsymbol{R_i}$; increment ***i***.

2) Merge the runs in a single merge step:

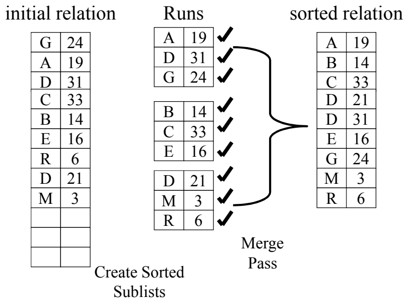
⇨ Suppose for now that ***i < M***. Use ***i*** blocks of memory to buffer input runs.

⇨ We will write output to disk instead of using 1 block to buffer output.

- Repeat these steps until all input buffer pages are empty:
  - ⇨ (a) Select the first record in sorted order from each of the buffers.
  - ⇨ (b) Write the record to the output.
  - ⇨ (c) Delete the record from the buffer page. If the buffer page is empty, read the next block (if any) of the run (sublist) into the buffer.

External Sort-Merge Example

Sort by column #1. M=3. (Note: Not using an output buffer.)



initial relation

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

Runs

A	19	✓
D	31	✓
G	24	✓

B	14	✓
C	33	✓
E	16	✓

D	21	✓
M	3	✓
R	6	✓

sorted relation

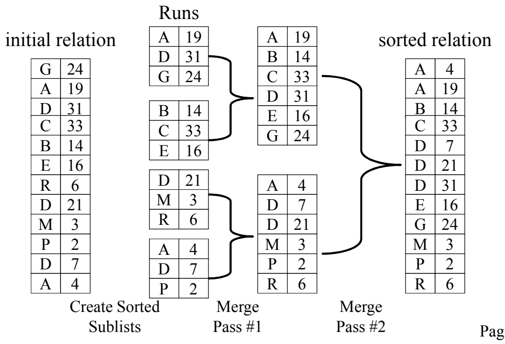
A	19
B	14
C	33
D	21
D	31
E	16
G	24
M	3
R	6

Multi-Pass External Sort-Merge

If ***i* ≥ *M***, several merge passes are required as we can not buffer the first block of all sublists in memory at the same time.

- In this case, use an output block to store the result of a merge.
- In each pass, contiguous groups of ***M-1*** runs are merged.
- A pass reduces the number of runs by a factor of ***M-1***, and creates runs longer by the same factor.
- Repeated passes are performed until all runs are merged.

External Sort-Merge Example 2 Multi-Pass Merge



Initial relation:

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3
P	2
D	7
A	4

Runs:

A	19
D	31
G	24

B	14
C	33
E	16

D	21
M	3
R	6

A	4
D	7
P	2

Merge Pass #1:

A	19
B	14
C	33
D	31
E	16
G	24

A	4
D	7
D	21
M	3
P	2
R	6

Sorted relation:

A	4
A	19
B	14
C	33
D	7
D	21
D	31
E	16
G	24
M	3
P	2
R	6

External Sort-Merge Analysis

Cost analysis:

- Two-pass external sort cost is: ***3*B***. (B=B(R))
  - ⇨ Each block is read twice: once for initial sort, once for merge.
  - ⇨ Each block is written once after the first pass.
  - ⇨ The cost is *4*B* if we include the cost of writing the output.
- Multi-pass external sort cost is: $B*(2\lceil \log_{M-1}(B/M)\rceil + 1)$.
  - ⇨ Disk accesses for initial run creation as well as in each pass is 2*B (except for final pass that does not write out results).
  - ⇨ Total number of merge passes required: $\lceil \log_{M-1}(B/M)\rceil$
    - *B/M* is the # of initial runs, and # decreases by factor of *M-1* every pass.
    - Each pass reads/writes each block (*2*B*) except final run has no write.

Sort analysis:

- A two-pass external sort can sort M^2 blocks.
- A *N*-pass external sort can sort M^N blocks.

External Sort-Merge Analysis Example

A main memory size is 64 MB, the block size is 4 KB, and the record size is 160 bytes.

- 1) How many records can be sorted using a two-pass sort?
  - ⇨ Sort can sort M^2 memory blocks.
  - ⇨ # of memory blocks = memory size/block size
  - ⇨ Total # of blocks sorted = (64 MB / 4 KB $)^2$ = approx. 268 million
  - ⇨ Total # of records sorted = #blocks *blockingFactor = approx. 6.8 billion!
  - ⇨ Total size is approximately **1 terabyte**.
- 2) How many records can be sorted using a three-pass sort?
  - ⇨ Sort can sort M^3 memory blocks.
  - ⇨ Same calculation results in 112 trillion records of total size **16 petabytes**!

Bottom-line: Two way sort is sufficient for most purposes!

External Sort-Merge Usage

The external sort-merge algorithm can be used when:

- 1) SQL queries specify a sorted output.
- 2) For processing a join algorithm using merge-join algorithm.
- 3) Duplicate elimination.
- 4) Grouping and aggregation.
- 5) Set operations.

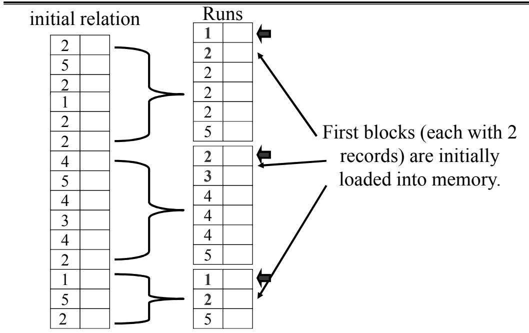
We will see how the basic external sort-merge algorithm can be modified for these operations.

Duplicate Elimination Using Sorting

Algorithm (two-pass):

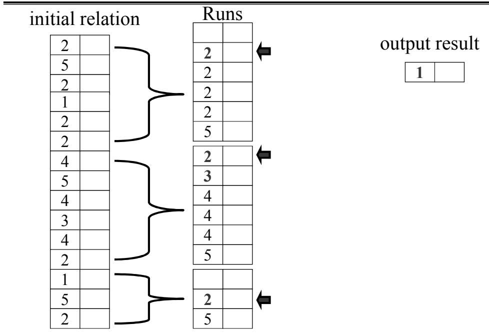
- Sort the tuples of ***R*** into sublists using the available memory buffers ***M***.
- In the second pass, buffer one block of each sublist in memory like the sorting algorithm.
- However, in this case, instead of sorting the tuples, only copy one to output and ignore all tuples with duplicate values.
  - ⇨ Every time we copy one value to the output, we search forward in all sublists removing all copies of this value.

Duplicate Elimination Example

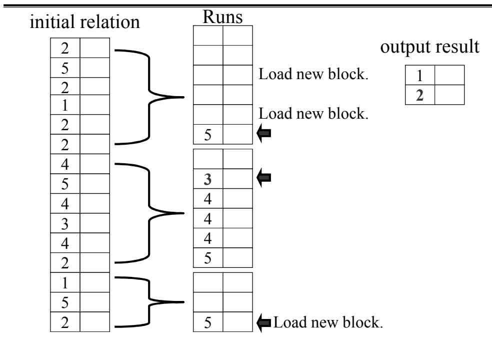


Duplicate elimination on column #1. M=3. blocking factor=2.

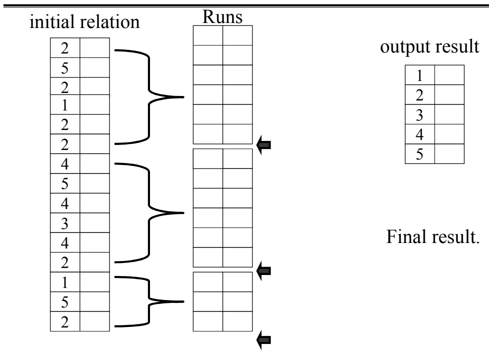
Duplicate Elimination Example (2)



Duplicate Elimination Example (3)



Duplicate Elimination Example (4)



Duplicate Elimination Analysis

The number of disk operations is always ***3*B(R)***.

- *2*B(R)* to read/write each block to create sorted sublists.
- *B(R)* to read each block of each sublist when performing duplicate elimination.

Remember the single pass algorithm was ***B(R)***.

The two-pass algorithm can handle relations where ***B(R)<=M²***.

Grouping and Aggregation Using Sorting

Algorithm (two-pass):

- Sort the tuples of ***R*** into sublists using the available memory buffers ***M***.
- In the second pass, buffer one block of each sublist in memory like the sorting algorithm.
- Find the smallest value of the sort key (grouping attributes) in all the sublists. This value becomes the next group.
  - Prepare to calculate all aggregates for this group.
  - Examine all tuples with the given value for the sort key and calculate aggregate functions accordingly.
  - Read blocks from the sublists into buffers as required.
  - When there are no more values for the given sort key, output a tuple containing the grouped values and the calculated aggregate values.

Analysis: This algorithm also performs ***3*B(R)*** disk operations.

Grouping Question

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Calculate the output for a query that groups by the given integer attribute and returns a count of the # of records that contains that attribute.

Assume M=3 and blocking factor=2.

Set Operations Using Sorting

The set operations can also be implemented using a sorting based algorithm.

- All algorithms start with an initial sublist creation step where both relations *R* and *S* are divided into sorted sublists.
- Use one main memory buffer for each sublist of *R* and *S*.
- Many of the algorithms require counting the # of tuples of *R* and *S* that are identical to the current minimum tuple *t*.

Special steps for each algorithm operation:

- ***Set Union*** - Find smallest tuple *t* of all buffers, copy *t* to output, and remove all other copies of *t*.
- ***Set Intersection*** - Find smallest tuple *t* of all buffers, copy *t* to output if it appears in both *R* and *S*.
- ***Bag Intersection*** - Find smallest tuple *t* of all buffers, output *t* the minimum # of times it appears in *R* and *S*.

Set Operations Using Sorting (2)

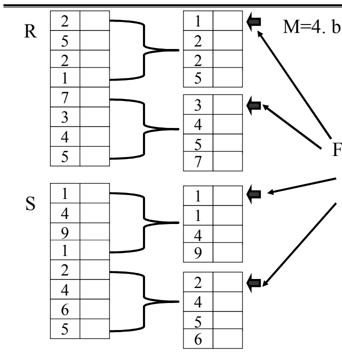
- ***Set Difference*** - Find smallest tuple *t* of all buffers, output *t* only if it appears in *R* but not in *S*. ($R -_S S$).
- ***Bag difference*** - Find smallest tuple *t* of all buffers, output *t* the number of times it appears in *R* minus the number of times it appears in *S*.

Analysis: All algorithms for set operations perform ***3*(B(R)+B(S))*** disk operations, and the two-pass versions will only work if ***B(R)+B(S) <= M²***.

⇨ Note: More precisely the two-pass set algorithms only work if:

$\lceil B(R)/M \rceil + \lceil B(S)/M \rceil <= M$

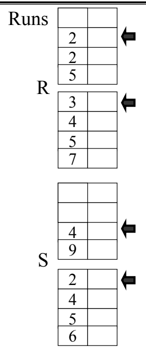
Set Operations Example - Intersection



M=4. blocking factor=1.

First blocks (each with 1 record) are initially loaded into memory.

Set Operations Example - Intersection (2)

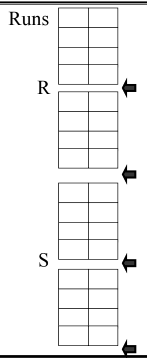


1 occurs in both R and S.

Output

1

Set Operations Example - Intersection (3)



Final Result.

Output

1
2
4
5

Set Operations Questions

R

2	
5	
2	
1	
7	
3	
4	
5	

S

1	
4	
9	
1	
2	
4	
6	
5	

Show how the following operations are performed using two-pass sorting based algorithms:
1) Set Union
2) Set Difference ($R -_S S$)
3) Bag Difference
4) Bag Intersection

Assume M=4 and bfr=1.

For set operators, first eliminate duplicates in R and S.

Sort-Based Join Algorithm

Sorting can be used to join two relations *R(X,Y)* and *S(Y,Z)*.

One of the challenges of any join algorithm is that the number of tuples of the two relations that share a common value of the join attribute(s) must be in memory at the same time.

- ◆This is difficult if the number exceeds the size of memory.
  - ⇨Worse-case: Only one value for the join attribute(s). All tuples join to each other. If this is the case, nested-loop join is used.

We will look at two different algorithms based on sorting:

- ◆***Sort-join*** - Allows for the most possible buffers for joining.
- ◆***Sort-merge-join*** - Has fewer I/Os, but more sensitive to large numbers of tuples with common join attribute.

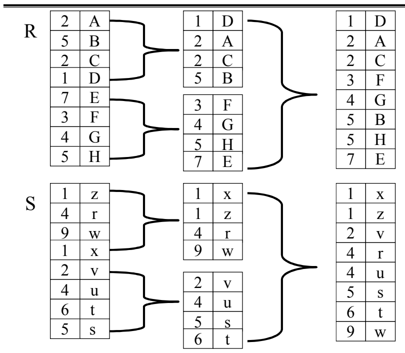
Sort-Join Algorithm

1) Sort *R* and *S* using an external merge sort with *Y* as the key.
2) Merge the sorted *R* and *S* using one buffer for each relation.

- ◆a) Find the smallest value *y* of join attributes *Y* in the start of blocks for *R* and *S*.
- ◆b) If *y* does not appear in the other relation, remove the tuples with key *y*.
- ◆c) Otherwise, identify all tuples in both relations that have the value *y*.
  - ⇨May need to read many blocks from *R* and *S* into memory. Use the *M* main memory buffers for this purpose.
- ◆d) Output all tuples that can be formed by joining tuples of *R* and *S* with common value *y*.
- ◆e) If either relation has no tuples buffered in memory, read the next block of the relation into a memory buffer.

Sort-Join Example Sort Phase

M=4. blocking factor=1.



R

2	A		1	D		1	D
5	B		2	A		2	A
2	C		2	C		2	C
1	D		5	B		3	F
7	E		3	F		4	G
3	F		4	G		5	B
4	G		5	H		5	H
5	H		7	E		7	E

S

1	z		1	x		1	x
4	r		1	z		1	z
9	w		4	r		2	v
1	x		9	w		4	r
2	v		2	v		4	u
4	u		4	u		5	s
6	t		5	s		6	t
5	s		6	t		9	w

Sort-Join Example Merge Phase

M=4. blocking factor=1.

R

1	D	⬅
2	A	⇦ In memory after join on 1.
2	C	
3	F	
4	G	
5	B	
5	H	
7	E	

S

1	x	⬅
1	z	⇦ Brought in for join on 1.
2	v	⇦ In memory after join on 1.
4	r	
4	u	
5	s	
6	t	
9	w	

Buffer

1	D	R
1	x	S
1	z	extra
		extra

Output

1	D	x
1	D	z

Notes:
- Only one block of R and S in memory at a time.
- Use other two buffers to bring in records with attribute values that match current join attribute.

Sort-Join Example Merge Phase (2)

M=4. blocking factor=1.

R

1	D	
2	A	⬅
2	C	⇦ Brought in for join on 2.
3	F	⇦ In memory after join on 2.
4	G	
5	B	
5	H	
7	E	

S

1	x	
1	z	
2	v	⬅
4	r	⇦ In memory after join on 2.
4	u	
5	s	
6	t	
9	w	

Buffer

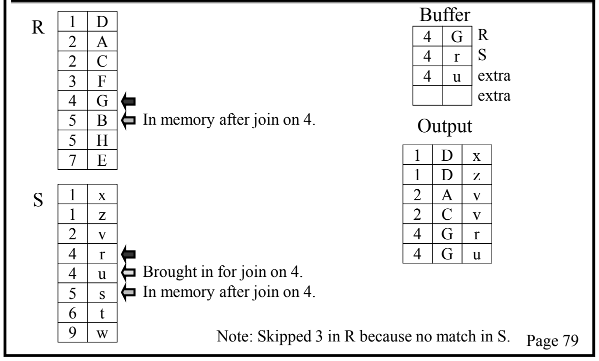
2	A	R
2	v	S
2	C	extra
		extra

Output

1	D	x
1	D	z
2	A	v
2	C	v

Sort-Join Example Merge Phase (3)

M=4. blocking factor=1.



R

1	D
2	A
2	C
3	F
4	G
5	B
5	H
7	E

In memory after join on 4.

S

1	x
1	z
2	v
4	r
4	u
5	s
6	t
9	w

Brought in for join on 4.
In memory after join on 4.

Buffer

4	G	R
4	r	S
4	u	extra
		extra

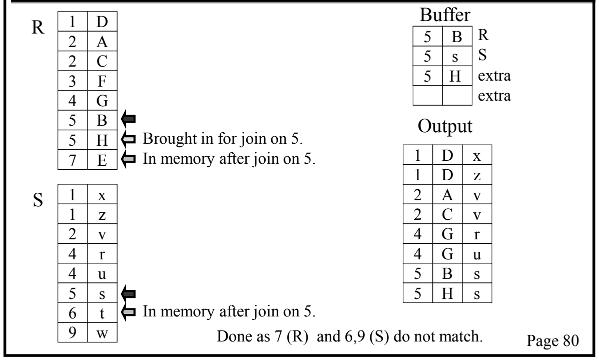
Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u

Note: Skipped 3 in R because no match in S.

Sort-Join Example Merge Phase (4)

M=4. blocking factor=1.



R

1	D
2	A
2	C
3	F
4	G
5	B
5	H
7	E

Brought in for join on 5.
In memory after join on 5.

S

1	x
1	z
2	v
4	r
4	u
5	s
6	t
9	w

In memory after join on 5.

Buffer

5	B	R
5	s	S
5	H	extra
		extra

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
5	H	s

Done as 7 (R) and 6,9 (S) do not match.

Sort-Join Analysis

The sort-join algorithm performs ***5*(B(R)+B(S))*** disk operations.

- *4*B(R)+4*B(S)* to perform the external merge sort on relations.
  - Counting the cost to output relations after sort - hence, *4*B* not *3*B*.
- *1*B(R)+1*B(S)* as each block of each relation read in merge phase to perform join.
- Algorithm limited to relations where ***B(R)<=M***^{2} and ***B(S)<=M***^{2}.

The algorithm can use all the main memory buffers *M* to merge tuples with the same key value.

- If more tuples exist with the same key value than can fit in memory, then we could perform a nested-loop join just on the tuples with that given key value.
  - Also possible to do a one-pass join if the tuples with the key value for one relation all fit in memory.

Sort-Based Join Algorithm Algorithm #1 - Example

Let relations *R* and *S* occupy 6,000 and 3,000 blocks respectively. Let *M* = 101 blocks.

Simple sort-join algorithm cost:

= *5*(B(R)+B(S))* = **45,000 disk I/Os**

- Algorithm works because 6,000<=101^2 and 3,000 <=101^2.

- Requires that there is no join value *y* where the total # of tuples from *R* and *S* with value *y* occupies more than 101 blocks.

Block nested-loop join cost:

= *B(S) + B(S)*B(R)/(M-1)* = 183,000 (*S* as smaller relation)

= *B(S) + B(S)*B(R)/(M-1)* = 186,000 (*S* as larger relation)

or approximately **180,000 disk I/Os**

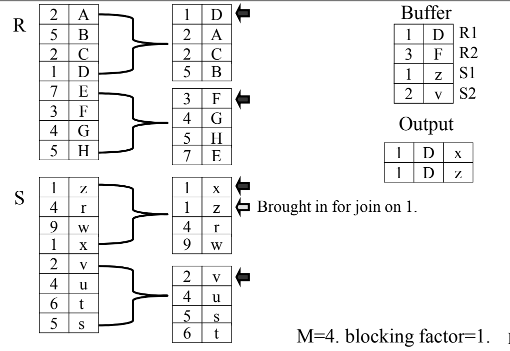
Sort-Merge-Join Algorithm

Idea: Merge the sorting steps and join steps to save disk I/Os.

Algorithm:

- 1) Create sorted sublists of size *M* using *Y* as the sort key for both *R* and *S*.
- 2) Buffer first block of all sublists in memory.
  - Assumes no more than *M* sublists in total.
- 3) Find the smallest value *y* of attribute(s) *Y* in all sublists.
- 4) Identify all tuples in *R* and *S* with value *y*.
  - May be able to buffer some of them if currently using less than *M* buffers.
- 5) Output the join of all tuples of *R* and *S* that share value *y*.

Sort-Merge-Join Example



R

2	A
5	B
2	C
1	D
7	E
3	F
4	G
5	H

1	D
2	A
2	C
5	B

3	F
4	G
5	H
7	E

S

1	z
4	r
9	w
1	x
2	v
4	u
6	t
5	s

1	x
1	z
4	r
9	w

Brought in for join on 1.

2	v
4	u
5	s
6	t

Buffer

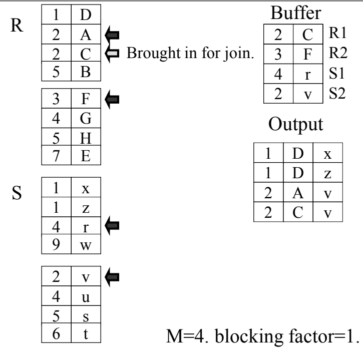
1	D	R1
3	F	R2
1	z	S1
2	v	S2

Output

1	D	x
1	D	z

M=4. blocking factor=1.

Sort-Merge-Join Example (2)



R

1	D
2	A
2	C
5	B

3	F
4	G
5	H
7	E

S

1	x
1	z
4	r
9	w

2	v
4	u
5	s
6	t

Brought in for join.

Buffer

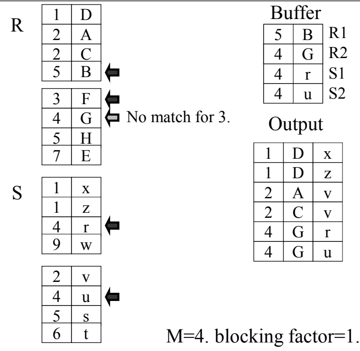
2	C	R1
3	F	R2
4	r	S1
2	v	S2

Output

1	D	x
1	D	z
2	A	v
2	C	v

M=4. blocking factor=1.

Sort-Merge-Join Example (3)



R

1	D
2	A
2	C
5	B

3	F
4	G
5	H
7	E

S

1	x
1	z
4	r
9	w

2	v
4	u
5	s
6	t

No match for 3.

Buffer

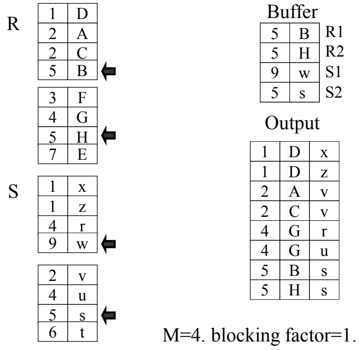
5	B	R1
4	G	R2
4	r	S1
4	u	S2

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u

M=4. blocking factor=1.

Sort-Merge-Join Example (4)



R

1	D
2	A
2	C
5	B

3	F
4	G
5	H
7	E

S

1	x
1	z
4	r
9	w

2	v
4	u
5	s
6	t

Buffer

5	B	R1
5	H	R2
9	w	S1
5	s	S2

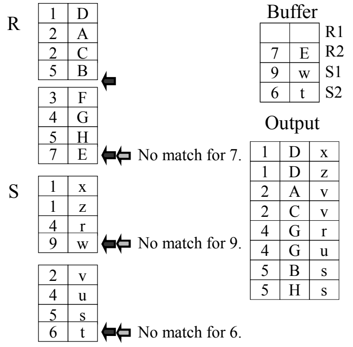
Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
5	H	s

M=4. blocking factor=1.

Sort-Merge-Join Example (5)

Done!



R

1	D
2	A
2	C
5	B

3	F
4	G
5	H
7	E

S

1	x
1	z
4	r
9	w

2	v
4	u
5	s
6	t

No match for 7.

No match for 9.

No match for 6.

Buffer

		R1
7	E	R2
9	w	S1
6	t	S2

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
5	H	s

Sort-Merge-Join Analysis

Sort-merge-join algorithm performs ***3*(B(R)+B(S))*** disk I/Os.

- *2*B(R)+2*B(S)* to create the sublists for each relation.
- *1*B(R)+1*B(S)* as each block of each relation read in merge phase to perform join.

The algorithm is limited to relations where ***B(R)+B(S)<=M²***.

Sort-Merge-Join Example

Let relations *R* and *S* occupy 6,000 and 3,000 blocks respectively. Let *M* = 101 blocks.

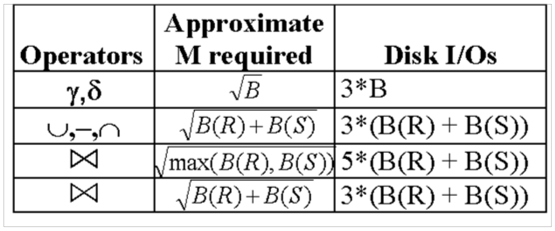
Merge-sort-join algorithm cost:

= *3*(B(R)+B(S))* = **27,000 disk I/Os**

- Algorithm works because 6,000+3,000<=101^2.
- # of memory blocks for sublists = 90
- 11 blocks free to use where there exists multiple join records with same key value *y*.

Summary of Sorting Based Methods

Performance of sorting based methods:



Operators	Approximate M required	Disk I/Os
γ,δ	$\sqrt{B}$	3*B
$\cup,-,\cap$	$\sqrt{B(R)+B(S)}$	3*(B(R) + B(S))
$\bowtie$	$\sqrt{\max(B(R),B(S))}$	5*(B(R) + B(S))
$\bowtie$	$\sqrt{B(R)+B(S)}$	3*(B(R) + B(S))

Hashing-based Two-Pass Algorithms

Hashing-based two-pass algorithms use a hash function to group all tuples with the same key in the same bucket.

The basic process is as follows:

- 1) Use a hash function on each tuple to hash the tuple using a key to a ***bucket*** (or ***partition***).
- 2) Perform the required operation by working on one bucket at a time. If there are *M* buffers available, *M-1* is the number of buckets.

We start with the general external hash partitioning algorithm.

Partitioning Using Hashing Algorithm

1) Partition relation *R* using *M* buffers into *M-1* buckets of roughly equal size.

2) Use a buffer for the input, and one buffer for each of the *M-1* buckets.

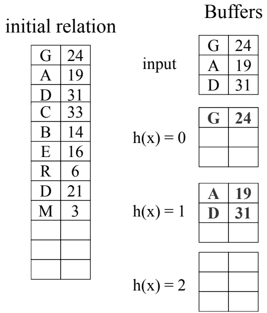
3) When a tuple of relation *R* is read, it is hashed using the hash function *h(x)* and stored in the appropriate bucket.

4) As output buffers (for the buckets) are filled they are written to disk. As the input buffer for *R* is exhausted, a new block is read.

The cost of the algorithm is ***2*B(R)***.

Partitioning using Hashing Example

M=4, bfr=3, h(x) = x % 3 (Hash on column #2.)



Buffers

initial relation

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

G	24
A	19
D	31

h(x) = 0

G	**24**

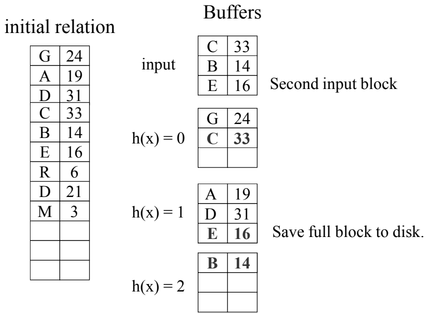
h(x) = 1

A	**19**
D	**31**

h(x) = 2

Partitioning using Hashing Example (2)

M=4, bfr=3, h(x) = x % 3 (Hash on column #2.)



Buffers

initial relation

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

C	33
B	14
E	16

Second input block

h(x) = 0

G	24
C	**33**

h(x) = 1

A	19
D	31
E	**16**

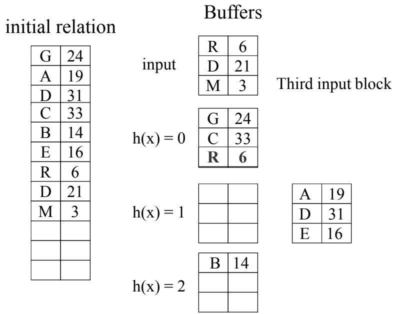
Save full block to disk.

h(x) = 2

B	**14**

Partitioning using Hashing Example (3)

M=4, bfr=3, h(x) = x % 3 (Hash on column #2.)



Buffers

initial relation

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

R	6
D	21
M	3

Third input block

h(x) = 0

G	24
C	33
R	**6**

h(x) = 1

A	19
D	31
E	16

h(x) = 2

B	14

Partitioning using Hashing Example (4)

M=4, bfr=3, h(x) = x % 3 (Hash on column #2.)

initial relation

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

Buffers

input

R	6
D	21
M	3

Third input block

h(x) = 0

D	**21**
M	**3**

G	24
C	33
R	6

h(x) = 1

A	19
D	31
E	16

h(x) = 2

B	14

Duplicate Elimination Using Hashing

Algorithm (two-pass):

- ◆Partition tuples of *R* using hashing and *M-1* buckets.
- ◆Two copies of the same tuple will hash to the same bucket.
- ◆One-pass algorithm can be used on each bucket to eliminate duplicates by loading entire bucket into memory.

Analysis:

- ◆If all buckets are approximately the same size, each bucket R_i will be of size *B(R)/(M-1)*.
- ◆The two-pass algorithm will work if ***B(R) <= M*(M-1)***.
- ◆The # of disk operations is the same as for sorting, ***3*B(R)***.

Grouping and Aggregation Using Hashing

Algorithm (two-pass):

- ◆Partition tuples of *R* using hashing and *M-1* buckets.
- ◆The hash function should *ONLY* use the grouping attributes.
- ◆Tuples with the same values of the grouping attributes will hash to the same bucket.
- ◆A one-pass algorithm is used on each bucket to perform grouping/aggregation by loading the entire bucket into memory.

The two-pass algorithm will work if ***B(R) <= M*(M-1)***.

On the second pass, we only need store one record per group.

- ◆Thus, even if a bucket size is larger than *M*, we may be able to process it if all the group records in the bucket fit into *M* buffers.

The number of disk operations is ***3*B(R)***.

Grouping using Hashing Question

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Calculate the output for a query that groups by the given integer attribute and returns a count of the # of records that contains that attribute.

Assume M=4 and blocking factor=2.

Set Operations Using Hashing

Set operations can be done using a hash-based algorithm.

- ◆Start by hash partitioning *R* and *S* into *M-1* buckets.
- ◆Perform a one-pass algorithm for the set operation on each of the buckets produced.

All algorithms perform ***3*(B(R) + B(S))*** disk operations.

Algorithms require that ***min(B(R),B(S)) <= M***^{2}, since one of the operands must fit in memory after partitioning into buckets in order to perform the one-pass algorithm.

Hash Partitioning Question

Question: Given ***M*** memory buffers, how many hash buckets are used when hash partitioning?

A) 1

B) ***M*** -1

C) ***M***

D) ***M*** +1

Hash-Join Algorithm

Hashing can be used to join two relations *R(X,Y)* and *S(Y,Z)*.

Algorithm:

- ◆Hash partition *R* and *S* using the hash key *Y*.
- ◆If any tuple t_R of *R* will join with a tuple t_S of *S*, then t_R will be in bucket R_i and, t_S will be in bucket S_i. (same bucket index)
- ◆For each bucket pair *i*, load the smaller bucket R_i or S_i into memory and perform a one-pass join.

Important notes for hash-based joins:

- ◆The smaller relation is called the ***build relation***, and the other relation is the ***probe relation***. We will assume *S* is smaller.
- ◆The size of the smaller relation dictates the number of partitioning steps needed.

Hash Join Example
Partition Phase

R

2	A
5	B
2	C
1	D
7	E
3	F
4	G
5	H

S

1	z
4	r
9	w
1	x
2	v
4	u
6	t
5	s

Partitions for R

h(x) = 0

3	F

h(x) = 1

1	D		4	G
7	E			

h(x) = 2

2	A		2	C
5	B		5	H

Partitions for S

h(x) = 0

9	w
6	t

h(x) = 1

1	z		1	x
4	r		4	u

h(x) = 2

2	v
5	s

M=4, bfr=2, h(x) = x % 3

Hash Join Example
Join Phase on Partition 1

Partition 1 for R

h(x) = 1

1	D		4	G
7	E			

Partition 1 for S

h(x) = 1

1	z		1	x
4	r		4	u

Buffers

1	D
7	E

4	G

1	z
4	r

1	x
4	u

Output

1	D	x
1	D	z
4	G	r
4	G	u

Note that both relations fit entirely in memory, but can perform join by having only one relation in memory and reading 1 block at a time from the other one.

Hash-Join Analysis

The hash-join algorithm performs ***3*(B(R)+B(S))*** disk I/Os.

- ◆*2*B(R)+2*B(S)* to perform the hash partitioning on the relations.
- ◆*1*B(R)+1*B(S)* as each block of each relation read in to perform join (one bucket at a time).

Algorithm limited to relations where ***min(B(R),B(S))<=M***^{2}.

Hash-Join Example

Let relations *R* and *S* occupy 6,000 and 3,000 blocks respectively. Let *M* = 101 blocks.

Hash-join algorithm cost:

= *3*(B(R)+B(S))* = **27,000 disk I/Os**

- Average # of blocks per bucket is 60 (for *R*) and 30 (for *S*).

- Algorithm works because min(60,30)<=101.

Hybrid-Hash Join Algorithm

Hybrid hash join uses any extra space beyond what is needed for buffers for each bucket to store one of the buckets in memory. This reduces the number of I/Os. Idea:

- ◆Assume that we need *k* buckets in order to guarantee that the partitions of the smaller relation *S* fit in memory after partitioning.
- ◆Of the *M* buffers, allocate k-1 buffers for each of the buckets except the first one. Expected bucket size is *B(S)/k.*
- ◆Give bucket 0 the rest of the buffers (*M-k+1*) to store its tuples in memory. The rest of the buckets are flushed to disk files.
- ◆When hash relation *R*, if tuple *t* of *R* hashes to bucket 0, we can join it immediately and produce output. Otherwise, we put it in the buffer for its partition (and flush this buffer as needed).
- ◆After read *R,* process all on-disk buckets using a one-pass join.

Hybrid-Hash Join Analysis

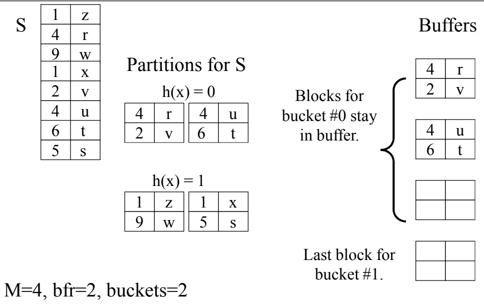
This approach saves two disk I/Os for every block of the buckets of S that remain in memory.

Overall cost is:

$$(3-\frac{2M}{B(S)})(B(R)+B(S))$$

⇨ Note: We are making the simplification that the in-memory partition takes up all of memory *M* (in practice it gets *M-k+1*) buffers. This is usually a small difference for large *M* and small *k*.

Hash Join Example Partition Phase



S

1	z
4	r
9	w
1	x
2	v
4	u
6	t
5	s

Partitions for S

h(x) = 0

4	r	4	u
2	v	6	t

h(x) = 1

1	z	1	x
9	w	5	s

Buffers

4	r
2	v

4	u
6	t

Blocks for bucket #0 stay in buffer.

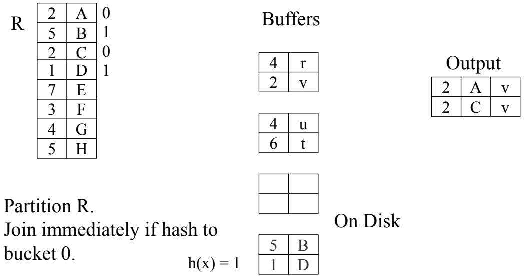
Last block for bucket #1.

M=4, bfr=2, buckets=2

Keep bucket 0 in memory.

Bucket 0 can use up to 3 blocks.

Hash Join Example Buffered Join Phase



R

2	A	0
5	B	1
2	C	0
1	D	1
7	E	
3	F	
4	G	
5	H	

Buffers

4	r
2	v

4	u
6	t

h(x) = 1

5	B
1	D

On Disk

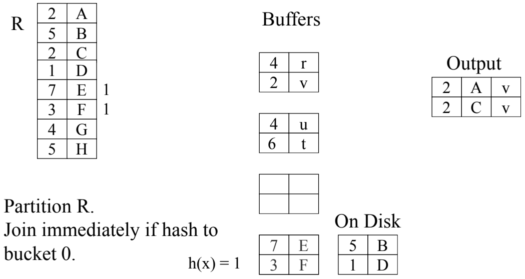
Output

2	A	v
2	C	v

Partition R.

Join immediately if hash to bucket 0.

Hash Join Example Buffered Join Phase (2)



R

2	A	
5	B	
2	C	
1	D	
7	E	1
3	F	1
4	G	
5	H	

Buffers

4	r
2	v

4	u
6	t

h(x) = 1

7	E
3	F

On Disk

5	B
1	D

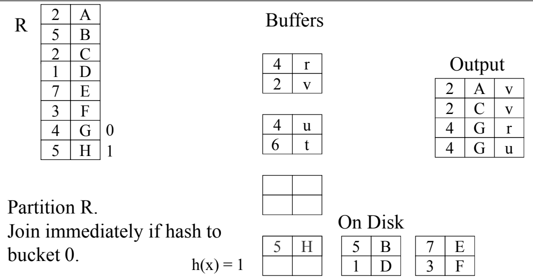
Output

2	A	v
2	C	v

Partition R.

Join immediately if hash to bucket 0.

Hash Join Example Buffered Join Phase (3)



R

2	A	
5	B	
2	C	
1	D	
7	E	
3	F	
4	G	0
5	H	1

Buffers

4	r
2	v

4	u
6	t

h(x) = 1

5	H

On Disk

5	B	7	E
1	D	3	F

Output

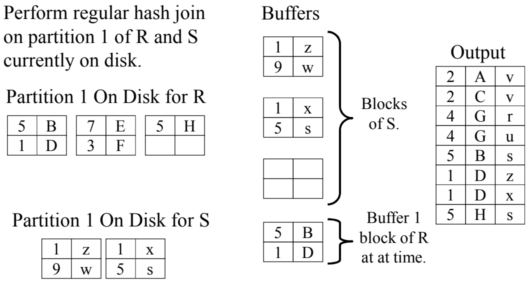
2	A	v
2	C	v
4	G	r
4	G	u

Partition R.

Join immediately if hash to bucket 0.

Hash Join Example Disk Join Phase

Perform regular hash join on partition 1 of R and S currently on disk.



Buffers

Partition 1 On Disk for R

5	B	7	E	5	H
1	D	3	F		

Partition 1 On Disk for S

1	z	1	x
9	w	5	s

1	z
9	w

1	x
5	s

Blocks of S.

5	B
1	D

Buffer 1 block of R at at time.

Output

2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
1	D	z
1	D	x
5	H	s

Hash-Join Example Analysis

Hash-join algorithm cost **26 total block I/Os**. (Expected 24!)

- Total partition cost = 17 I/Os.
  - Partition of *R*: 4 reads, 5 writes.
  - Partition of *S*: 4 reads, 4 writes.
- Join phase cost = 9 reads (5 for *R* and 4 for *S*).
- Total cost of 26 is larger than expected cost of 24 because tuples did not hash evenly into buckets.

Hybrid-hash join algorithm cost **16 block I/Os**. (Expected 16!)

- Partition cost is 12 disk I/Os.
  - Partition of *R*: 4 reads, 2 writes (for bucket #1) (do not write last block).
  - Partition of *S*: 4 reads, 2 writes.
  - Memory join is free.
- Regular hash join: 2 read for *R*, 2 reads for *S*.

Hash Join Question

Question: Select a true statement.

A) The probe relation is the smallest relation.

B) The probe relation has an in-memory hash table built on its tuples.

C) The build relation is the smallest relation.

D) The probe relation is buffered in memory.

Multi-Pass Hash Joins

We have examined two-pass hash joins where only one partitioning step is needed. Hash-based joins can be extended to support larger relations by performing recursive partitioning.

Unlike sort-based joins where the number of partition steps is determined by the larger relation, for hash-based joins the number of partition steps is determined by the smaller build relation. This is often a significant advantage.

Adaptive Hash Join

During its execution, a join algorithm may be required to give up memory or be given memory from the execution system based on system load and execution factors.

An ***adaptive hash join*** algorithm [Zeller90] is able to adapt to changing memory conditions by allowing the partition buckets to change in size.

Basic idea (that makes it different from hybrid hash):

- Each partition can hold a certain number of buffers and all are initially memory resident. Tuples are inserted as usual.
- When memory is exhausted, a victim partition is flushed to disk and frozen (no new tuples can be added). This is repeated until partitioning is complete.

The description of adaptive join algorithm above is for the simpler version called dynamic hash join [DeWitt95].

Local Research
Skew-Aware Hash Join

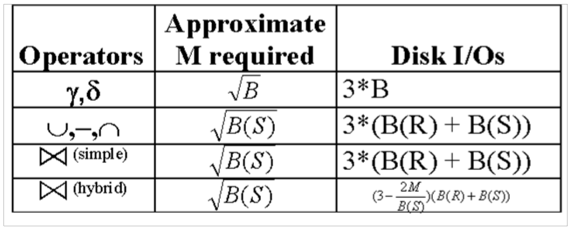
Skew-aware hash join [Cutt09] selects the build partition tuples to buffer based on their frequency of occurrence in the probe relation.

When data is skewed (some data is much more common than others), this can have a significant improvement on the number of I/Os performed.

Algorithm optimization is currently in PostgreSQL hash join implementation.

Summary of Hashing Based Methods

Performance of hashing based methods:



Operators	Approximate M required	Disk I/Os
γ,δ	$\sqrt{B}$	3*B
$\cup,-,\cap$	$\sqrt{B(S)}$	3*(B(R) + B(S))
$\bowtie$ (simple)	$\sqrt{B(S)}$	3*(B(R) + B(S))
$\bowtie$ (hybrid)	$\sqrt{B(S)}$	$(3-\frac{2M}{B(S)})(B(R)+B(S))$

Comparison of Sorting versus Hashing Methods

Speed and memory requirements for the algorithms are almost identical. However, there are some differences:

- 1) Hash-based algorithms for binary operations have size requirement based on the size of the smaller of the two arguments rather than the sum of the argument sizes.
- 2) Sort-based algorithms allow us to produce the result in sorted order and use this for later operations.
- 3) Hash-based algorithms depend on the buckets being of equal size.
  - Hard to accomplish in practice, so generally, we limit bucket sizes to slightly smaller values to handle this variation.
- 4) Sort-based algorithms may be able to write sorted sublists to consecutive disk blocks saving rotational and seek times.
- 5) Both algorithms can save disk access time by writing/reading several blocks at once if memory is available.

Comparison of Sorting versus Hashing Methods (2)

- 6) Hash based joins are usually best if neither of the input relations are sorted or there are no indexes for equi-join.

Note that for small relation sizes, the simple nested-block join is faster than both the sorting and hashing based methods!

Join Question

Question: For what percentage of join memory available compared to the smaller relation size (i.e. M / B(S)) is block nested-loop join faster than hybrid hash join?

A) 0% to 10%

B) 10% to 25%

C) 25% to 50%

D) 50% to 100%

Index-Based Algorithms

Index-based algorithms use index structures to improve performance.

Indexes are especially useful for selections instead of performing a table scan.

For example, if the query is $\sigma_{a=v}$(R), and we have a B+-tree index on attribute *a* then the query cost is the time to access the index plus the time to read all the records with value *v*.

Index-Based Algorithms Query Costs Example

Let *B(R)* = 1,000 and *T(R)* = 20,000.

- That is, *R* has 20,000 tuples, and 20 tuples fit in a block.

Let *a* be an attribute of *R*, and evaluate the operation $\sigma_{a=v}$(R).

Evaluation Cases (# of disk I/Os):

- 1) *R* is clustered and index is not used = *B(R)* = 1000.
- 2) *V(R,a)* = 100 and use a clustering index=(20,000/100)/20= 10.
- 3) *V(R,a)* = 10 and use a non-clustering index = 20,000/10 = 2000 I/Os.
  - Must retrieve on average 2000 tuples for condition and possible that each tuple can be on a separate block.
- 4) *V(R,a)* = 20,000 (*a* is a key) - cost = 1 (+ index cost)

Cost Estimate Example with Indices

Query: $\sigma_{Major = \text{“CS”}}(Student)$

- Evaluate query cost assuming:
  - *V(Student, Major)*=4 (4 different Major values: "BA", "BS", "CS", "ME")
  - *B(Student)* = 500, *T(Student)* = 10,000, blocking factor = 20
- Cost estimate for query using *Major* index:
  - Since *V(Student,Major)*=4 , we expect that 10000/4 = 2,500 tuples have "CS" as the value for the *Major* attribute.
  - If the index is a clustering index, 2,500/20 = 125 block reads are required to read the *Student* tuples. (What would be the strategy?)
  - If the index is non-clustering, how many index blocks are read?
    - The height of the index depends on the # of unique entries which is 4. The B+-tree index would be of depth 1. We can assume that it would be in main memory, only the pointer blocks would have to be read. If a leaf node can store 200 pointers, then 2,000/200 = 13 index blocks would have to be read.
  - How many block I/Os in total for a non-clustering index?
  - How does this compare to doing a sequential scan?

Index-Based Algorithms Complicated Selections

Indexes can also be used to answer other types of selections:

- 1) A B-tree index allows efficient range query selections such as $\sigma_{a<=v}(R)$ and $\sigma_{a>=v}(R)$.
- 2) Complex selections can be implemented by an index-scan followed by another selection on the tuples returned.

Complex selections involve more than one condition connected by boolean operators.

- For example, $\sigma_{a=v \text{ AND } b>=10}(R)$ is a complex selection.

This query can be evaluated by using the index to find all tuples where *a*=*v*, then apply the second condition *b* >=10 to all the tuples returned from the index scan.

Index Joins

An index can also be used to speed-up certain types of joins.

Consider joining *R(X,Y)* and *S(Y,Z)* by using a nested-block join with *S* as the outer relation and *R* as the inner relation. We have an index on *R* for attribute(s) *Y*.

We can modify the algorithm that for every tuple *t* of *S*, we use the value of *Y* for *t* to lookup in the index for *R*.

This lookup will return only the tuples of *R* with matching values for *Y*, and we can compute the join with *t*.

Cost: ***T(S)*(T(R)/V(R,Y))*** tuples will be read

- *T(S)*T(R)/V(R,Y)* (non-clustered)
- *T(S)*B(R)/V(R,Y)* (clustered)
- Not always faster than a nested-block join! Makes sense when *V(R,Y)* is large and *R* is small.

Index-Merge Join Variant Example

Join *R(X,Y)* with *S(Y,Z)* by sorting R and read S using index.

- with *B(R)*=6000,*B(S)*=3000,*M* = 101 blocks.

1) Assume only index on *S* for *Y*:

Sort *R* first = *2*B(R)* = 12,000 disk I/Os (to form sorted sublists)

Merge with *S* using 60 buffers for *R* and 1 for index block for *S*.

Read all of *R* and *S* = 9,000 disk I/Os

Total = **21,000 disk I/Os**

2) Assume index for both *R* and *S* for *Y*:

Do not need to sort either *R* or *S*.

Read all of *R* and *S* = **9,000 disk I/Os**

Remember that there is always a small overhead of accessing the index itself.

Multi-Pass Algorithms

The two-pass algorithms based on sorting and hashing can be extended to any number of passes using recursion.

- Each pass partitions the relations into smaller pieces.
- Eventually, the partitions will entirely fit in memory (base case).

Analysis of *k*-pass algorithm:

- Memory requirements $M = (B(R))^{1/k}$
- Maximum relation size $B(R) <= M^k$
- Disk operations = *2*k*B(R)*
  - Note: If do not count write in final *k* pass, cost is: *2*k*B(R) - B(R)*.

Parallel Operators

We have discussed implementing selection, project, join, duplicate elimination, and aggregation on a single processor.

Many algorithms have been developed to exploit parallelism in the form of additional CPUs, memory, and hard drives.

We will not study these algorithms, but realize that they exist.

Join Algorithms that Produce Results Early

One of the problems of join algorithms is that they must read either one (hash-based) or both (sort-based) relations before any join input can be produced.

- This is not desirable in interactive settings where the goal is to get answers to the user as soon as possible.

Research has been performed to define algorithms that can produced results early and are capable of joining sources over the Internet. These algorithms also handle network issues.

- **Sort-based algorithms:** Progressive-Merge join [Dittrich02] produces results early by sorting and joining both inputs simultaneously in memory.
- **Hash-based algorithms**: Hash-merge join [Mokbel04], X-Join [Urban00] and Early Hash Join [Lawrence05] use two hash tables. As tuples arrive they are inserted in their table and probe the other.

Research Challenges

There are several open research challenges for database algorithms:

- 1) Optimizing algorithms for cache performance
- 2) Examination of CPU costs as well as I/O costs
- 3) The migration to solid-state drives changes many of the algorithm assumptions.
  - Random I/O does NOT cost more any more which implies algorithms that performed more random I/O (index algorithms) may be more competitive on the new storage technology.

Conclusion

Every relational algebra operator may be implemented using many different algorithms. The performance of the algorithms depend on the data, the database structure, and indexes.

Classify algorithms by:

- 1) **# of passes:** Algorithms only have a fixed buffered memory area to use, and may require one, two, or more passes depending on input size.
- 2) **Type of operator:** selection, projection, grouping, join.
- 3) Algorithms can be based on sorting, hashing, or indexing.

The actual algorithm is chosen by the query optimizer based on its query plan and database statistics.

Major Objectives

The "One Things":

- Diagram the components of a query processor and explain their function (slide #5).
- Calculate block access for one-pass algorithms.
- Calculate block accesses for tuple & block nested joins.
- Perform two-pass sorting methods including all operators, sort-join and sort-merge-join and calculate performance.
- Perform two-pass hashing methods including all operators, hash-join and hybrid hash-join and calculate performance.

Major Theme:

- The query processor can select from many different algorithms to execute each relational algebra operator. The algorithm selected depends on database characteristics.

Objectives

- Explain the goal of query processing.
- Review: List the relational and set operators.
- Diagram and explain query processor components.
- Explain how index and table scans work and calculate the block operations performed.
- Write an iterator in Java for a relational operator.
- List the tuple-at-a-time relational operators.
- Illustrate how one-pass algorithms for selection, project, grouping, duplicate elimination, and binary operators work and be able to calculate performance and memory requirements.
- Calculate performance of tuple-based and block-based nested loop joins given relation sizes (memorize formulas!).

Objectives (2)

- Perform and calculate performance of two-pass sorting based algorithms - sort-merge algorithm, set operators, sort-merge-join/sort-join.
- Perform and calculate performance of two-pass hashing based algorithms - hash partitioning, operation implementation and performance, hash join, hybrid-hash join.
- Compare/contrast sorting versus hashing methods
- Calculate performance of index-based algorithms - cost estimate, complicated selections, index joins
- Explain how two-pass algorithms are extended to multi-pass algorithms.
- List some recent join algorithms: adaptive, hash-merge, XJoin, progressive-merge.

COSC 404
Database System Implementation

Query Optimization

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Query Optimization Overview

The query processor performs four main tasks:

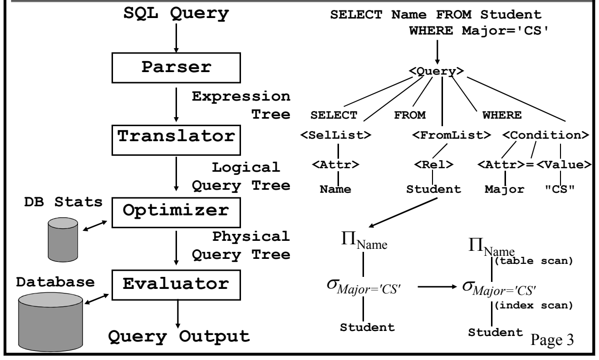
1) Verifies the correctness of an SQL statement

2) Converts the SQL statement into relational algebra

3) Performs heuristic and cost-based optimization to build the more efficient execution plan

4) Executes the plan and returns the results

Components of a Query Processor



Query Processor Components The Parser

The role of the parser is to convert an SQL statement represented as a string of characters into a parse tree.

A ***parse tree*** consists of nodes, and each node is either an:

- ***Atom*** - lexical elements such as words (`WHERE`), attribute or relation names, constants, operator symbols, etc.
- ***Syntactic category*** - are names for query subparts.
  - E.g. <SFW> represents a query in select-from-where form.

Nodes that are atoms have no children. Nodes that correspond to categories have children based on one of the rules of the grammar for the language.

A Simple SQL Grammar

A ***grammar*** is a set of rules dictating the structure of the language. It exactly specifies what strings correspond to the language and what ones do not.

- Compilers are used to parse grammars into parse trees.
  - Same process for SQL as programming languages, but somewhat simpler because the grammar for SQL is smaller.

Our simple SQL grammar will only allow queries in the form of `SELECT-FROM-WHERE`.

- We will not support grouping, ordering, or `SELECT DISTINCT`.
- We will support lists of attributes in the `SELECT` clause, lists of relations in the `FROM` clause, and conditions in the `WHERE` clause.

Simple SQL Grammar

```
<Query> ::= <SFW>
<Query> ::= ( <Query> )

<SFW> ::= SELECT <SelList> FROM <FromList> WHERE
                                            <Condition>
<SelList> ::= <Attr>
<SelList> ::= <Attr> , <SelList>

<FromList> ::= <Rel>
<FromList> ::= <Rel> , <FromList>

<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Tuple> IN <Query>
<Condition> ::= <Attr> = <Attr>
<Condition> ::= <Attr> LIKE <Value>
<Condition> ::= <Attr> = <Value>
<Tuple> ::= <Attr> // Tuple may be 1 attribute
```

A Simple SQL Grammar Discussion

The syntactic categories of `<Attr>`, `<Rel>`, and `<Value>` are special because they are not defined by the rules of the grammar.

- ◆`<Attr>` - must be a string of characters that matches an attribute name in the database schema.
- ◆`<Rel>` - must be a character string that matches a relation name in the database schema.
- ◆`<Value>` - is some quoted string that is a legal SQL pattern or a valid numerical value.

Query Example Database

```
Student(Id,Name,Major,Year)
Department(Code,DeptName,Location)
```

Student Relation

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

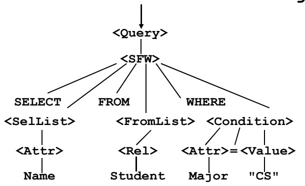
Department Relation

Code	DeptName	Location
BA	Bachelor of Arts	English Building
BS	Bachelor of Science	Physics Building
CS	Computer Science	MacLean Hall
ME	Mechanical Engineering	Engineering Building

Query Parsing Example

Return all students who major in computer science.

`SELECT Name FROM Student WHERE Major='CS'`



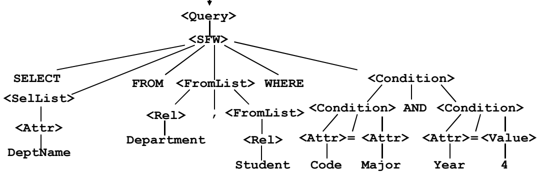
<Query>
<SFW>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attr>
<Rel>
<Attr>=<Value>
Name
Student
Major
"CS"

```
Rules applied:
<Query> ::= <SFW>
<SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
<SelList> ::= <Attr>                (<Attr> = "Name")
<Condition> ::= <Attr> = <Value>    (<Attr>="Major", <Value>="CS")
<FromList> ::= <Rel>                (<Rel> = "Student")
```

Query Parsing Example 2

Return all departments who have a 4th year student.

```
SELECT DeptName FROM Department, Student
WHERE Code = Major AND Year = 4
```



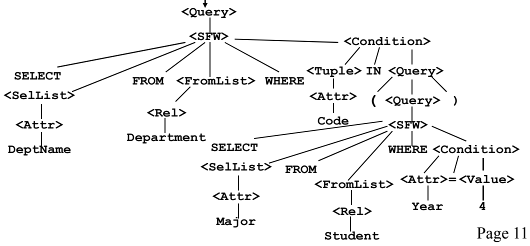
<Query>
<SFW>
SELECT
<SelList>
<Attr>
DeptName
FROM
<FromList>
WHERE
<Condition>
<Rel>
Department
,
<FromList>
<Rel>
Student
<Condition>
AND
<Condition>
<Attr>= <Attr>
Code
Major
<Attr>=<Value>
Year
4

Can you determine what rules are applied?

Query Parsing Example 3

Return all departments who have a 4th year student.

```
SELECT DeptName FROM Department WHERE Code IN
   (SELECT Major FROM Student WHERE Year=4)
```



<Query>
<SFW>
<Condition>
SELECT
<SelList>
<Attr>
DeptName
FROM
<FromList>
<Rel>
Department
WHERE
<Tuple> IN <Query>
<Attr>
Code
(<Query>)
<SFW>
SELECT
<SelList>
<Attr>
Major
FROM
<FromList>
<Rel>
Student
WHERE
<Condition>
<Attr>=<Value>
Year
4

Query Processor Components
The Parser Functionality

The parser converts an SQL string to a parse tree.

- ◆This involves breaking the string into tokens.
- ◆Each token is matched with the grammar rules according to the current parse tree.
- ◆Invalid tokens (not in grammar) generate an error.
- ◆If there are no rules in the grammar that apply to the current SQL string, the command will be flagged to have a syntax error.

We will not concern ourselves with how the parser works. However, we will note that the parser is responsible for checking for ***syntax errors*** in the SQL statement.

- ◆That is, the parser determines if the SQL statement is valid according to the grammar.

Query Processor Components The Preprocessor

The preprocessor is a component of the parser that performs ***semantic validation***.

The preprocessor runs ***after*** the parser has built the parse tree. Its functions include:

- ◆Mapping views into the parse tree if required.
- ◆Verify that the relation and attribute names are actually valid relations and attributes in the database schema.
- ◆Verify that attribute names have a corresponding relation name specified in the query. (Resolve attribute names to relations.)
- ◆Check types when comparing with constants or other attributes.

If a parse tree passes syntax and semantic validation, it is called a ***valid parse tree***.

A valid parse tree is sent to the logical query processor, otherwise an error is sent back to the user.

Query Parsing Question

Question: Select a true statement.

A) The SQL grammar contains information to validate if a given field name is a valid field in the database.

B) The preprocessor runs before the parsing process.

C) SQL syntax errors are checked by the preprocessor.

D) Errors indicating a table does not exist are generated by the preprocessor.

Query Processor Components Translator

The ***translator***, or ***logical query processor***, is the component that takes the parse tree and converts it into a logical query tree.

A ***logical query tree*** is a tree consisting of relational operators and relations. It specifies what operations to apply and the order to apply them. A logical query tree does *not* select a particular algorithm to implement each relational operator.

We will study some rules for how a parse tree is converted into a logical query tree.

Parse Trees to Logical Query Trees

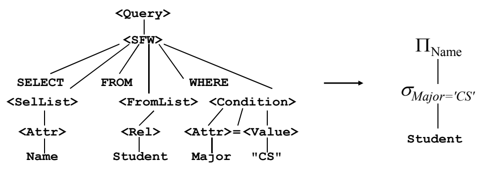
The simplest parse tree to convert is one where there is only one select-from-where (`<SFW>`) construct, and the `<Condition>` construct has no nested queries.

The logical query tree produced consists of:

- ◆1) The cross-product (×) of all relations mentioned in the `<FromList>` which are inputs to:
- ◆2) A selection operator, σ_C, where *C* is the `<Condition>` expression in the construct being replaced which is the input to:
- ◆3) A projection, π_L, where *L* is the list of attributes in the `<SelList>`.

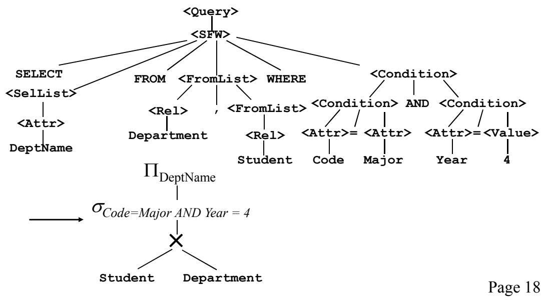
Parse Tree to Logical Tree Example

```
SELECT Name FROM Student WHERE Major='CS'
```



Parse Tree to Logical Tree Example 2

```
SELECT DeptName FROM Department, Student
     WHERE Code = Major AND Year = 4
```



Converting Nested Parse Trees to Logical Query Trees

Converting a parse tree that contains a nested query is slightly more challenging.

A nested query may be ***correlated*** with the outside query if it must be re-computed for every tuple produced by the outside query. Otherwise, it is ***uncorrelated***, and the nested query can be converted to a non-nested query using joins.

We will define a two-operand selection operator σ that takes the outer relation ***R*** as one input (left child), and the right child is the condition applied to each tuple of ***R***.

- ◆The condition is the subquery involving `IN`.

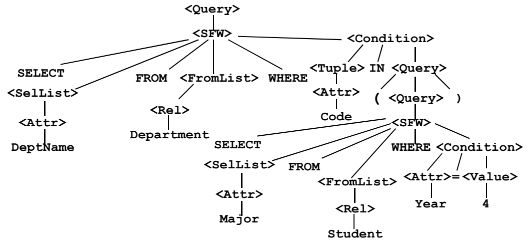
Converting Nested Parse Trees to Logical Query Trees (2)

The nested subquery translation algorithm involves defining a tree from root to leaves as follows:

- ◆1) Root node is a projection, π_L, where *L* is the list of attributes in the `<SelList>` of the outer query.
- ◆2) Child of root is a selection operator, σ_C, where *C* is the `<Condition>` expression in the outer query ignoring the subquery.
- ◆3) The two-operand selection operator σ with left-child as the cross-product (×) of all relations mentioned in the `<FromList>` of the outer query, and right child as the `<Condition>` expression for the subquery.
- ◆4) The subquery itself involved in the `<Condition>` expression is translated to relational algebra.

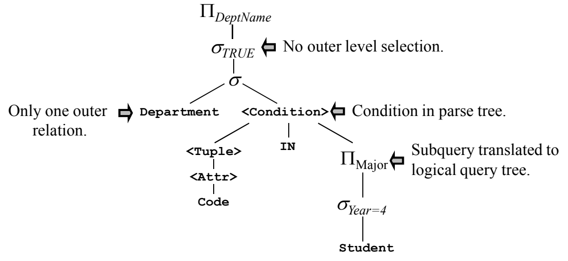
Parse Tree to Logical Tree Example 3

```
SELECT DeptName FROM Department WHERE Code IN
  (SELECT Major FROM Student WHERE Year=4)
```



Parse Tree to Logical Tree Example 3 (2)

```
SELECT DeptName FROM Department WHERE Code IN
  (SELECT Major FROM Student WHERE Year=4)
```



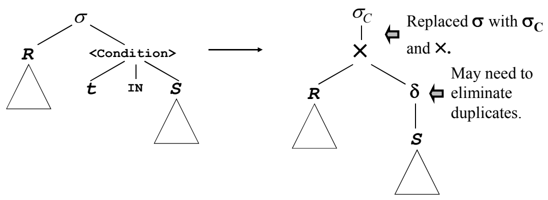
Converting Nested Parse Trees to Logical Query Trees (3)

Now, we must remove the two-operand selection and replace it by relational algebra operators.

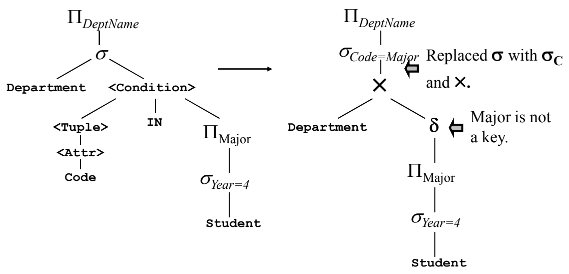
Rule for replacing two-operand selection (uncorrelated):

- ◆Let *R* be the first operand, and the second operand is a `<Condition>` of the form *t* IN *S*. (*S* is uncorrelated subquery.)
- ◆1) Replace `<Condition>` by the tree that is expression for *S*.
  - ⇨May require applying duplicate elimination if expression has duplicates.
- ◆2) Replace two-operand selection by one-argument selection, σ_C, where *C* is the condition that equates each component of the tuple *t* to the corresponding attribute of relation *S*.
- ◆3) Give σ_C an argument that is the product of *R* and *S*.

Parse Tree to Logical Tree Conversion


σ
R
<Condition>
t
IN
S
σ_C
Replaced σ with σ_C and ×.
×
R
δ
May need to eliminate duplicates.
S

Parse Tree to Logical Tree Example 3 (3)



Correlated Nested Subqueries

Translating correlated subqueries is more difficult because the result of the subquery depends on a value defined outside the query itself.

Correlated subqueries may require the subquery to be evaluated for each tuple of the outside relation as an attribute of each tuple is used as the parameter for the subquery.

- We will not study translation of correlated subqueries.

Example:

```
Return all students that are more senior than the
              average for their majors.

SELECT Name FROM Student s WHERE year >
  (SELECT Avg(Year) FROM student AS s2
       WHERE s.major = s2.major)
```

Logical Query Tree Question

Question: True or False: A logical query tree has relational algebra operators and specifies the algorithm used for each of them.

A) True

B) False

Logical Query Tree Question (2)

Question: True or False: A logical query tree is the final plan used for executing the query.

A) True

B) False

Parsing Review Question

Build the parse tree for the following SQL query then convert it into a logical query tree.

```
SELECT Name, DeptName FROM Department, Student
      WHERE Code = Major and Code = 'CS'
```

Optimizing the Logical Query Plan

The translation rules converting a parse tree to a logical query tree do not always produce the best logical query tree.

It is possible to optimize the logical query tree by applying relational algebra laws to convert the original tree into a more efficient logical query tree.

Optimizing a logical query tree using relational algebra laws is called ***heuristic optimization*** because the optimization process uses common conversion techniques that result in more efficient query trees in most cases, but not always.

- The optimization rules are heuristics.

We begin with a summary of relational algebra laws.

Relational Algebra Laws

Just like there are laws associated with the mathematical operators, there are laws associated with the relational algebra operators.

These laws often involve the properties of:

- ◆***commutativity*** - operator can be applied to operands independent of order.
  - ⇨E.g. A + B = B + A - The "+" operator is commutative.
- ◆***associativity*** - operator is independent of operand grouping.
  - ⇨E.g. A + (B + C) = (A + B) + C - The "+" operator is associative.

Associative and Commutative Operators

The relational algebra operators of cross-product (×), join ($\bowtie$), set and bag union ($\cup_S$ and $\cup_B$), and set and bag intersection ($\cap_S$ and $\cap_B$) are all associative and commutative.

Commutative	**Associative**
$R \times S = S \times R$	$(R \times S) \times T = R \times (S \times T)$
$R \bowtie S = S \bowtie R$	$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
$R \cup S = S \cup R$	$(R \cup S) \cup T = R \cup (S \cup T)$
$R \cap S = S \cap R$	$(R \cap S) \cap T = R \cap (S \cap T)$

Laws Involving Selection

1) Complex selections involving `AND` or `OR` can be broken into two or more selections: (***splitting laws***)

$$\sigma_{C_1 \texttt{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$$
$$\sigma_{C_1 \texttt{ OR } C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$$

2) Selection operators can be evaluated in any order:

$$\sigma_{C_1 \texttt{ AND } C_2}(R) = \sigma_{C_2}(\sigma_{C_1}(R)) = \sigma_{C_1}(\sigma_{C_2}(R))$$

3) Selection can be done before or after set operations and joins:

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$
$$\sigma_C(R - S) = \sigma_C(R) - S = \sigma_C(R) - \sigma_C(S)$$
$$\sigma_C(R \cap S) = \sigma_C(R) \cap S = \sigma_C(R) \cap \sigma_C(S)$$
$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$$

Laws Involving Selection and Joins

1) Selection and cross-product can be converted to a join:

$$\sigma_C(R \times S) = R \bowtie_C S$$

2) Selection and join can also be combined:

$$\sigma_C(R \bowtie_D S) = R \bowtie_{C \text{ AND } D} S$$

Laws Involving Selection Examples

1) Example relation is *R(a,b,c)*.

Given expression: $\sigma_{(a=1 \texttt{ OR } a=3) \texttt{ AND } b<c}(R)$

Can be converted to: $\sigma_{a=1 \texttt{ OR } a=3}(\sigma_{b<c}(R))$

then to: $\sigma_{a=1}(\sigma_{b<c}(R)) \cup \sigma_{a=3}(\sigma_{b<c}(R))$

There is another way to divide up the expression. What is it?

2) Given relations *R(a,b)* and *S(b,c)*.

Given expression: $\sigma_{(a=1 \texttt{ OR } a=3) \texttt{ AND } b<c}(R \bowtie S)$

Can be converted to: $\sigma_{(a=1 \texttt{ OR } a=3)}\ \sigma_{b<c}(R \bowtie S))$

then to: $\sigma_{(a=1 \texttt{ OR } a=3)}(R \bowtie \sigma_{b<c}(S))$

finally to: $\sigma_{(a=1 \texttt{ OR } a=3)}(R) \bowtie \sigma_{b<c}(S)$

Is there anything else we could do?

Laws Involving Projection

Like selections, it is also possible to push projections down the logical query tree. However, the performance gained is less than selections because projections just reduce the number of attributes instead of reducing the number of tuples.

- ◆Unlike selections, it is common for a pushed projection to also remain where it is.

General principle: We may introduce a projection anywhere in an expression tree, as long as it eliminates only attributes that are never used by any of the operators above, and are not in the result of the entire expression.

Note that discussion considers ***bag projection*** as normally implemented in SQL (duplicates are not eliminated).

Laws Involving Projection (2)

1) Projections can be done before joins as long as all attributes required are preserved.

$$\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$$
$$\pi_L(R \bowtie S) = \pi_L((\pi_M(R) \bowtie \pi_N(S))$$

⇨ *L* is a set of attributes to be projected. *M* is the attributes of *R* that are either join attributes or are attributes of *L*. *N* is the attributes of *S* that are either join attributes or attributes of *L*.

2) Projection can be done before bag union but *NOT* before set union or set/bag intersection and difference.

$$\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$$

3) Projection can be done before selection.

$$\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$$

4) Only the last projection operation is needed:

$$\pi_L(\pi_M(R)) = \pi_L(R)$$

Laws Involving Projection Examples

1) Given relations *R(a,b,c)* and *S(c,d,e).*

Given expression: $\pi_{b,d}(R \bowtie S)$

Can be converted to: $\pi_{b,d}(\pi_{b,c}(R) \bowtie \pi_{c,d}(S))$

2) Using *R(a,b,c)* and the expression: $\pi_b(\sigma_{a=5}(R))$

Can be converted to: $\pi_b(\sigma_{a=5}(\pi_{a,b}(R))$

Laws Involving Duplicate Elimination

Duplicate elimination (δ) can be done before many operators.

Note that δ(*R*) = *R* occurs when *R* has no duplicates:

- 1) *R* may be a stored relation with a primary key.
- 2) *R* may be the result after a grouping operation.

Laws for pushing duplicate elimination operator (δ):

$$\delta(R \times S) = \delta(R) \times \delta(S)$$
$$\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$$
$$\delta(R \bowtie_D S) = \delta(R) \bowtie_D \delta(S)$$
$$\delta(\sigma_C(R) = \sigma_C(\delta(R))$$

Duplicate elimination (δ) can also be pushed through bag intersection, but not across union, difference, or projection.

$$\delta(R \cap_B S) = \delta(R) \cap_B \delta(S)$$

Laws Involving Grouping

The grouping operator (γ) laws depend on the aggregate operators used.

There is one general rule, however, that grouping subsumes duplicate elimination:

$$\delta(\gamma_L(R)) = \gamma_L(R)$$

The reason is that some aggregate functions are unaffected by duplicates (`MIN` and `MAX`) while other functions are (`SUM`, `COUNT`, and `AVG`).

Relational Algebra Question

Question: How many of the following equivalences are **true**? Let *C* = predicate with only *R* attributes, *D* = predicate with only *S* attributes, and *E* = predicate with only *R* and *S* attributes.

$$\sigma_{C \text{ AND } D}(R \bowtie \text{S}) = \sigma_C(R) \bowtie \sigma_D(S)$$
$$\sigma_{C \text{ AND } D \text{ AND } E}(R \bowtie \text{S}) = \sigma_E(\sigma_C(R) \bowtie \sigma_D(S))$$
$$\sigma_{C \text{ OR } D}(R \bowtie \text{S}) = [\sigma_C(R) \bowtie \text{S}] \cup_S [\text{R} \bowtie \sigma_D(S)]$$
$$\pi_L(R \cup_S S) = \pi_L(R) \cup_S \pi_L(S)$$

A) 0

B) 1

C) 2

D) 3

E) 4

Relational Algebra Question

Give examples to show that:

- a) *Bag* projection cannot be pushed below set union.

$$\pi_L(R \cup_S S) \mathrel{!=} \pi_L(R) \cup_S \pi_L(S)$$

- b) Duplicate elimination cannot be pushed below *bag* projection.

$$\delta(\pi_L(R)) \mathrel{!=} \pi_L(\delta(R))$$

Heuristic Query Optimization

Heuristic query optimization takes a logical query tree as input and constructs a more efficient logical query tree by applying equivalence preserving relational algebra laws.

Equivalence preserving transformations insure that the query result is identical before and after the transformation is applied. Two logical query trees are ***equivalent*** if they produce the same result.

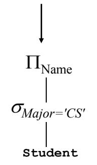
Note that heuristic optimization does not always produce the most efficient logical query tree as the rules applied are only heuristics!

Rules of Heuristic Query Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations.
2. Move selection operations down the query tree for the earliest possible execution.
3. Replace Cartesian product operations that are followed by a selection condition by join operations.
4. Execute first selection and join operations that will produce the smallest relations.
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed.

Heuristic Optimization Example

```
SELECT Name FROM Student WHERE Major="CS"
```

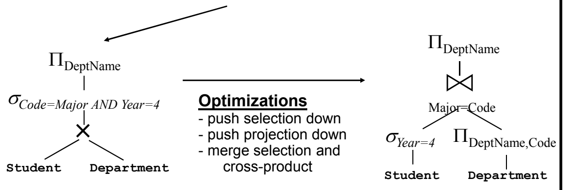


No optimization possible.

$\pi_{\text{Name}}(\sigma_{Major=\text{“CS”}}(Student))$

Heuristic Optimization Example 2

```
SELECT DeptName FROM Department, Student
      WHERE Code = Major AND Year = 4
```



Optimizations
- push selection down
- push projection down
- merge selection and cross-product

Original:

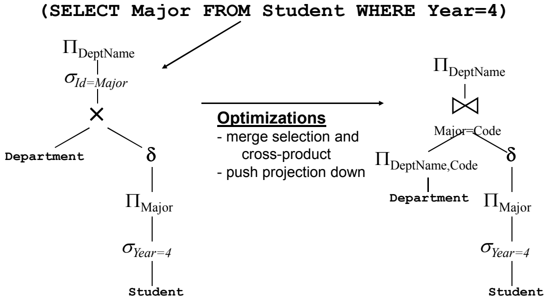
$\pi_{\text{DeptName}}(\sigma_{Code=Major\ \text{AND}\ Year=4}(Student \times Department))$

Optimized:

$\pi_{\text{DeptName}}((\sigma_{Year=4}(Student)) \bowtie_{Code=Major} (\pi_{\text{DeptName,Code}}(Department)))$

Heuristic Optimization Example 3

```
SELECT DeptName FROM Department WHERE Id IN
  (SELECT Major FROM Student WHERE Year=4)
```



Optimizations
- merge selection and cross-product
- push projection down

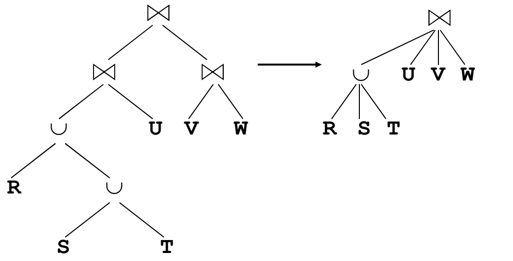
Canonical Logical Query Trees

A ***canonical logical query tree*** is a logical query tree where all associative and commutative operators with more than two operands are converted into multi-operand operators.

- ◆This makes it more convenient and obvious that the operands can be combined in any order.

This is especially important for joins as the order of joins may make a significant difference in the performance of the query.

Canonical Logical Query Tree Example



Original Query Tree

Canonical Query Tree

Canonical Query Tree Question

Question: What does the original logical query tree imply that the canonical tree does not?

A) an order of operator execution

B) the algorithms used for each relational operator

C) the sizes of each input

Query Optimization Physical Query Plan

A ***physical query plan*** is derived from a logical query plan by:

- 1) Selecting an order and grouping for operations like joins, unions, and intersections.
- 2) Deciding on an algorithm for each operator in the logical query plan.
  - e.g. For joins: Nested-loop join, sort join or hash join
- 3) Adding additional operators to the logical query tree such as sorting and scanning that are not present in the logical plan.
- 4) Determining if any operators should have their inputs materialized for efficiency.

Whether we perform cost-based or heuristic optimization, we eventually must arrive at a physical query tree that can be executed by the evaluator.

Query Optimization Heuristic versus Cost Optimization

To determine when one physical query plan is better than another, we must have an estimate of the cost of the plan.

Heuristic optimization is normally used to pick the best logical query plan.

Cost-based optimization is used to determine the best physical query plan given a logical query plan.

Note that both can be used in the same query processor (and typically are). Heuristic optimization is used to pick the best logical plan which is then optimized by cost-based techniques.

Query Optimization Estimating Operation Cost

To determine when one physical query plan is better than another for cost-based optimization, we must have an estimate of the cost of a physical query plan.

Note that the query optimizer will very rarely know the exact cost of a query plan because the only way to know is to execute the query itself!

- Since the cost to execute a query is much greater than the cost to optimize a query, we cannot execute the query to determine its cost!

It is important to be able to estimate the cost of a query plan without executing it based on statistics and general formulas.

Query Optimization Estimating Operation Cost (2)

Statistics for ***base relations*** such as ***B(R)***, ***T(R)***, and ***V(R,a)*** are used for optimization and can be gathered directly from the data, or estimated using statistical gathering techniques.

One of the most important factors determining the cost of the query is the size of the intermediate relations. An ***intermediate relation*** is a relation generated by a relational algebra operator that is the input to another query operator.

- The final result is not an intermediate relation.

The goal is to come up with general rules that estimate the sizes of intermediate relations that give accurate estimates, are easy to compute, and are consistent.

- There is no one set of agreed-upon rules!

Estimating Operation Cost Estimating Projection Sizes

Calculating the size of a relation after the projection operation is easy because we can compute it directly.

- ◆Assuming we know the size of the input, we can calculate the size of the output based on the size of the input records and the size of the output records.
- ◆The projection operator decreases the size of the tuples, not the number of tuples.

For example, given relation ***R(a,b,c)*** with size of *a* = size of *b* = 4 bytes, and size of *c* = 100 bytes. *T(R)* = 10000 and unspanned block size is 1024 bytes. If the projection operation is $\Pi_{a,b}$, what is the size of the output ***U*** in blocks?

```
T(U) = 10000.  Output tuples are 8 bytes long.
bfr = 1024/8 = 128  B(U) = 10000/128 = 79
B(R) = 10000 / (1024/108) = 1112
Savings = (B(R) - B(U))/B(R)*100% = 93%
```

Estimating Operation Cost Estimating Selection Sizes

A selection operator generally decreases the number of tuples in the output compared to the input. By how much does the operator decrease the input size?

The ***selectivity*** (***sf***) is the fraction of tuples selected by a selection operator. Common cases and their selectivities:

- ◆1) Equality: $S = \sigma_{a=v}(R)$ - *sf* = 1/*V(R,a)* *T(S) = T(R)/V(R,a)*
  - ⇨Reason: Based on the assumption that values occur equally likely in the database. However, estimate is still the best *on average* even if the values *v* for attribute *a* are not equally distributed in the database.
- ◆2) Inequality: $S = \sigma_{a<v}(R)$ - *sf* = 1/3 *T(S) = T(R)*/3
  - ⇨Reason: On average, you would think that the value should be *T(R)*/2. However, queries with inequalities tend to return less than half the tuples, so the rule compensates for this fact.
- ◆3) Not equals: $S = \sigma_{a!=v}(R)$ - *sf* = 1 *T(S) = T(R)*
  - ⇨Reason: Assume almost all tuples satisfy the condition.

Estimating Operation Cost Estimating Selection Sizes (2)

Simple selection clauses can be connected using AND or OR.

A complex selection operator using AND ($\sigma_{a=10 \text{ AND } b<20}(R)$) is the same as a cascade of simple selections ($\sigma_{a=10}(\sigma_{b<20}(R))$.

The selectivity is the ***product*** of the selectivity of the individual clauses.

Example: Given *R(a,b,c)* and $S = \sigma_{a=10 \text{ AND } b<20}(R)$, what is the best estimate for *T(S)*? Assume *T(R)*=10,000 and *V(R,a)* = 50.

```
The filter a=10 has selectivity of 1/V(R,a)=1/50.
The filter b<20 has selectivity of 1/3.
Total selectivity = 1/3 * 1/50 = 1/150.
T(S) = T(R)* 1/150 = 67
```

Estimating Operation Cost Estimating Selection Sizes (3)

For complex selections using OR ($S = \sigma_{C1 \text{ OR } C2}(R)$), the # of output tuples can be estimated by the ***sum*** of the # of tuples for each condition.

- ◆Measuring the selectivity with OR is less precise, and simply taking the sum is often an overestimate.

A better estimate assumes that the two clauses are independent, leading to the formula:

$$n * (1 - (1-m_1/n) * (1 - m_2/n))$$

- ⇨m_1 and m_2 are the # of tuples that satisfy C_1 and C_2 respectively.
- ⇨*n* is the number of tuples of *R* (i.e. *T(R)*).
- ⇨1-m_1/n and 1-m_2/n are the fraction of tuples that do not satisfy C_1 (resp. C_2). The product of these numbers is the fraction that do not satisfy either condition.

Estimating Operation Cost Estimating Selection Sizes (4)

Example: Given *R(a,b,c)* and $S = \sigma_{a=10 \text{ OR } b<20}(R)$, what is the best estimate for *T(S)*? Assume *T(R)*=10,000 and *V(R,a)* = 50.

```
The filter a=10 has selectivity of 1/V(R,a)=1/50.
The filter b<20 has selectivity of 1/3.
Total selectivity = (1 - (1 - 1/50)(1 - 1/3)) = .3466
T(S) = T(R) *.3466 = 3466

Simple method results in T(S) = 200 + 3333 = 3533.
```

Estimating Operation Cost Estimating Join Sizes

We will only study estimating the size of natural join.

- ◆Other types of joins are equivalent or can be translated into a cross-product followed by a selection.

The two relations joined are ***R(X,Y)*** and ***S(Y,Z)***.

- ◆We will assume *Y* consists of only one attribute.

The challenge is we do not know how the set of values of *Y* in *R* relate to the values of *Y* in *S*. There are some possibilities:

- ◆1) The two sets are disjoint. Result size = **0**.
- ◆2) *Y* may be a foreign key of *R* joining to a primary key of *S*. Result size in this case is ***T(R)***.
- ◆3) Almost all tuples of *R* and *S* have the same value for *Y*, so result size in the worst case is ***T(R)*T(S)***.

Estimating Join Sizes (2)

The result size of joining relations ***R(X,Y)*** and ***S(Y,Z)*** can be approximated by:

$$\frac{T(R) * T(S)}{\max(V(R,Y), V(S,Y))}$$

- Argument:
  - ⇨ Every tuple of *R* has a 1/*V(S,Y)* chance of joining with every tuple of *S*. On average then, each tuple of *R* joins with *T(S)*/*V(S,Y)* tuples. If there are *T(R)* tuples of *R*, then the expected size is *T(R)* * *T(S)*/*V(S,Y)*.
  - ⇨ A symmetric argument can be made from the perspective of joining every tuple of *S*. Each tuple has a 1/*V(R,Y)* chance of joining with every tuple of *R*. On average, each tuple of *R* joins with *T(R)/V(R,Y)* tuples. The expected size is then *T(S)* * *T(R)*/*V(R,Y)*.
  - ⇨ In general, we choose the smaller estimate for the result size (divide by the maximum value).

Estimating Operation Cost Estimating Join Sizes Example

Example:

- *R(a,b)* with *T(R)* = 1000 and *V(R,b)* = 20.
- *S(b,c)* with *T(S)* = 2000, *V(S,b)* = 50, and *V(S,c)* = 100
- *U(c,d)* with *T(U)* = 5000 and *V(U,c)* = 500

Calculate the natural join *R* $\bowtie$ *S* $\bowtie$ *U*.

1) (*R* $\bowtie$ *S*) $\bowtie$ *U* -

T(*R* $\bowtie$ *S*) = *T*(*R*)*T*(*S*)/max(*V*(*R*,*b*),*V*(*S*,*b*))

= 1000 * 2000 / 50 = 40,000

Now join with *U*.

Final size = *T*(*R* $\bowtie$ *S*)**T*(*U*)/max(*V*(*R* $\bowtie$ *S*,*c*),*V*(*U*,*c*))

= 40000 * 5000 / 500 = 400,000

Now, calculate the natural join like this: *R* $\bowtie$ (*S* $\bowtie$ *U*).

- Which of the two join orders is better?

Estimating Join Sizes Estimating V(R,a)

The database will keep statistics on the number of distinct values for each attribute *a* in each relation *R*, ***V(R,a)***.

When a sequence of operations is applied, it is necessary to estimate *V(R,a)* on the intermediate relations.

For our purposes, there will be three common cases:

- *a* is the primary key of *R* then **V(R,a) = T(R)**
  - ⇨ The number of distinct values is the same as the # tuples in *R*.
- *a* is a foreign key of *R* to another relation *S* then **V(R,a) = T(S)**
  - ⇨ In the worst case, the number of distinct values of *a* cannot be larger than the number of tuples of *S* since *a* is a foreign key to the primary key of *S*.
- If a selection occurs on relation R before a join, then **V(R,a)** after the selection is the same as **V(R,a)** before selection.
  - ⇨ This is often strange since V(R,a) may be greater than # of tuples in intermediate result! V(R,a) <> # of tuples in result.

Estimating Operation Cost Estimating Sizes of Other Operators

The size of the result of set operators, duplicate elimination, and grouping is hard to determine. Some estimates are below:

- Union
  - ⇨ bag union = sum of two argument sizes
  - ⇨ set union = minimum is the size of the largest relation, maximum is the sum of the two relations sizes. Estimate by taking average of min/max.
- Intersection
  - ⇨ minimum is 0, maximum is size of smallest relation. Take average.
- Difference
  - ⇨ Range is between *T(R)* and *T(R)* - *T(S)* tuples. Estimate: *T(R)* - 1/2**T(S)*
- Duplicate Elimination
  - ⇨ Range is 1 to *T(R)*. Estimate by either taking smaller of 1/2**T(R)* or product of all $V(R,a_i)$ for all attributes a_i.
- Grouping
  - ⇨ Range and estimate is similar to duplicate elimination.

Query Optimization Cost-Based Optimization

Cost-based optimization is used to determine the best physical query plan given a logical query plan.

The cost of a query plan in terms of disk I/Os is affected by:

- 1) The logical operations chosen to implement the query (the logical query plan).
- 2) The sizes of the intermediate results of operations.
- 3) The physical operators selected.
- 4) The ordering of similar operations such as joins.
- 5) If the inputs are materialized.

Cost-Based Optimization Obtaining Size Estimates

The cost calculations for the physical operators relied on reasonable estimates for *B(R)*, *T(R)*, and *V(R,a)*.

Most DBMSs allow an administrator to explicitly request these statistics be gathered. It is easy to gather them by performing a scan of the relation. It is also common for the DBMS to gather these statistics independently during its operation.

- Note that by answering one query using a table scan, it can simultaneously update its estimates about that table!

It is also possible to produce a histogram of values for use with *V(R,a)* as not all values are equally likely in practice.

- Histograms display the frequency that attribute values occur.

Since statistics tend not to change dramatically, statistics are computed only periodically instead of after every update.

Using Size Estimates in Heuristic Optimization

Size estimates can also be used during heuristic optimization.

In this case, we are not deciding on a physical plan, but rather determining if a given logical transformation will make sense.

By using statistics, we can estimate intermediate relation sizes (independent of the physical operator chosen), and thus determine if the logical transformation is useful.

Using Size Estimates in Cost-based Optimization

Given a logical query plan, the simplest algorithm to determine the best physical plan is an exhaustive search.

In an ***exhaustive search***, we evaluate the cost of every physical plan that can be derived from the logical plan and pick the one with minimum cost.

The time to perform an exhaustive search is extremely long because there are many combinations of physical operator algorithms, operator orderings, and join orderings.

Using Size Estimates in Cost-based Optimization (2)

Since exhaustive search is costly, other approaches have been proposed based on either a top-down or bottom-up approach.

Top-down algorithms start at the root of the logical query tree and pick the best implementation for each node starting at the root.

Bottom-up algorithms determine the best method for each subexpression in the tree (starting at the leaves) until the best method for the root is determined.

Cost-Based Optimization Choosing a Selection Method

In building the physical query plan, we will have to pick an algorithm to evaluate each selection operator.

Some of our choices are:

- table scan
- index scan

There also may be several variants of each choice if there are multiple indexes.

We evaluate the cost of each choice and select the best one.

Cost-Based Optimization Choosing a Join Method

In building the physical query plan, we will have to pick an algorithm to evaluate each join operator:

- **nested-block join** - one-pass join or nested-block join used if reasonably sure that relations will fit in memory.
- **sort-join** is good when arguments are sorted on the join attribute or there are two or more joins on the same attribute.
- **index-join** may be used when an index is available.
- **hash-join** is generally used if a multipass join is required, and no sorting or indexing can be exploited.

Cost-Based Optimization Pipelining versus Materialization

The default action for iterators is ***pipelining*** when the inputs to the operator provide results a tuple-at-a-time.

However, some operators require the ability to scan the inputs multiple times. This requires the input operator to be able to support ***rescan***.

An alternative to using rescan is to materialize the results of an input to disk. This has two benefits:

- Operators do not have to implement rescan.
- It may be more efficient to compute the result once, save it to disk, then read it from disk multiple times than to re-compute it each time.

Plans can use a materialization operator at any point to materialize the output of another operator.

Selecting a Join Order

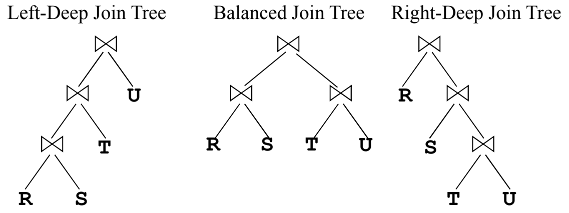
Since joins are the most costly operation, determining the best possible join order will result in more efficient queries.

Selecting a join order is most important if we are performing a join of three or more relations. However, a join of two relations can be evaluated in two different ways depending on which relation is chosen to be the left argument.

- ◆Some algorithms (such as nested-block join and one-pass join) are more efficient if the left argument is the smaller relation.

A ***join tree*** is used to graphically display the ***join order***.

Join Tree Examples



Join Tree Question

Question: How many possible join tree shapes (different trees ignoring relations at leaves) are there for joining 4 nodes?

A) 3
B) 4
C) 5
D) 6
E) 8

Join Tree Question (2)

Question: Assuming that every relation can join with every other relation, how many distinct join trees (considering different relations at leaf nodes) are there for joining 4 nodes?

A) 256
B) 120
C) 60
D) 20
E) 5

Cost-Based Optimization Selecting a Join Order

Dynamic programming is used to select a join order.

Algorithm to find best join tree for a set of *n* relations:

- ◆1) Find the best plan for each relation.
  - ⇨File scan, index scan
- ◆2) Find the best plan to combine pairs of relations found in step #1. If have two plans for *R* and *S*, test
  - ⇨$R \bowtie S$ and $S \bowtie R$ for all types of joins.
  - ⇨May also consider interesting sort orders.
- ◆3) Of the plans produced involving two relations, add a third relation and test all possible combinations.

In practice the algorithm works top down recursively and remembers the best subplans for later use.

Join Order Dynamic Programming Algorithm

```
// S is set of relations to join
procedure findBestPlan(S)
{     if (bestplan[S].cost ≠ ∞)          // bestplan stores computed plans
              return bestplan[S];
      // else bestplan[S] has not been computed. Compute it now.
      for each non-empty subset S1 of S such that S1 ≠ S
      {       P1= findBestPlan(S1);
              P2= findBestPlan(S - S1);
              A = best algorithm for join of P1 and P2;
              cost = P1.cost + P2.cost + cost of A;
              if (cost < bestplan[S].cost)
              {       bestplan[S].cost = cost;
                      bestplan[S].plan = P1 ⋈ P2 using A;
              }
      }
      return bestplan[S];
}
```

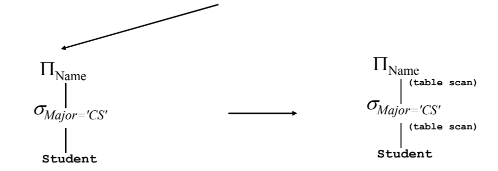
Cost-Based Optimization Example

We will perform cost-based optimization on the three example queries giving the following statistics:

- ◆ *T(Student)* = 200,000 ; *B(Student)* = 50,000
- ◆ *T(Department)* = 4 ; *B(Department)* = 4
- ◆ *V(Student, Major)* = 4 ; *V(Student, Year)* = 4
- ◆ *Student* has B+-tree secondary indexes on *Major* and *Year*, and primary index on *Id*.
- ◆ *Department* has a primary index on *Code*.

Cost-Based Optimization Example

```
SELECT Name FROM Student WHERE Major="CS"
```



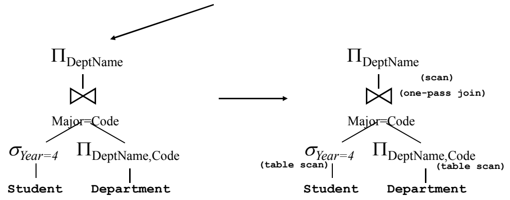
Logical Query Tree

Physical Query Tree

Selection will return *T(Student)*/*V(Student,Major)* = 200,000/4 = 50,000 tuples.
Since tuples are not sorted by *Major*, each read may potentially require reading another block (results in another seek + rotational latency).
Thus, table scan will be more efficient.
Projection performed using table scan of pipelined output from selection.

Cost-Based Optimization Example 2

```
SELECT DeptName FROM Department, Student
WHERE Code = Major AND Year = 4
```



Logical Query Tree

Physical Query Tree

Selection uses table scan again due to high selectivity.
One-pass join chosen as result from *Department* subtree is small. Index-join cannot be used as already performed projection on base relation.

Cost-Based Optimization Example 3

Consider a query involving the join of relations:

- ◆ *Enrolled(StudentID, Year, CourseID)*
- ◆ *Course(CID, Name)*
- ◆ and the relations *Student* and *Department*.
- ◆ That is, *Student* ⋈ *Department* ⋈ *Enrolled* ⋈ *Course*.

Determine the best join ordering given this information:

- ◆ *T(Enrolled)* = 1,000,000; *B(Enrolled)* = 200,000
- ◆ *V(Enrolled,StudentID)* = 180,000 ; *V(Enrolled,CourseID)* = 900
- ◆ *T(Course)* = 1000 ; *B(Course)* = 100

The best join ordering would have the minimum sizes for the intermediate relations, and we would like to perform the join with the greatest selectivity first.

Cost-Based Optimization Example 3 (2)

Possible join pairs and intermediate result sizes:

- ◆ *Student* ⋈ *Department* = 200,000 * 4 / max(4,4) = **200,000**
- ◆ *Student* ⋈ *Enrolled*
  = 200,000*1,000,000 / max(200,000,180,000) = **1,000,000**
- ◆ *Enrolled* ⋈ *Course*
  =1,000,000 * 1,000 / max(900,1000) = **1,000,000**

Conclusion: Join *Student* and *Department* first as it results in smallest intermediate relation. Then, join that result with *Enrolled*, finally join with *Course*.

Cost-based Optimization Question

Question: Would it be better or worse if we joined *Enrolled* with *Course* then joined that with the result of *Student* and *Department*?

A) same

B) better

C) worse

Join Ordering Example

Query:

```
SELECT * FROM Course C, Enrolled E, Student S
   WHERE Year = 4 AND C.cid = 'COSC404' AND
        E.cid = E.cid and E.sid = S.sid
```

Relation statistics:

- *B(C) = 100, B(E) = 200,000, B(S) = 20,000*
- *T(C)* = 1,000 ; *T(E)* = 1,000,000 ; T(S) = 200,000
- Assume block size = 1000 bytes.
- *Tuple sizes: C = 100 bytes ; E = 200 bytes ; S = 100 bytes*
- *V(E,sid)* = 180,000 ; *V(E,cid)* = 900
- *Student* has secondary B-tree index on *Year*.
- *Course* has primary B-tree index on *cid*.

Join Ordering Example (2)

The first step is to calculate best plan for each relation:

Enrolled

- only choice is file scan at cost = 200,000

Course with filter `cid = 'COSC404'`:

- file scan cost = 100
- index scan cost = 1 (assume get record in 1 block with index)
- Best plan = index scan with cost = 1

Student with filter `Year = 4`:

- file scan cost = 20,000
- index scan will return approximately ¼ of records (50,000). If assume each does a block access that is 50,000 cost.
- Best plan = file scan with cost = 20,000

Join Ordering Example (3)

Now calculate all pairs of relations (sets of size two). Test all types of joins (sort, hash, block). Assume left is build input and M= 1000.

Enrolled, Course: (output size tuples = 1111 blocks = 334)

- Enrolled ⋈ Course
  - ⇨ Sort = 600,003 ; Hash = 598,003 ; Block nested = 200,201
- Course ⋈ Enrolled
  - ⇨ Sort = 600,003 ; **Hash = 200,001**; Block nested = 200,001

Enrolled, Student: (output size tuples = 1,000,000 blocks = 300,000)

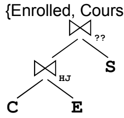
- Enrolled ⋈ Student
  - ⇨ Sort = 660,000 ; Hash = 657,800 ; Block nested = 4,040,000
- Student ⋈ Enrolled
  - ⇨ Sort = 660,000 ; **Hash = 638,000** ; Block nested = 4,220,000

Student, Course (Note: This may not be done if cross-products are not allowed.)

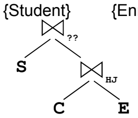
- Student X Course cost = 20,000 output size = 40,000 blocks

Join Ordering Example (4)

{Enrolled, Course}, {Student}

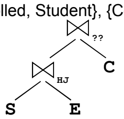


HJ = 20,334
SJ = 61,002
NLJ = 20,334
Overall: 220,335

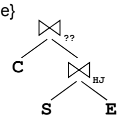


HJ = 58,969
SJ = 61,002
NLJ = 27,014
Overall: 227,015

{Enrolled, Student}, {Course}

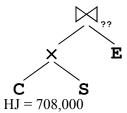


HJ = 898,002
SJ = 900,003
NLJ = 300,301
Overall = 938,301

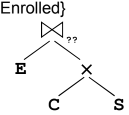


HJ = 300,001
SJ = 900,003
NLJ = 300,001
Overall = 938,001

{Student, Course}, {Enrolled}

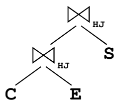


HJ = 708,000
SJ = 720,000
NLJ = 8,240,000
Overall = 728,000



HJ = 717,600
SJ = 720,000
NLJ = 8,240,000
Overall = 737,000

Best plan:



Overall: 220,335

Conclusion

A query processor first parses a query into a parse tree, validates its syntax, then translates the query into a relational algebra ***logical query plan***.

The logical query plan is optimized using ***heuristic optimization*** that uses equivalence preserving transformations.

Cost-based optimization is used to select a join ordering and build an execution plan which selects an implementation for each of the relational algebra operations in the logical tree.

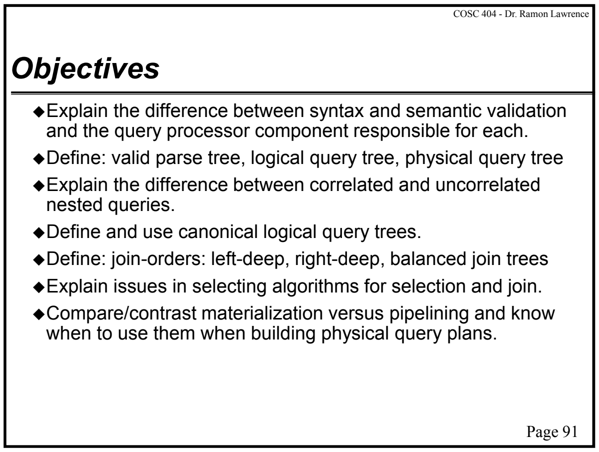
Major Objectives

The "One Things":

- Convert an SQL query to a parse tree using a grammar.
- Convert a parse tree to a logical query tree.
- Use heuristic optimization and relational algebra laws to optimize logical query trees.
- Convert a logical query tree to a physical query tree.
- Calculate size estimates for selection, projection, joins, and set operations.

Major Theme:

- The query optimizer uses heuristic (relational algebra laws) and cost-based optimization to greatly improve the performance of query execution.



Objectives

- Explain the difference between syntax and semantic validation and the query processor component responsible for each.
- Define: valid parse tree, logical query tree, physical query tree
- Explain the difference between correlated and uncorrelated nested queries.
- Define and use canonical logical query trees.
- Define: join-orders: left-deep, right-deep, balanced join trees
- Explain issues in selecting algorithms for selection and join.
- Compare/contrast materialization versus pipelining and know when to use them when building physical query plans.

COSC 404
Database System Implementation

Transaction Management

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Transaction Management Overview

The database system must ensure that the data stored in the database is always ***consistent***.

There are several possible types of ***failures*** that may cause the data to become inconsistent.

A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.

- ◆The input to a transaction is a consistent database, AND the output of the transaction must also be a consistent database.
- ◆A transaction must execute completely or not at all.

Transaction Management Motivating Example

Consider a person who wants to transfer $50 from a savings account with balance $1000 to a checking account with current balance = $250.

- ◆1) At the ATM, the person starts the process by telling the bank to remove $50 from the savings account.
- ◆2) The $50 is removed from the savings account by the bank.
- ◆3) Before the customer can tell the ATM to deposit the $50 in the checking account, the ATM "crashes."

Where has the $50 gone?

It is lost if the ATM did not support transactions!
The customer wanted the withdraw and deposit to both happen in one step, or neither action to happen.

Transaction Definition

A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.

The basic assumption is that when a transaction starts executing the database is consistent, and when it finishes executing the database is still in a consistent state.

- ◆Do not consider malicious or incorrect transactions.
- ◆This assumption is called **The Correctness Principle**.

Note that the database may be inconsistent during transaction execution.

- ◆For the bank example, the $50 is removed from the savings account and is not yet in the checking account at some point in time.

Consistency Definition

A database is ***consistent*** if the data satisfies all constraints specified in the database schema. A ***consistent database*** is said to be in a ***consistent state***.

A ***constraint*** is a predicate (rule) that the data must satisfy.

- ◆Examples:
  - ⇨*StudentID* is a key of relation *Student*.
  - ⇨*StudentID* → *Name* holds in *Student*.
  - ⇨No student may have more than one major.
  - ⇨The field *Major* can only have one of the 4 values: {"BA","BS","CS","ME"}.
  - ⇨The field *Year* must be between 1 and 4.

Note that the database may be internally consistent but not reflect the real-world reality.

Consistency Issues

There are two major challenges in preserving consistency:

- ◆1) The database system must handle ***failures*** of various kinds such as hardware failures and system crashes.
- ◆2) The database system must support ***concurrent execution*** of multiple transactions and guarantee that this concurrency does not lead to inconsistency.

ACID Properties

To preserve integrity, transactions have the following properties:

- ***Atomicity* -** Either all operations of the transaction are properly reflected in the database or none are.
- ***Consistency* -** Execution of a transaction in isolation preserves the consistency of the database.
- ***Isolation* -** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
  - Intermediate transaction results must be hidden from other concurrently executing transactions. That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- ***Durability* -** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction Operations

Since a transaction is a general program, there are an enormous number of potential operations that a transaction can perform.

However, there are two really important operations:

- `read(A,t)` (or `read(A)` when *t* is not important)
  - Read database element *A* into local variable *t*.
- `write(A,t)` (or `write(A)` when *t* is not important)
  - Write the value of local variable *t* to the database element *A*.

For most of the discussion, we will assume that the buffer manager insures that database element is in memory. We could make the memory management more explicit by using:

- `input(A)`
  - Read database element *A* into local memory buffer.
- `output(A)`
  - Write the block containing *A* to disk.

Fund Transfer Transaction Example

Transaction to transfer $50 from account *A* to account *B*:

```
1.    read(A,t)
2.    t := t - 50
3.    write(A,t)
4.    read(B,t)
5.    t := t + 50
6.    write(B,t)
```

Fund Transfer Transaction Example (2)

Atomicity requirement – If the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, or inconsistency will result.

Consistency requirement – The sum of *A* and *B* is unchanged by the execution of the transaction.

Isolation requirement – If between steps 3 and 6, another transaction accesses the partially updated database, it will see an inconsistent database (*A* + *B* is less than it should be).

- Can be ensured trivially by running transactions *serially,* that is one after the other. However, executing multiple transactions concurrently has significant benefits.

Durability requirement – Once the user has been notified that the transaction has completed (i.e., the $50 transfer occurred), the updates by the transaction must persist despite failures.

ACID Properties

Question: Two transactions running at the same time can see each other's updates. What ACID property is violated?

A) atomicity

B) consistency

C) isolation

D) durability

E) none of them

ACID Properties (2)

Question: A company stores a customer's address in the database. The customer moves and does not tell the company to update its database. What ACID property is violated?

A) atomicity

B) consistency

C) isolation

D) durability

E) none of them

Transaction Questions

Example database:

`Student(Id,Name,Major,Year)`

1) Write a transaction to change the name of a student to "Joe Smith." Let *A* represent the database object currently storing the name.

2) Write a transaction to swap the names of two students with names *A* and *B*.

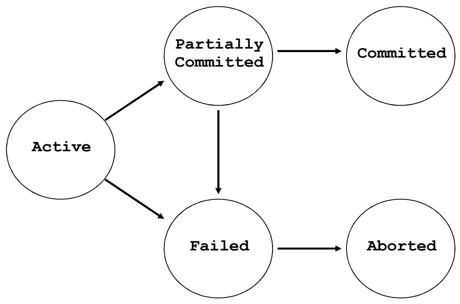
3) Write a transaction to increase the *Year* attribute of all students by 1.

Transaction States

An executing transaction can be in one of several states:

- ***Active*** - is the initial state. The transaction stays in this state while it is executing.
- ***Partially committed* -** A transaction is partially committed after its final statement has been executed.
- ***Failed* -** A transaction enters the failed state after the discovery that normal execution can no longer proceed.
- ***Aborted* -** A transaction is aborted after it has been rolled back and the database restored to its prior state before the transaction. There are two options after abort:
  - ⇨ restart the transaction – only if no internal logical error
  - ⇨ kill the transaction - problem with transaction itself
- ***Committed* -** Commit state occurs after *successful completion*.
  - ⇨ May also consider **terminated** as a transaction state.

Transaction State Diagram



Transaction States

Question: Is it possible for a transaction to be in the aborted and committed states at different times during its lifetime?

A) yes

B) no

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

- Increased processor and disk utilization, leading to better transaction ***throughput****:* one transaction can be using the CPU while another is reading from or writing to the disk.
- Reduced ***average response time*** for transactions as short transactions need not wait behind long ones.

Concurrency control schemes are mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

- We will study concurrency control schemes after examining the notion of correctness of concurrent executions.

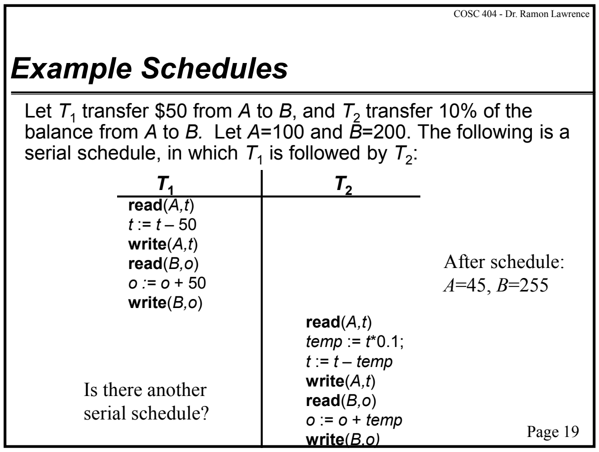
Schedules

A s***chedule*** is the chronological order in which instructions of concurrent transactions are executed.

- A schedule for a set of transactions must consist of all instructions of those transactions.
- We must preserve the order in which the instructions appear in each individual transaction.
- It is useful to think of a schedule as a journal of the database actions. It is a ***historical record*** that the database keeps as it is processing transactions.

A ***serial schedule*** is a schedule where the instructions belonging to each transaction appear together.

- i.e. There is no *interleaving* of transaction operations.
- For *n* transactions, there are *n*! different serial schedules.



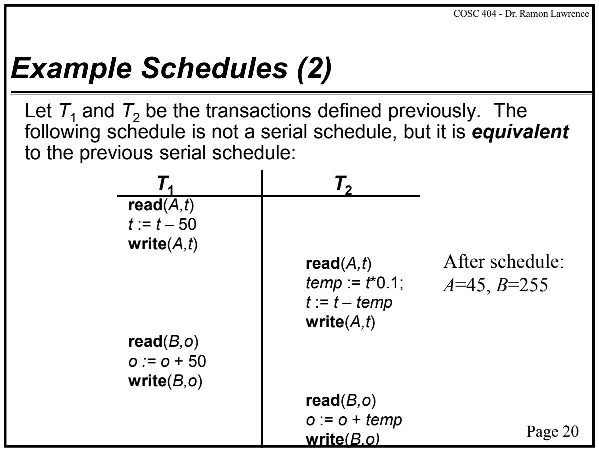
Example Schedules

Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*. Let *A*=100 and *B*=200. The following is a serial schedule, in which T_1 is followed by T_2:

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
write(*A*,*t*)	
read(*B*,*o*)	
o := *o* + 50	
write(*B*,*o*)	
	read(*A*,*t*)
	temp := *t**0.1;
	t := *t* – *temp*
	write(*A*,*t*)
	read(*B*,*o*)
	o := *o* + *temp*
	write(*B*,*o*)

After schedule: *A*=45, *B*=255

Is there another serial schedule?

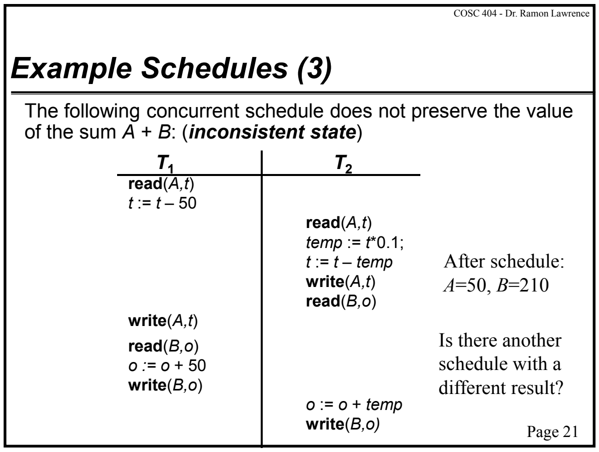


Example Schedules (2)

Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is ***equivalent*** to the previous serial schedule:

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
write(*A*,*t*)	
	read(*A*,*t*)
	temp := *t**0.1;
	t := *t* – *temp*
	write(*A*,*t*)
read(*B*,*o*)	
o := *o* + 50	
write(*B*,*o*)	
	read(*B*,*o*)
	o := *o* + *temp*
	write(*B*,*o*)

After schedule: *A*=45, *B*=255



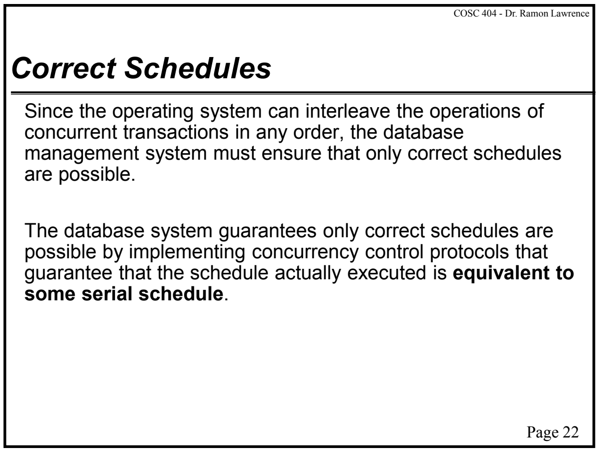
Example Schedules (3)

The following concurrent schedule does not preserve the value of the sum *A* + *B*: (***inconsistent state***)

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
	read(*A*,*t*)
	temp := *t**0.1;
	t := *t* – *temp*
	write(*A*,*t*)
	read(*B*,*o*)
write(*A*,*t*)	
read(*B*,*o*)	
o := *o* + 50	
write(*B*,*o*)	
	o := *o* + *temp*
	write(*B*,*o)*

After schedule: *A*=50, *B*=210

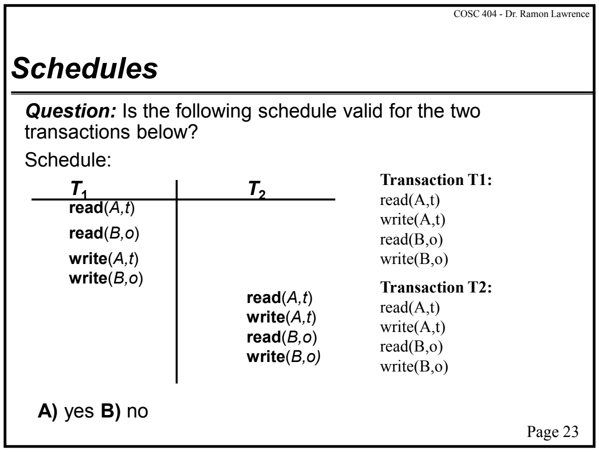
Is there another schedule with a different result?



Correct Schedules

Since the operating system can interleave the operations of concurrent transactions in any order, the database management system must ensure that only correct schedules are possible.

The database system guarantees only correct schedules are possible by implementing concurrency control protocols that guarantee that the schedule actually executed is **equivalent to some serial schedule**.



Schedules

Question: Is the following schedule valid for the two transactions below?

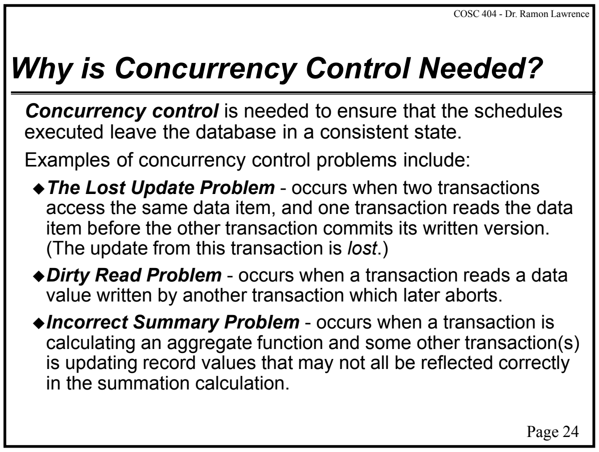
Schedule:

T_1	T_2
read(*A*,*t*)	
read(*B*,*o*)	
write(*A*,*t*)	
write(*B*,*o*)	
	read(*A*,*t*)
	write(*A*,*t*)
	read(*B*,*o*)
	write(*B*,*o)*

Transaction T1:
read(A,t)
write(A,t)
read(B,o)
write(B,o)

Transaction T2:
read(A,t)
write(A,t)
read(B,o)
write(B,o)

A) yes **B)** no



Why is Concurrency Control Needed?

Concurrency control is needed to ensure that the schedules executed leave the database in a consistent state.

Examples of concurrency control problems include:

- ***The Lost Update Problem*** - occurs when two transactions access the same data item, and one transaction reads the data item before the other transaction commits its written version. (The update from this transaction is *lost*.)
- ***Dirty Read Problem*** - occurs when a transaction reads a data value written by another transaction which later aborts.
- ***Incorrect Summary Problem*** - occurs when a transaction is calculating an aggregate function and some other transaction(s) is updating record values that may not all be reflected correctly in the summation calculation.

Lost Update Example

The ***lost update problem*** occurs when two transactions read the same value before either of them commits their write.

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
	read (*A*,*t*)
	temp := *t* *0.1
	t = *t* – temp
	write(*A*,*t*) ⇦
read(*B*,*o*)	
write(*A*,*t*)	
write(*B*,*o*)	

A is written without T_1's changes!

Dirty Read Example

The ***dirty read*** (or temporary update) problem occurs when a transaction reads a value of a later aborted transaction.

T_1	T_2
read(*A*,*t*)	
t := *t* – 50	
write(*A*,*t*)	
	read (*A*,*t*)
	temp := *t* *0.1
	t = *t* – temp
	write(*A*,*t*)
read(*B*,*o*)	
abort	

If T_1 aborts, then T_2 has used its incorrect value of A, and should not be allowed to commit.

Incorrect Summary Example

The ***incorrect summary*** problem occurs when a transaction updates values when another transaction is calculating a sum.

T_1	T_2
	sum = 0
	read(A)
	sum = sum + A
	...
read(X)	
X = X -100	X is updated before its value is used in summation.
write(X)	
	read (X)
	sum = sum + X
	read (Y)
	sum = sum + Y
	...
read(Y)	
Y = Y +100	Y is updated after its value is used in summation. (not consistent with X)
write(Y)	

Consistency Issues

Question: What consistency issue does this schedule have?

T_1	T_2
read(*A*,*t*)	
	read (*A*,*t*)
write(*A*,*t*)	
write(*B*, *10*)	
	read(*B*,*u*)
write(*C*,*t*)	
	write(*C*,*t*+*u*)

A) lost update **B)** dirty read **C)** incorrect summary **D)** none
E) more than one

Serializability

A schedule is ***serializable*** if it is equivalent to a serial schedule.

There are two different forms of serializability:

1. **conflict serializability**
2. **view serializability**

We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability
Conflicting Operations

To understand conflict serializability, we must understand what it means for two operations to conflict.

Operations O_i and O_j of transactions T_i and T_j respectively, ***conflict*** if and only if there exists some item *Q* accessed by both O_i and O_j, and at least one of these operations wrote *Q*.

Possibilities:

1. O_i = `read(Q)`, O_j = `read(Q)`. O_i and O_j do not conflict.
2. O_i = `read(Q)`, O_j = `write(Q)`. Conflict - order is important
3. O_i = `write(Q)`, O_j = `read(Q)`. Conflict - reverse of #2
4. O_i = `write(Q)`, O_j = `write(Q)`. Conflict - who writes last?

Intuitively, a conflict between O_i and O_j forces a (logical) temporal order between them. If O_i and O_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are ***conflict equivalent***.

We say that a schedule S is ***conflict serializable*** if it is conflict equivalent to a serial schedule.

Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Conflict Serializability (3)

The schedule below can be transformed into a serial schedule by a series of swaps of non-conflicting instructions. It is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read (A)
	write(A)
read (B)	
write(B)	
	read (B)
	write(B)

What is the serial schedule?

Conflict Serializability Question

Question: Is this schedule conflict serializable?

T_1	T_2
read(A)	
	write(A)
read(B)	
	write(B)
read(C)	
	read(C)
write(C)	

A) yes **B)** no

Serializability Questions

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

⇦ Note shorthand notation!
E.g. $r_1(A) = T_1$ does read(A)

Questions:

- 1) How many possible serial schedules are there?
- 2) How many schedules are conflict equivalent to the serial order (T_1 ,T_2)?
- 3) Write one non-serial schedule that is conflict equivalent to the serial execution (T_2 ,T_1), if possible.

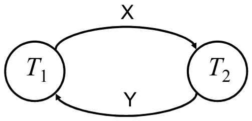
Testing for Serializability

It is possible to determine if some schedule of transactions T_1, T_2, ..., T_n is serializable using a precedence graph.

A ***precedence graph*** is a directed graph where the vertices are the transactions, and there is an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which they conflict earlier.

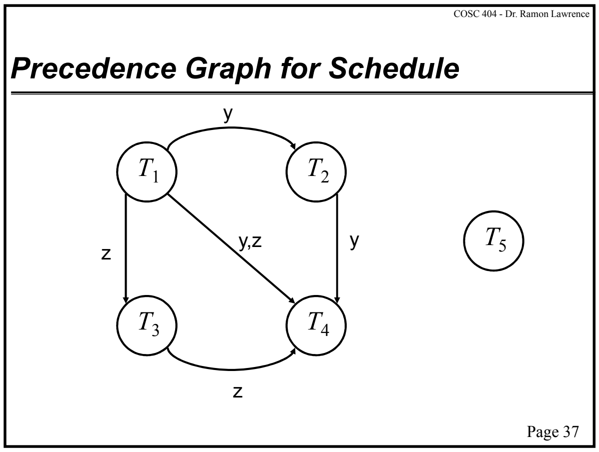
- We may label the arc using the item that was accessed.

Example: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$;

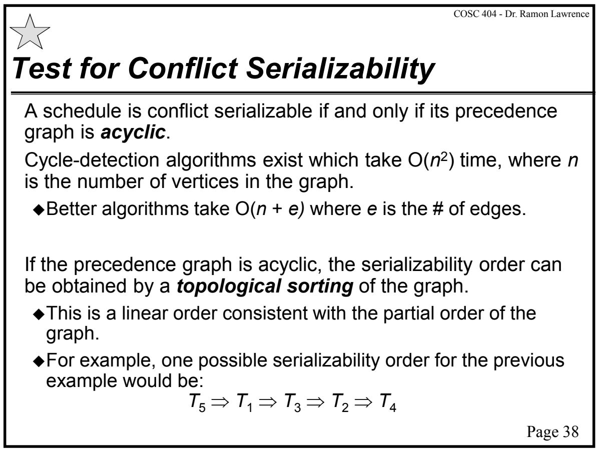


Precedence Graph Example Schedule

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Precedence Graph for Schedule



Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is ***acyclic***.

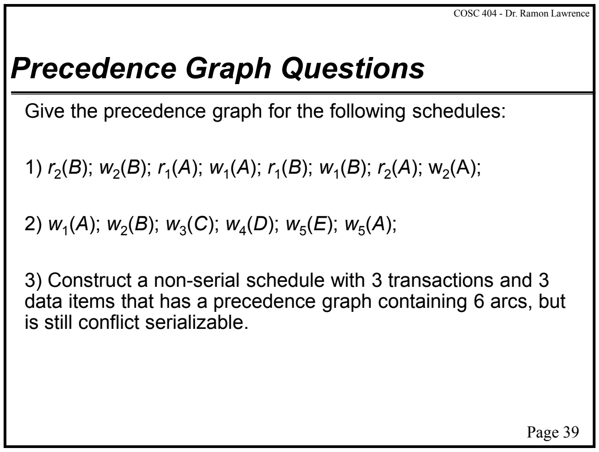
Cycle-detection algorithms exist which take O(n^2) time, where *n* is the number of vertices in the graph.

- Better algorithms take O(*n* + *e)* where *e* is the # of edges.

If the precedence graph is acyclic, the serializability order can be obtained by a ***topological sorting*** of the graph.

- This is a linear order consistent with the partial order of the graph.
- For example, one possible serializability order for the previous example would be:

$$T_5 \Rightarrow T_1 \Rightarrow T_3 \Rightarrow T_2 \Rightarrow T_4$$



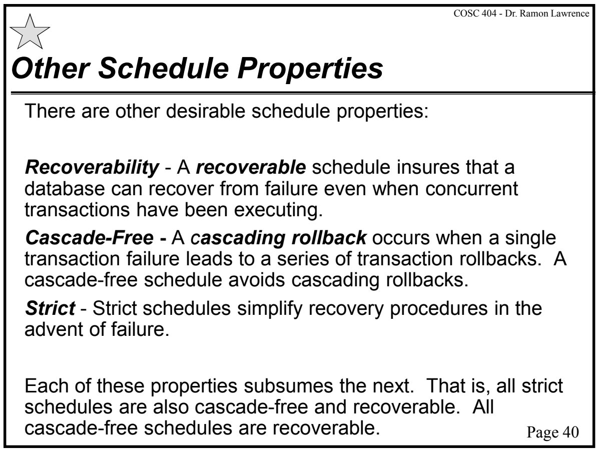
Precedence Graph Questions

Give the precedence graph for the following schedules:

1) $r_2(B)$; $w_2(B)$; $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; $r_2(A)$; $\mathrm{w}_2(\mathrm{A})$;

2) $w_1(A)$; $w_2(B)$; $w_3(C)$; $w_4(D)$; $w_5(E)$; $w_5(A)$;

3) Construct a non-serial schedule with 3 transactions and 3 data items that has a precedence graph containing 6 arcs, but is still conflict serializable.



Other Schedule Properties

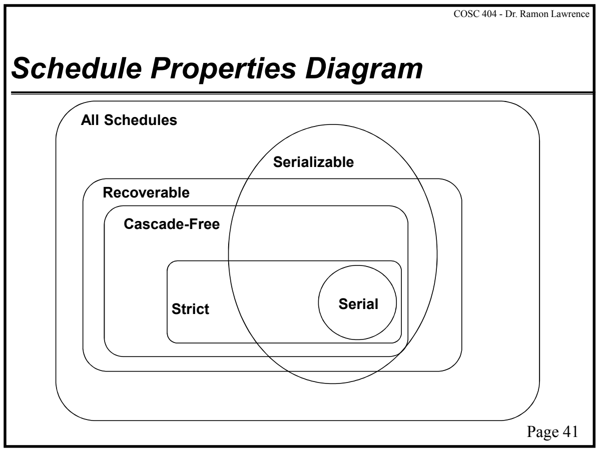
There are other desirable schedule properties:

Recoverability - A ***recoverable*** schedule insures that a database can recover from failure even when concurrent transactions have been executing.

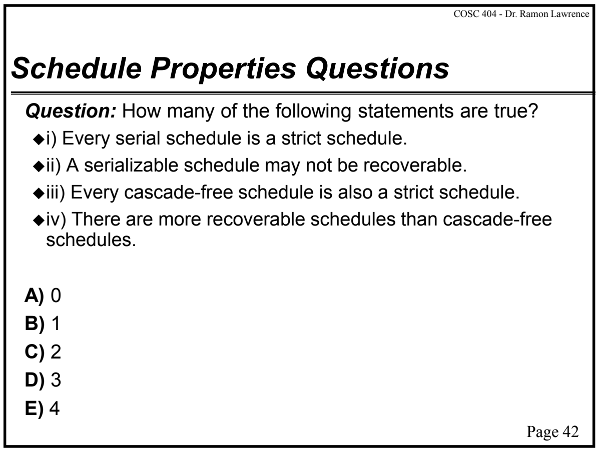
Cascade-Free - A c***ascading rollback*** occurs when a single transaction failure leads to a series of transaction rollbacks. A cascade-free schedule avoids cascading rollbacks.

Strict - Strict schedules simplify recovery procedures in the advent of failure.

Each of these properties subsumes the next. That is, all strict schedules are also cascade-free and recoverable. All cascade-free schedules are recoverable.



Schedule Properties Diagram



Schedule Properties Questions

Question: How many of the following statements are true?

- i) Every serial schedule is a strict schedule.
- ii) A serializable schedule may not be recoverable.
- iii) Every cascade-free schedule is also a strict schedule.
- iv) There are more recoverable schedules than cascade-free schedules.

A) 0
B) 1
C) 2
D) 3
E) 4

Recoverability

We need to address the effect of transaction failures on concurrently running transactions.

- ◆Let a transaction T_j read a data value written by another transaction T_i . If T_i aborts, then T_j should also abort because the data it read was inconsistent.

A ***recoverable*** schedule has the property that if a transaction T_j reads a data item previously written by a transaction T_i , the commit of T_i appears before the commit of T_j.

- ◆Note that if T_i aborts ***before*** T_j commits then the schedule is recoverable. It is not recoverable if T_i aborts ***after*** T_j commits.

Obviously, the database system wants to only allow recoverable schedules in advent of failures.

Non-Recoverable Schedules

The following schedule is not recoverable if T_9 commits immediately after the read:

T_8	T_9
read(*A*)	
write(*A*)	
	read(*A*)
	commit
read(*B*)	
abort	

⇦ T_8 aborts, but T_9 is already committed based on update of T_8!

The schedule is **not recoverable** because the commit for T_9 cannot be undone, but it should be because T_8 was never committed!

Recoverable Schedule Question

Question: Is this schedule recoverable?

T_8	T_9
read(*A*)	
write(*A*)	
	read(*A*)
	commit
read(*B*)	
commit	

A) yes **B)** no

Cascading Rollback

Cascading rollback occurs when a single transaction failure leads to a series of transaction rollbacks.

Consider the following schedule where no transactions have yet committed (so the schedule is recoverable):

T_{10}	T_{11}	T_{12}
read(*A*)		
read(*B*)		
write(*A*)		
	read(*A*)	
	write(*A*)	
		read(*A*)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- ◆Can lead to the undoing of a significant amount of work!
  - ⇨Note that T10 does not have to abort for the schedule to have cascading rollback. T11 and T12 will be **FORCED** to abort if T10 aborts. However, even if T10 commits, the schedule is not cascade-free because it has the *potential* for cascading aborts (but they did not occur)

Cascadeless Schedules

In a ***cascadeless*** schedule, cascading rollbacks cannot occur.

- ◆For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit of T_i appears before the read operation of T_j.
- ◆That is, transactions only read committed values.

Every cascadeless schedule is also recoverable.

A recoverable schedule never rolls back committed transactions, but may cascade rollback *uncommitted* transactions.

Cascade-Free Schedule Question

Question: Is this schedule cascade-free?

T_8	T_9
read(*A*)	
write(*A*)	
	read(*B*)
read(*B*)	
commit	
	commit

A) yes **B)** no

Strict Schedules

In a ***strict*** schedule, a transaction can neither read nor write a data item until the last transaction that wrote the data item commits (or aborts).

- ◆Strict schedules simplify recovery procedures because undoing an item write of an aborted transaction just involves restoring the before image (old value) of the item.
- ◆A strict schedule is always recoverable and cascadeless, but not vice versa.

Example:

T_{10}	T_{11}
read(*A*)	
read(*B*)	
write(*A*)	
	write(*A*)
	commit
abort	

Schedule Questions

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; c_1

T_2: $r_2(A)$; $\mathrm{w}_2(\mathrm{A})$; $r_2(B)$; $w_2(B)$; c_2

T_3: $r_3(B)$; $r_3(A)$; $\mathrm{w}_3(\mathrm{B})$; c_3

Given the three transactions T_1, T_2, T_3, come up with the following schedules:

- ◆a) A serial schedule
- ◆b) A conflict serializable schedule (non-serial)
- ◆c) A non-conflict serializable schedule
- ◆d) A non-recoverable, non-serial schedule
- ◆e) A cascade-free, non-serial schedule
- ◆f) A strict, non-serial schedule

View Serializability

Let *S* and *S´* be two schedules with the same transactions. *S* and *S´* are ***view equivalent*** if these three conditions are met:

1. For each data item *Q*, if transaction T_i reads the initial value of *Q* in schedule *S*, then transaction T_i must also read the initial value of *Q* in schedule *S´*.
2. For each data item *Q*, if transaction T_i executes `read`(*Q)* in schedule *S*, and that value was produced by transaction T_j, then transaction T_i must also read the value of *Q* that was produced by transaction T_j in schedule *S´*.
3. For each data item *Q*, the transaction (if any) that performs the final `write`(*Q*) operation in schedule *S* must perform the final `write`(*Q*) operation in schedule *S´*.

Conditions 1 and 2 ensure each transaction reads the same values, and condition 3 ensures the same final result.

View Serializability (2)

A schedule *S* is ***view serializable*** if it is view equivalent to a serial schedule.

- ◆Every conflict serializable schedule is also view serializable.
  - ⇨Every view serializable schedule which is not conflict serializable has *blind writes.* (A write without a read.)

This schedule is view serializable but *not* conflict serializable:

T_3	T_4	T_8
read(*Q*)		
	write(*Q*)	
write(*Q*)		
		write (*Q*)

Schedule is equivalent to serial schedule: $T_3 \Rightarrow T_4 \Rightarrow T_8$

Test for View Serializability

The precedence graph test for conflict serializability can be modified to test for view serializability:

- ◆Construct a *labeled precedence graph.*
- ◆Look for an acyclic graph that is derived from the labeled precedence graph by choosing one edge from every pair of edges with the same non-zero label. (2^n such graphs)
- ◆Schedule is view serializable if and only if such an acyclic graph can be found.

The problem of looking for such an acyclic graph falls in the class of *NP*-complete problems.

- ◆Thus existence of an efficient algorithm is unlikely. However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

Other Notions of Serializability

The schedule below produces the same outcome as the serial schedule < T_1, T_5 >, yet is not conflict or view equivalent.

T_1	T_5
read(*A*) *A* := *A* – 50 **write**(*A*)	
	read(*B*) *B* := *B* – 10 **write**(*B*)
read(*B*) *B* := *B* + 50 **write**(*B*)	
	read(*A*) *A* := *A* + 10 **write**(*A*)

Why DO these schedules result in the same answer?

Determining such equivalence requires analysis of operations other than read and write.

Concurrency Control and Serializability Tests

Testing a schedule for serializability ***after*** it has executed is a little too late!

The goal is to develop concurrency control protocols that will ensure serializability.

- They do not use the precedence graph as it is being created.
- Instead a protocol will impose a discipline that avoids non-serializable schedules.

Tests for serializability help understand why a concurrency control protocol is correct.

Transaction Management Summary

A ***transaction*** is a unit of program execution that accesses and may update data values and must be executed atomically.

Transactions should demonstrate the ***ACID properties***:

- atomicity, consistency, isolation, and durability

A ***schedule*** is the sequence of operations (possibly interleaved) from multiple concurrent transactions. A schedule is serializable if it can be proven equivalent to a serial schedule.

- Two types: conflict serializability and view serializability
- Tests for conflict serializability involves defining a precedence graph and checking for cycles.
- A schedule may also be recoverable, cascade-free, or strict.

Serializability tests are re-active, concurrency control protocols are pro-active. (prevent non-serializability)

Major Objectives

The "One Things":

- List and explain the ACID properties of transactions.
- Test for conflict serializability using a precedence graph.

Major Theme:

- Transactions are used to guarantee a set of operations are performed in an atomic manner. The DBMS must ensure interleaving of concurrent transactions is (conflict) serializable using a concurrency control method.

Objectives

- Define: transaction, atomic, consistent, constraint
- Explain the two challenges in preserving consistency.
- List and explain the ACID properties of transactions.
- Write a transaction using read/write operations.
- List the transactions states and draw the state diagram.
- Define schedules and serial schedules.
- List three problems that motivate concurrency control.
- Define conflict serializability and conflicting operations.
- Test for conflict serializability using a precedence graph.
- Define, recognize, and create examples of recoverable, cascade-free, and strict schedules.
- Draw the Venn diagram for schedules.

Objectives (2)

- Define view serializability and the 3 rules for view equivalent schedules.
- Define and give an example of a blind write.
- Recognize and create view serializable schedules.

COSC 404
Database System Implementation

Concurrency Control

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Concurrency Control Overview

Concurrency control (CC) is a mechanism for guaranteeing that concurrent transactions in the database exhibit the ACID properties. Specifically, the isolation property.

There are different concurrency control protocols:

- lock-based protocols
- timestamp protocols
- validation protocols
- snapshot isolation

Lock-Based Protocols

A ***lock*** is a mechanism to control concurrent access to data.

- An item can only be accessed through the lock.

Data items can be locked in two modes:

- ***exclusive (X) mode*:** Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
- ***shared (S) mode*:** Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock requests are made to the concurrency control manager. A transaction can only proceed after the request is *granted* and must follow the restrictions of the lock.

Lock-Based Protocols (2)

Lock-compatibility matrix:

	S	*X*
S	true	false
X	false	false

A transaction may be granted a lock on an item if the requested lock is *compatible* with locks already held on the item by other transactions.

- Any # of transactions can hold shared locks on an item.
- If any transaction holds an exclusive lock on the item, no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to ***wait*** until all incompatible locks held by other transactions are released. The lock is then granted.

Lock-Based Protocol Example

Example of a transaction performing locking:

```
lock-S(A);
read (A);
unlock(A);
lock-S(B);  ⇦ Another transaction updates B here.
read (B);
unlock(B);
display(A+B)
```

Simple locking is not sufficient to guarantee serializability.

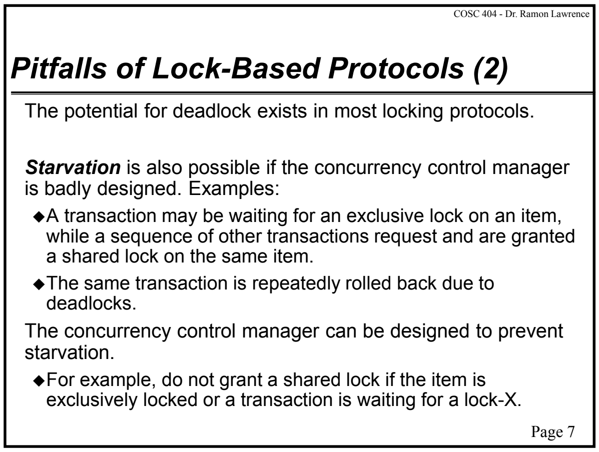
- If *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A ***locking protocol*** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

Consider the partial schedule:

T_3	T_4
lock-X*(B)*	
read*(B)*	
B:- B-50	
write*(B)*	
	lock-S*(A)*
	read*(A)*
	lock-S*(B)*
lock-X*(A)*	

- Neither T_3 nor T_4 can make progress as executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a ***deadlock***. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.



Pitfalls of Lock-Based Protocols (2)

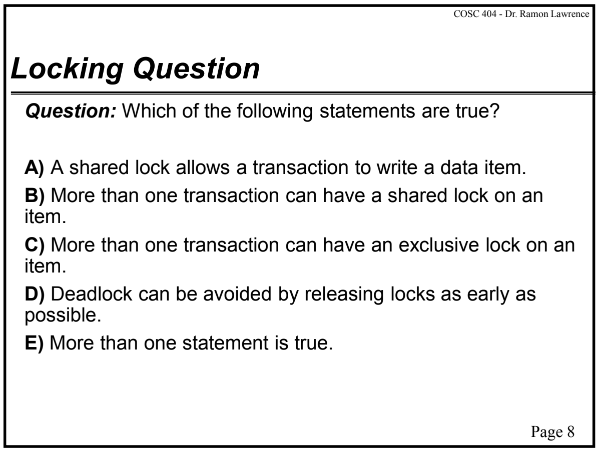
The potential for deadlock exists in most locking protocols.

Starvation is also possible if the concurrency control manager is badly designed. Examples:

- A transaction may be waiting for an exclusive lock on an item, while a sequence of other transactions request and are granted a shared lock on the same item.
- The same transaction is repeatedly rolled back due to deadlocks.

The concurrency control manager can be designed to prevent starvation.

- For example, do not grant a shared lock if the item is exclusively locked or a transaction is waiting for a lock-X.



Locking Question

Question: Which of the following statements are true?

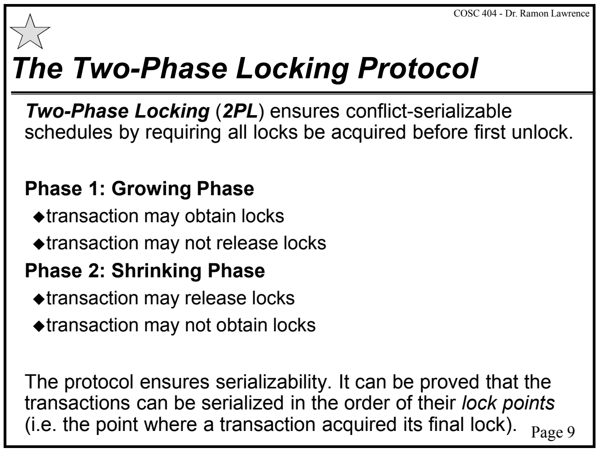
A) A shared lock allows a transaction to write a data item.

B) More than one transaction can have a shared lock on an item.

C) More than one transaction can have an exclusive lock on an item.

D) Deadlock can be avoided by releasing locks as early as possible.

E) More than one statement is true.



The Two-Phase Locking Protocol

Two-Phase Locking (***2PL***) ensures conflict-serializable schedules by requiring all locks be acquired before first unlock.

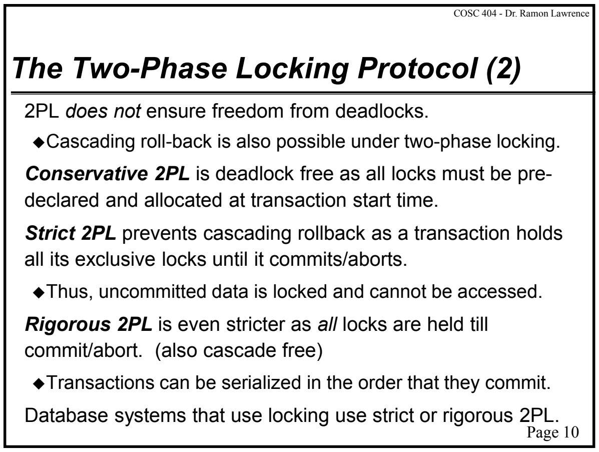
Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

The protocol ensures serializability. It can be proved that the transactions can be serialized in the order of their *lock points* (i.e. the point where a transaction acquired its final lock).



The Two-Phase Locking Protocol (2)

2PL *does not* ensure freedom from deadlocks.

- Cascading roll-back is also possible under two-phase locking.

Conservative 2PL is deadlock free as all locks must be pre-declared and allocated at transaction start time.

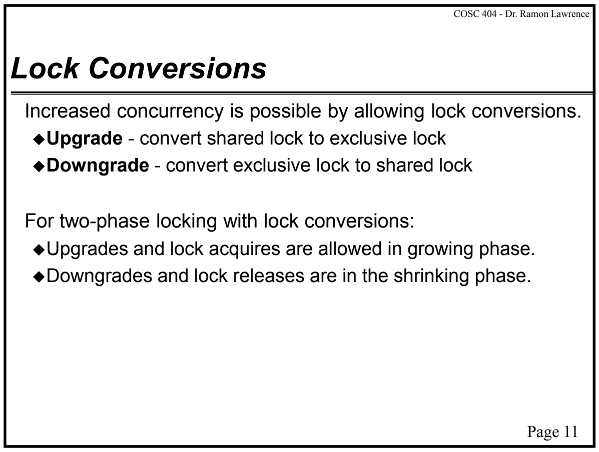
Strict 2PL prevents cascading rollback as a transaction holds all its exclusive locks until it commits/aborts.

- Thus, uncommitted data is locked and cannot be accessed.

Rigorous 2PL is even stricter as *all* locks are held till commit/abort. (also cascade free)

- Transactions can be serialized in the order that they commit.

Database systems that use locking use strict or rigorous 2PL.



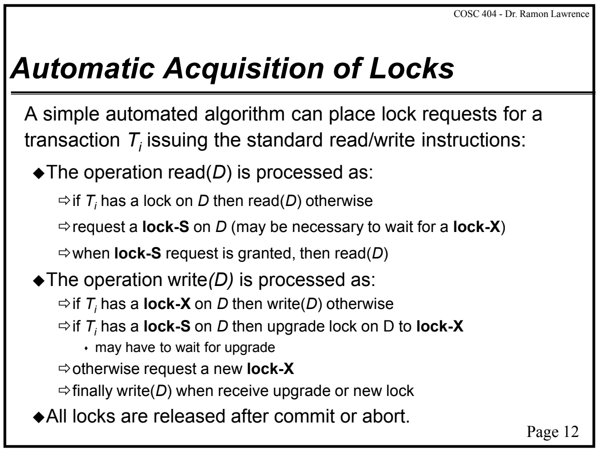
Lock Conversions

Increased concurrency is possible by allowing lock conversions.

- **Upgrade** - convert shared lock to exclusive lock
- **Downgrade** - convert exclusive lock to shared lock

For two-phase locking with lock conversions:

- Upgrades and lock acquires are allowed in growing phase.
- Downgrades and lock releases are in the shrinking phase.



Automatic Acquisition of Locks

A simple automated algorithm can place lock requests for a transaction T_i issuing the standard read/write instructions:

- The operation read(*D*) is processed as:
  - if T_i has a lock on *D* then read(*D*) otherwise
  - request a **lock-S** on *D* (may be necessary to wait for a **lock-X**)
  - when **lock-S** request is granted, then read(*D*)
- The operation write*(D)* is processed as:
  - if T_i has a **lock-X** on *D* then write(*D*) otherwise
  - if T_i has a **lock-S** on *D* then upgrade lock on D to **lock-X**
    - may have to wait for upgrade
  - otherwise request a new **lock-X**
  - finally write(*D*) when receive upgrade or new lock
- All locks are released after commit or abort.

Example on Auto Lock Insertion

Abbreviations:

- ◆A transaction T_i requesting a **lock-S** on D is given as: $sl_i(D)$.
- ◆A transaction T_i requesting a **lock-X** on D is given as: $xl_i(D)$.
- ◆A transaction T_i unlocking a data item D is given as: $ul_i(D)$.

Given transaction T_1, insert lock operations according to 2PL:

T_1: $r_1(A)$; $r_1(C)$; $w_1(B)$; $w_1(C)$;

Basic 2PL:

locks may be released anytime after this operation when not needed

$sl_1(A)$; $r_1(A)$; $sl_1(C)$; $r_1(C)$; $xl_1(B)$; $ul_1(A)$; $w_1(B)$; $ul_1(B)$; $xl_1(C)$; $w_1(C)$; $ul_1(C)$; c_1;

Example on Auto Lock Insertion (2)

Conservative 2PL:

atomic($sl_1(A)$, $xl_1(C)$, $xl_1(B)$)

locks may be released after they are no longer needed

$r_1(A)$; $r_1(C)$; $w_1(B)$; $w_1(C)$; c_1;$ul_1(A)$; $ul_1(B)$; $ul_1(C)$;

Strict 2PL:

$sl_1(A)$; $r_1(A)$; $xl_1(C)$; $r_1(C)$; $xl_1(B)$; $w_1(B)$; $xl_1(C)$; $ul_1(A)$; $w_1(C)$; c_1; $ul_1(B)$; $ul_1(C)$;

read locks may be released before commit (after last lock operation)

Rigorous 2PL:

$sl_1(A)$; $r_1(A)$; $xl_1(C)$; $r_1(C)$; $xl_1(B)$; $w_1(B)$;); $xl_1(C)$; $w_1(C)$; c_1; $ul_1(A)$; $ul_1(B)$; $ul_1(C)$;

all locks released after commit

2PL Question

Question: How many of the following statements are true?

- ◆i) Conservative 2PL is deadlock-free.
- ◆ii) Rigorous 2PL releases only write locks after commit.
- ◆iii) Lock upgrades are allowed during the shrinking phase of 2PL.
- ◆iv) Strict 2PL produces strict schedules.

A) 0
B) 1
C) 2
D) 3
E) 4

Questions on 2PL

1) Given the following transactions, insert lock operations according to 2PL:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $\mathrm{w}_2(\mathrm{A})$;

2) Write one non-serial schedule that obeys to 2PL, or argue why one is not possible.

3) Repeat #1 and #2 for these transactions:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; c_1
T_2: $r_2(A)$; $\mathrm{w}_2(\mathrm{A})$; $r_2(B)$; $w_2(B)$; c_2
T_3: $r_3(C)$; $r_3(A)$; $\mathrm{w}_3(\mathrm{C})$; c_3

Multiple Granularity

To this point, we have been locking individual data items. It is beneficial to allow locking of various size data items.

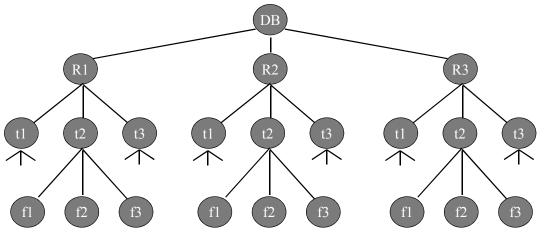
- ◆Define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- ◆Can be represented graphically as a tree.

When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

Granularity of locking (level in tree where locking is done):

- ◆*fine granularity* (lower in tree): high concurrency, high locking overhead (e.g. record locking, attribute locking)
- ◆*coarse granularity* (higher in tree): low locking overhead, low concurrency (e.g. table locking, database locking)

Example of Granularity Hierarchy



The highest level in the hierarchy is the entire database.

The levels below are *relation, tuple* and *field* in that order.

Intention Lock Modes

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

- ***intention-shared* (IS):** indicates explicit locking at a lower level of the tree but only with shared locks.
- ***intention-exclusive* (IX):** indicates explicit locking at a lower level with exclusive or shared locks
- ***shared and intention-exclusive* (SIX):** the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

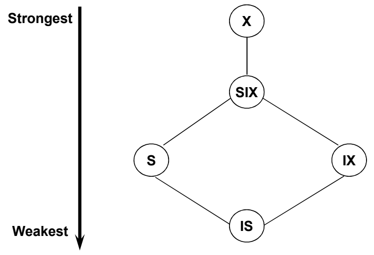
Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

Multi Granularity Lock "Strength"



Multiple Granularity Locking

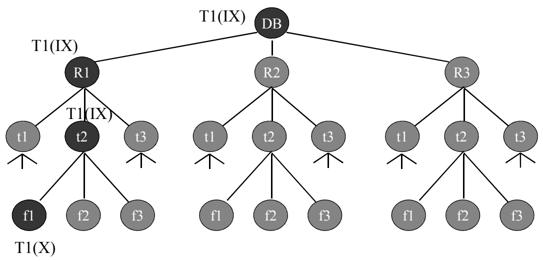
Transaction T_i can lock a node Q using the rules:

- The lock compatibility matrix must be observed.
- The root of the tree must be locked first (in any mode).
- A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
- A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
- T_i can lock a node only if it has not previously unlocked any node (that is, this is a variant of two-phase locking).
- T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Locks are acquired in root-to-leaf order, and released in leaf-to-root order.

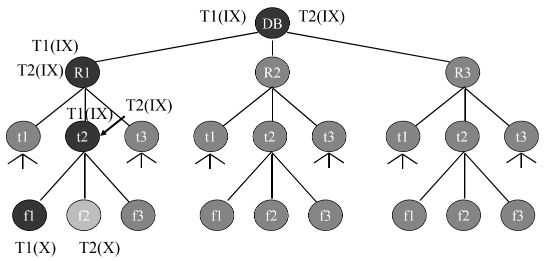
Multiple Granularity Locking Example

T1 wants to lock R1.t2.f1 in X-mode.



Multiple Granularity Locking Example (2)

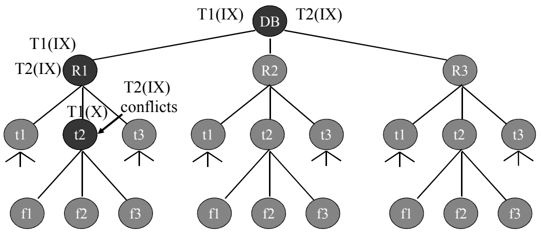
T2 wants to lock R1.t2.f2 in X-mode. Does it work?



Yes, it works!

Multiple Granularity Locking Example (3)

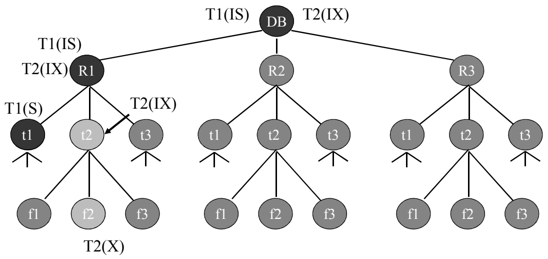
T2 wants to lock R1.t2.f2 in X-mode. Does it work?



No, conflict at t2!

Multiple Granularity Locking Example (4)

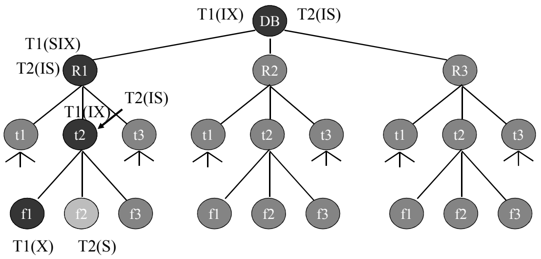
T2 wants to lock R1.t2.f2 in X-mode. Does it work?



Yes, it works!

Multiple Granularity Locking Example (5)

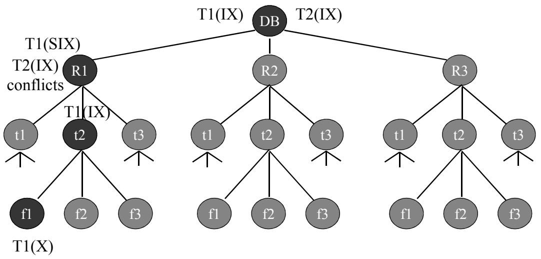
T2 wants to lock R1.t2.f2 in S-mode. Does it work?



Yes, it works!

Multiple Granularity Locking Example (6)

T2 wants to lock R1.t2.f2 in X-mode. Does it work?



No, conflict at R1!

Multiple Granularity Locking Question

Question: How many of the following statements are true?

- ◆i) The protocol always must lock the root node first.
- ◆ii) If a child node is locked, its parent node must also be locked.
- ◆iii) The protocol allows locking several tables at the same time.
- ◆iv) The protocol is deadlock free.

A) 0
B) 1
C) 2
D) 3
E) 4

Deadlock Handling

A system is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Two mechanisms for deadlock handling:

- ◆***deadlock prevention*** - do not allow system to enter deadlock state
- ◆***deadlock detection*** - detect deadlock condition and abort transactions to remove deadlock state

Cost of deadlock handling includes:

- ◆overhead of scheme itself
- ◆potential losses in transaction processing due to rollbacks

Deadlock Prevention

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state.

Some strategies:

- Require that each transaction locks all its data items before it begins execution (predeclare locks, e.g. conservative 2PL).
- Impose a partial ordering on data items and require that a transaction lock data items only in the order specified.
- Wound-wait and wait-die strategies use timestamps to determine transaction age and determine if a transaction should wait or be rolled back on a lock conflict.

Wound-Wait and Wait-Die Strategies

Wait-Die scheme — non-preemptive

- Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
- A transaction may die several times before acquiring needed data item.

Wound-Wait scheme — preemptive

- Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
- May cause fewer rollbacks than *wait-die* scheme.

Note: A rolled back transaction is restarted with its original timestamp. Older transactions have precedence over newer ones, and starvation is avoided.

Timeout-Based Schemes

In a Timeout-Based Schemes:

- A transaction waits for a lock only for a specified amount of time. After that, the transaction times out and is rolled back.
- Thus deadlocks are not possible.
- Simple to implement, but starvation is possible.
- Difficult to determine good value of the timeout interval.
  - Too short - false deadlocks (unnecessary rollbacks)
  - Too long - wasted time while system is in deadlock

Deadlock Detection & Recovery

If deadlocks are not prevented, then a detection and recovery procedure is needed to recover when the system enters the deadlock state.

An algorithm is run periodically to check for deadlock. If the system is in deadlock, then transactions are aborted to resolve the deadlock.

Deadlock detection requires the system:

- Maintain information about currently allocated locks.
- Provide an algorithm to detect a deadlock state.
- Recover from deadlock by aborting transactions efficiently.

Wait-for Graphs

Deadlocks can be detected using a ***wait-for graph***, *G* = (*V*,*E*):

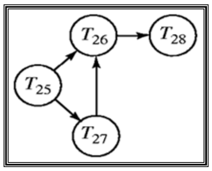
- *V* is a set of vertices (all the transactions in the system).
- *E* is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in *E*, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted into the graph.

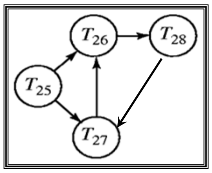
- This edge is removed only when T_j is no longer holding a data item needed by T_i.

The system is in a deadlock state if and only if the wait-for graph has a ***cycle***. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for Graph Examples



Wait-for graph with no cycle



Wait-for graph with a cycle

Deadlock Recovery

When a deadlock is detected three factors to consider:

- ***Victim selection*** - Some transaction will have to rolled back (made a victim) to break deadlock.
  - ⇨ Select the victim transaction that will incur minimum cost (computation time, data items used, *etc.*).
- ***Rollback*** - determine how far to roll back transaction
  - ⇨ **Total rollback:** Abort the transaction and then restart it.
  - ⇨ More effective to roll back transaction only as far as necessary to break deadlock. (requires system store additional information)
- ***Starvation*** happens if same transaction is always chosen as victim.
  - ⇨ Include the number of rollbacks in the cost factor to avoid starvation.

Deadlock Question

Question: How many of the following statements are true?

- i) A deadlock prevention protocol ensures deadlock never occurs.
- ii) In Wound-Wait, an older transaction waits on a younger one.
- iii) A wait-for graph has undirected edges between transactions.
- iv) A wait-for graph with 5 nodes but only 3 in a cycle is not in a deadlock state.

A) 0

B) 1

C) 2

D) 3

E) 4

Questions on Deadlocks

1) Assume a read-lock is requested before each read, and a write lock before each write. All unlocks occur after the last operation of a transaction. Explain what operations are denied during each schedule, draw the wait-for graph, and pick a transaction to abort if a deadlock does occur.

a) $r_1(A)$; r_2(B); w_1(C); r_3(D); $r_4(E)$; w_3(B); w_2(C); w_4(A); w_1(D);

b) $r_1(A)$; r_2(B); r_3(C); $w_1(B)$; w_2(C); w_3(D);

c) $r_1(A)$; $r_2(B)$; r_3(C); w_1(B); w_2(C); w_3(A);

Timestamp-Based Protocol

A ***timestamp protocol*** serializes transactions in the order they are assigned timestamps by the system.

Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.

- If an ***old*** transaction T_i has timestamp TS(T_i), a ***new*** transaction T_j has timestamp TS(T_j) where TS(T_i) < TS(T_j).
- The timestamp can be assigned using the system clock or some logical counter that is incremented for every timestamp.

Timestamp protocols do not use locks, so deadlock cannot occur!

Timestamp-Based Protocol Read and Write Timestamps

To ensure serializability, the protocol maintains for each data *Q* two timestamp values:

- **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully.
- **R-timestamp**(*Q*) is the largest timestamp of any transaction that executed **read**(*Q*) successfully.

The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Timestamp-Based Protocol Rules

Suppose a transaction T_i issues a **read**(*Q*):

- If TS(T_i) < **W**-timestamp(*Q*), then T_i needs to read a value of *Q* that was already overwritten.
  - ⇨ Hence, the **read** operation is rejected, and T_i is rolled back.
- If **TS(*T*$_i$)≥ W-timestamp(*Q*),** then the **read** operation is executed.
  - ⇨ The R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

Suppose that transaction T_i issues a **write**(*Q*):

- If **TS(*T*$_i$)≥ R-timestamp(*Q*) AND TS(*T*$_i$)≥ W-timestamp(*Q*),** then the **write** operation is executed.
- If TS(T_i) < R-timestamp(*Q*), then the value of *Q* that T_i is producing was previously read by newer transaction.
  - ⇨ Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of *Q*. T_i is rolled back.

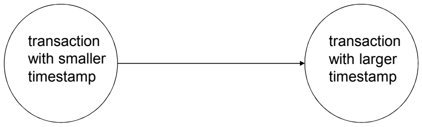
Timestamp Example

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5:

T_1	T_2	T_3	T_4	T_5
				read(X)
	read(Y)			
read(Y)				
		write(Y)		
				read(Z)
	write(X) **abort**			
read(X)				
		write(Z) **abort**		
				write(Y)
				write(Z)

Correctness of Timestamp-Ordering Protocol

The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:


transaction with smaller timestamp
transaction with larger timestamp

Thus, there will be no cycles in the precedence graph.

Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

Protocol is not recoverable or cascade-free.

- Can achieve both properties if perform all writes atomically at end of the transaction.

Thomas' Write Rule

Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances:

- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}. Hence, rather than rolling back T_i as the timestamp ordering protocol would have done, this **write** operation can be ignored. Otherwise protocol is unchanged.

Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

Timestamp Protocol Question

Question: How many of the following statements are true?

- i) Deadlock is not possible with timestamp protocols.
- ii) A transaction that arrives later to the system always has a smaller timestamp.
- iii) The precedence graph for the timestamp algorithm has edges from smaller timestamp transactions to larger ones.
- iv) A write is only performed if transaction has a timestamp >= the read timestamp for the data item.

A) 0

B) 1

C) 2

D) 3

E) 4

Questions on Timestamping

1) Indicate what happens during each of these schedules where concurrency control is performed using timestamps:

a) st_1; st_2; $r_1(A)$; r_2(B); w_2(A); $w_1(B)$;

b) st_1; $r_1(A)$; st_2; w_2(B); r_2(A); $w_1(B)$;

c) st_1; st_2; st_3; $r_1(A)$; $r_2(B)$; w_1(C); r_3(B); r_3(C); w_2(B); w_3(A);

d) st_1; st_3; st_2; $r_1(A)$; $r_2(B)$; w_1(C); r_3(B); r_3(C); w_2(B); w_3(A);

Validation Protocols

Validation or ***optimistic concurrency control protocols*** assume that the number of conflicts is low and verify correctness after a transaction is completed. Three phases:

- 1) ***Read phase*** – Transaction reads data items and performs operations. Writes are stored in local transaction memory.
- 2) ***Validation phase*** – Transaction checks if can proceed to write phase without violating serializability.
- 3) ***Write phase*** – All writes are copied to the database.

The validation test uses timestamps to guarantee that for two transactions T_i and T_j with TS(T_i) < TS(T_j) either:

- 1) T_i finished before T_j started OR
- 2) Set of data items written by T_i does not intersect with items read by T_j and T_i completes writes before T_j validates.

Multiversion Schemes

Multiversion schemes keep old versions of data to increase concurrency. This is especially useful for read transactions.

Each successful **write** creates a new version of the data item. Use timestamps or transaction ids to label versions.

When a **read** operation is issued, select an appropriate version of the data item based on the timestamp.

Reads never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

Each data item Q has a sequence of versions <Q_1, Q_2, ..., Q_m>. Each version Q_k contains three fields:

- ◆**Content** - the value of version Q_k
- ◆**W-timestamp**(Q_k) - timestamp of the transaction that created (wrote) version Q_k
- ◆**R-timestamp**(Q_k) - largest timestamp of a transaction that successfully read version Q_k

When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).

R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Scheme

The following scheme ensures serializability:

- ◆Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

If transaction T_i issues a **read**(Q) then:

- ◆ The value returned is the content of version Q_k.

If transaction T_i issues a **write**(Q):

- ◆If TS(T_i) < R-timestamp(Q_k), then T_i is rolled back.
- ◆If TS(T_i) = W-timestamp(Q_k), Q_k is overwritten.
- ◆Otherwise a new version of Q is created.

Multiversion Timestamp Scheme (2)

Reads always succeed; writes may be rejected if:

- ◆Some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.

Challenges:

- ◆Must have an efficient way of handling versions (and discarding when no longer needed).
- ◆Conflicts resolved through rollbacks rather than waiting so user application must be prepared to resubmit failed transactions.
  - ⇨Only update transactions can be rolled back.

Multiversion 2PL

Multiversion 2PL requires:

- ◆1) An integer counter used for timestamps for items and transactions.
- ◆2) Read-only transactions retrieve counter at start of transaction and use it to determine version to read. No locking used.
- ◆3) Update transactions perform rigorous 2PL. At commit, transaction increments timestamp counter and sets timestamp on every item it created.

Multiversion 2PL allows read transactions to never wait on locks and produces schedules that are recoverable and cascadeless.

Snapshot Isolation

Snapshot isolation is a widely-used protocol that gives each transaction its own "snapshot" of the database to execute on.

A snapshot consists of committed data values in the database before the transaction starts.

Read-only transactions never wait and are never aborted.

Update transactions keep updates private until commit when they are written to the database atomically. A validation is performed before writing the updates are allowed.

Snapshot Isolation Validation Test

Two ways to validate:

First committer wins:

- Transaction *T* enters prepared to commit state and checks:
  - If any concurrent transaction has updated any item *T* wants to update.
  - If yes, *T* is aborted. If no, *T* commits and updates written to database.

First update wins:

- If transaction T wants to update, it must get write lock on item.
- When lock is acquired, check if item has been updated by a concurrent transaction. If so, abort, otherwise proceed.

Snapshot Isolation Serializability Issues

Despite its advantages and being widely implemented (Oracle, PostgreSQL, SQL Server), snapshot isolation does not ensure serializability.

There are cases where particular transaction schedules are not serializable.

However, these issues can be often ignored or avoided, especially since primary and foreign key constraints are validated after snapshot validation and will often detect conflicts.

Multiversion and Snapshot Isolation Question

Question: How many of the following statements are true?

- i) Reads always succeed with a multiversion scheme.
- ii) Writes always succeed and create a new version each write.
- iii) Snapshot isolation guarantees serializability.
- iv) In a multiversion scheme, a read for a transaction may occur on a data value that is not the most recent.

A) 0

B) 1

C) 2

D) 3

E) 4

Insert and Delete Operations

In addition to read/write operations, the system must handle ***delete*** and ***insert*** operations.

Deletion with two-phase locking:

- May only be performed if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.

Insertion with two-phase locking:

- A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple.

The Phantom Phenomenon

Inserts/deletes can lead to the ***phantom phenomenon***:

- A transaction that scans a relation (e.g., find all students) and a transaction that inserts a tuple in the relation (e.g., inserts a new student) may conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result: the scan transaction may not see the new tuple, yet may be serialized before the insert transaction.
- Transactions conflict over a ***phantom tuple***.

The transaction scanning the relation reads information that indicates what tuples the relation contains. A transaction inserting a tuple updates the same info.

This information should be locked.

The Phantom Phenomenon (2)

Can prevent problem by:

- Accepting the issue (read committed isolation)
- Locking the entire relation (multi-granularity locking)
- Using index-locking or predicate-locking to guarantee that conflicts within the relation are detected.
- Having a special lock associated with the entire file. Read transactions that scan the whole relation must get a read lock on it and update transactions must get a write lock.

Transaction Definition in SQL

In SQL, a transaction begins implicitly.

A transaction in SQL ends by:

- **Commit** accepts updates of current transaction.
- **Rollback** aborts current transaction and discards its updates. Failures may also cause a transaction to be aborted.

An ***isolation level*** reflects how a transaction perceives the results of other transactions. It applies only to your perspective of the database, not other transactions/users. Lowering isolation level improves performance but may potentially sacrifice consistency.

Example Transactions

Transaction to deposit $50 into a bank account:

```
BEGIN TRANSACTION;
   UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
```

Transaction to calculate totals for all accounts (twice):

```
BEGIN TRANSACTION;
   SELECT SUM(balance) as total1 FROM Account;
   SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
```

Transaction to add a new account:

```
BEGIN TRANSACTION;
   INSERT INTO ACCOUNT (num, balance) VALUES ('S5', 100);
COMMIT T3;
```

Levels of Consistency in SQL-92

The isolation level can be specified by:

`SET TRANSACTION ISOLATION LEVEL =` *X* where *X* is

- **Serializable -** transactions behave like executed one at a time.
- **Repeatable read -** repeated reads must return same data. Does not necessarily read newly inserted records.
- **Read committed -** only committed values can be read, but successive reads may return different values.
- **Read uncommitted** - even uncommitted records may be read. Reading an uncommitted value is called a *dirty read*.

Scheduling of Transactions

Each transaction in a database is a separate executing program.

- A transaction may be its own program or a thread of execution.

The operating system schedules the execution of programs outside of the control of the DBMS.

- Thus, transactions may be executed in any order (as long as the order of operations within a transaction are the same). This interleaving is what produces different schedules.

The DBMS uses its concurrency control protocol to restrict the schedules to those that respect the consistency specified by the user for the transaction isolation level.

- All transactions must write lock any data item updated and the relation lock if inserting.
- Isolation level only affects read locks.

Scheduling Question

Question: **TRUE** or **FALSE**: The database has complete control over the scheduling of transactions.

A) True

B) False

Isolation Example
Serializable

A ***serializable*** schedule requires that regardless of the interleaving of the operations, the final result is the same as some serial ordering of the transactions.

- Read and write locks are held to commit. Also have a relation-level lock.

For three transactions, there are 3! = 6 serial schedules.

For these examples, assume that the total amount of money in all accounts is $5000 before the transactions begin.

Isolation Example Serializable (2)

Example schedule for T1, T2, T3:

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
SELECT SUM(balance) as total1 FROM Account;
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT T3;
```

After execution, total1 = $5050 and total2 = $5050.

- ◆The results for all six serial schedules are:
  - ⇨T1, T2, T3 – total1 = $5050 ; total2 = $5050
  - ⇨T1, T3, T2 – total1 = $5150 ; total2 = $5150
  - ⇨T2, T1, T3 – total1 = $5000 ; total2 = $5000
  - ⇨T2, T3, T1 – total1 = $5000 ; total2 = $5000
  - ⇨T3, T1, T2 – total1 = $5150 ; total2 = $5150
  - ⇨T3, T2, T1 – total1 = $5100 ; total2 = $5100

Isolation Example Repeatable read

With ***repeatable read***, a transaction is guaranteed to get the same data back on multiple reads but may see ***phantom records*** inserted in between reads.

- ◆Read and write locks are held to commit.

Example schedule:

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
SELECT SUM(balance) as total1 FROM Account;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT T3;
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
```

After execution, total1 = $5050 and total2 = $5150 as the second read sees the newly inserted tuple.

Isolation Example Read Committed

With ***read committed***, each read will get the most recently committed values even if different than an earlier read.

- ◆Read locks are released after every statement. Write locks released at commit.

Example schedule:

```
SELECT SUM(balance) as total1 FROM Account;
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT T1;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT T3;
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
```

After execution, total1 = $5000 and total2 = $5150 as the second read sees the newly inserted tuple and T1's update.

Isolation Example Read Uncommitted

Read uncommitted allows a transaction to read dirty data that has not been (and may never be) committed.

- ◆Transaction acquires no read locks.

Example schedule:

```
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
SELECT SUM(balance) as total1 FROM Account;
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
SELECT SUM(balance) as total2 FROM Account;
COMMIT T2;
ABORT T3;
ABORT T1;
```

After execution, total1 = $5050 and total2 = $5150 as T2's sees even uncommitted data. Note that both T1 and T3 abort so T2 sees incorrect data. **It is very dangerous to use read uncommitted if the transaction updates the database!**

Summary of Isolation Levels

Isolation Level	Problems	Lock Usage	Speed	Comments
Serializable	None	Read locks held to commit ; read lock on relation	Slowest	Only level that guarantees correctness.
Repeatable read	Phantom tuples	Read locks held to commit	Medium	Useful for modify transactions.
Read committed	Phantom tuples, values may change	Read locks released after each statement	Fast	Useful for transactions where operations are separable but updates are all or none.
Read uncommitted	Phantoms, values may change, dirty reads	No read locks	Fastest	Useful for read-only transactions that tolerate inaccurate results

Isolation Levels Question

Question: How many of the following statements are true?

- ◆i) Serializability guarantees that there are no phantom tuples.
- ◆ii) Read committed may be affected by phantom tuples.
- ◆iii) In read committed, two reads at separate times may retrieve different values.
- ◆iv) Read uncommitted is the fastest isolation level.

A) 0

B) 1

C) 2

D) 3

E) 4

Transaction Practice Question

Given these transactions and table `Bid(itemID, price)` that initially contains the two tuples: `(i1,10)` and `(i2,20)`:

```
T1: BEGIN TRANSACTION;
        S1: UPDATE Bid SET price = price + 5;
        S2: INSERT INTO Bid VALUES (i3,30);
        COMMIT;

T2: BEGIN TRANSACTION;
        S1: SELECT SUM(price) AS p1 FROM Bid;
        S2: SELECT MAX(price) AS p2 FROM Bid;
        COMMIT;
```

Assume that T1 executes with isolation level serializable and both transactions successfully commit.

- 1) If T2 executes with isolation level serializable, what are all the possible pairs of values for p1 and p2 returned by T2?
- 2) If T2 executes with isolation level read committed, what are all the possible pairs of values for p1 and p2 for T2?

Concurrency Control in PostgreSQL

PostgreSQL uses snapshot isolation for DML and 2PL for DDL.

- Snapshot isolation implementation is referred to as multi-version concurrency control (MVCC).
  - Uses first updater wins policy. Uses x-locks on written rows.
  - Each transaction has id (logical counter). Each tuple has transaction id that created it. Keeps track of snapshot info for each transaction.
  - Tradeoff: Reads never wait but more space used that must be handled.
- Uses deadlock detection with timeouts (default 1 sec.).

Isolation levels supported:

- read committed (default), serializable
  - For read committed, timestamp is at statement level. For serializable, timestamp is transaction's first timestamp.
  - A transaction will wait for a lock on a row currently being updated. If update committed by another transaction, waiting transaction issues error "could not serialize access due to concurrent update". Only possible for update/deletes.

Concurrency Control in MySQL

MySQL with the InnoDB storage engine uses snapshot isolation (multi-version concurrency control) for reads and 2PL for updates.

Supports all 4 isolation levels with different locks acquired for different levels. Default is repeatable read.

Concurrency Control in Microsoft SQL Server

Microsoft SQL Server uses 2PL and optimistic concurrency control.

Supports all four isolation levels plus two snapshot isolation levels.

Uses multiple granularity locking and automatically determines correct sizes (table, extent, page, rows).

Older snapshots are stored in temporary database.

Deadlock detection performed every 5 seconds by default.

Concurrency Control in Oracle

Oracle uses ***multiversion read consistency*** (snapshots).

- No locks for a read operation, so a read never blocks for a write.
- Uses row-level locking and transaction will wait if tries to change row updated by uncommitted transactions.
- System change number (SCN) used for ordering operations.
- Stores row lock on data block where row is stored.
- Locks held throughout transaction, released at commit/abort
  - Different types of locks; DDL, DML, mutex, latches
- Does deadlock detection using wait-for graphs
- Oracle Flashback Technology allows recovering a table to a point in time. Can be used to recover deleted rows or dropped tables without doing full restore from backup.

Implements: read committed and serializable isolation levels

Concurrency Control in MongoDB

MongoDB is a NoSQL document database. Performs atomic updates at document-level with no support for transactions.

MongoDB does not support any of the traditional isolation levels directly.

Uses reader-writer locks to ensure a data item can be read by many but only written by one at a time.

- Waiting writers have precedence over readers.
- Until Mongo 3.0, locking was at the database level. Mongo 3.0 and above perform multiple granularity locking (database, collection, document).

Concurrency Control Summary

Concurrency control protocols are used to ensure concurrent transactions maintain their isolation.

- ***Two-phase locking*** (**2PL**) and multigranularity locking schemes are commonly used.
- ***Deadlocks*** must be handled by either deadlock prevention or deadlock detection and recovery.
  - Prevention: wound-wait and wait-die schemes
  - Detection: wait-for graphs and transaction rollback

Multiversion schemes and snapshots create new versions on every update and determine the correct version for reads.

- Allows higher concurrency but uses more space. Very common.

SQL isolation levels are read uncommitted, read committed, repeatable read, and serializable.

- Differ on handling of dirty reads and phantom tuples.

Major Objectives

The "One Things":

- Explain how two-phase locking (2PL) works and detect valid 2PL schedules.
- Perform deadlock detection and recovery using wait-for graphs.
- Explain and use the timestamp based protocol.
- Perform multiple granularity locking using lock modes, rules, and compatibility matrix.
- Understand difference between snapshot based approaches (MVCC) and using 2PL.

Objectives

- Define concurrency control, locking protocol, deadlock, starvation, exclusive and shared locks (compatibility matrix).
- Define and use conservative, strict, and rigorous 2PL.
- Explain the use of lock conversions (upgrades/downgrades).
- Insert locks into a schedule using automatic algorithm.
- List some methods for deadlock prevention.
- List three factors with deadlock recovery.
- Define and motivate a validation based protocol.
- Explain the motivation for multiversion 2PL and timestamping.
- Explain the general approach for snapshot protocols.
- Explain how the phantom phenomenon occurs.
- List consistency levels in SQL-92 and determine which schedules are valid under each consistency level.

COSC 404
Database System Implementation

Recovery

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Recovery Motivation

A database system like any computer system is subject to various types of failures.

The database system must ensure the ACID properties (specifically durability and atomicity) despite failures.

We will categorize the various types of failures, and provide approaches for *recovering* from failures.

The process of restoring the database to a consistent state after a failure is called ***recovery****,* and is performed by the ***recovery system***.

Why is Recoverability Needed?

Recoverability is needed because the database system can fail for many reasons during transaction processing:

- **Computer Failure** - computer crash due to hardware, software, or network problems.
- **Disk Failure** - disk fails to correctly read/write blocks
- **Physical Problems/Catastrophes** - external problems resulting in data loss or system destruction (e.g. earthquake)

Transaction failures (but not database system failures):

- **Transaction Error** - error in transaction (e.g. divide by 0)
- **Exception Conditions** - transaction detects exception condition (e.g. data not present, insufficient bank funds)
- **Concurrency Control Enforcement** - transaction can be forced to abort to resolve deadlock or for serializability.

Failure Classification

The various types of failures can be classified in three categories:

- Transaction Failures:
  - **Logical errors**: Transaction cannot complete due to some internal error condition (bad input, data not found).
  - **System errors**: The database system must terminate an active transaction due to an error condition (e.g. deadlock).
- Software Failures:
  - **System crash**: A failure causes the system to crash, but non-volatile storage contents are not corrupted.
  - Examples: software design errors, bugs, buffer/stack overflows
- Hardware Failures:
  - **Disk failure**: A head crash destroys all or part of disk storage.
  - Examples: overutilization/overloading (used beyond its design), wearout failure, poor manufacturing

Terminology

A system is ***reliable*** if it functions as per specifications and produces a correct output for a given input.

A system ***failure*** occurs if it does not function according to specifications and fails to deliver the service desired.

An ***error*** occurs if the system assumes an undesirable state.

A ***fault*** is detected when either an error is propagated from one component to another or the failure of a component is detected.

Reliability Mechanisms

Fault Avoidance

- Attempt to eliminate all forms of hardware and software errors.

Fault Tolerance

- Provide component redundancies that cater to faults occurring within the system and its components.

Tradeoff:

- Fault tolerance requires more components.
- More components means more faults.
- Therefore, more components are need to handle the increasing faults.

Storage Structure (review)

Volatile storage does not survive system crashes.

- main memory, cache memory

Nonvolatile storage survives system crashes.

- Hard drive, solid-state drive

Stable storage is a *theoretical* form of storage that survives all failures.

- Approximated by maintaining multiple copies on distinct nonvolatile media.
- Practically achieving stable storage requires duplication of information such as maintaining multiple copies of each block on separate disks (RAID), or sending copies to remote sites to protect against disasters such as fire or flooding.
  - e.g. Multiple availability zones with Amazon hosting

Data Access

Physical blocks are those blocks residing on the disk. ***Buffer blocks*** are the blocks residing temporarily in main memory.

Block movements between disk and main memory are initiated through the following two operations:

- **input**(*B*) transfers the physical block *B* to main memory.
- **output**(*B*) transfers the buffer block *B* to the disk.

Each transaction T_i has its private work area in which local copies of all data items accessed and updated by it are kept. Assume that T_i's local copy of a data item *X* is called x_i.

Data Access (2)

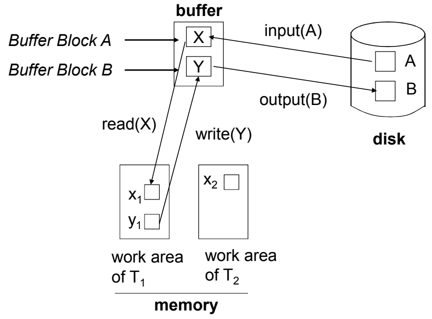
A transaction transfers data items between system buffer blocks and its private work-area using operations:

- **read**(X, x_i) assigns the value of item *X* to the local variable x_i.
- **write**(X, x_i) assigns the value of local variable x_i to data item *X* in the buffer block.
- Both these commands may require an **input**(B_X), if the block B_X in which *X* resides is not already in memory.

Transactions perform **read**(*X*) while accessing *X* for the first time; all subsequent accesses are to the local copy. After last access, transaction executes **write**(*X*).

output(B_X) need not immediately follow **write**(*X*). System can perform the **output** operation when it deems fit.

Example of Data Access



Buffer Management

The blocks in a database buffer are managed by a ***replacement policy*** (such as LRU).

Other considerations:

- **steal** vs. **no-steal** – no-steal prevents a buffer that is written by an uncommitted transaction to be saved to disk (removed from the buffer). Steal policy allows writing uncommitted updates.
  - Implemented using a pin bit on each buffer block.
- **force** vs. **no-force** – A force approach writes updates for committed transactions to disk immediately. No-force allows a committed update to remain in the buffer for some time.

Databases typically implement steal/no-force as it provides the most flexibility and best performance.

Log-Based Recovery

In log-based recovery, a ***log*** is kept on stable storage, and consists of a sequence of *log records*.

The log will record the sequence of database operations, and can be used to replay the database actions after a failure. The recovery manager uses the log to restore data items to their consistent state.

Recovery is related to concurrency control. We will assume that strict 2PL is performed that guarantees an item updated by a transaction T cannot be updated by another transaction until transaction T commits or aborts.

Log-Based Recovery Log Records

There are several types of log records:

- ***Start Records:*** When transaction T_i starts, it registers by writing a <T_i **start**> log record.
- ***Commit Records:*** When T_i finishes its last statement and successfully commits, the record <T_i **commit**> is written.
- ***Abort Records:*** When T_i aborts for whatever reason, the record <T_i **abort**> is written.
- ***Update Records:*** *Before* T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
    - That is, T_i has performed a write on data item X. X had value V_1 before the write, and will have value V_2 after the write.

Log records are written to stable storage.

Log Record Buffering

Log records are buffered in main memory, instead of being output directly to stable storage. Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.

- Several log records can thus be output using a single output operation, reducing the I/O cost.

These rules must be followed if log records are buffered:

- Log records are output in the order in which they are created.
- Transaction T_i enters the commit state after the log record <T_i **commit**> has been output to stable storage.
- Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. (This rule is called the **write-ahead logging** or **WAL** rule.)

Undo/Redo Logging

Undo/redo logging performs recovery by:

- **undo** updates for transactions that are not committed
- **redo** updates for transactions that were committed before failure

Redo/undo logging (WAL) rule:

- Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that update record <T, X, V_1, V_2> appear on disk.

Write-Ahead Logging

Question: Write-ahead logging means:

A) If a data item is updated, it must be written to storage before the log record.

B) If a data item is read, it must read a written, committed value.

C) An updated data item must only be written to storage after the log record for the update is written to storage.

D) None of the above

Recovery with Undo/Redo Logging

The recovery system must:

- Redo all the committed transactions in the order earliest-first.
- Undo all uncompleted transactions in the order latest-first.

When the system recovers, it does the following:

- 1) Initialize *undo-list* and *redo-list* to empty.
- 2) **First pass**: Scan the log backwards from end to build list of transactions to undo and redo.
- 3) **Second pass**: Scan the log forwards from the beginning and redo updates of committed transactions.
- 4) **Third pass**: Scan the log backwards from end and undo updates of uncommitted transactions.
- 5) For each undo transaction T, write a <T **abort**> log record. Flush the log and resume normal operation.

Undo/Redo Recovery Example

The log as it appears at three instances of time:

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
(a)	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
	(b)	
		(c)

Recovery actions in each case above are:

- (a) undo (T_0): B is restored to 2000 and A to 1000.
- (b) redo (T_0) and undo (T_1): A set to 950 and B set to 2050 then C is restored to 700.
- (c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

Undo/Redo Logging

Question: How many of the following statements are true?

- i) The first pass scans log forward to build undo and redo lists.
- ii) The second pass scans log forward performing redo.
- iii) The third pass scans log forward performing undo.
- iv) An update that is "redone" may or may not change the actual value in storage.

A) 0

B) 1

C) 2

D) 3

E) 4

Checkpoints

Recovery using the entire log would be expensive as the log grows in size over time.

To reduce the size of the log in order to make recovery faster, ***checkpoints*** are used to speed up recovery.

Checkpointing (blocking)

Checkpointing approach that blocks new transactions:

- 1) Stop accepting new transactions.
- 2) Wait until all currently running transactions either commit or abort.
- 3) Output all log records currently residing in main memory onto stable storage. (flush log) Output all updated buffers.
- 4) Write a log record <**checkpoint**> and flush log again.
- 5) Resume accepting transactions.

This guarantees all transactions before the checkpoint have their results reflected in the database. Recovery only needs to focus on log after the checkpoint.

Online (fuzzy) Checkpointing

The biggest problem with the previous technique is the system must stop processing transactions during the checkpoint.

Online checkpointing allows transactions to continue to run and be submitted during the procedure:

- 1) Write a log record <checkpoint start (T_1 ... T_N)> where T_1...T_N are the currently executing transactions. (flush log)
- 2) Write to disk all ***dirty*** buffers that have been modified before the checkpoint start. The buffers written include buffers changed by uncommitted transactions.
  - Note that the checkpoint procedure does not write dirty buffers that get modified between the checkpoint start and the checkpoint end records.
- 3) After all dirty buffers (recorded at checkpoint start) have been flushed, write a log record <checkpoint end> and flush the log.

Online Checkpointing

Question: How many of the following statements are true?

- i) Transactions may still run during an online checkpoint.
- ii) All updates in the buffer (committed or not) when the checkpoint starts are written to storage by end of checkpoint.
- iii) Updates in the buffer done after checkpoint start are written to storage.
- iv) The checkpoint start record contains all transactions, running and committed, before the checkpoint.

A) 0

B) 1

C) 2

D) 3

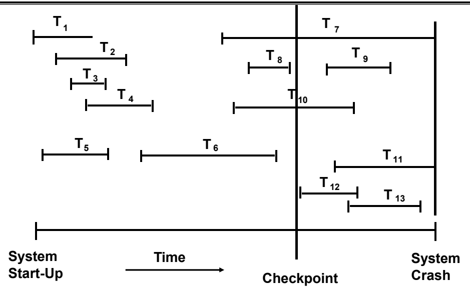
E) 4

Recovery using Undo/Redo and Checkpointing

Steps for recovery using undo/redo and checkpointing:

- 1) First pass backwards scan stops at the first start checkpoint log record found with a matching end checkpoint.
  - This scan will enumerate all transactions since last checkpoint and all active transactions when checkpoint began.
  - Divide these transactions into undo and redo lists.
- 2) Second pass forward scan starts at start checkpoint record and ends when all transactions are redone.
- 3) Third pass backwards scan starts at end of log and stops when all transactions in the undo list have been undone.
  - We know a transaction has no more operations when we encounter its transaction start log record.

Undo/Redo Checkpoints Example



What transactions are undone, redone, or committed?

Undo/Redo Recovery Example

The recovery algorithm on the following log:

<T_0 **start**>	**First Backwards Pass. (build lists from end)**
<T_0, A, 0, 10>	**Forwards Pass - Redo (start at checkpoint)**
<T_0 **commit**>	**Backwards Pass - Undo (start at end)**
<T_1 **start**>	⬅ Undo T1 complete. (Undo complete.)
<T_1, B, 0, 10>	⬅ Undo T1 write on B value now 0.
<T_2 **start**>	⬅ Undo T2 complete.
<T_2, C, 0, 10>	⬅ Undo T2 write on C value now 0.
<T_2, C, 10, 20>	⬅ Undo T2 write on C value now 10.
<checkpoint start (T_1, T_2)>	⇦ Checkpoint: T1, T2 were active (undo-list)
<checkpoint end>	
<T_3 **start**>	
<T_3, A, 10, 20>	⇦ Redo T3 write on A value now 20.
<T_3, D, 0, 10>	⇦ Redo T3 write on D value now 10.
<T_3 **commit**>	⇦ T3 in redo-list.
<T_1 abort>	⇦ Write abort transaction to log.
<T_2 abort>	⇦ Write abort transaction to log.

Undo/Redo Recovery with Checkpoints

Question: How many of the following statements are true?

- i) The first pass stops at the last checkpoint end record.
- ii) The second pass starts at the last checkpoint start record with a matching checkpoint end record.
- iii) The third pass stops when the start record for all transactions to be undone have been seen.
- iv) The second pass stops at the end of the log.
- v) The first pass starts at the end of the log.

A) 0

B) 1

C) 2

D) 3

E) 4

ARIES Recovery Algorithm

Recovery algorithm described is a simplification of the ***ARIES*** recovery algorithm that is widely used in databases.

Three steps:

- 1) Analysis – determine dirty pages in buffer, active transactions, and starting point for REDO step
- 2) REDO – reapplies updates of committed transactions
- 3) UNDO – scan log backwards undoing updates for non-committed transactions

Implementation details:

- Every log record has a log sequence number (LSN).
- Also stores Transaction Table and Dirty Page Table.
- Handles failure during recovery by logging undo operations so do not have to be repeated (uses compensation log records).

Nonvolatile Storage Failures

Solution: Periodically **dump** the entire contents of the database to stable storage.

No transaction may be active during the dump procedure. A procedure similar to checkpointing must take place:

- Output all log records currently residing in main memory onto stable storage.
- Output all buffer blocks onto the disk.
- Copy the contents of the database to stable storage.
- Output a record <**dump**> to log on stable storage.

To recover from disk failure, restore database from most recent dump. Then log is consulted and all transactions that committed since the dump are redone.

- Can be extended to allow transactions to be active during dump; known as *fuzzy* or *online* dump.

Advanced Recovery Techniques

Support high-concurrency locking techniques, such as those used for B$^+$-tree concurrency control.

Operations like B$^+$-tree insertions and deletions release locks early. They cannot be undone by restoring old values (***physical undo***), since once a lock is released, other transactions may have updated the B$^+$-tree.

Instead, insertions/deletions are undone by executing a deletion/insertion operation (known as ***logical undo***).

- For such operations, undo log records should contain the undo operation to be executed; called ***logical undo logging***, in contrast to ***physical undo logging***.
- Redo information is logged *physically* (that is, new value for each write) even for such operations.

Undo/Redo Logging Questions

Explain undo/redo logging recovery for the following log as it appears at three instances of time:

<T_1 start>	<T_1 start>	<T_1 start>
<T_1, A, 4, 5>	<T_1, A, 4, 5>	<T_1, A, 4, 5>
<T_2 start>	<T_2 start>	<T_2 start>
<T_1 commit>	<T_1 commit>	<T_1 commit>
<T_2, B, 9, 10>	<T_2, B, 9, 10>	<T_2, B, 9, 10>
System Failure	<checkpoint start (T_2)>	<checkpoint start (T_2)>
(a)	<T_2, C, 14, 15>	<T_2, C, 14, 15>
	<T_3 start>	<T_3 start>
	<T_3, D, 19, 20>	<T_3, D, 19, 20>
	System Failure	<checkpoint end>
	(b)	<T_2 commit>
		System Failure
		(c)

Summary

A database system must be able to ***recover*** in the presence of hardware and software failures. The database system must ensure a consistent database after failure and preserve the ACID properties.

Log-based recovery records all updates in a log and undo/redo operations are used to restore the database to a consistent state (*write-ahead logging* is used).

Checkpointing reduces the cost of log-based recovery.

Database backups are needed to handle catastrophic failures.

Advanced (logical) recovery is necessary for B+-tree indexes.

Major Objectives

The "One Things":

- ◆Perform Undo/Redo logging with checkpoints.

Major Theme:

- ◆The recovery system rebuilds the database into a consistent state after failure using the log records saved to stable store while the database was operational. Various methods including checkpoints are used to speed-up recovery after failures.

Objectives

- ◆Define: recovery and recovery system
- ◆List the types of failures and motivation for recovery.
- ◆Define: reliable, failure, error, fault, stable storage
- ◆Compare/contrast fault avoidance versus fault tolerance.
- ◆Read and write log records in a log.
- ◆Define: write-ahead logging rule (WAL), log force operation
- ◆Motivate the importance of checkpoints and online checkpointing.
- ◆Compare/contrast physical versus logical logging.

COSC 404
Database System Implementation
Scaling Databases
Distribution, Parallelism, Virtualization

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Scaling Database Systems

Scaling a database system involves handling:

- ◆larger data sets and queries that involve more data
- ◆larger number of users/transactions/queries
- ◆handling failures and concurrency issues when supporting more users and servers

Scaling is achieved by adding more servers (i.e. cluster) and replicating/distributing/partitioning data across those servers that handle the data and query load.

There are a variety of architectures and approaches.

Performance Measures for Parallel and Distributed Systems

Throughput - the number of tasks that can be completed in a given time interval.

Response time - the amount of time it takes to complete a single task from the time it is submitted.

Speedup - how much faster a fixed-sized problem can be executed on hardware that is *N*-times faster.

- ◆*speedup = time on basic system / time on N-times faster system*
- ◆Speedup is *linear* if equation equals N.

Scaleup - is the ability of a system *N* times larger to perform a job *N* times larger, in the same time as the original system.

- ◆*scaleup* = $\dfrac{\textit{time to execute small problem on small system}}{\textit{time to execute large problem on large system}}$
- ◆Scale up is *linear* if equation equals 1.

Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- ◆**Startup costs**: Cost of starting up multiple processes may dominate computation time if the degree of parallelism is high.
- ◆**Interference**: Processes accessing shared resources (e.g. system bus, disks, or locks) compete with each other and spend time waiting on other processes, rather than performing work.
- ◆**Skew**: Increasing the degree of parallelism increases the variance in service times of parallel executing tasks. Overall execution time determined by ***slowest*** of executing tasks.

Parallel Performance Measures

Question: How many of the following statements are true?

- ◆i) Response time is how long it takes to complete a given task from the time it is submitted.
- ◆ii) Throughput is the rate at which tasks can be completed.
- ◆iii) Interference is one factor that can limit scaleup.
- ◆iv) When a company wants to grow its database, scaleup is an important factor.

A) 0
B) 1
C) 2
D) 3
E) 4

Parallel Database Systems

A ***parallel database system*** consist of multiple processors and disks connected by a fast interconnection network.

Parallel database systems are used for:

- ◆storing large volumes of data
- ◆processing time-consuming decision-support queries
- ◆providing high throughput for transaction processing

Parallel execution occurs *within* a system in the form of exploiting parallelism available in CPUs and graphics cards.

Distributed Database System

A ***distributed database system*** (DDBS) is a database system distributed across several network nodes that appears to the user as a single system.

A DDBS processes complex queries by coordinating among the individual nodes. Processing may be done at a site other than the location of query submission. This requires cooperation on transaction management, concurrency control, and query optimization.

Parallel and distributed databases have many features in common and the line between them is not always clear. One main difference is that a distributed system is designed to be ***physically/geographically distributed*** where a parallel DBMS may be in a single server/data center.

Parallel and Distributed Databases Advantages and Disadvantages

Advantages:

- ◆**PERFORMANCE, availability, reliability**
- ◆Local autonomy
- ◆Reflects organization structure
- ◆Economics (smaller systems)
- ◆Less network traffic compared to centralized

Disadvantages:

- ◆Complexity
- ◆Lack of control
- ◆Cost
- ◆Security
- ◆More complex database design

Parallel/Distributed Architectures

Shared memory - processors share a common memory.

- ◆Memory shared using a bus allowing fast communication between processors. Good for small parallel systems.
- ◆Architecture is not scalable since the bus is a bottleneck.

Shared disk - processors share a common disk.

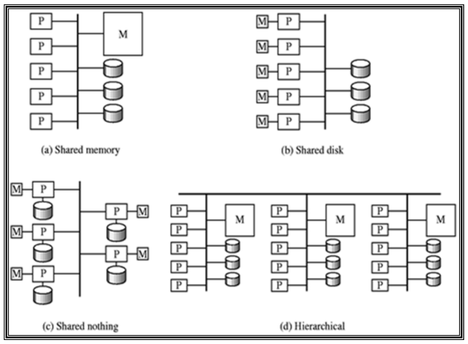
- ◆Processors shared data on disk but have private memories.
- ◆Bottleneck at disk system instead of bus. Slower data sharing.

Shared nothing - processors share no memory or disks.

- ◆A node consists of a processor, memory, and one or more disks. Nodes communicate over the network.
- ◆Can be scaled up to thousands of processors.

Hierarchical - hybrid combination of the above architectures.

Parallel/Distributed Architectures



(a) Shared memory

(b) Shared disk

(c) Shared nothing

(d) Hierarchical

Parallel/Distributed Architectures

Question: How many of the following statements are true?

- ◆i) Shared memory is used when servers are in different locations.
- ◆ii) Shared nothing is the architecture that is the least popular.
- ◆iii) MongoDB assumes/uses a shared disk architecture.
- ◆iv) The shared disk architecture is the hardest to implement.

A) 0
B) 1
C) 2
D) 3
E) 4

Types of Database Parallelism

A database can exploit a parallel hardware system by:

- ◆**Partitioning/Sharding** - dividing the data across hardware to allow for parallel I/O and query processing.
- ◆**Interquery Parallelism -** executing multiple queries concurrently using the parallel hardware resources.
- ◆**Intraquery Parallelism** - executing operators of a query plan in parallel or parallelizing individual operators.

Distributed Data Storage Replication and Partitioning

A key decision in a parallel/distributed database is how to allocate the data across nodes.

This allocation involves both ***replication*** and ***partitioning***:

- ◆**Replication -** system maintains multiple copies of data stored at different sites for faster retrieval and fault tolerance.
- ◆**Partitioning -** relation is partitioned into several fragments/partitions stored in distinct sites.
- ◆**Replication and partitioning** - relation is partitioned into several partitions and system maintains several identical replicas of each partition.

Data Replication Discussion

Replication is good for reads and bad for writes!

Advantages of Replication:

- ◆**Availability** - failure of a site containing a relation does not result in unavailability if replicas exist.
- ◆**Parallelism** - queries on a relation may be processed by several nodes in parallel.
- ◆**Reduced data transfer -** relation is available locally at each site containing a replica of it.

Disadvantages of Replication:

- ◆**Increased update cost** - each replica must be updated.
- ◆**Increased complexity of concurrency control** - concurrent updates to distinct replicas may lead to inconsistent data.
- ◆**Increased space requirements** - more storage is needed.

Maintaining Consistency with Replication – CAP Theorem

The ***CAP Theorem*** (Brewer 2000) proves that a distributed system can have only two of these three properties: consistency, availability, and partition-tolerance.

In a large system, partitions cannot be prevented, so must sacrifice either availability or consistency.

Many new NoSQL databases select availability over consistency which means that the replicas are not always consistent in time, which is called weak or ***eventual consistency***.

- ◆Strong consistency – all replicas same value at end of update
- ◆Weak consistency – may take some time for all replicas to become consistent

BASE Properties – Not ACID

In eventually consistent systems, the ACID properties do not hold. We may consider these systems to have BASE properties:

Basically **A**vailable

- ◆If server is accessible, can do reads and updates (even if have network partition). Availability at the cost of consistency.

Soft state

- ◆Each replica may have different values (due to partitioning or time delay in update propagation).

Eventually consistent

- ◆Replicas are not consistent at instance of update but will be come become consistent eventually as updates are propagated and conflicts resolved.
- ◆Merging inconsistent updates is still a challenge.

Master/Slave Configuration for Handling Replication and Ensuring Consistency

In a ***master/slave configuration***, one master server is responsible for updates to each partition and sends the updates to the slaves that contain copies of the partition.

- ◆***Primary copy ownership*** – one site owns data, perform updates on that site, and updates are sent out to subscribers who update their replicas. These updates may be sent out by shipping the log to the slave sites.
- ◆The master node is read/write. The slave nodes are read only. This requires a way to specify a read-only transaction (e.g. set at connection or statement level before executing query) so that it can be processed by a slave node.

Master/Master Configuration for Handling Replication and Consistency

In a ***master/master configuration***, more than one server is able to perform updates on a given partition. This requires co-ordination by the masters.

Techniques:

- ◆Any update must be "approved" by all (or a majority) of the master servers. This approval may be done before commit (online) using a distributed algorithm (e.g. two phase commit).
- ◆Updates may be allowed on multiple servers simultaneously, but there must be some system or user-configured resolution mechanism to handle conflicts.

Data Partitioning

Partitioning is the process of dividing a relation *r* into partitions $r_1, r_2, \ldots, r_n$ that can be combined to reconstruct *r*.

- **Horizontal partitioning (sharding)** - each tuple of *r* is assigned to one or more partitions (shards).
  - Partition can be defined using selection from *r*.
  - Reconstruct *r* from partitions by performing union.
- **Vertical partitioning** - the schema for relation *r* is split into several smaller schemas.
  - Partitions are defined using projection on *r*.
  - Reconstruct *r* by joining partitions.
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, such as a tuple id attribute may be added to each schema to serve as a candidate key.
- Vertical and horizontal partitioning can be mixed.

Horizontal Data Partitioning and Sharding

Horizontal partitioning/sharding – tuples are divided among many servers such that each tuple resides on one server.

- Partitioning techniques (assuming *n* servers):
  - **Round-robin**: Send the i^{th} tuple in the relation to server *i* mod *n*.
  - **Hash partitioning**: Use a hash function *h*(*x*) on partitioning attributes *x* that maps each tuple to one of the *n* servers.
  - **Range partitioning**: Chose a partitioning attribute V and divide the domain of V using a partitioning vector $[v_o, v_1, \ldots, v_{n\text{-}2}]$. For each tuple with value v, if $v \leq v_i$ then tuple goes on server *i*. If $v \geq v_{n\text{-}2}$ go to server *n*-1.

Question: How does each partitioning technique perform for these different types of queries?

- 1) Scanning the entire relation
- 2) Lookup queries (on the partition attribute)
- 3) Range queries (on the partition attribute)
- 4) Lookup or range queries not on the partition attribute

Horizontal Partitioning Example

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

account₁

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

account₂

Partitioned Account relation on branch-name attribute.

Vertical Partitioning Example

deposit₁

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

deposit₂

account number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

Partitioned Deposit relation.

Advantages of Partitioning

Horizontal:

- allows parallel processing on a relation
- allows a relation to be split so that tuples are located where they are most frequently accessed

Vertical:

- allows for further decomposition from what can be achieved with normalization
- tuple id attribute allows efficient joining of vertical fragments
- allows parallel processing on a relation
- allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed

Skew

Skew is when the distribution of data is not uniform. Skew reduces the performance of algorithms such as partitioning that intend for the data to be uniformly distributed across hardware.

Types of skew:

- **Attribute-value skew**
  - Some values appear in the partitioning attributes of many tuples. All the tuples with the same value for the partitioning attribute end up in the same partition.
    - E.g. Ages of students are skewed between 18-25.
  - Can occur with range-partitioning and hash-partitioning.
- **Partition skew**
  - With range-partitioning, badly chosen partition vector may assign too many tuples to some partitions and too few to others.
  - Less likely with hash-partitioning if a good hash-function is chosen.

Partitioning

Question: How many of the following statements are true?

- i) Sharding is another name for horizontal partitioning.
- ii) Vertical partitioning divides a relation by its attributes.
- iii) Skew is beneficial when performing partitioning.
- iv) Replication and partitioning can be used together.

A) 0
B) 1
C) 2
D) 3
E) 4

Interquery Parallelism

Interquery parallelism is when queries execute in parallel with one another.

- Increases throughput but does not improve response time.
- Easiest form of parallelism to support particularly in a shared memory database.

More complicated to implement on shared disk or shared nothing architectures when dealing with updates:

- Locking and logging must be coordinated by passing messages between processors if system guarantees consistency.
  - ⇨ *Cache coherency* has to be maintained as reads and writes of data in buffer must find latest version of data.
- Sharding can often help as data within a shard (partition) is only located on one server. (Replication will be an issue though).

Intraquery Parallelism

Intraquery parallelism is the execution of a single query in parallel.

- Reduces response time (especially for long-running queries).

Two forms of intraquery parallelism:

- **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
- **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

Intraoperation parallelism scales better because the number of tuples processed by each operator is typically more than the number of query operators.

Parallel Processing of Relational Operations

Our discussion assumes a shared-nothing architecture of *n* processors, P_0, ..., P_{n-1} and *n* disks D_0, ..., D_{n-1}, where disk D_i is associated with processor P_i.

For all algorithms, we will assume that we have already partitioned relation *R* across the *n* processors uniformly using either range or hash partitioning.

Implementing parallel selection and projection:

- Each processor performs local selection (projection) on its partition. Result is sent to client.
- This also works for duplicate elimination and aggregation.

Parallel Sorting

Parallel External Sort-Merge

- Assume the relation is partitioned among disks D_0, ..., D_{n-1}.
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.

Optimizations:

- The merge is trivial if the relation was range partitioned on the sort attribute.
- Note that range partitioning can be used after the local sort to parallelize the merge as well (less of a benefit).

Parallel Join

Parallel join algorithms partition the relations across the processors such that two tuples will join if and only if they are in the same partition at a single processor.

- Range or hash partitioning can be used on the *join attributes*.
- Each processor computes its local join and the final result is the union of the results of all local joins.

Interoperation Parallelism

There are two types of interoperation parallelism:

- **Pipelined parallelism -** output tuples of one operation are consumed as input by another operation. (Iterators)
  - ⇨ Avoids writing intermediate results to disk.
  - ⇨ With parallel systems, operations can be performed at different processors. Output of one processor is input for another processor.
  - ⇨ Useful for sequences of joins but limited parallelism scaling.
- **Independent parallelism** - operations in a query that do not depend on each other can be performed in parallel.
  - ⇨ Different branches of operator tree.
  - ⇨ E.g. Join of four relations can be computed as join of two temporary joins of relations r_1 and r_2 and relations r_3 and r_4.

Distributed Query Optimization

Distributed query optimization is even more complex than with a centralized system.

Issues:

- Query cost estimation – must consider processing capabilities of each node as well as location of data and transfer cost
- Query decomposition – how to divide query across nodes
- Data localization – ***goal is to move query to data***
- Global optimization – optimize query overall
- Local optimization – optimize part of query on particular node
- Distributed operations – parallelizing and distributing work of joins/sorts over multiple nodes

Semijoin

The ***semijoin*** of r_1 with r_2, is denoted by $r_1 \ltimes r_2$.

Semijoin is computed by:

$$\Pi_{r1} (r_1 \bowtie r_2)$$

$r_1 \ltimes r_2$ selects tuples of r_1 that are present in the join of r_1 and r_2.

The semijoin operation is used to reduce the number of tuples in a relation before transferring it to another site.

- The basic idea is that one site sends all the values of the join key to the other site which then knows which tuples will participate in the join (and will only send those tuples back).

Semijoin Example

Let *Emp(ssn, name, deptName)* be at site S_1 and *Dept (name, mgrssn)* be at S_2. Compute $Emp \bowtie_{ssn=mgrssn} Dept$.

Algorithm:

- Compute $temp_1 \leftarrow \Pi_{mgrssn}(Dept)$ at S_2. Send $temp_1$ to S_1.
- At S_1 compute $temp_2 \leftarrow Emp \bowtie temp_1$ and send back to S_2.
- Compute $Dept \bowtie temp_2$ at S_1. This is the result of $Emp \bowtie Dept$.
- In this operation sequence, $temp_2 = Emp \ltimes Dept$.

Performance question:

- T(*Emp*)=100,000 and T(*Dept*)=500. Size of *ssn* and *mgrssn* = 9 bytes. The size of *name* and *deptName* is 50 bytes.
- Compute the network cost of this algorithm.

Parallel Operators

Question: How many of the following statements are true?

- i) A parallel sort can perform sorting on each node and then send the sorted sublists to a single node to be merged.
- ii) A semijoin gets its efficiency by only sending tuples that participate in the join.
- iii) The #1 rule for optimization is move the data to the query.
- iv) Intraquery parallelism is the execution of a single query in parallel.

A) 0

B) 1

C) 2

D) 3

E) 4

Distributed Transaction Model

Features of a distributed transaction model:

- Transactions may access data at several sites.
  - ⇨ A **local transaction** accesses data in the *single* site at which the transaction was initiated.
  - ⇨ A **global transaction** either accesses data in a site different from the one at which the transaction was initiated or accesses data in several sites.
- Each site has a local **transaction manager** responsible for:
  - ⇨ Maintaining a log for recovery purposes.
  - ⇨ Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a **transaction coordinator** responsible for:
  - ⇨ Starting the execution of transactions that originate at the site.
  - ⇨ Distributing subtransactions at appropriate sites for execution.
  - ⇨ Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed or aborted at all sites.

Distributed Concurrency Control

Concurrency control protocols must be modified to handle distributed databases.

- ◆Locking protocols may have to determine how to share lock information.
- ◆Propagating updates may be eager (immediate) or lazy (delayed).
- ◆Deadlock detection using wait-for graphs must handle detecting deadlocks across multiple servers.

Commit Protocols

Commit protocols are used to ensure atomicity across all sites:

- ◆A transaction which executes at multiple sites must either be committed at all the sites or aborted at all the sites.
- ◆It is not acceptable to have a transaction committed at one site and aborted at another.

The ***two-phase commit*** (**2*PC***) protocol is widely used.

The ***three-phase commit*** (**3*PC***) allows for faster recovery than 2PC as no site must wait. However, the protocol is more complicated/costly and does not handle network partitioning.

Two-Phase Commit Protocol (2PC)

The ***two-phase commit*** (**2*PC***) protocol is widely used to ensure atomicity across all sites.

The two-phase commit protocol assumes a *fail-stop* model.

- ◆Failed sites simply stop working and do not cause any other harm, such as sending incorrect messages to other sites.

Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.

The protocol involves all the local sites at which the transaction executed.

Phase 1: Obtaining a Decision

After all processing of a transaction is complete, the coordinator asks all participants to ***prepare to commit*** transaction *T*:

- ◆**"Prepare" Request:** Coordinator sends (**prepare** *T*) messages to all sites at which *T* executed.
  - ⇨Coordinator adds the record <**prepare** *T*> to the log and forces log to stable storage. Will wait for response with a timeout.

Upon receiving "Prepare" message, transaction manager at site determines if it can commit the transaction.

- ◆**"Abort" Response:** send (**abort** *T*) message to coordinator
  - ⇨Write <**abort** *T*> to the log and send (**abort** *T*) message to coordinator
- ◆**"Ready" Response:** send (**ready** *T*) message to coordinator if the transaction can be committed.
  - ⇨Write <**ready** *T*> to the log
  - ⇨force *all records* for *T* to stable storage
  - ⇨send (**ready** *T*) message to coordinator

Phase 2: Recording the Decision

T can be committed if coordinator received a (**ready** *T*) message from all the participating sites, otherwise *T* is aborted.

Coordinator adds a decision record <**commit** *T*> or <**abort** *T*> to the log and forces record onto stable storage.

Coordinator sends a message to each participant informing it of the decision (commit or abort).

Participants take appropriate action locally.

Two-Phase Commit (2PC) Protocol

Question: How many of the following statements are true?

- ◆i) The protocol uses two phases of message passing.
- ◆ii) The first phase sends a prepare to commit message to each site involved in the transaction.
- ◆iii) A site can respond to the prepare to commit message by sending either "ready" or "abort".
- ◆iv) If all sites respond with "ready" the coordinator, sends out a "commit" message to all participating sites.

A) 0

B) 1

C) 2

D) 3

E) 4

Handling Failures during 2PC

There are various possible failures during 2PC such as *site failure*, *coordinator failure*, and *network partitioning*.

Handling Site Failure:

- When site S_i recovers after failure, it examines its log to determine the fate of transactions active at the time of failure.
- If log contains <**commit** *T*> record, site executes **redo**(*T*).
- If log contains <**abort** *T*> record, site executes **undo**(*T*).
- If log contains <**ready** *T*> record, site must consult coordinator to determine the fate of *T*:
  - If *T* committed, **redo** (*T*) otherwise if *T* aborted, **undo** (*T*).
- If the log contains no control records concerning *T* means that site failed before responding to the <**prepare** *T*> message.
  - Since the failure of the site precludes the sending of such a response to the coordinator, site must abort *T* and executes **undo** (*T*).

Handling Failures during 2PC (2)

Handling Coordinator Failure:

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate.
  - If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  - If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  - If some active site does not contain a <**ready** *T*> record in its log, then the failed coordinator cannot have decided to commit *T*. Therefore abort *T*.
  - If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <abort T> of <commit T>). In this case active sites must wait for coordinator to recover, to find decision.

Blocking problem: Active sites may have to wait for failed coordinator to recover.

Handling Failures during 2PC (3)

Handling Network Partitioning:

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
  - Again, no harm results

Recovery and Concurrency Control

Recovery system must handle *in-doubt* transactions.

- Transactions that have a <**ready** *T*>, but neither a <**commit** *T*> nor an <**abort** *T*> log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites.
  - This can be slow and potentially block recovery.

Thus, recovery algorithms note lock info in the log:

- Instead of <**ready** *T*>, write out <**ready** *T*, *L*> where *L* = list of write locks held by *T* when the log is written.
- For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.

After re-acquiring locks, processing can resume.

- The commit/abort of in-doubt transactions is performed concurrently with execution of new transactions.
  - Note that new transactions may still have to wait on locks.

Handling Failures with 2PC

Question: How many of the following statements are true?

- i) If a site fails in 2PC, the transaction is always aborted.
- ii) If a coordinator fails in 2PC, the transaction is always aborted.
- iii) If a site fails and in recovery sees a "commit" entry in its log for a transaction, it performs "redo" as transaction is committed.
- iv) If a site is in a different network partition than the transaction coordinator, it always must wait for communication to coordinator to be fixed.

A) 0

B) 1

C) 2

D) 3

E) 4

2PC Question

Assume that a transaction T executes at 3 sites (S1,S2,S3) and was started at S2. The transaction completed its execution and the controller at S2 sent out prepare to commit message to all sites.

What happens if?

- 1) Site S3 replies with (abort T) message?
- 2) All sites reply with (ready T) messages but the coordinator S2 fails before it can make a decision?
- 3) All sites reply with (ready T) messages, the coordinator locally commits T and sends out commit messages but S1 fails before it gets the commit message.

Two-Phase Commit (2PC) Exercise

In groups of at least 3, act out the possible failure modes and how they are handled:

- 1) Failure of a site
- 2) Failure of coordinator
  - One site has <commit> in log
  - One site has <abort> in log
  - All sites have <ready> in log but no <commit> or <abort>
- 3) Network partitioned
  - All participants in same partition
  - Coordinator and one participant in a partition and another participant in the other partition

What is Integration/Virtualization?

Database integration and virtualization is combining the data in more than one database to have a consistent, global view.

- Typically, databases were developed independently and organization needs to combine data for reporting/analysis.
- Alternative to data warehousing which would involving moving data into a new system.

Database integration/virtualization systems must handle different operating systems, database systems, database schema designs, and query languages.

Other integration challenges:

- data model differences, naming conflicts, different database capabilities, no control over systems (autonomous)

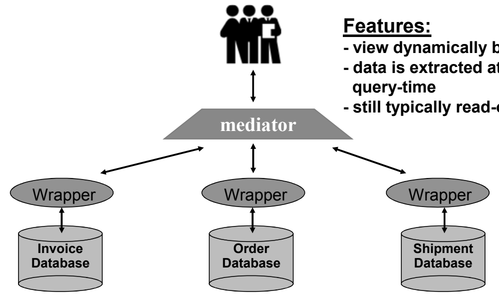
Integration/Virtualization using Mediators/Wrappers

Unlike integration using a data warehouse, integration architectures that use wrappers and mediators provide online access to operational systems.

Wrappers are software that converts global level queries into queries that the local database can handle. A ***mediator*** is global-level software that receives global queries and divides them into subqueries for execution by wrappers.

Unlike data warehouses, these systems are not suitable for very large decision-support queries because the data must be dynamically extracted from operational systems. They are useful for integrating operational systems without creating a single, unified database.

Query-Driven Dynamic Approach



Features:
- view dynamically built
- data is extracted at query-time
- still typically read-only

Cust(id,name,addr,city,state,cty)
Invoice(invId,custId,shipId,iDate)
InvProd(invId,prodId,amt,pr)
Prod(id,name,pr,desc)

Cust(id,name,addr,city,state,cty)
Order(oid,cid,odate)
OrdProd(oid,pid,amt,pr)
Prod(id,name,pr,desc)

Cust(id,name,addr,city,state,cty)
Shipment(shipid,oid,cid,shipdate)
ShipProd(shipid,prodid,amt)
Prod(id,name,pr,desc, inv)

Database Integration/Virtualization vs. Distributed Database Systems

Integrated database systems are similar to distributed database systems as they consist of a set of databases distributed over the network.

The major difference is that all databases in an integrated database system are ***autonomous***.

- They have their own unique schema, database administrator, transaction protocols, structures, and unique function.

This autonomy introduces complexities in determining an integrated view of the data, processing local and global transactions and concurrency control, and handling database system and model heterogeneity.

Key point: Nodes in a distributed database system work together while those in a multidatabase (virtualized) system do not.

Integration/Virtualization Challenges

Database integration is an active area of research. Common problems include:

- 1) **Schema matching and merging** - How can we create a single, global schema for users to query? Can this be done automatically?
- 2) **Global Query Optimization** - How do we optimize the execution of queries over independent data sources?
- 3) **Global Transactions and Updates** - Is it possible to efficiently support transactions over autonomous databases?

Using a Global View

Once a global view has been constructed, it can be used to query the entire system:

- ◆A user writes a query on the global view.
- ◆The mediator converts the query into queries on the local sources (views).
- ◆The queries are executed on the local sources and the answers integrated at the mediator before presentation to the user.

Schema Matching and Model Management

One challenging research problem is how do you automatically construct the global view?

Bernstein *et al.* have proposed model management and schema matching algorithms for this problem.

The schema matching problem takes as input two schemas and uses the names and types to determine matches between them.

- ◆A very challenging problem involving semantics, linguistics, and ontologies.

Transaction Management

Transaction management is somewhat similar to distributed databases with the existence of local and global transactions.

However, global transactions and local transactions are managed differently:

- ◆Local transactions are executed by each local DBMS, outside of the global system control. (*autonomy*)
- ◆Global transactions are executed under global system control and appear as regular local transactions at each local database system.

Transaction Management (2)

Respecting *local autonomy* requires that each LDBS cannot communicate directly to synchronize global transaction execution and the MDBS has no control over local transaction execution.

Thus, the global level mediation software must guarantee *global serializability* since each LDBS only guarantees local serializability.

- ◆Local concurrency control scheme needed to ensure that DBMS's schedule is serializable and must be able to guard against local deadlocks.

A schedule is globally serializable if there exists an ordering of committing global transactions such that all subtransactions of the global transactions are committed in the same order at all sites.

Approaches to MultiDatabase Transaction Management

Transaction management in a multidatabase has proceeded in 3 general directions:

- ◆Weakening autonomy of local databases
- ◆Enforcing serializability by using local conflicts
- ◆Relaxing serializability constraints by defining alternative notions of correctness

Still a great potential to make a contribution in this area!

Global Serializability using Tickets

Architecture:

- ◆Each site S_i has a special data item called a *ticket*.
- ◆Transaction T_j that runs at site S_i writes the ticket at site S_i.
- ◆Before a global transaction is allowed to commit, verify that there are no cycles based on tickets (optimistic protocol).
- ◆Pessimistic protocol allows global transaction manager to decide serial ordering of global transactions by controlling order in which tickets are accessed.

Ensures global transactions are serialized at each site, regardless of local concurrency control method, so long as the method guarantees local serializability.

Problems include hot spot at ticket and frequent aborts under heavy transaction loads (optimistic version).

Global Serialization Graph

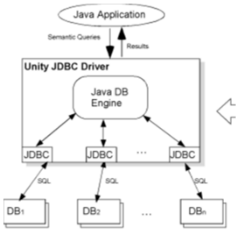
A global serialization graph (GSG) is used to determine if a global transaction can be committed using the tickets.

- The nodes of a GSG are “recently” committed transactions.
- An edge G_i -> G_j exists if at least one of the subtransactions of G_i preceded (had a smaller ticket that) one of G_j at any site.
- Initially the GSG contains no cycles.
- Add a node for the global transaction G to be committed and the appropriate edges.
- If a cycle exists abort G otherwise commit G.

My Research

My integration research built a JDBC driver called ***UnityJDBC*** that can query multiple databases at the same time.

- The system is based on the virtualization, mediator architecture.
- Contains a query parser, optimizer, and execution engine.
- Allows for cross-database joins (executed client-side).
- Previous students have worked on schema matching, high-level query languages, and optimization techniques.
- Still opportunities for further work.
- Driver is used as basis for MongoDB JDBC driver that allows querying MongoDB with SQL.



Summary

Parallel and distributed databases allow scalability by using more hardware for data storage and query processing.

- Goal is for increased performance, reliability, and availability.
- Data may be distributed, partitioned, and replicated.
- Queries are distributed across nodes.
- Specialized parallel algorithms and 2PC for transactions.

Database integration/virtualization combines data from multiple databases into a single virtual system.

- The ***global view*** may be *materialized* as in ***data warehouses*** or *virtual* as in ***mediator/wrapper systems***.
- Integrated databases must handle issues in concurrency control and recovery, global view generation and maintenance, and query execution and optimization.

Major Objectives

The "One Things":

- Explain the two phase commit (2PC) protocol and how sites recover after failure.

Major Theme:

- Distributed/parallel databases allow for increased performance but complicate concurrency control and recovery.

Objectives:

- Define replication and partitioning (horizontal and vertical).
- List advantages/disadvantages of partitioning.
- Explain how semijoins are used in distributed query processing.
- Use the 4 metrics for parallel systems.
- List some factors limiting speedup and scaleup.
- Define and give an example of skew.

Objectives (2)

Objectives:

- Be able to explain some challenges in constructing an integrated database system.
- Compare/contrast integrated databases and DDBS.
- Discuss and draw the mediator architecture.
- Give an example of naming and structural conflicts.
- Define the schema matching problem.
- Define globally serializable.
- Explain the ticket protocol.

COSC 404
Database System Implementation

Database Architectures

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Databases Architectures Not "One Size Fits All"

Relational databases are still the dominant database architecture and apply to many data management problems.

- Over $20 billion annual market in 2014.

However, recent research and commercial systems have demonstrated that "one size fits all" is not true. There are better architectures for classes of data management problems:

- Transactional systems: In-memory architectures
- Data warehousing: Column stores, parallel query processing
- Big Data: Massive scale-out with fault tolerance
- "NoSQL": simplified query languages/structures for high performance, consistency relaxation

Variety of Database Architectures

A database system provides independence from data storage and processing challenges. There are many different architectures/systems which are good for different use cases.

- Single (centralized) server database – easy to deploy/use
- Parallel database – for large query loads and data sizes
- Distributed database – for large-scale deployments (shared-nothing) with physical/geographical distribution
- Virtual (multi-)database – for integrating existing, autonomous databases
- Data warehouses – for decision support queries
- NoSQL databases – MongoDB, Cassandra, etc. supporting different data models

There are also lots of ways for implementing these architectures with associated algorithms.

Single (Centralized) Server Database

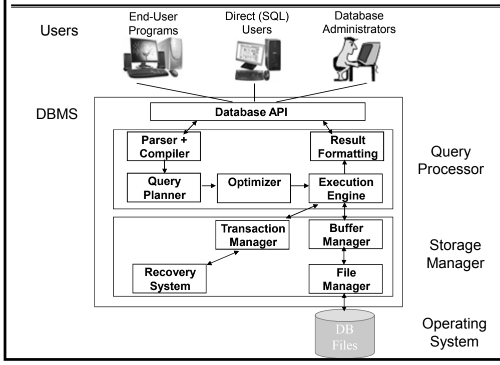
Single server centralized database systems such as MySQL, PostgreSQL, Oracle, and SQL Server have fairly standardized features and properties.

Ideal for: General-purpose databases (low cost/complexity)

Implementation details we studied:

- Data storage system, buffer manager
- Indexing algorithms and using indexes in practice
- Query processing/optimization of SQL
- Transactions, concurrency control, recovery
- Many systems also support distribution/replication/partitioning.
- Often, no parallelism within a query but can execute many queries simultaneously.
- Using JDBC API including PreparedStatements

Traditional Database System Architecture



Users

End-User Programs

Direct (SQL) Users

Database Administrators

DBMS

Database API

Parser + Compiler

Result Formatting

Query Planner

Optimizer

Execution Engine

Query Processor

Transaction Manager

Buffer Manager

Recovery System

File Manager

Storage Manager

DB Files

Operating System

Parallel Database Systems

A ***parallel database system*** consists of multiple processors and storage connected by a fast interconnection network.

Ideal for: processing time-consuming decision-support queries or providing high throughput for transaction processing within a single server/data center

Implementation details:

- replication and partitioning used for availability/performance
- parallel algorithms for relational operators
- modified algorithms for concurrency control and transactions
- query optimization must consider data location

Parallel Database Systems Greenplum

Greenplum is a shared-nothing, massively parallel (MPP) system where each node runs PostgreSQL.

Implementation:

- Cost-based optimizer factors in cost of moving data across nodes.
- Join and sort algorithms implemented in parallel across nodes and can move data between them.
- Utilizes log shipping and segment-level replication for fail-over.
- Supports SQL and Map-Reduce.
- Developed by Pivotal software (formerly part of EMC).

Distributed Database System

A ***distributed database system*** is a database system distributed across several network nodes that appears to the user as a single system.

Ideal for: high availability/reliability where large data set can be partitioned and queried across servers (often geographically)

Implementation details:

- Shared-nothing, massively parallel (MPP) architectures
- Concurrency control must determine how to handle replication and partitioning (eager versus lazy consistency)
- Scaling requires dividing workload across servers and intelligent data placement and query processing

Master/Slave Replication

Master/slave replication is supported by all major relational database systems (MySQL, PostgreSQL, Oracle, etc.).

Implementation details:

- 1) How are updates sent to slaves? Log shipping or real-time.
- 2) Slave nodes can except read requests but need to indicate when a transaction is read-only.
- 3) Slave nodes can take over from master if it fails.

Master/Master Replication

Master/master (multi-master) replication allows the data to be modified at more than one server. This requires coordination by the masters.

Techniques:

- Any update must be "approved" by all (or a majority) of the master servers. This approval may be done before commit (online) using a distributed algorithm (e.g. two phase commit).
- Updates may be allowed on multiple servers simultaneously, but there must be some system or user-configured resolution mechanism to handle conflicts.

Oracle

Oracle supports multiple servers with distributed features:

- database links between databases for querying other databases as if the data was local to Oracle (virtualization)
- supports remote/distributed transactions that involve one or more nodes (via database links and 2PC)
- does not perform auto-fragmentation/location transparency but does support user configurable horizontal partitioning
- Oracle supports both master/slave and multi-master replication using either synchronous or asynchronous propagation of changes between masters.
  - Different techniques for conflict resolution that user can control.
- Parallel execution of single SQL statement (joins, scans, sorts)

Oracle Real Application Clusters

Oracle Real Application Clusters (RAC) is a shared-storage architecture with multiple server nodes.

Provides support for clustering and high availability with multiple servers having concurrent access to the database and any server can process a transaction.

SQL Server

Microsoft SQL Server supports different use cases within its product including warehousing and in-memory databases.

- ◆In-memory tables and query processing for transactional
- ◆Data warehousing extensions and algorithms for analytics
- ◆Replication using master-slave and multi-master via log shipping, publish/subscribe, and merge conflict resolution
- ◆Linked servers (ODBC) for heterogeneous query processing and virtualization
- ◆Ability to scale from single server to multiple servers with high availability
- ◆Most "reasonably-priced" of the commercial systems
- ◆Very active database research laboratory

Database Architectures: NoSQL vs Relational

"NoSQL" databases are useful for several problems not well-suited for relational databases with some typical features:

- ◆**Variable data:** semi-structured, evolving, or has no schema
- ◆**Massive data:** terabytes or petabytes of data from new applications (web analysis, sensors, social graphs)
- ◆**Parallelism:** large data requires architectures to handle massive parallelism, scalability, and reliability
- ◆**Simpler queries:** may not need full SQL expressiveness
- ◆**Relaxed consistency:** more tolerant of errors, delays, or inconsistent results ("eventual consistency")
- ◆**Easier/cheaper:** less initial cost to get started

NoSQL is not really about SQL but instead developing data management architectures designed for scale.

- ◆NoSQL – "Not Only SQL"

Data Warehouse Architectures

A ***data warehouse*** is a historical database that summarizes, integrates, and organizes data from one or more operational databases in a format that is more efficient for analytical queries.

Ideal for: Large-scale analytic and decision-support queries

Implementation details:

- ◆Special storage formats (compressed, column stores)
- ◆Special index structures (bitmap indexes)
- ◆Optimized for reads over writes
- ◆Large query rather than large number of queries/updates so parallelism within a query is critical
- ◆May be relational or multidimensional (cubes).

In-Memory Databases

An ***in-memory database*** stores its working set of data in memory for improved response time.

Ideal for: high-volume, low-latency transactional systems

Implementation details:

- ◆May be single or multiple server
- ◆Data must be in memory. Persistent store used only in failure. Specialized memory queries (often have user pre-declare queries/transactions) – VoltDB, SQL Server, SAP HANA
- ◆Concurrency control and recovery system optimized for high throughput and unlikely failures

Batch Systems Map-Reduce

Batch systems like ***Map-Reduce*** designed for processing large-scale queries where the data may not be well-structured or pre-processed into a database engine.

Implementation Details:

- ◆Data often has limited structure (flat files, log files, CSV).
  - ⇨Massive amounts of data that may not be worth loading into a database.
- ◆Queries may take a LONG time so query processor must be resistant to failures with the ability to restart parts of the query that failed.
- ◆Many database vendors have ability to integrate with Hadoop File System and perform Map-Reduce queries.

Cloud Databases

Cloud databases are databases hosted by a service provider that allow for easy setup, administration and scaling.

- ◆Database as a service – databases hosted by provider, provide monitoring, backup, fail-over, high-availability, and ability to scale.

Examples: Google BigTable, Amazon RDS, DynamoDB, Redshift

Ideal for: Quick start without a server, minimal administration, scaling without expertise

Multi-Tenancy

Multi-tenancy is the ability to handle multiple customers (tenants) on the same database infrastructure. Approaches:

- ◆**Separate server** – each tenant has there own physical hardware, OS, DBMS
- ◆**Shared server, separate DBMS** – shared hardware but have multiple different DBMS running on hardware (maybe VMs)
- ◆**Shared database server, separate databases** – shared DBMS but different databases
- ◆**Shared database, separate schema** – same database but multiple schemas (user collection of objects)
- ◆**Shared database, shared schema** – customer data is differentiated by tenant id in all tables designed

Multi-Tenancy Issues

Multi-tenancy issues to consider:

- ◆Hardware and software costs
- ◆Efficient use of hardware resources
- ◆Isolation and security
- ◆Query performance
- ◆Ease of backup

Bottom Line

Bottom line: **No one size fits all.**

Select a database system based on your application and use case.

Understanding how database systems work and their architectures will help you make informed decisions on database systems to use and how to deploy them properly.

Survey Question: Lecture Value

Question: On a scale of 1 to 5 with 5 being the highest, how valuable/useful was the lecture time?

A) 1
B) 2
C) 3
D) 4
E) 5

Survey Question: Lab Value

Question: On a scale of 1 to 5 with 5 being the highest, how valuable/useful was the lab time and assignments?

A) 1
B) 2
C) 3
D) 4
E) 5

Survey Question: Workload

Question: On a scale of 1 to 5 with 1 being very low and 5 being very high, how was the overall workload compared to other courses and your expectations?

A) 1
B) 2
C) 3
D) 4
E) 5

Survey Question: Clicker Value

Question: On a scale of 1 to 5 with 5 being the highest, how valuable/useful were the clicker questions used in-class?

A) 1
B) 2
C) 3
D) 4
E) 5

Summary of Course

Our course goals were to understand database systems to:

1) Be a better, "expert" user of database systems.
2) Be able to use and compare different database systems.
3) Adapt the techniques when developing your own software.

We opened the database system "**black box**".

- Inside was storage, indexing, query processing/optimization, transactions, concurrency, recovery, distribution, lots of stuff!

You gained ***lots*** of industrial experience using a variety of databases and became a better, more experienced developer.

- MySQL, PostgreSQL, Microsoft SQL Server, MongoDB, JUnit, VoltDB, Java, JDBC, javacc, JSON, Map-Reduce, SQL

Thank you for a great course!

Good luck on the exam!