CMSC424: Database Design

Module: Introduction/Overview

Instructor: Amol Deshpande
amol@umd.edu

Motivation

- Book Chapters (6th Edition)
  - 1.1, 1.2
- Key Topics
  - Data-driven world and Big Data
  - Why managing large volumes of data is difficult
  - Drawbacks of using File Systems to store data
  - What we will cover in this course

Motivation: Data Overload

- Explosion of data, in pretty much every domain
  - Sensing devices and sensor networks that can monitor everything 24/7 from temperature to pollution to vital signs
  - Increasingly sophisticated smart phones
  - Internet, social networks makes it easy to publish data
  - Scientific experiments and simulations produce astronomical volumes of data
  - Internet of Things
  - Dataification: taking all aspects of life and turning them into data (e.g., what you like/enjoy turned into a stream of your "likes")
- How to handle that data? How to extract interesting actionable insights and scientific knowledge?
- Data volumes expected to get much worse

Four V's of Big Data

- Increasing data Volumes
  - Scientific data: 1.5GB/genome -- can be sequenced in .5 hrs; LHC generates 100TB of data a day
  - 500M tweets per day
  - As of 2012: 2.5 Exabytes of data created every day
  - EBay: Two data warehouses with 7.5PB and 40PB
  - Walmart: 583 terabytes of sales and inventory data
  - FICO monitors 2.5 billion active accounts worldwide
- Variety:
  - Structured data, spreadsheets, photos, videos, natural text, ...
- Velocity
- Veracity

Four V's of Big Data

- Increasing data Volumes
- Variety
- Velocity
  - Sensors, smart watches, etc., everywhere -- can generate tremendous volumes of "data streams"
  - Real-time analytics requires data to be consumed as fast as it is generated
- Veracity
  - How do you decide what to trust? How to remove noise? How to fill in missing values?
  - By various accounts, 90% or so of the time is spent in data cleaning and preparation, vs 10% or so on the machine learning/data science

Big Data and Data Science to the Rescue

- Terms increasingly used synonymously: also data analytics, data mining, business intelligence
  - Loosely used for any process where interesting things are inferred from data
  - Google search: “How Big Data Will Change”
- Data scientist called the sexiest job of the 21st century
  - The term has becoming very muddled at this point

Is it all hype?

- No: Extracting insights and knowledge from data very important, and will continue to increase in importance
  - Big data techniques are revolutionizing things in many domains like Education, Food Supply, Disease Epidemics, ...
- But: it is not much different from what we, especially statisticians, have been doing for many years
- What is different?
  - Much more data is digitally available than was before
  - Inexpensive computing + Cloud + Easy-to-use programming frameworks = Much easier to analyze it
  - Often: large-scale data + simple algorithms > small data + complex algorithms
    - Changes how you do analysis dramatically

Motivation: Data Overload

- How do we do anything with this data?
- Where and how do we store it ?
  - Disks are doubling every 18 months or so -- not enough
  - In many cases, the data is not actually recorded as it is; *summarized* first
- What if the disks crash ?
  - Very common, especially with 10,000's of disks

- How do we ensure "correctness" ?
  - What if the system crashes in the middle of an ATM transaction ?
    - Can't have money disappearing
  - What happens when a million people try to buy tickets to *<your favorite artist>'s* concert at the same time ?

Motivation: Data Overload

- What to do with the data ? How to process/analyze it ?
  - text search ?
    - Very limited
  - "find the stores with the maximum increase in sales in last month"
    - We can't expect the users to write Java programs
  - "how much time from here to Pittsburgh if I start at 2pm ?"
    - Data is there; more will be soon (GPS, live traffic data)
    - Requires predictive capabilities
  - Increasing need to convert "information" to "knowledge": **Data mining/Machine Learning**
    - "How many DVDs should we order?" (Netflix)
    - Find videos with this type of an event (say car break-ins)
    - Mine the "blogs" to detect "buzz"

9

Motivation: Data Overload

- Speed !!
  - With TB's of data, just finding something (even if you know what), is not easy
    - Reading a file with TB of data can take hours
  - Imagine a bank and millions of ATMs
    - How much time does it take you to do a withdrawal ?
    - The data is not local
- How do we guarantee the data will be there 10 years from now ?
- Privacy and security !!!
  - Every other day we see some database leaked on the web
  - How to make sure different users' data is protected from each other

10

Why not use file systems ?

- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
    - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task
  - Data isolation — multiple files and formats
  - Integrity problems
    - Integrity constraints (e.g., account balance > 0) become "buried" in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones

Why not use file systems ?

- Drawbacks of using file systems to store data:
  - Atomicity of updates
    - Failures may leave database in an inconsistent state with partial updates carried out
    - Example: Transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
    - Concurrent access needed for performance
    - Uncontrolled concurrent accesses can lead to inconsistencies
      - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
  - Security problems
    - Hard to provide user access to some, but not all, data

What we will cover…

- We will mainly discuss structured data
  - That can be represented in tabular forms (called Relational data)
  - We will spend some time on JSON/Document Data Model (MongoDB)
  - We will also spend some time on Mapreduce-like stuff (Apache Spark)
- Still the biggest and most important business (?)
  - Well defined problem with really good solutions that work
    - Contrast XQuery for XML vs SQL for relational
  - Solid technological foundations
- Many of the basic techniques however are directly applicable
  - E.g. reliable data storage etc.
  - Cf. Many recent attempts to add SQL-like capabilities, transactions to Mapreduce and related technologies
    - E.g., Spark DataFrames

Structure of the Course

- Introduction
  - Motivation, data abstraction, common data systems architectures today
- Relational Model + SQL (Two programming assignments)
- Schema Design: Entity-relationship Models and Normalization (Long-form Assgn)
  - How to create a database schema, and how to ensure it is "good"
- Implementation Issues (Programming assignment)
  - Different types of storage, and how to ensure reliability in presence of failures
  - Indexes for faster retrieval of data
  - How an SQL query is processed and optimized
- NoSQL (somewhat of a misnomer) (Programming assignment)
  - Document, key-value, and graph data models
  - MongoDB and its Query Language
  - Map-reduce Model and Apache Spark
- Transactions (Long-form Assignment)
  - How to do concurrent updates correctly
  - How to ensure consistency in presence of failures

Programming assignments may have small non-programming component, and vice versa

Summary

- Why study databases ?
  - Shift from *computation* to *information*
    - Always true in *corporate* domains
    - Increasing true for *personal* and *scientific* domains
  - Need has exploded in recent years
    - Data is growing at a very fast rate
  - Solving the data management problems is going to be a key
- Database Management Systems provide
  - Data abstraction: Key in evolving systems
  - Guarantees about data integrity
    - In presence of concurrent access, failures…
  - Speed !!

CMSC424: Database Design

Module: Introduction/Overview

Data Models and Data Abstraction

Instructor: Amol Deshpande
amol@cs.umd.edu

Motivation

- Book Chapters (6th Edition)
  - 1.3
- Key Topics
  - Data Models and Why Capturing "Structure" is Important
  - Data Abstraction, and Views
  - Logical and Physical Data Independence

Database Management Systems

- Provide a systematic way to solve data management issues
- Aim is to allow easy management of high volumes of data
  - Storing , Updating, Querying, Analyzing ….
- What is a Database ?
  - A large, integrated collection of (mostly *structured*) data
  - Typically models and captures information about a real-world ***enterprise***
    - Entities *(e.g. courses, students)*
    - Relationships *(e.g. **John** is taking **CMSC 424**)*
    - Usually also contains:
      - Knowledge of constraints on the data *(e.g. course capacities)*
      - Business logic *(e.g. pre-requisite rules)*
      - Encoded as part of the data model (preferable) or through external programs

Database Management Systems

- Massively successful for *highly structured data*
  - Why ? Structure in the data (if any) can be exploited for ease of use and efficiency
    - If there is no structure in the data, hard to do much
    - Contrast managing emails vs managing photos
  - Much of the data we need to deal with is highly structured
  - Some data is *semi-structured*
    - E.g.: Resumes, Webpages, Blogs etc.
  - Some has complicated structure
    - E.g.: Social networks
  - Some has no structure
    - E.g.: Text data, Video/Image data etc.

Structured vs Unstructured Data

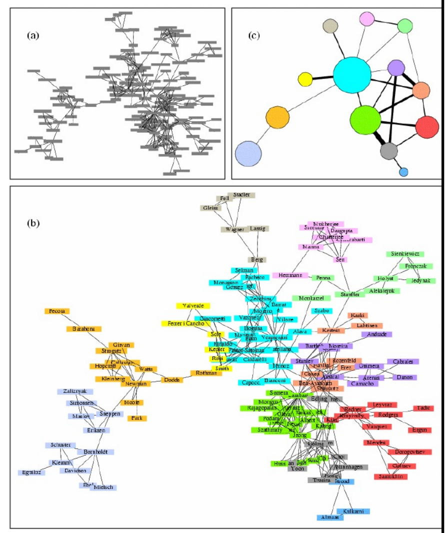
- A lot of the data we encounter is structured
  - Some have very simple structures
    - E.g. Data that can be represented in tabular forms
  - Significantly easier to deal with
  - We will focus on such data for much of the class

Account		
bname	**acct_no**	**balance**
Downtown	A-101	500
Mianus	A-215	700
Perry	A-102	400
R.H	A-305	350

Customer		
cname	**cstreet**	**ccity**
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield

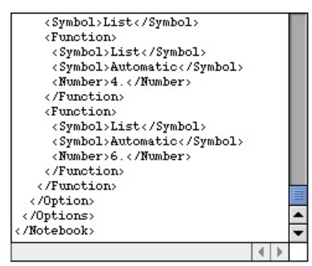
Structured vs Unstructured Data

- Some data has a little more complicated structure
  - E.g graph structures
    - Map data, social networks data, the web link structure etc
  - Can convert to tabular forms for storage, but may not be optimal
  - Queries often reason about graph structure
    - *Find my "Erdos number"*
    - *Suggest friends based on current friends*
  - Growing importance in recent years in a variety of domains: Biological, social networks, web…

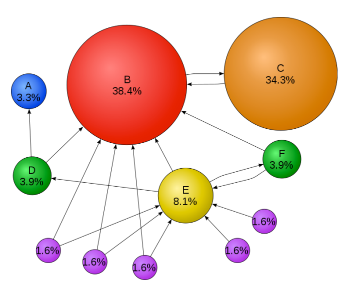


Structured vs Unstructured Data

- Increasing amount of data in a semi-structured format
  - XML – Self-describing tags (HTML ?)
  - Complicates a lot of things
  - We will discuss this toward the end



- A huge amount of data is unfortunately unstructured
  - Books, WWW
  - Amenable to pretty much only text search… so far
    - Information Retreival research deals with this topic
  - What about Google search ?
    - Google search is mainly successful because it uses the structure (in its original incarnation)
- Video ? Music ?
  - Can represent in DBMS's, but can't really operate on them



circle size == page importance == **pagerank**
more incoming links → higher pagerank
incoming links from important pages → higher pagerank

Database Management Systems

- Massively successful for *highly structured data*
  - Why ? Structure in the data (if any) can be exploited for ease of use and efficiency
  - How ?
  - Two Key Concepts:
    - Data Modeling: Allows reasoning about the data at a high level
      - e.g. "emails" have "sender", "receiver", "…"
      - Once we can describe the data, we can start "querying" it
    - Data Abstraction/Independence:
      - Layer the system so that the users/applications are insulated from the low-level details

Data Modeling

- Data modeling
  - **Data model**: A collection of concepts that describes how data is represented and accessed
  - **Schema:** A description of a specific collection of data, using a given data model
  - Some examples of data models that we will see
    - Relational, Entity-relationship model, XML, JSON…
    - Object-oriented, object-relational, semantic data model, RDF…
  - Why so many models ?
    - Tension between descriptive power and ease of use/efficiency
    - More powerful models → more data can be represented
    - More powerful models → harder to use, to query, and less efficient

Data Abstraction

- Probably *the* most important purpose of a DBMS
- Goal: Hiding *low-level details* from the users of the system
  - Alternatively: the principle that
    - *applications and users should be insulated from how data is structured and stored*
  - Also called *data independence*

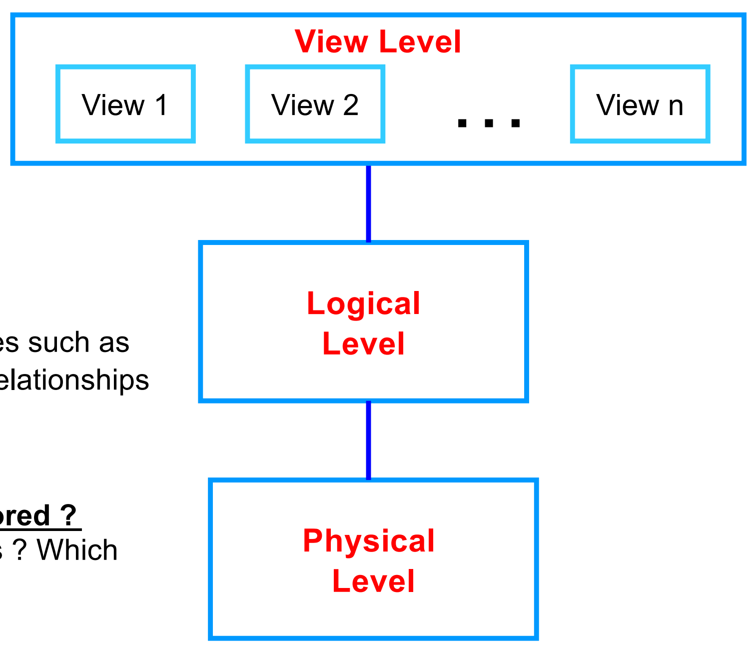
- Through use of *logical abstractions*

Data Abstraction

What data users and application programs see ?

What data is stored ?
describe data properties such as data semantics, data relationships

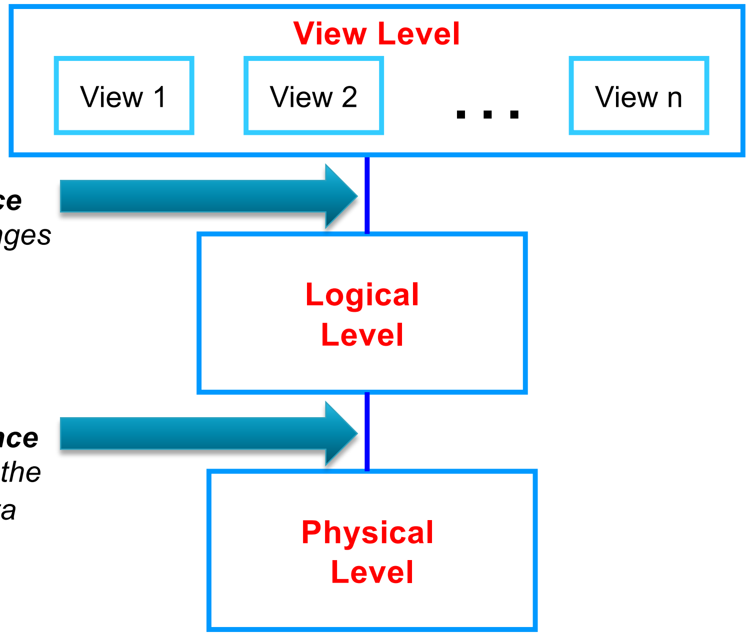
How data is actually stored ?
e.g. are we using disks ? Which file system ?



Data Abstraction

Logical Data Independence
Protection from logical changes to the schema

Physical Data Independence
Protection from changes to the physical structure of the data

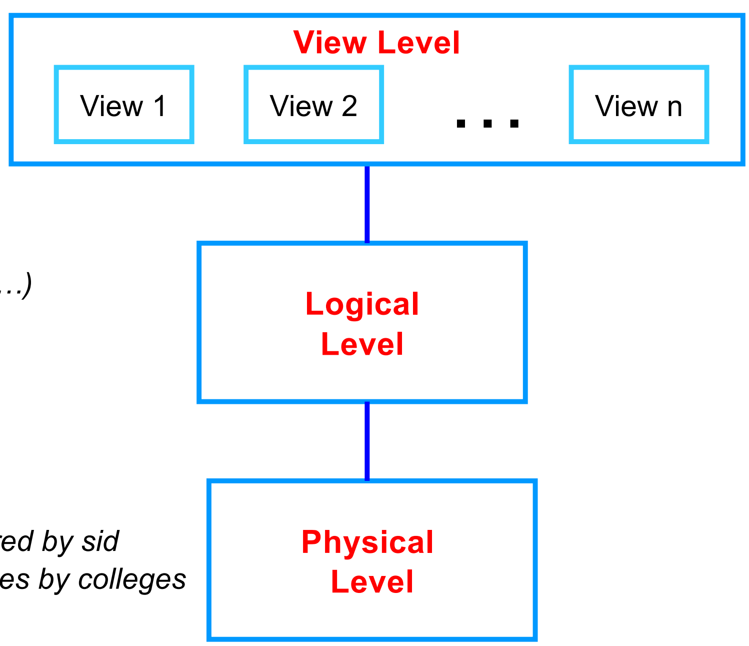


Data Abstractions: Example

A View Schema
course_info(#registered,…)

Logical Schema
students(sid, name, major, …)
courses(cid, name, …)
enrolled(sid, cid, …)

Physical Schema
all students in one file ordered by sid
courses split into multiple files by colleges



CMSC424: Database Design

Module: Introduction/Overview

DBMS Architectures; Industry Outlook

Instructor: Amol Deshpande
amol@umd.edu

Motivation

- Book Chapters (6th Edition)
  - 1.4, 1.9 (to some extent)
- Key Topics
  - Data Definition and Data Manipulation Languages
  - Typical Database Architecture
  - Current Industry Outlook

Database System

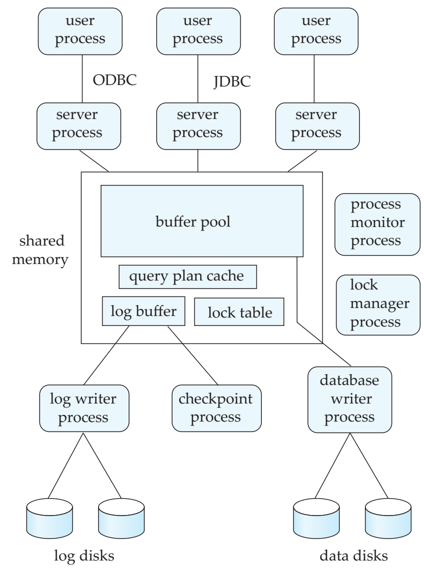
- A DBMS is a software system designed to store, manage, facilitate access to databases
  - Typically uses a specific ***data model***, and
  - Supports some level of ***physical and logical data independence***
- Provides:
  - Data Definition Language (DDL)
    - For defining and modifying the schemas
  - Data Manipulation Language (DML)
    - For retrieving, modifying, analyzing the data itself
  - Guarantees about correctness in presence of failures and concurrency, data semantics etc.
- Common use patterns
  - Handling transactions (e.g. ATM Transactions, flight reservations)
  - Archival (storing historical data)
  - Analytics (e.g. identifying trends, **Data Mining**)

Example: Relational DBMS and SQL

- **SQL** (sequel): Structured Query Language
- **Data definition (DDL)**
  - **create table** *instructor* (
    ID **char**(5),
    name **varchar**(20)**,**
    dept_name **varchar**(20),
    salary **numeric**(8,2))
- **Data manipulation (DML)**
  - Example: Find the name of the instructor with ID 22222
    select *name*
    from *instructor*
    where *instructor.ID* = ‘22222’

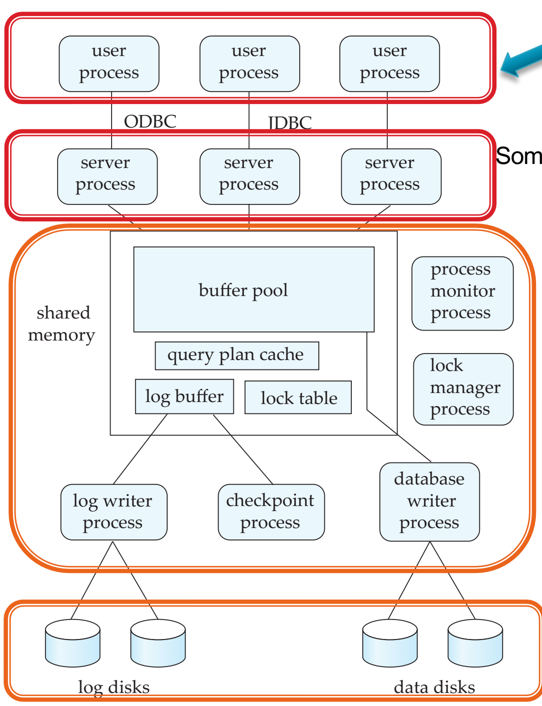
Database Architecture: Pre-2000's

- All data was typically in hard disks or arrays of hard disks
- RAM (Memory) was never enough
  - So always had to worry about what was in memory vs not
- Almost no real "distributed" execution
  - Different from "parallel", i.e., on co-located clusters of computers
- Relatively well-understood use cases
  - Report generation
  - Interactive data analysis and exploration
  - Supporting transactions



From Chapter 20

Traditional RDBMS Architecture



Clients may be anywhere – e.g., ATMs, desktops, laptops, web apps etc.

Talk to the database using standard protocols like JDBC/ODBC, SOAP, or REST (today), or proprietary protocols

Some sort of load balancer or intake mechanism

Typical components in a database system: some for queries, some for transactions

Maybe on a single physical computer or a cluster connected by a fast network

Data Storage Systems:
(1) Punch cards (long time ago)
(2) Hard disks (still prevalent)
(3) SSDs

Need "redundancy" and "fault-tolerance"
Data once stored should always be there

RAID = Redundant Array of Independent Disks

Database Architecture : Today

- Much more diversity in the architectures that we see
  - More modern hardware architectures
    - Massively parallel computers
    - SSDs
    - Massive amounts of RAM – often don't need to worry about data fitting in memory
    - Much faster networks, even over a wide area
    - Virtualization and Containerization
    - Cloud Computing
  - As a result: Data and execution typically distributed all over the place
- Much more diversity in data processing applications
  - Much more non-relational data (images, text, video)
  - Data Analytics/Machine learning more common use-cases
- Much more diversity in "data models"
  - Document data models (JSON, XML), Key-value data model, Graph data model, RDF

Current Industry Outlook

- Relational DBMSs
  - Oracle, IBM DB2, Microsoft SQL Server, Sybase, Amazon RDS/Aurora
- Open source alternatives
  - MySQL, PostgreSQL, BerkeleyDB (mainly a storage engine – no SQL) …
- Other Data Models
  - Neo4j (Graph), MongoDB (Document), CosmosDB (many)
- Data Warehousing Solutions
  - Geared towards very large volumes of data and on analyzing them
  - Long list: Teradata, Oracle Exadata, Netezza (based on FPGAs), Aster Data (founded 2005), Vertica (column-based), Kickfire, Xtremedata..
  - Usually sell package/services and charge per TB of managed data
  - Many (especially recent ones) start with MySQL or PostgreSQL and make them parallel/faster etc..

Web Scale Data Management, Analysis

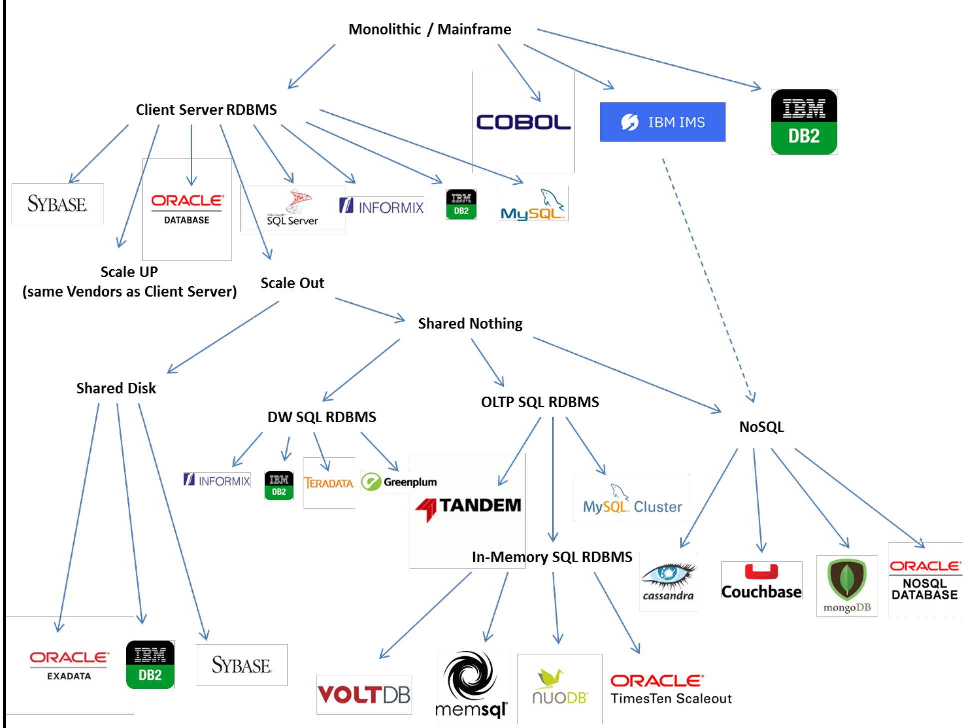
- Ongoing debate/issue
  - Cloud computing seems to eschew DBMSs in favor of homegrown solutions
  - E.g. Google, Facebook, Amazon etc…
- MapReduce: A paradigm for large-scale data analysis
  - Hadoop: An open source implementation
  - Apache Spark: a better open source implementation
- Why ?
  - DBMSs can't scale to the needs, not fault-tolerant enough
    - These apps don't need things like transactions, that complicate DBMSs (???)
  - Mapreduce favors Unix-style programming, doesn't require SQL
    - Try writing SVMs or decision trees in SQL
  - Cost
    - Companies like Teradata may charge $100,000 per TB of data managed

Current Industry Outlook

- Bigtable-like
  - Called "key-value stores"
  - Think highly distributed hash tables
  - Allow some transactional capabilities – still evolving area
  - PNUTS (Yahoo), Apache Cassandra (Facebook), Dynamo (Amazon), and many many others
- Mapreduce-like
  - Hadoop (open source), Pig (@Yahoo), Dryad (@Microsoft), Spark
  - Amazon EC2 Framework
  - Not really a database – but increasing declarative SQL-like capabilities are being added (e.g. HIVE at Facebook)
- Much ongoing research in industry and academia

In This Class…

- We have to limit the scope drastically
- Focus on:
  - Single-server Relational Databases
  - Assume hard disks are still important and memory is limited
  - Go deep into different ways to execute queries, and find the best queries
- Will briefly discuss:
  - Parallel architectures and query processing there
  - Map-reduce architectures and considerations there-in
- Most of the key concepts valid in modern databases (including NoSQL) and Big Data Frameworks



From: https://blogs.oracle.com/timesten/the-evolution-of-db-architectures
(Oracle-focused)

Data Warehouses

For: Large-scale data processing (TBs to PBs)

Parallel architectures (lots of co-located computers)
SQL and Reporting
No transactions

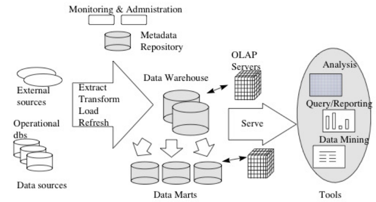
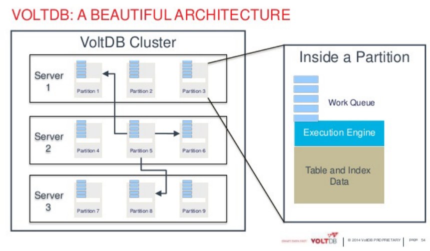


Figure 1. Data Warehousing Architecture

In-memory OLTP (on-line transaction processing)

For: Extremely fast transactions

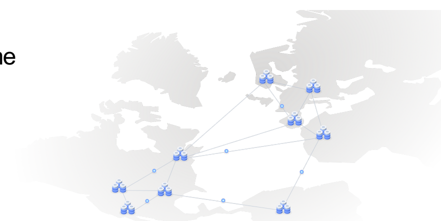
Many-core or parallel architectures
Very limited SQL – mostly focused on "writes"
Typically assume data fits in memory across servers



Highly available, distributed OLTP

For: Distributed scenarios where clients are all over the world

Focus on "consistency" – how to make sure all users see the same data
Limited SQL – mostly focused on "writes"
Considerations of memory vs disk less important

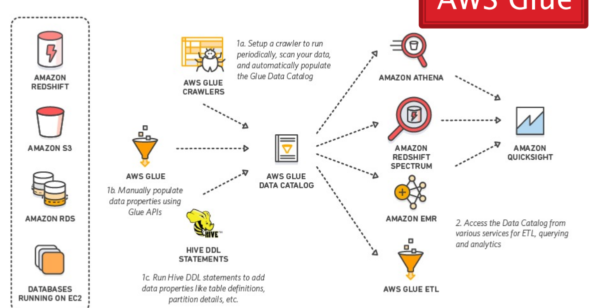


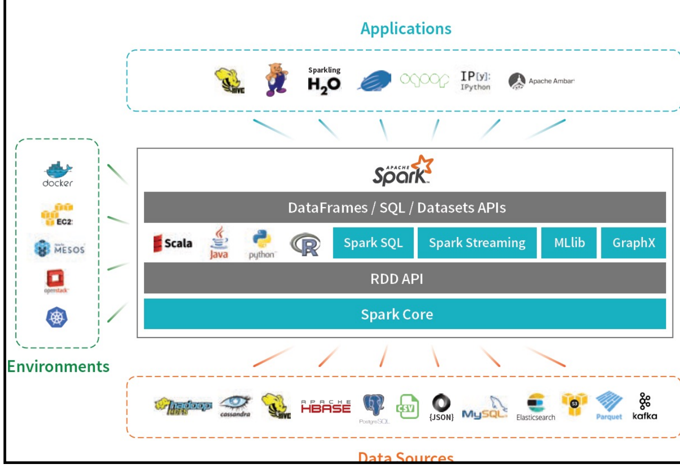
Extract-Transform-Load Systems, or Map-Reduce, or Big Data Frameworks

For: Large-scale, "ad hoc" data analysis

Mix of parallel and distributed architectures
Data usually coming from many different sources
Mix of SQL, Machine Learning, and ad hoc tasks (e.g., do image analysis, followed by SQL)

AWS Glue



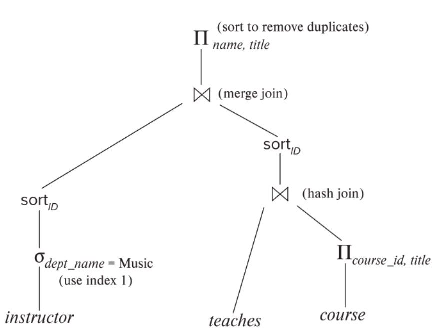


Apache Spark

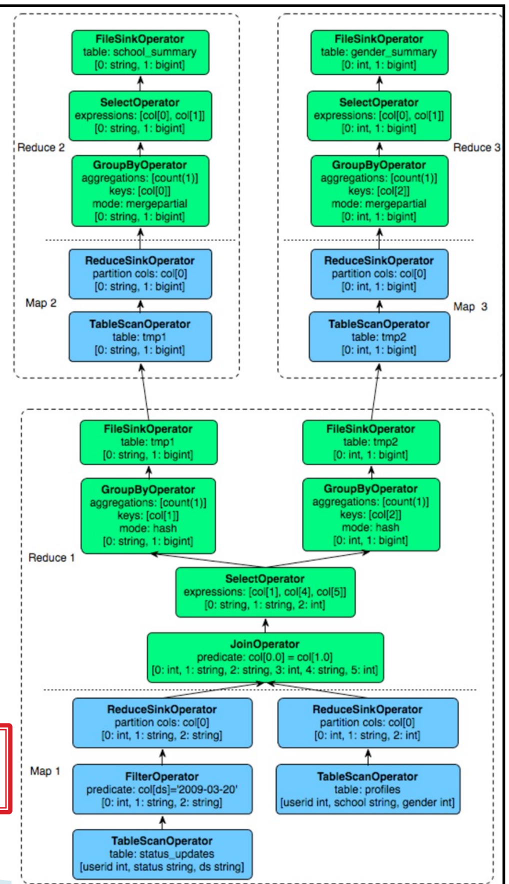
Okay…

- Key takeaway: Modern data architectures are a mess
  - We haven't talked about NoSQL (MongoDB, etc.), Machine Learning, "Streaming"…
- Fundamentals haven't changed that much though
  - We are still either:
    - Going from some "input datasets" to an "output dataset" (queries/analytics)
    - Modifying data (transactions)
  - SQL is still very common, albeit often disguised
    - Spark RDD operations map nicely to SQL joins and aggregates (unified now)
    - MongoDB lookups, filters, and aggregates map to joins, selects, and aggregates in SQL
- But "performance trade-offs" are all over the place now
  - 30 years ago, we worried a lot about hard disks and things fitting in memory
  - Today, focus more on networks
- Focus has shifted to other aspects of data processing pipelines
  - Analytics/Machine learning, data cleaning, statistics

Query Plans vs…

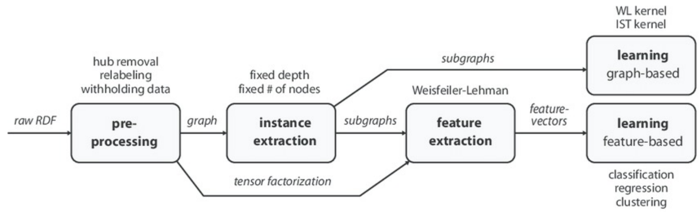


SQL "Query Plan"



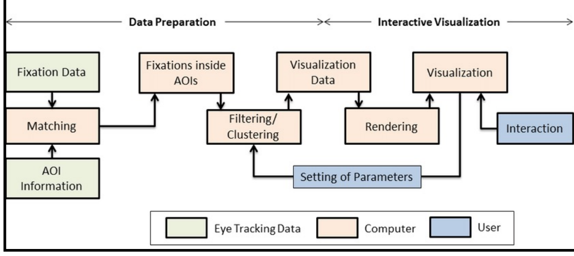
Apache Hive "Query Plan"
(Hive is an SQL layer on top of Hadoop)

vs … Data Transformation Pipelines



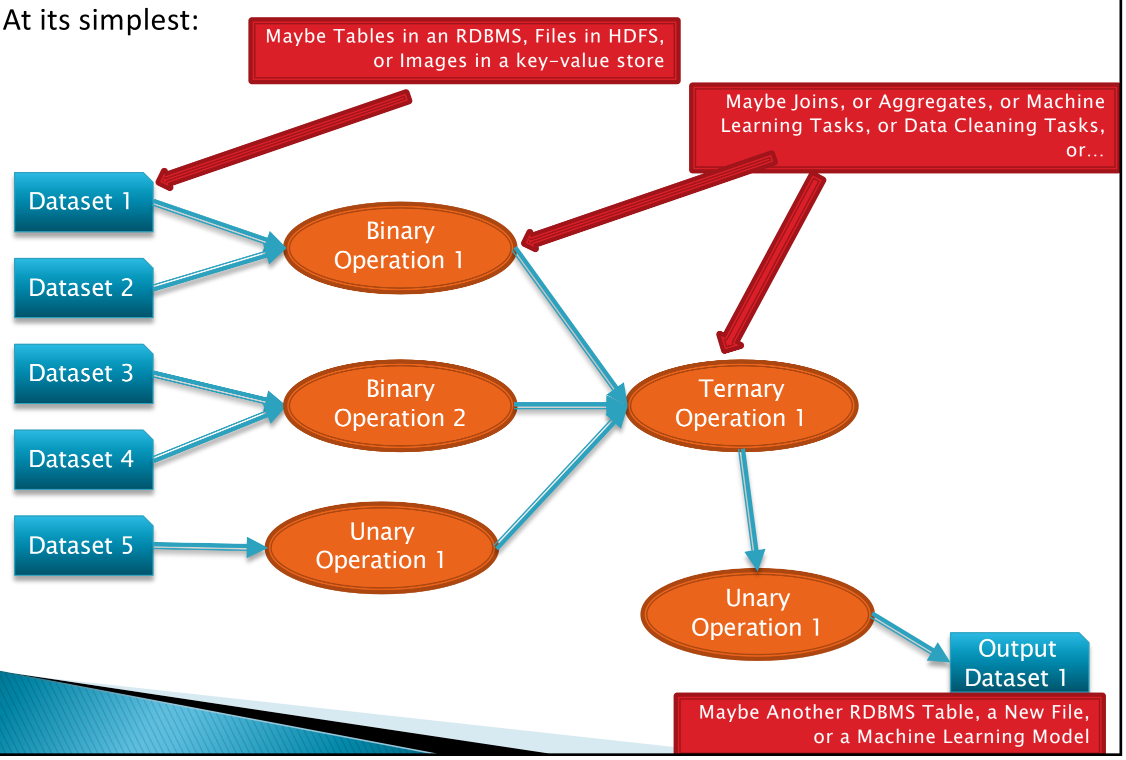
Machine Learning Pipeline

Data Preparation and Visualization Pipeline



Okay…

- Many similarities across different ways to process and analyze data
- At its simplest:



Okay…

- Many similarities across different ways to process and analyze data
- Some considerations that we see repeated:
  - Are there multiple ways to accomplish the goals?
    - i.e., are there multiple pipelines or SQL Query Plans that will accomplish the same task
  - How to "enumerate" all of them?
    - i.e., how to automatically come up with all the different options?
  - How to decide which is the "best"?
    - Ideally based on some consideration of total cost (e.g., total CPU time)
  - How to "find" the best plan?
    - Called "query optimization" in databases
- RDBMSs have been doing this for 4-5 decades now
  - The classic paper on SQL query optimization is from 1979
    - Outlined the approach still in use today
- Same ideas re-discovered repeatedly in other contexts (e.g., Hadoop)

How important is this today?

- Trade-offs shifted drastically over last 10-15 years
  - Especially with fast network, SSDs, and high memories
  - However, the volume of data is also growing quite rapidly
- Some observations:
  - Cheaper to access another computer's memory than accessing your own disk
  - Cache is playing more and more important role
  - Enough memory around that data often fits in memory of a single machine, or a cluster of machines
  - "Disk" considerations less important
    - Still: Disks are where most of the data lives today
  - Similar reasoning/algorithms required though

CMSC424: Database Design Module: Relational Model; SQL

Instructor: Amol Deshpande
amol@umd.edu

CMSC424: Database Design

Module: Relation Model + SQL

Relational Model

Instructor: Amol Deshpande
amol@cs.umd.edu

Relational Model

- Book Chapters (6th Edition)
  - 2.1, 2.2, 2.4
- Key Topics
  - Relational Model Key Concepts
  - Domains of Table Attributes
  - Null Values
  - Schema Diagrams

Relational Data Model

Introduced by Ted Codd (late 60's – early 70's)

- *Before = "Network Data Model" (Cobol as DDL, DML)*
- *Very contentious: Database Wars (Charlie Bachman vs. Ted Codd)*

Relational data model contributes:

1. *Separation of logical, physical data models (data independence)*
2. *Declarative query languages*
3. *Formal semantics*
4. *Query optimization (key to commercial success)*

1st prototypes:

- *Ingres → CA*
- *Postgres → Illustra → Informix → IBM*
- *System R → Oracle, DB2*

Key Abstraction: Relation

Account =

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Terms:

- Tables (aka: Relations)

Why called Relations?

Closely correspond to mathematical concept of a ***relation***

Relations

Account =

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Considered equivalent to…

{ (Downtown, A-101, 500),
(Brighton, A-201, 900),
(Brighton, A-217, 500) }

Relational database semantics defined in terms of mathematical relations

Relations

Account =

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Considered equivalent to…

{ (Downtown, A-101, 500),
(Brighton, A-201, 900),
(Brighton, A-217, 500) }

Terms:

- Tables (aka: Relations)
- Rows (aka: tuples)
- Columns (aka: attributes)
- Schema (e.g.: Acct_Schema = (bname, acct_no, balance))

Definitions

Relation Schema (or Schema)

A list of attributes and their domains

E.g. **account**(account-number, branch-name, balance)

Programming language equivalent: A variable (e.g. x)

Relation Instance

A particular instantiation of a relation with actual values

Will change with time

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Programming language equivalent: Value of a variable

Definitions

Domains of an attribute/column

The set of permitted values

e.g., bname must be String, balance must be a positive real number

We typically assume domains are **atomic,** i.e., the values are treated as indivisible (specifically: you can't store lists or arrays in them)

Null value

A special value used if the value of an attribute for a row is:

unknown (e.g., don't know address of a customer)

inapplicable (e.g., "spouse name" attribute for a customer)

withheld/hidden

Different interpretations all captured by a single concept – leads to major headaches and problems

Tables in a University Database

classroom(building, room_number, capacity)

department(dept_name, building, budget)

course(course_id, title, dept_name, credits)

instructor(ID, name, dept_name, salary)

section(course_id, sec_id, semester, year, building, room_number, time_slot_id)

teaches(ID, course_id, sec_id, semester, year)

student(ID, name, dept_name, tot_cred)

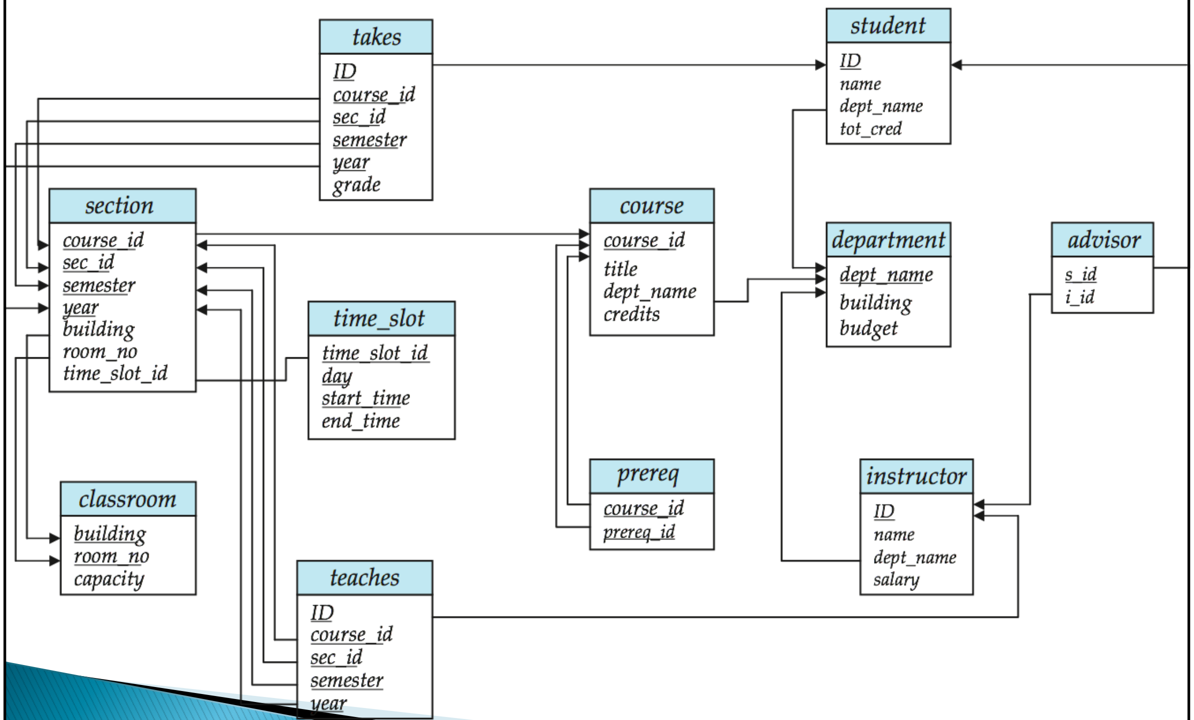
takes(Id, course_id, sec_id, semester, year, grade)

advisor(s_ID, i_ID)

time_slot(time_slot_id, day, start_time, end_time)

prereq(course_id, prereq_id)

Schema Diagram for University Database



CMSC424: Database Design

Module: Relation Model + SQL

SQL: Basics and DDL

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Basics and DDL

- Book Chapters (6th Edition)
  - 3.1, 3.2
- Key Topics
  - SQL Overview
  - How to create relations using SQL
  - How to insert/delete/update tuples

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.
- Several alternative syntaxes to write the same queries

Different Types of Constructs

- **Data definition language** (DDL): Defining/modifying schemas
  - **Integrity constraints:** Specifying conditions the data must satisfy
  - **View definition:** Defining views over data
  - **Authorization:** Who can access what
- **Data-manipulation language** (DML): Insert/delete/update tuples, queries
- **Transaction control:**
- **Embedded SQL:** Calling SQL from within programming languages
- **Creating indexes, Query Optimization control…**

Data Definition Language

The SQL **data-definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- Also: other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

SQL Constructs: Data Definition Language

- CREATE TABLE <name> (<field> <domain>, …)

```
create table department
   (dept_name varchar(20),
    building varchar(15),
    budget numeric(12,2) check (budget > 0),
    primary key (dept_name)
   );
```

```
create table instructor (
    ID        char(5),
    name      varchar(20) not null,
    dept_name varchar(20),
    salary    numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department
)
```


SQL Constructs: Data Definition Language

- CREATE TABLE <name> (<field> <domain>, …)

```
create table department
   (dept_name varchar(20) primary key,
    building varchar(15),
    budget numeric(12,2) check (budget > 0)
);
```

```
create table instructor (
    ID        char(5) primary key,
    name      varchar(20) not null,
    d_name    varchar(20),
    salary    numeric(8,2),
    foreign key (d_name) references department
)
```

SQL Constructs: Insert/Delete/Update Tuples

- INSERT INTO <name> (<field names>) VALUES (<field values>)

  insert into *instructor* **values** ('10211', 'Smith', 'Biology', 66000);

  insert into *instructor (name, ID)* **values** ('Smith', '10211');

  -- NULL for other two

  insert into *instructor (ID) values ('10211');*

  -- FAIL

- DELETE FROM <name> WHERE <condition>

  delete from department **where** budget < 80000;

  - Syntax is fine, but this command **may be rejected** because of referential integrity constraints.

SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>

  delete from department **where** budget < 80000;

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.

ID	*name*	*salary*	*dept_name*
10101	Srinivasan	65000	Comp. Sci.
12121	Wu	90000	Finance
15151	Mozart	40000	Music
22222	Einstein	95000	Physics
32343	El Said	60000	History
33456	Gold	87000	Physics
45565	Katz	75000	Comp. Sci.
58583	Califieri	62000	History
76543	Singh	80000	Finance
76766	Crick	72000	Biology
83821	Brandt	92000	Comp. Sci.
98345	Kim	80000	Elec. Eng.

Instructor relation

We can choose what happens:
(1) Reject the delete, or
(2) Delete the rows in Instructor (may be a cascade), or
(3) Set the appropriate values in Instructor to NULL

SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>

  delete from department **where** budget < 80000;

```
create table instructor
    (ID         varchar(5),
     name            varchar(20) not null,
     dept_name       varchar(20),
     salary          numeric(8,2) check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department
        on delete set null
    );
```

We can choose what happens:
(1) Reject the delete (nothing), or
(2) Delete the rows in Instructor (on delete cascade), or
(3) Set the appropriate values in Instructor to NULL (on delete set null)

SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>
  - Delete all classrooms with capacity below average

    delete from classroom **where** capacity <
    (**select avg(**capacity) **from** classroom);

  - Problem: as we delete tuples, the average capacity changes
  - Solution used in SQL:
    - First, compute **avg** capacity and find all tuples to delete
    - Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)
  - E.g. consider the query: delete the smallest classroom

SQL Constructs: Insert/Delete/Update Tuples

- UPDATE <name> SET <field name> = <value> WHERE <condition>
  - Increase all salaries's over $100,000 by 6%, all other receive 5%.
  - Write two update statements:

```
update instructor
 set salary = salary * 1.06
 where salary > 100000;

update instructor
 set salary = salary * 1.05
 where salary ≤ 10000;
```

  - The order is important
  - Can be done better using the <u>case</u> statement

SQL Constructs: Insert/Delete/Update Tuples

- UPDATE <name> SET <field name> = <value> WHERE <condition>
  - Increase all salaries's over $100,000 by 6%, all other receive 5%.
  - Can be done better using the <u>case</u> statement

```
update instructor
set salary =
        case
            when salary > 100000
                then salary * 1.06
            when salary <= 100000
                then salary * 1.05
        end;
```

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Querying Basics

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Querying Basics

- Book Chapters (6th Edition)
  - 3.3
- Key Topics
  - Single-table Queries in SQL
  - Multi-table Queries using Cartesian Product
  - Difference between Cartesian Product and “Natural Join”
  - Careful with using “natural join” keyword

Basic Query Structure

select A_1, A_2, ..., A_n ← Attributes or expressions
from r_1, r_2, ..., r_m ← Relations (or queries returning tables)
where P ← Predicates

Find the names of all instructors:
select *name*
from *instructor*

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of "**select** *name* **from** *instructor*".

Basic Query Structure

select A_1, A_2, ..., A_n ← Attributes or expressions
from r_1, r_2, ..., r_m ← Relations (or queries returning tables)
where P ← Predicates

Find the names of all instructor departments:
select *dept_name*
from *instructor*

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of "**select** *dept_name* **from** *instructor*".

Basic Query Structure

select $A_1, A_2, ..., A_n$ ← Attributes or expressions

from $r_1, r_2, ..., r_m$ ← Relations (or queries returning tables)

where P ← Predicates

Find the names of all instructors:
select *name*
from *instructor*

Remove duplicates:
select distinct *name*
from *instructor*

Order the output:
select distinct *name*
from *instructor*
order by *name* **asc**

Apply some filters (predicates):
select *name*
from *instructor*
where salary > 80000 **and** dept_name = 'Finance';

Basic Query Constructs

Find the names of all instructors:
select *name*
from *instructor*

Select all attributes:
select *
from *instructor*

Expressions in the select clause:
select *name, salary < 100000*
from *instructor*

More complex filters:
select *name*
from *instructor*
where (dept_name != 'Finance' **and** salary > 75000)
or (dept_name = 'Finance' **and** salary > 85000);

A filter with a subquery:
select *name*
from *instructor*
where dept_name in (**select** dept_name **from**
department **where** budget < 100000);

Basic Query Constructs

Renaming tables or output column names:
select *i.name, i.salary * 2* **as** *double_salary*
from *instructor i*
where *i.salary < 80000* **and** *i.name like '%g_';*

Find the names of all instructors:
select *name*
from *instructor*

More complex expressions:
select *concat(name, concat(', ', dept_name))*
from *instructor;*

Careful with NULLs:
select *name*
from *instructor*
where *salary < 100000* **or** *salary >= 100000;*

Wouldn't return the instructor with NULL salary (if any)

Multi-table Queries

Cartesian product:
select *
from *instructor, teaches*

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 3.6 The Cartesian product of the *instructor* relation with the *teaches* relation.

Multi-table Queries

Use predicates to only select "matching" pairs:
select *
from *instructor i, teaches t*
where *i.ID = t.ID;*

Cartesian product:
select *
from *instructor, teaches*

Identical (in this case) to using a natural join:
select *
from *instructor* **natural join** *teaches;*

Multi-table Queries

Cartesian product:
select *
from *instructor* ***natural join*** *teaches*

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.8 The natural join of the *instructor* relation with the *teaches* relation.

Multi-table Queries

Cartesian product:
select *
from *instructor, teaches*

Use predicates to only select "matching" pairs:
select *
from *instructor i, teaches t*
where *i.ID = t.ID;*

Identical (in this case) to using a natural join:
select *
from *instructor* **natural join** *teaches;*

Natural join does an equality on common attributes – doesn't work here:
select *
from *instructor* **natural join** *advisor;*

Instead can use "on" construct (or where clause as above):
select *
from *instructor* **join** *advisor* **on** *(i_id = id);*

Multi-table Queries

3-Table Query to get a list of instructor-teaches-course information:

select *i.name* **as** *instructor_name, c.title* **as** *course_name*
from *instructor i, course c, teaches*
where *i.ID = teaches.ID and c.course_id = teaches.course_id;*

Beware of unintended common names (happens often)
You may think the following query has the same result as above – it doesn't

select *name*, *title*
from *instructor* **natural join** *course* **natural join** *teaches;*

I prefer avoiding "natural joins" for that reason

Note: On the small dataset, the above two have the same answer, but not on the large dataset. Large dataset has cases where an instructor teaches a course from a different department.

CMSC424: Database Design

Module: Relation Model + SQL

Keys

Instructor: Amol Deshpande
amol@cs.umd.edu

Relational Model: Keys

- Book Chapters (6th Edition)
  - 2.3
- Key Topics
  - Keys as a mechanism to uniquely identify tuples in a relation
  - Super key vs Candidate key vs Primary key
  - Foreign keys and Referential Integrity
  - How to identify keys of a relation

Keys

- Let K ⊆ R
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of any possible relation r(R)
  - *Example: {ID} and {ID,name} are both superkeys of instructor.*
- Superkey K is **a candidate key** if K is minimal (i.e., no subset of it is a superkey)
  - *Example: {ID} is a candidate key for Instructor*
- One of the candidate keys is selected to be the **primary key**
  - Typically one that is small and immutable (doesn't change often)
- Primary key typically highlighted (e.g., underlined)

Tables in a University Database

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)

Tables in a University Database

takes(ID, course_id, sec_id, semester, year, grade)

What about ID, course_id?

No. May repeat:

("1011049", "CMSC424", "101", "Spring", 2014, D)

("1011049", "CMSC424", "102", "Fall", 2015, null)

What about ID, course_id, sec_id?

May repeat:

("1011049", "CMSC424", "101", "Spring", 2014, D)

("1011049", "CMSC424", "101", "Fall", 2015, null)

What about ID, course_id, sec_id, semester?

Still no: ("1011049", "CMSC424", "101", "Spring", 2014, D)

("1011049", "CMSC424", "101", "Spring", 2015, null)

Tables in a University Database

classroom(building, room_number, capacity)

department(dept_name, building, budget)

course(course_id, title, dept_name, credits)

instructor(ID, name, dept_name, salary)

section(course_id, sec_id, semester, year, building, room_number, time_slot_id)

teaches(ID, course_id, sec_id, semester, year)

student(ID, name, dept_name, tot_cred)

takes(ID, course_id, sec_id, semester, year, grade)

advisor(s_ID, i_ID)

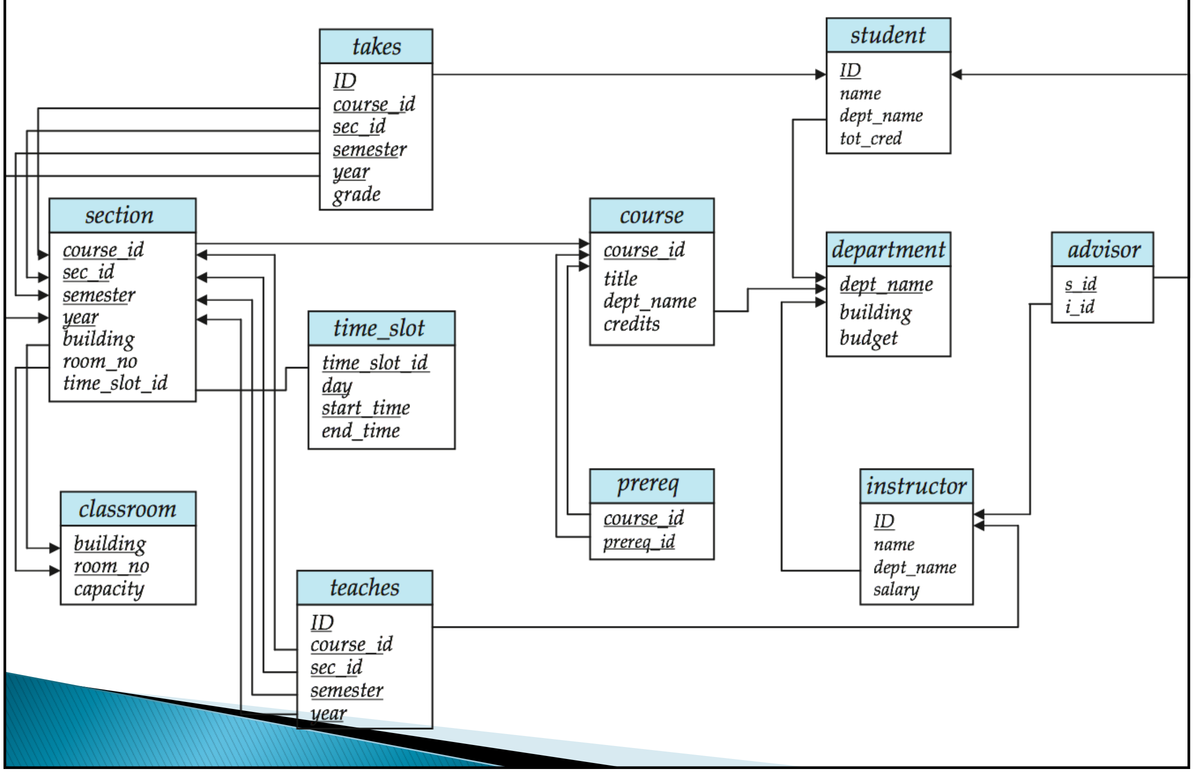
time_slot(time_slot_id, day, start_time, end_time)

prereq(course_id, prereq_id)

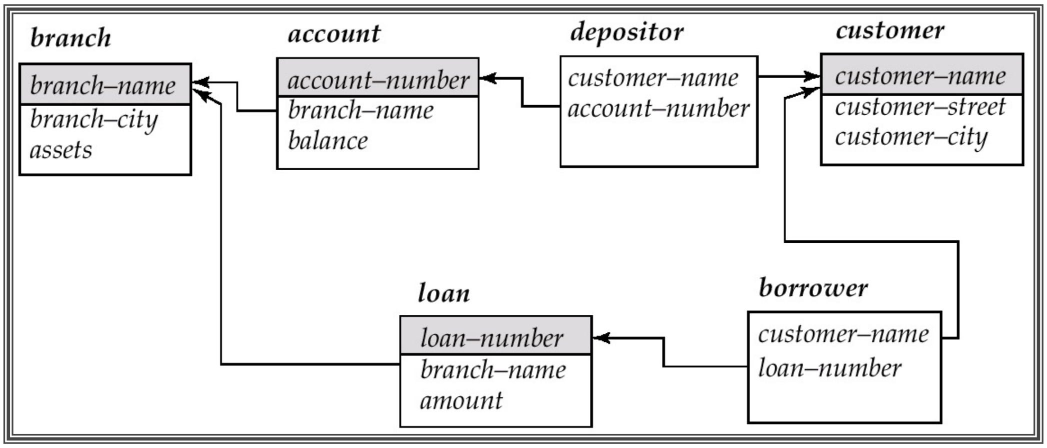
Keys

- **Foreign key:** Primary key of a relation that appears in another relation
  - {ID} from *student* appears in *takes, advisor*
  - *student* called ***referenced*** relation
  - *takes* is the ***referencing*** relation
  - Typically shown by an arrow from referencing to referenced
- **Foreign key constraint**: the tuple corresponding to that primary key must exist
  - Imagine:
    - Tuple: (‘student101’, ‘CMSC424’) in *takes*
    - But no tuple corresponding to ‘student101’ in *student*
  - Also called ***referential integrity constraint***

Schema Diagram for University Database



Schema Diagram for the Banking Enterprise



Examples

- Married(person1_ssn, person2_ssn, date_married, date_divorced)
- Account(cust_ssn, account_number, cust_name, balance, cust_address)
- RA(student_id, project_id, superviser_id, appt_time, appt_start_date, appt_end_date)
- Person(Name, DOB, Born, Education, Religion, …)
  - *Information typically found on Wikipedia Pages*

Examples

- Married(person1_ssn, person2_ssn, date_married, date_divorced)
- Account(cust_ssn, account_number, cust_name, balance, cust_address)
  - If a single account per customer, then: cust_ssn
  - Else: (cust_ssn, account_number)
    - In the latter case, this is not a good schema because it requires repeating information
- RA(student_id, project_id, superviser_id, appt_time, appt_start_date, appt_end_date)
  - Could be smaller if there are some restrictions – requires some domain knowledge of the data being stored
- Person(Name, DOB, Born, Education, Religion, …)
  - *Information typically found on Wikipedia Pages*
  - *Unclear what could be a primary key here: you could in theory have two people who match on all of those*

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Aggregates

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Aggregates

- Book Chapters (6th Edition)
  - 3.7.1-3.7.3
- Key Topics
  - Basic aggregates
  - Aggregation with "grouping"
  - "Having" clause to select among groups

Aggregates

Other common aggregates:
max, min, sum, count, stdev, …

select count (**distinct** *ID*)
from *teaches*
where *semester* = 'Spring' **and** *year* = 2010

Find the average salary of instructors in the Computer Science
select ***avg(****salary)*
from *instructor*
where *dept_name = 'Comp. Sci';*

Can specify aggregates in any query.

Find max salary over instructors teaching in S'10
select max(*salary)*
from *teaches* **natural join** *instructor*
where *semester* = 'Spring' **and** *year* = 2010;

Aggregate result can be used as a scalar.
Find instructors with max salary:
select *
from *instructor*
where *salary* **= (select max***(salary)* **from** *instructor);*

Aggregates

Aggregate result can be used as a scalar.
Find instructors with max salary:
select *
from *instructor*
where *salary* **= (select max***(salary)* **from** *instructor);*

Following doesn't work:

select *
from *instructor*
where *salary* **= max***(salary);*

select *name,* **max***(salary)*
from *instructor*
where *salary* **= max***(salary);*

Aggregates: Group By

Split the tuples into groups, and computer the aggregate for each group
select *dept_name*, **avg** (*salary*)
from *instructor*
group by *dept_name*;

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregates: Group By

Find the number of instructors in each department who teach a course in the Spring 2010 semester.

```
Partial Query 1:
select
from instructor natural join teaches
where semester = 'Spring' and year = 2010
```

ID	name	dept_name	salary	course_id	sec_id	semester	year
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~CS-101~~	~~1~~	~~Fall~~	~~2009~~
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~CS-347~~	~~1~~	~~Fall~~	~~2009~~
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
~~22222~~	~~Einstein~~	~~Physics~~	~~95000~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
~~76766~~	~~Crick~~	~~Biology~~	~~72000~~	~~BIO-101~~	~~1~~	~~Summer~~	~~2009~~
~~76766~~	~~Crick~~	~~Biology~~	~~72000~~	~~BIO-301~~	~~1~~	~~Summer~~	~~2010~~
~~83821~~	~~Brandt~~	~~Comp. Sci.~~	~~92000~~	~~CS-190~~	~~1~~	~~Spring~~	~~2009~~
~~83821~~	~~Brandt~~	~~Comp. Sci.~~	~~92000~~	~~CS-190~~	~~2~~	~~Spring~~	~~2009~~
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
~~98345~~	~~Kim~~	~~Elec. Eng.~~	~~80000~~	~~EE-181~~	~~1~~	~~Spring~~	~~2009~~

Aggregates: Group By

Find the number of instructors in each department who teach a course in the Spring 2010 semester.

```
Partial Query 2:
select dept_name, count(*)
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name
```

Doesn't work – double counts "Katz" who teaches twice in Spring 2010

```
Final:
select dept_name, count(distinct ID)
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name
```

Aggregates: Group By

Attributes in the select clause must be aggregates, or must appear in the group by clause. Following wouldn't work

select *dept_name*, ID, **avg** (*salary*)
from *instructor*
group by *dept_name*;

"having" can be used to select only some of the groups.

select *dept_name*, ID, **avg** (*salary*)
from *instructor*
group by *dept_name*
having avg(*salary***)** *> 42000;*

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Different Types of Joins, and Set Operations

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Querying Basics

- Book Chapters (6th Edition)
  - 4.1, 3.5

- Key Topics
  - Outer Joins
  - Anti-joins, Semi-joins
  - Set Operations

Multi-table Queries

Cartesian product:
select *
from *R*, *S*

A	B
a	1
b	1
c	2

R

×

B	C
1	x
3	y
4	z

S

=

R.A	R.B	S.B	S.C
a	1	1	x
a	1	3	y
a	1	4	z
b	1	1	x
b	1	3	y
b	1	1	x
c	2	3	y
c	2	1	x
c	2	3	y

Multi-table Queries

Natural Join:
select *
from *R natural join S*

A	B
a	1
b	1
c	2

R

⋈

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x

Equivalent to:
select *R.A, R.B, S.C*
from *R, S*
where R.B = S.B

Equivalent to:
select *R.A, R.B, S.C*
from *R* ***join*** *S* ***on*** (R.B = S.B)

Equivalent to:
select *R.A, R.B, S.C*
from *R* ***join*** *S* ***on*** (B)

Outer joins: Why?

Natural Join:
select *
from *R natural join S*

A	B
a	1
b	1
c	2

R

⋈

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x

Often need the ”non-matching” tuples in the result

"Left" Outerjoin

select *
from *R natural left outer join S*

R

A	B
a	1
b	1
c	2

⟕

S

B	C
1	x
3	y
4	z

=

R.A	B	S.C
a	1	x
b	1	x
c	2	NULL

select *
from *R left outer join S on (R.B = S.B)*

"Right" Outerjoin

select *
from *R right natural outer join S*

R

A	B
a	1
b	1
c	2

⟖

S

B	C
1	x
3	y
4	z

=

R.A	B	S.C
a	1	x
b	1	x
NULL	3	y
NULL	4	z

select *
from *R right outer join S on (R.B = S.B)*

"Full" Outerjoin

select ***
from *R natural full outer join S*

R

A	B
a	1
b	1
c	2

⟗

S

B	C
1	x
3	y
4	z

=

R.A	B	S.C
a	1	x
b	1	x
c	2	NULL
NULL	3	y
NULL	4	z

select ***
from *R full outer join S on (R.B = S.B)*

Semi-joins

R SEMI-JOIN S = tuples of R that do have a "match" in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

⋉

B	C
1	x
3	y
4	z

=

R.A	R.B
a	1
b	1

Can be written in SQL as:

```
select *
from R
where B in (select B from S);
```

Semi-joins

R SEMI-JOIN S = tuples of R that do have a “match” in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

$\ltimes$

B	C
1	x
3	y
4	z

=

R.A	R.B
a	1
b	1

A	B
a	1
b	1
c	2

$\rtimes$

B	C
1	x
3	y
4	z

=

S.B	S.C
1	x

Anti-joins

R ANTI-JOIN S = tuples of R that do NOT have a “match” in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

$\triangleright$

B	C
1	x
3	y
4	z

=

R.A	R.B
c	2

Can be written in SQL as:

```
select *
from R
where B not in (select B from S);
```

Anti-joins

R ANTI-JOIN S = tuples of R that do NOT have a "match" in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

▷

B	C
1	x
3	y
4	z

=

R.A	R.B
c	2

A	B
a	1
b	1
c	2

◁

B	C
1	x
3	y
4	z

=

S.B	S.C
3	y
4	z

Set operations

Find courses that ran in Fall 2009 or Spring 2010

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
union
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010);

In both:

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
intersect
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010);

In Fall 2009, but not in Spring 2010:

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
except
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010);

Set operations: Duplicates

Union/Intersection/Except eliminate duplicates in the answer (the other SQL commands don't) (e.g., try 'select dept_name from instructor').

Can use "union all" to retain duplicates.

NOTE: The duplicates are retained in a systematic fashion (for all SQL operations)

Suppose a tuple occurs m times in r and n times in s, then, it occurs:

- $m + n$ times in r **union all** s
- min(m,n) times in r **intersect all** s
- max(0, $m - n$) times in r **except all** s

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Nested Subqueries

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Nested Subqueries

- Book Chapters (6th Edition)
  - 3.8
- Key Topics
  - Subqueries
  - Boolean operations with Subqueries

Nested Subqueries

- A query within a query – can be used in select/from/where and other clauses

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
      course_id in (select course_id
                    from section
                    where semester = 'Spring' and year= 2010);
```

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

```
select dept_name,
       (select count(*)
        from instructor
        where department.dept_name = instructor.dept_name)
       as num_instructors
from department;
```

Nested Subqueries

- A query within a query – can be used in select/from/where and other clauses

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
      course_id in (select course_id
                    from section
                    where semester = 'Spring' and year= 2010);
```

Uncorrelated subquery – the subquery makes no reference to the enclosing queries, and can be evaluated by itself

* Correlated subquery – the subquery has a reference to the enclosing query
* For every tuple of department, the subquery returns a different result

```
select dept_name,
      (select count(*)
       from instructor
       where department.dept_name = instructor.dept_name)
      as num_instructors
from department;
```


Set Membership: "IN" and "NOT IN"

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
      course_id in (select course_id
                    from section
                    where semester = 'Spring' and year= 2010);
```

Can also be written using Set Intersection

```
(select course_id from section where semester = 'Fall' and year = 2009)
 intersect
(select course_id from section where semester = 'Spring' and year = 2010);
```

Set Membership: ”IN” and “NOT IN”

Can do this with “tuples” as well:

select count (**distinct** *ID*)
from *takes*
where (*course_id*, *sec_id*, *semester*, *year*) **in** (**select** *course_id*, *sec_id*, *semester*, *year*
from *teaches*
where *teaches*.*ID*= 10101);

Set Comparisons

select *name*
from *instructor*
where *salary* > **some** (**select** *salary*
from *instructor*
where *dept_name* = 'Biology');

select *name*
from *instructor*
where *salary* > **all** (**select** *salary*
from *instructor*
where *dept_name* = 'Biology');

Testing for Empty Results

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
          exists (select *
                    from section as T
                    where semester = 'Spring' and year= 2010 and
                              S.course_id= T.course_id);
```

Also: "Not Exists"

Uniqueness

```
select T.course_id
from course as T
where unique (select R.course_id
                    from section as R
                    where T.course_id= R.course_id and
                              R.year = 2009);
```

There are usually alternatives to using these constructs (e.g., group by + having instead of "unique"), but these can often make queries more readable and more compact

"With" Clause

Used for creating "temporary" tables within the context of the query

```
with dept_total (dept_name, value) as
      (select dept_name, sum(salary)
       from instructor
       group by dept_name),
dept_total_avg(value) as
      (select avg(value)
       from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```


Scalar Subqueries

A scalar subquery is one that returns exactly one tuple with exactly one attribute (so typically some sort of aggregate) – can be used in "select", "where", and "having" clauses

```
select dept_name,
         (select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
         as num_instructors
from department;
```

```
delete from instructor
where salary< (select avg (salary)
                from instructor);
```

CMSC424: Database Design

Module: Relation Model + SQL

SQL: NULLs

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL: NULLs

- Book Chapters (6th Edition)
  - 3.6, 3.7.4
- Key Topics
  - Operating with NULLs
  - ”Unknown” as a new Boolean value
  - Operating with UNKNOWNs
  - Aggregates and NULLs

SQL: Nulls

Can cause headaches for query semantics as well as query processing and optimization)

Can be a value of any attribute

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

What does this mean?

(unknown) We don't know Waltham's assets?

(inapplicable) Waltham has a special kind of account without assets

(withheld) We are not allowed to know

SQL: Nulls

Arithmetic Operations with `Null`

`n + NULL = NULL` (similarly for all *arithmetic ops*: `+, -, *, /, mod,` …)

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

```
SELECT bname, assets * 2 as a2
FROM branch
```

=

bname	a2
Downtown	18M
Perry	3.4M
Mianus	.8M
Waltham	NULL

Counter-intuitive: NULL * 0 = NULL

SQL: Nulls

Boolean Operations with `Null`

`n < NULL = UNKNOWN` (similarly for all *boolean ops*: `>, <=, >=, <>, =,` …)

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

assets < 10M will evaluate to UNKNOWN for the last tuple

But what about:

(assets < 10M) or (bcity = 'Boston') ?

(assets < 10M) and (bcity = 'Boston')?

SQL: Unknown

`FALSE OR UNKNOWN = UNKNOWN`

`TRUE AND UNKNOWN = UNKNOWN`

`FALSE AND UNKNOWN = FALSE`

`TRUE OR UNKNOWN = TRUE`

`UNKNOWN OR UNKNOWN = UNKNOWN`

`UNKNOWN AND UNKNOWN = UNKNOWN`

`NOT (UNKNOWN) = UNKNOWN`

Intuition: substitute each of TRUE, FALSE for unknown. If different answer results, results is unknown

Values	Expression	Result
x = NULL, y = 10	(x < 10) and (y = 20)	UNKNOWN and FALSE = FALSE
x = NULL, y = 10	(x is NULL) and (y = 20)	TRUE and FALSE = FALSE
x = NULL, y = 10	(x < 10) and (y = 10)	UNKNOWN and TRUE = UNKNOWN
x = NULL, y = 10	(x < 10) is UNKNOWN	TRUE
x = NULL, y = 10	((x < 10) is UNKNOWN) and (y = 10)	TRUE AND TRUE = TRUE

UNKNOWN tuples are not included in final result

Aggregates and NULLs

Given

branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

Aggregate Operations

```
SELECT SUM (assets) =
FROM branch
```

SUM
11.1 M

NULL *is ignored for SUM*

Same for AVG *(3.7M),* MIN *(0.4M),* MAX *(9M)*

Also for COUNT(assets) -- returns 3

But COUNT (*) *returns*

COUNT
4

Aggregates and NULLs

Given

branch =

bname	bcity	assets

```
SELECT SUM (assets) =
FROM branch
```

SUM
NULL

- *Same as* AVG, MIN, MAX
- *But* COUNT (assets) *returns*

COUNT
0

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Transactions, Functions, Procedures, Recursive Queries, Authorization

Instructor: Amol Deshpande
amol@cs.umd.edu

Miscellaneous SQL

- Book Chapters (6th Edition)
  - Sections 5.2, 5.3, 5.4, 5.5.1
  - Mostly at a high level
  - See Assignment 2
- Key topics
  - Transactions
  - Ranking over relations or results
  - Recursion in SQL (makes SQL Turing Complete)
  - Functions and Procedures
  - Triggers

Ranking

- Ranking is done in conjunction with an order by specification.
- Consider: *student_grades(ID, GPA)*
- Find the rank of each student.

select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
from *student_grades*
order by *s_rank*

- Equivalent to:

select *ID*, (1 + (**select count**(*)
from *student_grades B*
where *B.GPA* > *A.GPA*)) **as** *s_rank*
from *student_grades A*
order by *s_rank*;

Window Functions

- Similar to “Group By” – allows a calculation over “related” tuples
- Unlike aggregates, does not “group” them – rather rows remain separate from each other

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

https://www.postgresql.org/docs/9.3/tutorial-window.html

Recursion in SQL

- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
      select course_id, prereq_id
      from prereq
   union
      select rec_prereq.course_id, prereq.prereq_id,
      from rec_rereq, prereq
      where rec_prereq.prereq_id = prereq.course_id
   )
select ∗
from rec_prereq;
```

Makes SQL Turing Complete (i.e., you can write any program in SQL)

But: Just because you can, doesn't mean you should

SQL Functions

- Function to count number of instructors in a department

```
create function dept_count (dept_name varchar(20))
     returns integer
         begin
         declare d_count  integer;
              select count (* ) into d_count
              from instructor
              where instructor.dept_name = dept_name
         return d_count;
      end
```

- Can use in queries

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL Procedures

- Same function as a procedure

  create procedure *dept_count_proc* (**in** *dept_name* **varchar**(20),
  out *d_count* **integer)**
  begin
  select count(*) **into** *d_count*
  from *instructor*
  where *instructor.dept_name = dept_count_proc.dept_name*
  end

- But use differently:

  declare *d_count* **integer**;
  call *dept_count_proc*('Physics', *d_count*);

- HOWEVER: Syntax can be wildly different across different systems
  - Was put in place by DBMS systems before standardization
  - Hard to change once customers are already using it

SQL Functions/Procedures

- Stored procedures widely used in practice
  - Many benefits including reusability, better performance (reduce back and forth to the DB)
- Most database systems support multiple langauges
  - Purely SQL → Fully procedural (e.g., C, etc)
- PostgreSQL supports SQL, C, PL/pgSQL
  - Note PostgreSQL 10 (that we use) does not support PROCEDURE, only FUNCTION

```
CREATE FUNCTION c_overpaid (EMP, INTEGER) RETURNS BOOLEAN AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

Triggers

- A ***trigger*** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - 1. setting the account balance to zero
  - 2. creating a loan in the amount of the overdraft
  - 3. giving this loan a loan number identical to the account number of the overdrawn account

Trigger Example in SQL:1999

create trigger *overdraft-trigger* **after update on** *account*
referencing new row as *nrow*
for each row
when *nrow.balance* < 0
begin atomic
actions to be taken
end

Trigger Example in SQL:1999

create trigger *overdraft-trigger* **after update on** *account*
referencing new row as *nrow*
for each row
when *nrow.balance* < 0
begin atomic
insert into *borrower*
(select *customer-name, account-number*
from *depositor*
where *nrow.account-number = depositor.account-number*);
insert into *loan* **values**
(*nrow.account-number, nrow.branch-name, nrow.balance*);
update *account* **set** *balance* = 0
where *account.account-number = nrow.account-number*
end

PostgreSQL Trigger Syntax

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

https://www.postgresql.org/docs/12/sql-createtrigger.html

NOTE: We use PostgreSQL 10, which does not support PROCEUDRE

Triggers…

- External World Actions
  - How does the DB *order* something if the inventory is low ?
- Syntax
  - Every system has its own syntax
- Careful with triggers
  - Cascading triggers, Infinite Sequences…
- More Info/Examples:
  - http://www.adp-gmbh.ch/ora/sql/create_trigger.html
  - Google: "create trigger" oracle download-uk

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - commit work: makes all updates of the transaction permanent in the database
    - rollback work: undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by rollback work.
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within

```
begin atomic
  …
end
```


CMSC424: Database Design

Aside

Anatomy of a Web Application

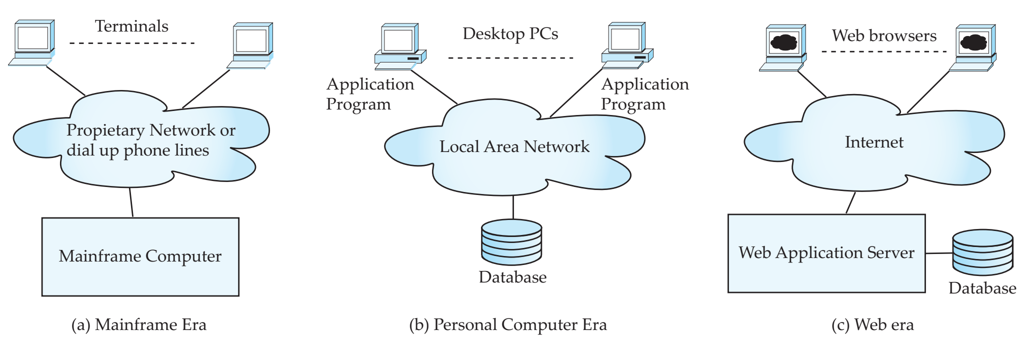
Instructor: Amol Deshpande
amol@umd.edu

Anatomy of a Web Application

- Book Chapters (6th Edition)
  - Sections 9.1, 9.2, 9.3.5, 9.3.6, 9.4.3
  - Much not covered in depth in the book, but lot of good tutorials on the web
- Key Topics
  - How Web Applications Work
  - Some of the underlying technologies
  - REST

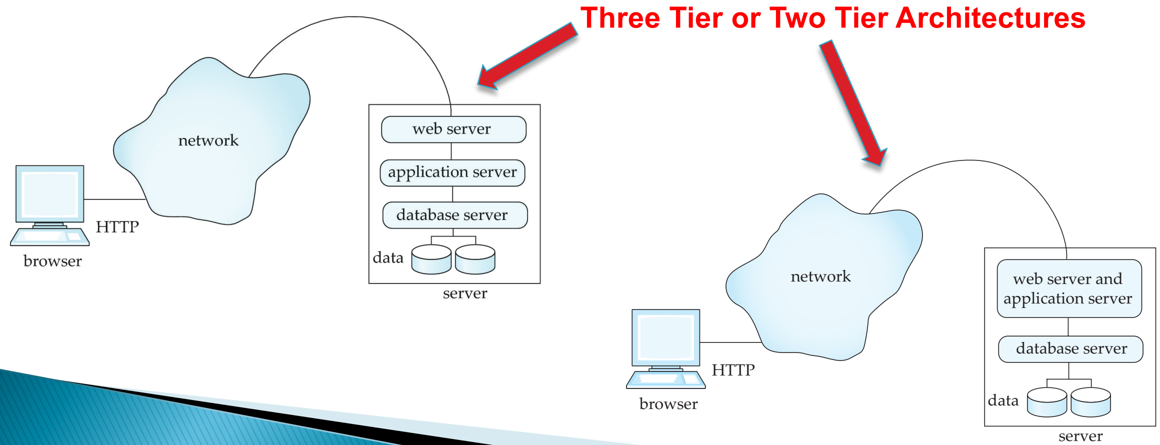
Application Architecture Evolution

- Three distinct eras of application architecture
  - Mainframe (1960's and 70's)
  - Personal computer era (1980's)
  - Web era (mid 1990's onwards)
  - Web and Smartphone era (2010 onwards)



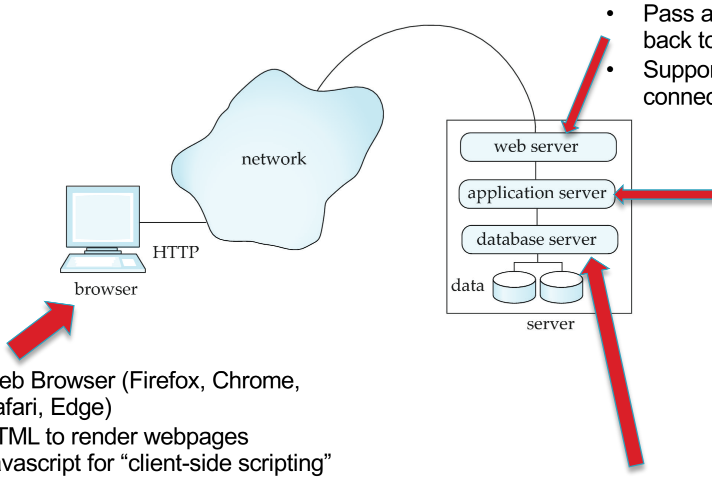
Web or Mobile Applications

- Web browsers and mobile applications have become de facto standard user interface
  - Wide cross-platform accessibility
  - No need to download something



What runs where?

- Flask, Django, Tomcat, Node.js, and others
- Accept requests from the client and pass to the application server
- Pass application server response back to the client
- Support HTTP and HTTPS connections



- Encapsulates business logic
- Needs to support different user flows
- Needs to handle all of the rendering and visualization
- Ruby-on-rails, Django, Flask, Angular, React, PHP, and many others

1. Web Browser (Firefox, Chrome, Safari, Edge)
2. HTML to render webpages
3. Javascript for "client-side scripting" (running code in your browser without contacting the server)
4. Flash (not supported much – too much security risk)
5. Java "applets" – less common today

- PostgreSQL, Oracle, SQL Server, Amazon RDS (Relational Databases)
- MongoDB (Document/JSON databases)
- SQLite --- not typically for production environments
- Pretty much any database can be used…

Some Key Technologies

- HTML
  - Controls display of content on webpages
- HTTP/HTTPS, Sessions, Cookies
  - How "clients" connect to "servers"
- Server-side vs client-side scripting
  - Some processing happens on the server, but increasingly on the client (though Javascript)
- REST, SOAP, GraphQL
  - Protocols for "clients" to requests things from the "servers" (or for two web services to talk to each other)
- Web APIs (typically REST or GraphQL)
  - Some services available on the Web

REST

- **Representation State Transfer**: use standard HTTP requests to execute a request (against a web or application server) and return data
  - Technically REST is a software architectural style -- APIs that conform to it are called RESTful APIs
- How REST uses the five standard HTTP request types:
  - POST: Invoke the method that corresponds to the URL, typically with data that is sent with the request
  - GET: Retrieve the data (no data sent with the request)
  - PUT: Reverse of GET
  - PATCH: Update some data
  - DELETE: Delete the data

As someone on Stackoverflow put it: "**REST** is the way **HTTP** should be *used*."

- **Alternative: GraphQL** -- uses HTTP POST calls, where the body of the call tells the web server what needs to be done

REST – GET Calls

https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

REST Example: Twitter

API reference contents ^

GET /2/tweets (lookup by list of IDs) GET /2/tweets/:id (lookup by single ID)

GET /2/tweets/:id (lookup by single ID)

Returns a variety of information about a single Tweet specified by the requested ID.

Run in Postman ›

Endpoint URL

`https://api.twitter.com/2/tweets/:id`

Authentication and rate limits

Authentication methods supported by this endpoint	OAuth 2.0 Bearer token OAuth 1.0a User context
Rate limit	300 requests per 15-minute window (app auth) 900 requests per 15-minute window (user auth)

CMSC424: Database Design

Module: Relational Model + SQL

SQL and Programming Languages

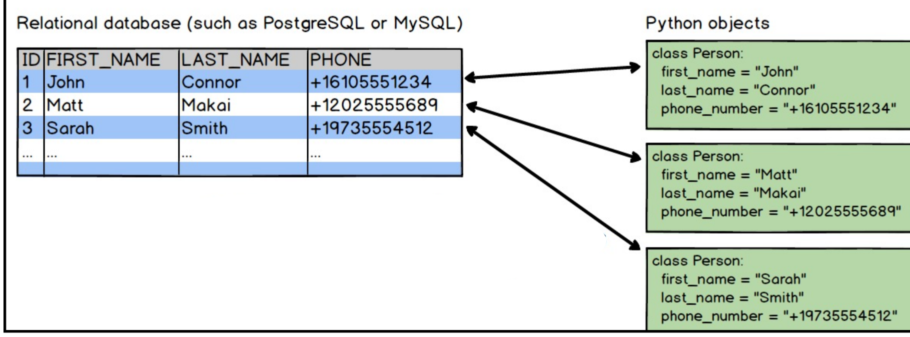
Instructor: Amol Deshpande
amol@umd.edu

SQL and Programming Languages

- Book Chapters (6th Edition)
  - Sections 5.1, 9.4.2
- Key Topics
  - Why use a programming language
  - Embedded SQL vs OBDC/JDBC
  - Object-relational impedance mismatch
  - Object-relational Mapping Frameworks

SQL and Programming Languages

- Programmers/developers more comfortable using a programming language like Java, Python, etc.
  - SQL not natural for many things
  - Performance issues in going back and forth to the database
- Need to deal with impedance mismatch between:
  - how data is represented in memory (typically as objects)
  - how it is stored (typically in a "normalized" relational schema)



Option 1: JDBC/ODBC

- Use a standard protocol like JDBC (Java Database Connectivity) to talk to the database from the programming language

```
>>> import jaydebeapi
>>> conn = jaydebeapi.connect("org.hsqldb.jdbcDriver",
...                           "jdbc:hsqldb:mem:.",
...                           ["SA", ""],
...                           "/path/to/hsqldb.jar",)
>>> curs = conn.cursor()
>>> curs.execute('create table CUSTOMER'
...              '("CUST_ID" INTEGER not null,'
...              ' "NAME" VARCHAR(50) not null,'
...              ' primary key ("CUST_ID"))'
...             )
>>> curs.execute("insert into CUSTOMER values (1, 'John')")
>>> curs.execute("select * from CUSTOMER")
>>> curs.fetchall()
[(1, u'John')]
>>> curs.close()
>>> conn.close()
```

- Doesn't solve impedance mismatch problem
  - Have to convert from the "result tuples" into "objects" and vice versa (when updating)

```java
import java.sql.*;

public class JDBCExample
{

    public static void main(String[] argv) {
        System.out.println("-------- PostgreSQL " + "JDBC Connection Testing ------------");
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("Where is your PostgreSQL JDBC Driver? " + "Include in your library path!");
            e.printStackTrace();
            return;
        }

        System.out.println("PostgreSQL JDBC Driver Registered!");
        Connection connection = null;
        try {
            connection = DriverManager.getConnection("jdbc:postgresql://localhost:5432/olympics","vagrant", "vagrant");
        } catch (SQLException e) {
            System.out.println("Connection Failed! Check output console");
            e.printStackTrace();
            return;
        }

        if (connection != null) {
            System.out.println("You made it, take control your database now!");
        } else {
            System.out.println("Failed to make connection!");
            return;
        }

        Statement stmt = null;
        String query = "select * from players;";
        try {
            stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String name = rs.getString("name");
                System.out.println(name + "\t");
            }
            stmt.close();
        } catch (SQLException e ) {
            System.out.println(e);
        }
    }
```


Option 1: JDBC/ODBC

- WARNING: always use prepared statements when taking an input from the user and adding it to a query (Related to the issue of SQL Injection attacks)
  - NEVER create a query by concatenating strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')"
  - What if name is "D'Souza"?

```java
PreparedStatement pStmt = conn.prepareStatement("insert into instructor
values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

- Python psycopg2 also has its own way of doing prepared statements

```python
cur = conn.cursor()
for i, j in parameters:
        cur.execute( "select * from tables where i = %s and j = %s", (i, j))
        for record in cur: do_something_with(record)
```

Option 1: JDBC/ODBC

- JDBC Features
  - Getting schemas, columns, primary keys
    - DatabaseMetaData dbmd = conn.getMetaData()
  - Transaction control
    - conn.commit(), conn.rollback()
  - Calling functions and procedures

- ODBC: Open Database Connectivity Standard
  - Similar in many ways
  - Older – designed by Microsoft and typically used in C, C++, like languages
    - Java supports as well but slower

Option 2: Embedded SQL

- SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
  - The language in which embedded is call "host" language

C++

```
int main() {
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int OrderID;          /* Employee ID (from user)        */
      int CustID;             /* Retrieved customer ID        */
      char SalesPerson[10]    /* Retrieved salesperson name     */
      char Status[6]          /* Retrieved order status       */
   EXEC SQL END DECLARE SECTION;

   /* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

   /* Prompt the user for order number */
   printf ("Enter order number: ");
   scanf_s("%d", &OrderID);

   /* Execute the SQL query */
   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;

   /* Display the results */
   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson: %s\n", SalesPerson);
   printf ("Status: %s\n", Status);
   exit();
```

Option 2: Embedded SQL

- SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
  - The language in which embedded is call “host” language
- Needs compiler support for the host language
  - The compiler needs to know what to do with the EXEC SQL commands
  - Hard to port
- Doesn’t solve impedance mismatch problem
  - Have to convert from the “result tuples” into “objects” and vice versa (when updating)
- Not a preferred approach today

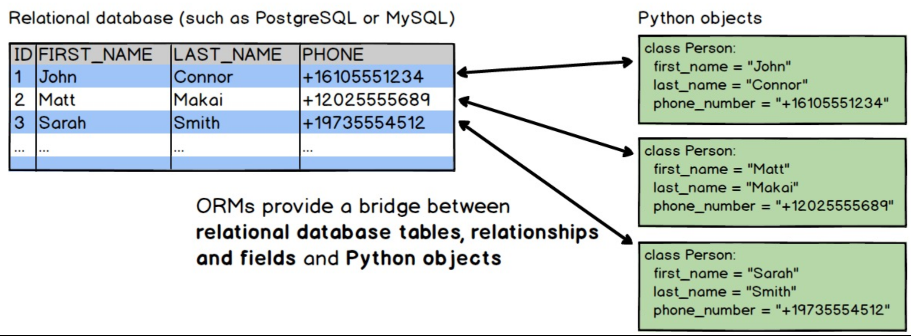
Option 3: Custom Libraries

- Often there are vendor-specific libraries that sometimes use internal protocols (and not JDBC/ODBC)
- e.g., python psycopg2 for PostgreSQL – although similar to JDBC calls, it uses the same proprietary protocol that ‘psql’ uses

```
conn = psycopg2.connect("dbname=olympics user=vagrant")
cur = conn.cursor()

totalscore = 0
for i in range(0, 14):
    # If a query is specified by -q option, only do that one
    if args.query is None or args.query == i:
        try:
            if interactive:
                os.system('clear')
            print("========== Executing Query {}".format(i))
            print(queries[i])
            cur.execute(queries[i])

            if i not in [5, 6, 8, 9]:
                ans = cur.fetchall()

                print("--------- Your Query Answer ---------")
                for t in ans:
                    print(t)
                print("")
```

Option 4: Object-relational Mappers

- Aimed at solving the impedance mismatch
  - Primarily for Web Application Development
- The ORM takes care of the mapping between objects and the database
  - Although largely designed around RDBMS, some ORMs support other databases as well
- The programmer works with objects, and never directly sees the SQL
  - Has pros (easier to use) and cons (performance and correctness issues)
- ORMs typically work with “Entities/Objects” and “Relationships”
  - Aligns well with the ER model that we will discuss next
  - We will cover Django constructs in more detail



Option 5: Other Mappers

- Many other “wrappers” on top of relational databases that offer different functionalities
  - In some cases, operations written in a higher-level language mapped to SQL
    - like what we saw for ORMs
    - Microsoft LINQ is also similar
    - Allows intermixing of code mapped to SQL and other code
  - In some cases, used to provide alternate data models to users
    - e.g., a thin layer that provides a graph data model, but stores data in a relational database
    - Most “RDF” databases built on top of SQL databases
- In today’s big data ecosystem, we see many many permutations how different tools (including databases) are combined together

CMSC424: Database Design

Module: Relation Model + SQL

Relational Algebra

Instructor: Amol Deshpande
amol@umd.edu

Relational Operations

- Book Chapters (6th Edition)
  - 2.5, 2.6, 6.1.1-6.1.3 (expanded treatment of 2.5, 2.6)
- Key Topics
  - Relational query languages and what purpose they serve
  - Basic unary and binary relational operations
  - Mapping between relational operations and SQL

Relational Query Languages

- Example schema: *R(A, B)*
- Practical languages
  - SQL
    - select A from R where B = 5;
  - Datalog (sort of practical)
    - q(A) :- R(A, 5)
- Formal languages
  - Relational algebra

    $\pi_A (\sigma_{B=5} (R))$
  - Tuple relational calculus

    $\{ t : \{A\} \mid \exists s : \{A, B\} (R(A, B) \wedge s.B = 5) \}$
  - Domain relational calculus
    - Similar to tuple relational calculus

Relational Operations

- Some of the languages are "procedural" and provide a set of operations
  - Each operation takes one or two relations as input, and produces a single relation as output
  - Examples: SQL, and Relational Algebra
- The "non-procedural" (also called "declarative") languages specify the output, but don't specify the operations
  - Relational calculus
  - Datalog (used as an intermediate layer in quite a few systems today)

Relational Algebra

- Procedural language
- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename
- The operators take one or more relations as inputs and give a new relation as a result.

Select Operation

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

SQL Equivalent:

select distinct *
from r
where A = B and D > 5

Unfortunate naming confusion

Project

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\pi_{A,D}$ (r)

A	D
α	7
α	7
β	3
β	10

A	D
α	7
β	3
β	10

SQL Equivalent:

select distinct A, D
from r

Set Union, Difference

Relation r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s:

A	B
α	1
α	2
β	1
β	3

r – s:

A	B
α	1
β	1

Must be compatible schemas

What about intersection ?

Can be derived

$r \cap s = r - (r - s);$

SQL Equivalent:

select * from r
union/except/intersect
select * from s;

This is one case where
duplicates are removed.

Cartesian Product

Relation r, s

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

$r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

SQL Equivalent:

```
select distinct *
from r, s
```

Does not remove duplicates.

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.

  Example:

  $$\rho_X(E)$$

  returns the expression *E* under the name *X*

  If a relational-algebra expression *E* has arity *n*, then

  $$\rho_{X\,(A1,\,A2,\,\ldots,\,An)}(E)$$

  returns the result of expression *E* under the name *X*, and with the attributes renamed to *A1, A2, …., An*.

Relational Algebra

- Those are the basic operations

- What about SQL Joins ?
  - Compose multiple operators together

    $\sigma_{A=C}(r \times s)$

- Additional Operations
  - Set intersection
  - Natural join
  - Division
  - Assignment

Additional Operators

- Set intersection ($\cap$)
  - $r \cap s = r - (r - s)$;
  - SQL Equivalent: intersect

- Assignment ($\leftarrow$)
  - A convenient way to right complex RA expressions
  - Essentially for creating "temporary" relations
    - $temp1 \leftarrow \prod_{R-S}(r)$

  - SQL Equivalent: "create table as…"

Additional Operators: Joins

- Natural join ($\bowtie$)
  - A Cartesian product with equality condition on common attributes
  - Example:
    - if *r* has schema *R(A, B, C, D)*, and if *s* has schema *S(E, B, D)*
    - Common attributes: *B* and *D*
    - Then:

$$r \bowtie s = \prod_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

- SQL Equivalent:
  - select r.A, r.B, r.C, r.D, s.E from r, s where r.B = s.B and r.D = s.D, OR
  - select * from r natural join s

Additional Operators: Joins

- Equi-join
  - A join that only has equality conditions

- Theta-join ($\bowtie_\theta$)
  - $r \bowtie_\theta s = \sigma_\theta(r \times s)$

- Left outer join (⟕)
  - Say *r(A, B), s(B, C)*
  - We need to somehow find the tuples in *r* that have no match in *s*
  - Consider: $(r - \pi_{r.A,\ r.B}(r \bowtie s))$

  - We are done:

$$(r \bowtie s) \quad \cup \quad \rho_{temp\ (A,\ B,\ C)} \left((r - \pi_{r.A,\ r.B}(r \bowtie s)) \times \{(NULL)\} \right)$$

Additional Operators: Join Variations

- Tables: r(A, B), s(B, C)

name	Symbol	SQL Equivalent	RA expression
cross product	$\times$	select * from r, s;	$r \times s$
natural join	$\bowtie$	natural join	$\pi_{r.A,\ r.B,\ s.C}\sigma_{r.B\ =\ s.B}(r\ x\ s)$
theta join	$\bowtie_\theta$	from .. where θ;	$\sigma_\theta(r\ x\ s)$
equi-join	$\bowtie_\theta$ *(theta must be equality)*		
left outer join	r ⟕ s	left outer join (with "on")	(see previous slide)
full outer join	r ⟗ s	full outer join (with "on")	–
(left) semijoin	$r \ltimes s$	none	$\pi_{r.A,\ r.B}(r \bowtie s)$
(left) antijoin	$r \triangleright s$	none	$r - \pi_{r.A,\ r.B}(r \bowtie s)$

Example Query

- Find the largest salary in the university
  - Step 1: find instructor salaries that are less than some other instructor salary (i.e. not maximum)
    - using a copy of *instructor* under a new name *d*
    - $\Pi_{instructor.salary}(\sigma_{instructor.salary < d,salary}$
      $(instructor\ x\ \rho_d\ (instructor)))$
  - Step 2: Find the largest salary
    - $\Pi_{salary}\ (instructor)$ –
      $\Pi_{instructor.salary}(\sigma_{instructor.salary < d,salary}$
      $(instructor\ x\ \rho_d\ (instructor)))$

Example Queries

- Find the names of all instructors in the Physics department, along with the *course_id* of all courses they have taught
  - Query 1

$$\Pi_{instructor.ID,course\_id}(\sigma_{dept\_name=\text{"Physics"}}(\sigma_{instructor.ID=teaches.ID}(instructor \times teaches)))$$

  - Query 2

$$\Pi_{instructor.ID,course\_id}(\sigma_{instructor.ID=teaches.ID}(\sigma_{dept\_name=\text{"Physics"}}(instructor) \times teaches))$$

CMSC424: Database Design

Module: Relation Model + SQL

SQL "Multi-Set/Bag" Semantics

Instructor: Amol Deshpande
amol@umd.edu

Relational Operations

- Book Chapters (6th Edition)
  - Multiset Relational Algebra Paragraph (Section 6.1, page 238)
- Key Topics
  - SQL ”Bag”/”Multiset” Semantics
  - Operations on multisets

Duplicates

- By definition, *relations* are *sets*
  - So → No duplicates allowed
- Problem:
  - Not practical to remove duplicates after every operation
  - Why ?
- So…
  - SQL by default does not remove duplicates
- SQL follows *bag* semantics, not *set* semantics
  - Implicitly we keep count of number of copies of each tuple

Formal Semantics of SQL

- RA can only express `SELECT DISTINCT` queries

- To express SQL, must extend RA to a bag algebra
  → *Bags (aka: multisets) like sets, but can have duplicates*

e.g: {5, 3, 3}

e.g: homes =

cname	ccity
Johnson	Brighton
Smith	Perry
Johnson	Brighton
Smith	R.H.

- Next: will define RA^*: a bag version of RA

Formal Semantics of SQL: RA*

1. σ^*_p (r): *preserves copies in* r

e.g: $\sigma^*_{city = \text{Brighton}}$ (homes) =

cname	ccity
Johnson	Brighton
Johnson	Brighton

2. $\pi^*_{A1, \ldots, An}$ (r): *no duplicate elimination*

e.g: π^*_{cname} (homes) =

cname
Johnson
Smith
Johnson
Smith

Formal Semantics of SQL: RA*

3. r ∪* s : *additive union*

r

A	B
1	α
1	α
2	β

∪*

s

A	B
2	β
3	α
1	α

=

A	B
1	α
1	α
2	β
2	β
3	α
1	α

4. r -* s: *bag difference*

e.g.: r -* s =

A	B
1	α

s -* r =

A	B
3	α

Formal Semantics of SQL: RA*

5. r ×* s: *cartesian product*

A	B
1	α
1	α
2	β

×*

C
+
-

=

A	B	C
1	α	+
1	α	-
1	α	+
1	α	-
2	β	+
2	β	-

Formal Semantics of SQL

Query:

```
SELECT      a1, ....., an
FROM        r1, ....., rm
WHERE       p
```

Semantics: $\pi^*_{A1, \ldots, An} (\sigma^*_p (r_1 \times^* \ldots \times^* r_m))$ (1)

Query:

```
SELECT DISTINCT   a1, ....., an
FROM              r1, ....., rm
WHERE             p
```

Semantics: *What is the only operator to change in (1)?*

$\pi_{A1, \ldots, An} (\sigma^*_p (r_1 \times^* \ldots \times^* r_m))$ (2)

Set/Bag Operations Revisited

- Set Operations
  - UNION ≡ ∪
  - INTERSECT ≡ ∩
  - EXCEPT ≡ -

Bag Operations

- UNION ALL ≡ ∪*
- INTERSECT ALL ≡ ∩*
- EXCEPT ALL ≡ -*

Duplicate Counting:

Given m copies of t in r, n copies of t in s, how many copies of t in:

r UNION ALL s? A: $m + n$

r INTERSECT ALL s? *A: min (m, n)*

r EXCEPT ALL s? *A: max (0, m-n)*

CMSC424: Database Design

Module: Relational Model + SQL

SQL: Views, Authorization

Instructor: Amol Deshpande
amol@umd.edu

SQL Views

- Book Chapters (6th Edition)
  - 3.8, 4.6
- Key Topics
  - Defining Views and Use Cases
  - Difference between a view and a table
  - Updating a view
  - Authorization

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

create view *v* **as** <query expression>

where:

<query expression> is any legal expression

The view name is represented by *v*

- Can be used in any place a normal table can be used
- For users, there is no distinction in terms of using it

Example Queries

- A view consisting of courses and sections for Physics in Fall 2009

create view *physics_fall_2009* **as**
select *course.course_id*, *sec_id*, *building*, *room_number*
from *course*, *section*
where *course.course_id* = *section.course_id*
and *course.dept_name* = 'Physics'
and *section.semester* = 'Fall'
and *section.year* = '2009';

Find all physics fall 2009 courses in a building.

select *course_id*
from *physics_fall_2009*
where *building*= 'Watson';

Views

- Is it different from DBMS's side ?
  - Yes; a view may or may not be *materialized*
  - Pros/Cons ?
- Updates into views have to be treated differently
  - In most cases, disallowed.

Views vs Tables

Creating	Create view V as (select * from A, B where …)	Create table T as (select * from A, B where …)
Can be used	In any select query. Only some update queries.	It's a new table. You can do what you want.
Maintained as	1. Evaluate the query and store it on disk as if a table. 2. Don't store. Substitute in queries when referenced.	It's a new table. Stored on disk.
What if a tuple inserted in A ?	1. If stored on disk, the stored table is automatically updated to be accurate. 2. If we are just substituting, there is no need to do anything.	T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger.

Views vs Tables

- Views strictly supercede "create a table and define a trigger to keep it updated"
- Two main reasons for using them:
  - Security/authorization
    - Can provide a user with "read" access to only the view
  - Ease of writing queries
    - E.g. *PresidentStateReturns* , or a view listing who won which state
- Perhaps the only reason to create a table is to force the DBMS to choose the option of "materializing"
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don't change

Update of a View

- Create a view of all instructors while hiding the salary

  create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*;

- Add a new tuple to the view

  insert into *faculty* **values** ('30765', 'Green', 'Music');

- Options:
  - Reject because we don't "salary" information, or
  - Insert into "instructors": ('30765', 'Green', 'Music', NULL);
- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Many SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Authorization/Security

- GRANT and REVOKE keywords
  - **grant select on** *instructor* **to** U_1, U_2, U_3
  - **revoke select on** *branch* **from** U_1, U_2, U_3

- Can provide select, insert, update, delete privileges

- Can provide this for tables, schemas, "functions/procedures", etc.
  - Some databases support doing this at the level of individual "tuples"
    - MS SQL Server: https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver15
    - PostgreSQL: https://www.postgresql.org/docs/10/ddl-rowsecurity.html

- Can also create "Roles" and do security at the level of roles

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Integrity Constraints

Instructor: Amol Deshpande
amol@umd.edu

SQL Integrity Constraints

- Book Chapters (6^{th} Edition)
  - 4.4
- Key Topics
  - Why Constraints
  - Different Types of Integrity Constraints
  - Referential Integrity
  - How to specify in SQL

IC's

- Goal: Avoid Semantic Inconsistencies in the Data
- An IC is a predicate on the database
- Must always be true (checked whenever DB gets updated)
- There are the following 4 types of IC's:
  - Key constraints (1 table)
    - e.g., *2 accts can't share the same acct_no*
  - Attribute constraints (1 table)
    - e.g., *accts must have nonnegative balance*
  - Referential Integrity constraints (2 tables)
    - E.g. *bnames* associated w/ *loans* must be names of real branches
  - Global Constraints (*n* tables)
    - E.g., all *loans* must be carried by at least 1 *customer* with a savings acct

Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. Primary Key:

```
CREATE TABLE branch(
    bname  CHAR(15)  PRIMARY KEY,
    bcity     CHAR(20),
    assets    INT);
```

or

```
CREATE TABLE depositor(
    cname   CHAR(15),
    acct_no  CHAR(5),
    PRIMARY KEY(cname, acct_no));
```

2. Candidate Keys:

```
CREATE TABLE customer (
    ssn     CHAR(9)    PRIMARY KEY,
    cname  CHAR(15),
    address CHAR(30),
    city         CHAR(10),
    UNIQUE (cname, address, city));
```


Key Constraints

Effect of SQL Key declarations
PRIMARY (A1, A2, .., An) or
UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, **reject insertion**

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)

1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by "foreign key" (ref integrity)
3. DBMS may treat primary key differently
   (e.g.: create an index on PK)

Attribute Constraints

- Idea:
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)
- In SQL:

1. NOT NULL

e.g.:
```
CREATE TABLE branch(
    bname   CHAR(15)  NOT NULL,
    ....
    )
```

Note: declaring bname as primary key also prevents null values

2. CHECK

e.g.:
```
CREATE TABLE depositor(
    ....
    balance int NOT NULL,
    CHECK(  balance >= 0),
    ....
    )
```

affect insertions, update in affected columns

Attribute Constraints

Domains: can associate constraints with DOMAINS rather than attributes

e.g: instead of:
```
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK  (balance >= 0)
    )
```

One can write:
```
CREATE DOMAIN  bank-balance INT (
    CONSTRAINT not-overdrawn CHECK (value >= 0),
    CONSTRAINT not-null-value CHECK( value NOT NULL));


CREATE TABLE depositor (
    .....
    balance    bank-balance,
    )
```

Advantages?

Attribute Constraints

Advantage of associating constraints with domains:

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints

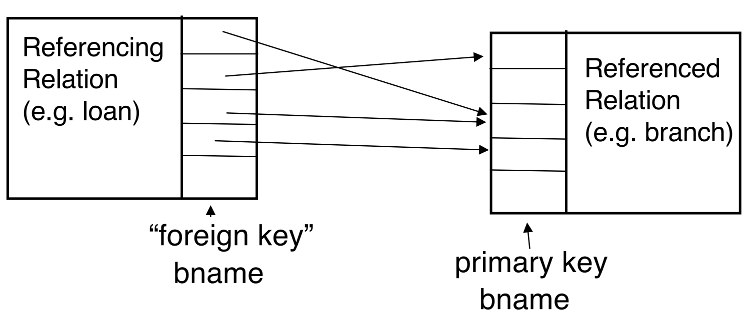
e.g.:
```
CREATE DOMAIN bank-balance INT (
    CONSTRAINT not-overdrawn
        CHECK (value >= 0),
    CONSTRAINT not-null-value
        CHECK( value NOT NULL));
```

allows one to:

1. add or remove:
```
ALTER DOMAIN bank-balance
    ADD CONSTRAINT capped
        CHECK( value <= 10000)
```
2. report better errors (know which constraint violated)

Referential Integrity Constraints

Idea: prevent "dangling tuples" (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)



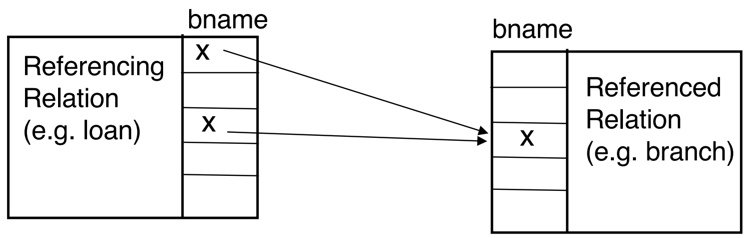
Ref Integrity:

ensure that:

foreign key value → primary key value

(note: don't need to ensure ←, i.e., not all branches have to have loans)

Referential Integrity Constraints



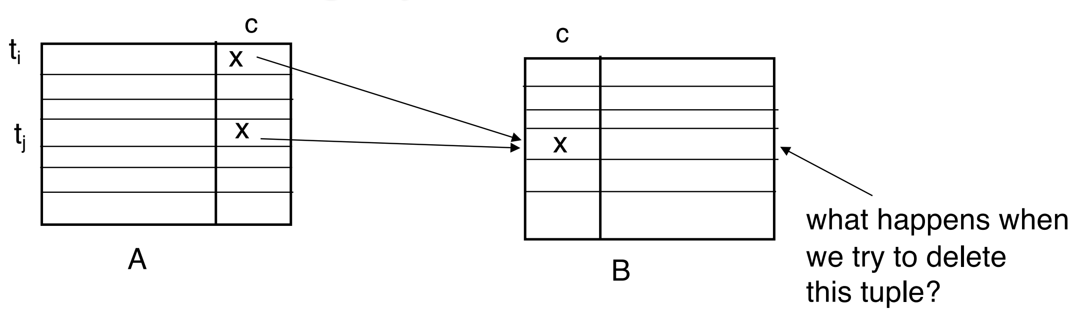
In SQL:

```
CREATE TABLE  branch(
      bname   CHAR(15)   PRIMARY KEY
      ....)

CREATE TABLE loan (
        .........
        FOREIGN KEY bname REFERENCES branch);
```

Affects:

1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation

Referential Integrity Constraints



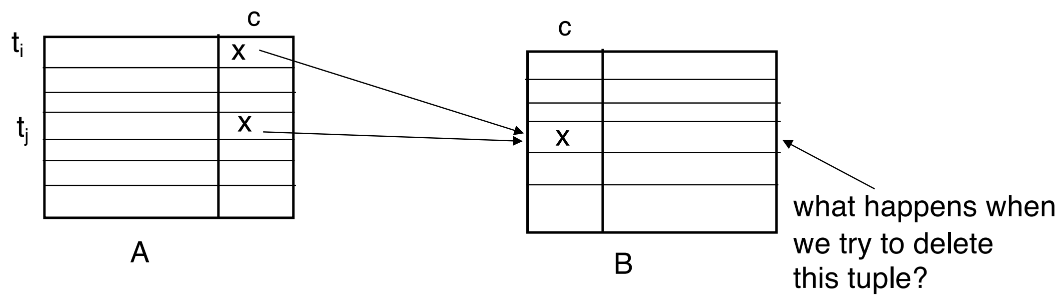
Ans: 3 possibilities

1) reject deletion/ update

2) set t_i [c], t_j[c] = NULL

3) propagate deletion/update
   DELETE: delete ti, tj
   UPDATE: set ti[c], tj[c] to updated values

Referential Integrity Constraints



CREATE TABLE A (.....
FOREIGN KEY c REFERENCES B action
..........)

Action: 1) left blank (deletion/update rejected)

2) ON DELETE SET NULL/ ON UPDATE SET NULL
sets ti[c] = NULL, tj[c] = NULL

3) ON DELETE CASCADE
deletes ti, tj
ON UPDATE CASCADE
sets ti[c], tj[c] to new key values

Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   - E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) multiple relations: CREATE ASSERTION

SQL examples:

1) single relation: All Bkln branches must have assets > 5M

```
CREATE TABLE branch (
        ..........
        bcity  CHAR(15),
        assets INT,
        CHECK (NOT(bcity = 'Bkln') OR assets > 5M))
```

Affects:

insertions into branch
updates of bcity or assets in branch

Global Constraints

SQL example:

2) Multiple relations: every loan has a borrower with a savings account

```
CHECK (NOT EXISTS (
        SELECT   *
        FROM loan AS L
        WHERE  NOT EXISTS(
               SELECT   *
               FROM borrower B, depositor D, account A
               WHERE B.cname = D.cname  AND
                     D.acct_no = A.acct_no  AND
                     L.lno  = B.lno)))
```

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

CREATE ASSERTION loan-constraint
CHECK(.....)

Checked with EVERY DB update!
very expensive.....

Summary: Integrity Constraints

Constraint Type	Where declared	Affects...	Expense
Key Constraints	CREATE TABLE (PRIMARY KEY, UNIQUE)	Insertions, Updates	Moderate
Attribute Constraints	CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)	Insertions, Updates	Cheap
Referential Integrity	Table Tag (FOREIGN KEY .... REFERENCES ....)	1.Insertions into referencing rel'n 2. Updates of referencing rel'n of relevant attrs 3. Deletions from referenced rel'n 4. Update of referenced rel'n	1,2: like key constraints. Another reason to index/sort on the primary keys 3,4: depends on a. update/delete policy chosen b. existence of indexes on foreign key
Global Constraints	Table Tag (CHECK) or outside table (CREATE ASSERTION)	1. For single rel'n constraint, with insertion, deletion of relevant attrs 2. For assesrtions w/ every db modification	1. cheap 2. very expensive

CMSC424: Database Design

NOT IN SYLLABUS

SQLMan: Wielding the Superpower of SQL

Instructor: Amol Deshpande
amol@umd.edu

Fun with SQL

- https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/
  - Long slide-deck linked off of this page
  - Complex SQL queries showing how to do things like: do Mandelbrot, solve subset sum problem etc.
- **The MADlib Analytics Library or MAD Skills, the SQL;** https://arxiv.org/abs/1208.4165
- https://www.red-gate.com/simple-talk/blogs/statistics-sql-simple-linear-regressions/

1. Everything is a Table

```
SELECT *
FROM (
  SELECT *
  FROM person
) t
```

```
SELECT *
FROM (
  VALUES(1),(2),(3)
) t(a)
```

Everything is a table. In PostgreSQL, even functions are tables:

```
SELECT *
FROM substring('abcde', 2, 3)
```

https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/

2. Recursion can be very powerful

```
WITH RECURSIVE t(v) AS (
  SELECT 1     -- Seed Row
  UNION ALL
  SELECT v + 1 -- Recursion
  FROM t
)
SELECT v
FROM t
LIMIT 5
```

Makes SQL Turing-Complete

It yields

```
v
---
1
2
3
4
5
```

https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/

3. Window Functions

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

https://www.postgresql.org/docs/9.3/tutorial-window.html

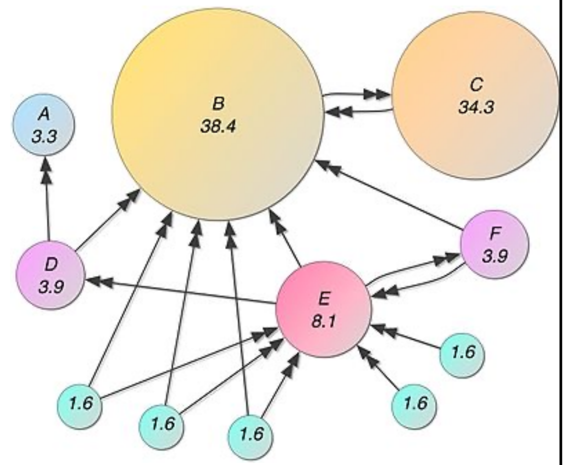
https://www.red-gate.com/simple-talk/blogs/statistics-sql-simple-linear-regressions/

4. Correlation Coefficient

```
SET ARITHABORT ON;

DECLARE @OurData TABLE
    (
    x NUMERIC(18,6) NOT NULL,
    y NUMERIC(18,6) NOT NULL
    );
  INSERT INTO @OurData
    (x, y)
  SELECT
   x,y
   FROM (VALUES
  (1,32),(1,23),(3,50),(11,37),(-2,39),(10,44),(27,32),(25,16),(20,23),
  (4,5),(30,41),(28,2),(31,52),(29,12),(50,40),(43,18),(10,65),(44,26),
  (35,15),(24,37),(52,66),(59,46),(64,95),(79,36),(24,66),(69,58),(88,56),
  (61,21),(100,60),(62,54),(10,14),(22,40),(52,97),(81,26),(37,58),(93,71)
  (64,82),(24,33),(112,49),(64,90),(53,90),(132,61),(104,35),(60,52),
  (29,50),(85,116),(95,104),(131,37),(139,38),(8,124)
  )f(x,y)
  SELECT
    ((Sy * Sxx) - (Sx * Sxy))
    / ((N * (Sxx)) - (Sx * Sx)) AS a,
    ((N * Sxy) - (Sx * Sy))
    / ((N * Sxx) - (Sx * Sx)) AS b,
    ((N * Sxy) - (Sx * Sy))
    / SQRT(
        (((N * Sxx) - (Sx * Sx))
         * ((N * Syy - (Sy * Sy))))) AS r
    FROM
      (
      SELECT SUM([@OurData].x) AS Sx, SUM([@OurData].y) AS Sy,
        SUM([@OurData].x * [@OurData].x) AS Sxx,
        SUM([@OurData].x * [@OurData].y) AS Sxy,
        SUM([@OurData].y * [@OurData].y) AS Syy,
        COUNT(*) AS N
        FROM @OurData
      ) sums;
```

5. Page Rank

- Recursive algorithm to assign weights to the nodes of a graph (Web Link Graph)
- Weight for a node depends on the weights of the nodes that point to it
- Typically done in iterations till “convergence”
- Not obvious that you can do it in SQL, but:
  - Each iteration is just a LEFT OUTERJOIN
  - Stopping condition is trickier
- Other ways to do it as well



https://devnambi.com/2013/pagerank.html


```
declare @DampingFactor decimal(3,2) = 0.85 --set the damping factor
        ,@MarginOfError decimal(10,5) = 0.001 --set the stable weight
        ,@TotalNodeCount int
        ,@IterationCount int = 1

-- we need to know the total number of nodes in the system
set @TotalNodeCount = (select count(*) from Nodes)

-- iterate!
WHILE EXISTS
(
        -- stop as soon as all nodes have converged
        SELECT *
        FROM dbo.Nodes
        WHERE HasConverged = 0
)
BEGIN

        UPDATE n SET
        NodeWeight = 1.0 - @DampingFactor + isnull(x.TransferWeight, 0.0)

        -- a node has converged when its existing weight is the same as the weight it would be given
        -- (plus or minus the stable weight margin of error)
        ,HasConverged = case when abs(n.NodeWeight - (1.0 - @DampingFactor + isnull(x.TransferWeight, 0.0))) < @MarginOfError then 1
else 0 end
        FROM Nodes n
        LEFT OUTER JOIN
        (
                -- Here's the weight calculation in place
                SELECT
                        e.TargetNodeId
                        ,TransferWeight = sum(n.NodeWeight / n.NodeCount) * @DampingFactor
                FROM Nodes n
                INNER JOIN Edges e
                  ON n.NodeId = e.SourceNodeId
                GROUP BY e.TargetNodeId
        ) as x
        ON x.TargetNodeId = n.NodeId

        -- for demonstration purposes, return the value of the nodes after each iteration
        SELECT
                @IterationCount as IterationCount
                ,*
        FROM Nodes

        set @IterationCount += 1
END
```

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Design Process

Instructor: Amol Deshpande
amol@umd.edu

Design Process; E/R Basics

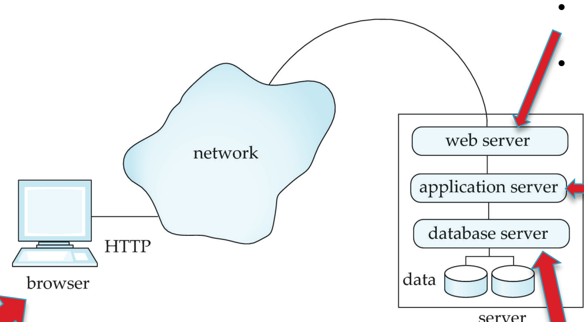
- Book Chapters (6th Edition)
  - Sections 7.1
- Key Topics
  - Steps in application and database design process
  - Two approaches to doing database design

Design Process

- To create an end-to-end database-backed application, we must:
  - Design the database schema for hosting the data
  - Design the application programs for accessing and updating the data
  - Design security schemes to control access to the data
- Typically an iterative process, involving many decision points and stakeholders
  - *computing environments, where to deploy, how to host, languages to use, data model, database systems, application frameworks, etc. etc.*
- Need clear understanding of user requirements
  - Followed by conceptual designs → functional requirements → physical designs → implementation
  - Need to keep revisiting earlier decisions as requirements evolve

What runs where?

- Flask, Django, Tomcat, Node.js, and others
- Accept requests from the client and pass to the application server
- Pass application server response back to the client
- Support HTTP and HTTPS connections



- Encapsulates business logic
- Needs to support different user flows
- Needs to handle all of the rendering and visualization
- Ruby-on-rails, Django, Flask, Angular, React, PHP, and many others

1. Web Browser (Firefox, Chrome, Safari, Edge)
2. HTML to render webpages
3. Javascript for "client-side scripting" (running code in your browser without contacting the server)
4. Flash (not supported much – too much security risk)
5. Java "applets" – less common today

- PostgreSQL, Oracle, SQL Server, Amazon RDS (Relational Databases)
- MongoDB (Document/JSON databases)
- SQLite --- not typically for production environments
- Pretty much any database can be used…

"Database" Design

- Goal: design the logical database schema
  - Try to avoid redundancy
    - Can lead to inconsistencies and require manual intervention
    - Makes it harder to program against the database
      - Need additional code/processes to update everywhere
      - Harder to make schema changes and migrate data
  - Ensure faithfulness to the requirements
    - Need to make sure it supports the use cases and the application requirements
    - Capturing all the data properly
      - Any data properties not captured cannot be stored in the database
    - Capture the constraints accurately
      - e.g., don't want to set \`s_id\` as the primary key for \`advisor(s_id, i_id)\` if we expect multiple advisors for a student
- Need a systematic way to do this for large schemas

"Database" Design

- Approach 1: Using a logical data model like the Entity-Relationship Model
  - Easier for humans to work with and visualize
  - Abstracts away the details, and allows focusing on the important issues
  - Richer than relational model, but allows easy conversion to relational for implementation
  - Harder to keep up to date – requires a lot of discipline
- Approach 2: Normalization Theory
  - Helps formalize the key design pitfalls and how to avoid them
- The two approaches are complementary and important to know both of them

Schema "Evolution" and Challenges

- Initial application schema nicely designed and normalized
- But as business requirements changes,
  - Schemas need to be modified
  - Data needs to be "migrated" from old schema to new schema
- Ideally the new schema is also normalized and properly designed
- However…
  - More changes to schema → More changes to applications running on top
  - Incremental schema changes often preferred by developers
  - Result: After a few iterations, the schema is not properly normalized any more
- No good solutions to date
  - Using "views" can help, but also requires discipline
  - Things we discuss here provide the foundations needed…

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Basics of E/R Models

Instructor: Amol Deshpande
amol@umd.edu

Basics of E/R Modeling

- Book Chapters (6th Edition)
  - Sections 7.2, 7.3.1, 7.3.3, 7.5.1-7.5.5
- Key Topics
  - Basics
  - Different types of attributes
  - Cardinalities of relationships
  - How to identify ”keys” for relationships

Entity-Relationship Model

- Two key concepts
  - *Entities*:
    - An object that *exists* and is *distinguishable* from other objects
      - Examples: Bob Smith, BofA, CMSC424
    - Have *attributes* (people have names and addresses)
    - Form *entity sets* with other entities of the same type that share the same properties
      - Set of all people, set of all classes
    - Entity sets may overlap
      - Customers and Employees

Entity-Relationship Model

- Two key concepts
  - *Relationships*:
    - Relate 2 or more entities
      - E.g. Bob Smith *has account at* College Park Branch
    - Form *relationship sets* with other relationships of the same type that share the same properties
      - Customers *have accounts at* Branches
    - Can have attributes:
      - *has account at* may have an attribute *start-date*
    - Can involve more than 2 entities
      - Employee *works at* Branch *at* Job

Entities and relationships

Two Entity Sets

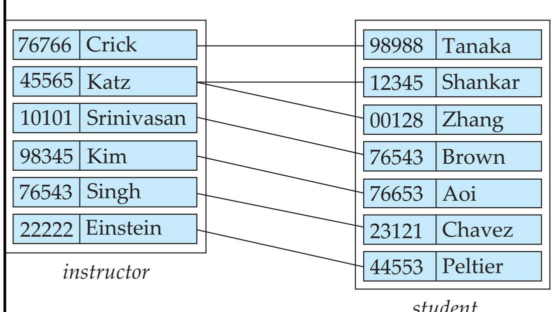
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

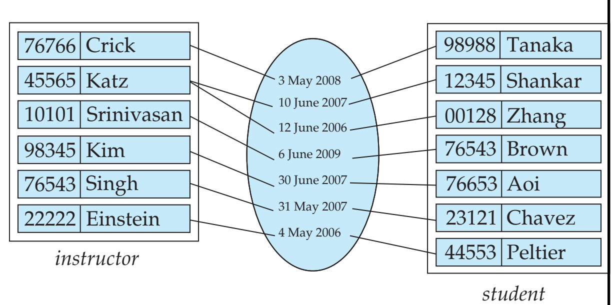
instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

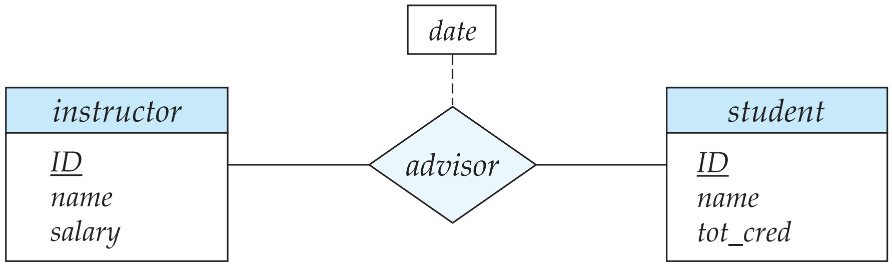
Advisor Relationship, with and without attributes





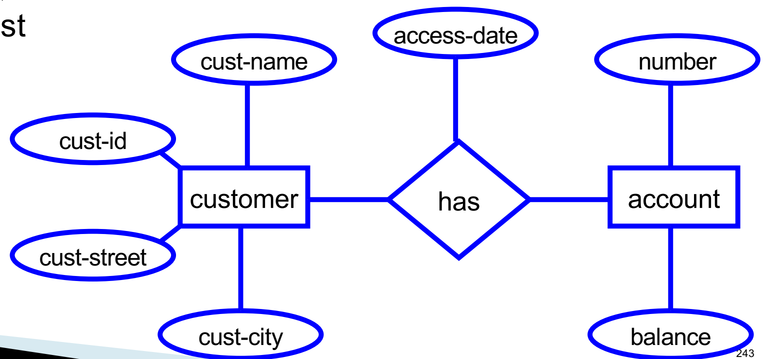
242

ER Diagram



Alternative representation,
used in the book in the past

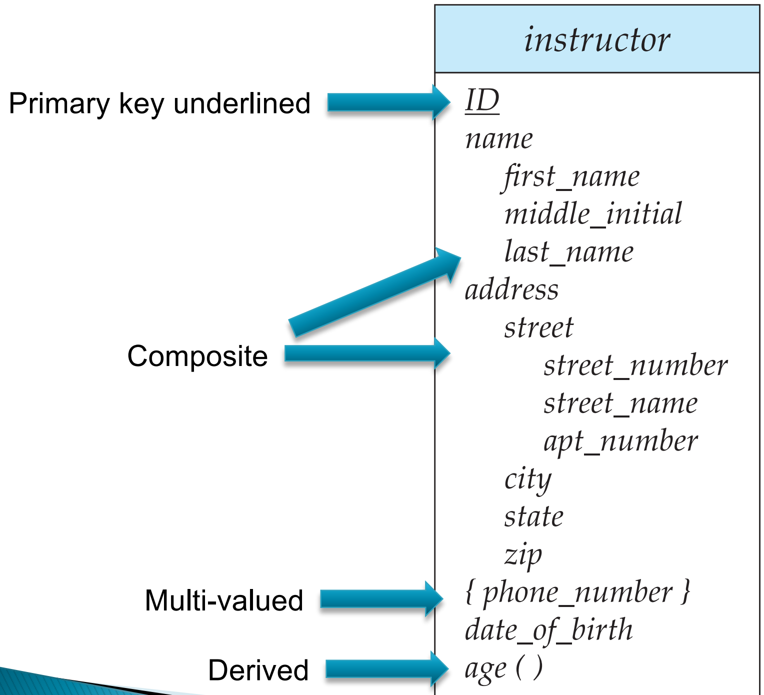
Both notations used commonly



Types of Attributes

- Simple vs Composite
  - Single value per attribute ?
- Single-valued vs Multi-valued
  - E.g. Phone numbers are multi-valued
- Derived
  - If date-of-birth is present, age can be derived
  - Can help in avoiding redundancy, enforcing constraints etc…

Types of Attributes



245

Relationship Cardinalities

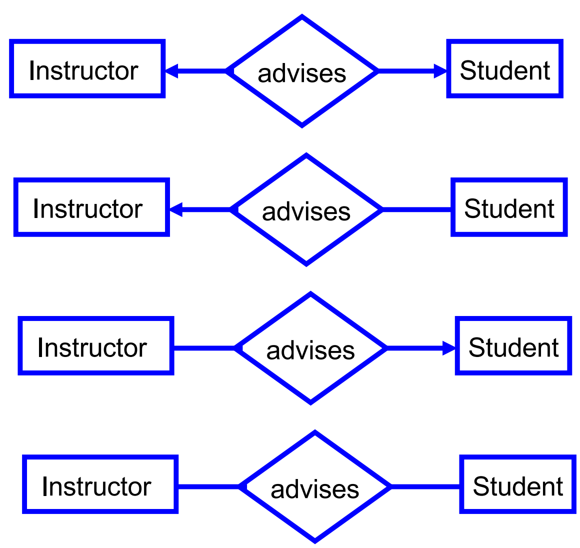
- We may know:
  - One customer can only open one account
  - OR
  - One customer can open multiple accounts
- Representing this is important
- Why ?
  - Better manipulation of data
    - If former, can store the account info in the customer table
  - Can enforce such a constraint
    - Application logic will have to do it; NOT GOOD
  - Remember: If not represented in conceptual model, the domain knowledge may be lost

Mapping Cardinalities

- Express the number of entities to which another entity can be associated via a relationship set
- Most useful in describing binary relationship sets

Mapping Cardinalities

- One-to-One
- One-to-Many
- Many-to-One
- Many-to-Many



Mapping Cardinalities

- Express the number of entities to which another entity can be associated via a relationship set

- Most useful in describing binary relationship sets

- N-ary relationships ?
  - More complicated
  - Details in the book

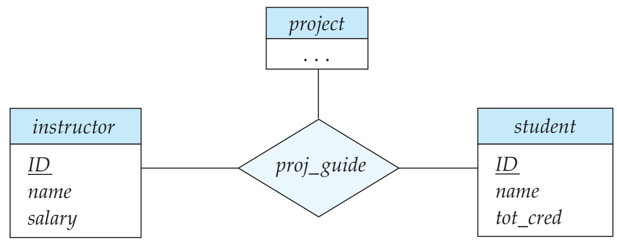


Figure 7.13 E-R diagram with a ternary relationship.

Relationship Set Keys

- What attributes are needed to represent a relationship completely and uniquely ?
  - Union of primary keys of the entities involved, and relationship attributes

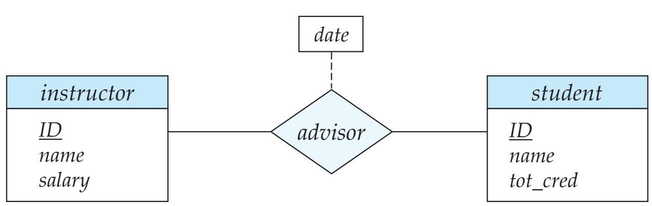


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

  - {instructor.ID, date, student.ID} describes a relationship completely

Relationship Set Keys

- Is *{student_id, date, instructor_id}* a candidate key ?
  - No. Attribute *date* can be removed from this set without losing key-ness
  - In fact, union of primary keys of associated entities is always a superkey

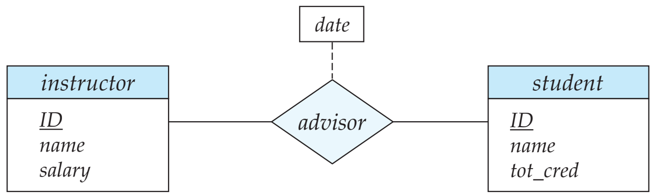


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

Relationship Set Keys

- Is {student_id, instructor_id} a candidate key ?
  - Depends

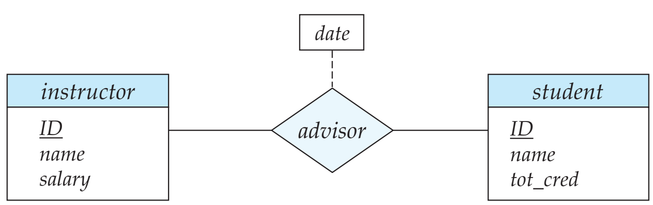


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

Relationship Set Keys

- Is {student_id, instructor_id} a candidate key ?
  - Depends

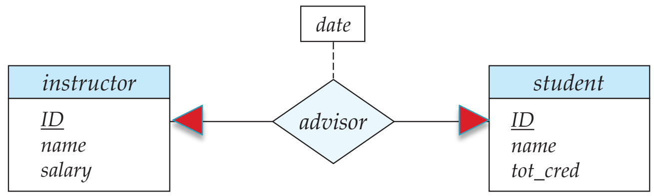


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

- If one-to-one relationship, either *{instructor_id}* or *{student_id}* sufficient
  - Since a given *instructor* can only have one *advisee*, an instructor entity can only participate in one relationship
  - Ditto *student*

Relationship Set Keys

- Is {student_id, instructor_id} a candidate key ?
  - Depends

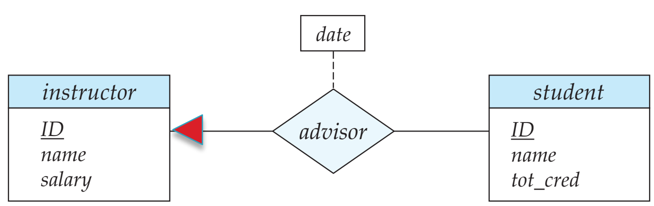


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

- If one-to-many relationship (as shown), *{student_id}* is a candidate key
  - A given instructor can have many advisees, but at most one advisor per student allowed

Relationship Set Keys

- General rule for binary relationships
  - one-to-one: primary key of either entity set
  - one-to-many: primary key of the entity set on the many side
  - many-to-many: union of primary keys of the associate entity sets

- n-ary relationships
  - More complicated rules

CMSC424: Database Design

Module: Design: E/R Models and Normalization

More E/R Constructs

Instructor: Amol Deshpande
amol@umd.edu

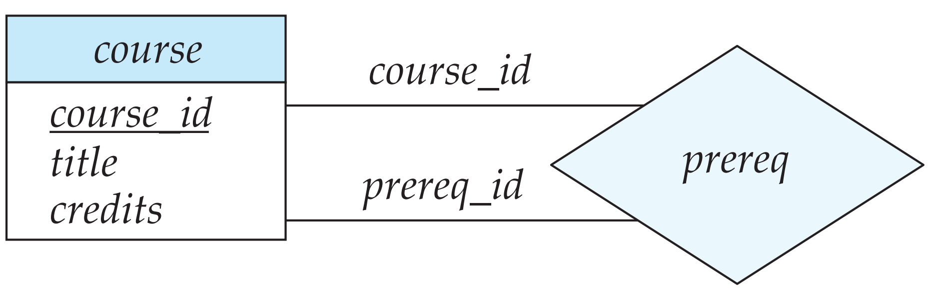
More E/R Constructs

- Book Chapters (6th Edition)
  - Sections 7.5.4, 7.5.6, 7.8
- Key Topics
  - Recursive Relationships and Roles
  - Weak Entity Sets
  - Specialization/Generalization
  - Aggregation

257

Recursive Relationships

- Sometimes a relationship associates an entity set to itself
- Need "roles" to distinguish

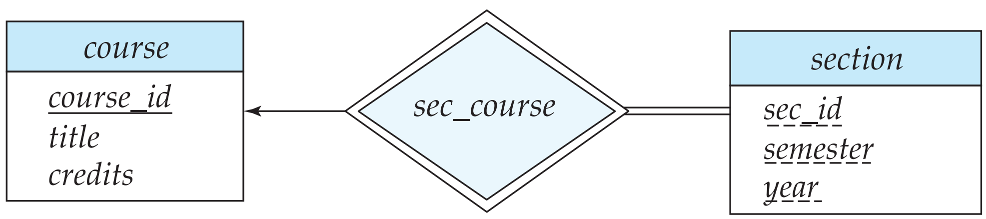


course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

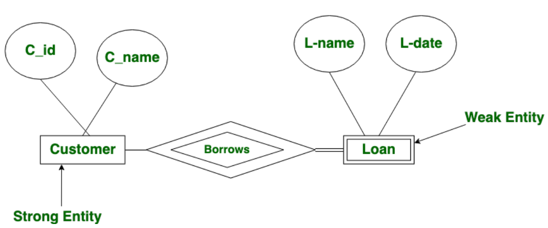
258

Weak Entity Sets

- An entity set without enough attributes to have a primary key
  - E.g. Section Entity
- Still need to be able to distinguish between weak entities
  - Called “discriminator attributes”: dashed underline

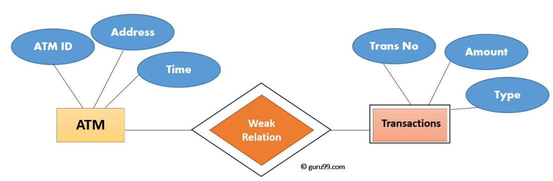


Examples of Weak Entity Sets



Loan may or may not have an extra unique identifier

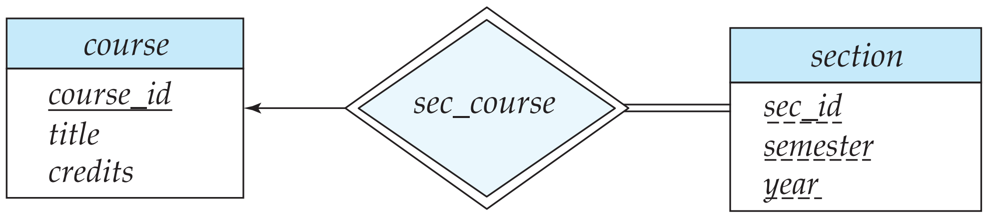
If transaction numbers are per ATM (i.e., first transaction from that ATM gets number 1, etc.), then Transactions is a weak entity



Apartments don’t have a unique identifer (across all buildings) without the building information

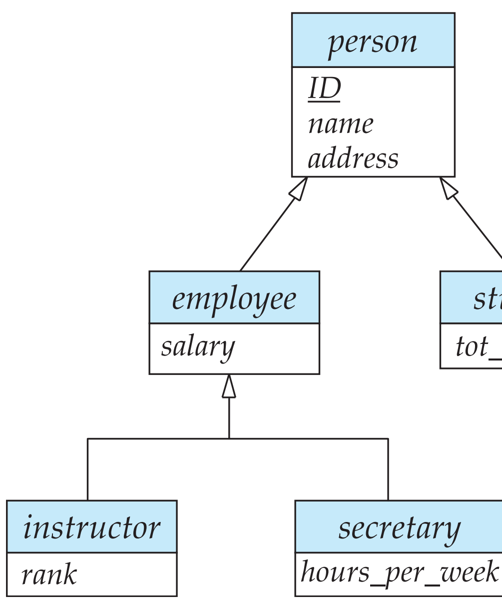
Participation Constraints

- Allow specifying full participation from an entity set in a relationship
  - i.e., every entity from that entity set ”must” participate in at least one relationship
  - Most common for Weak Entity Sets, but useful otherwise as well



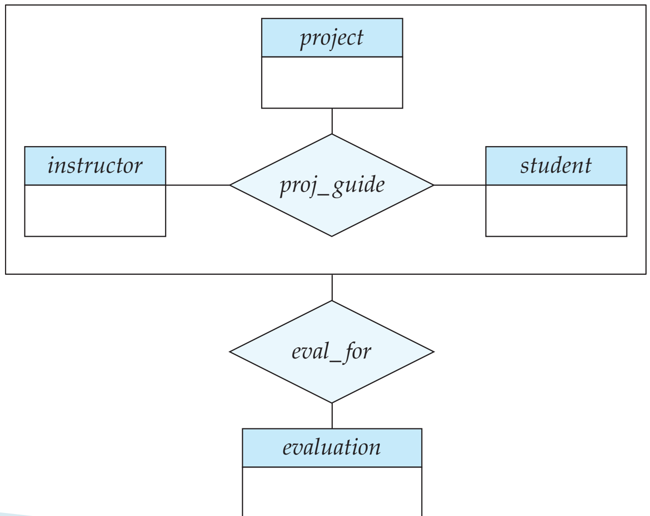
Specialization/Generalization

Similar to object-oriented programming: allows inheritance etc.



Aggregation

- No relationships allowed between relationships
- Suppose we want to record evaluations of a student by a guide on a project



CMSC424: Database Design

Module: Design: E/R Models and Normalization

Converting to Relational

Instructor: Amol Deshpande
amol@umd.edu

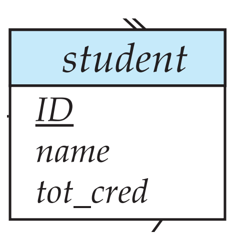
Converting E/R Models to Relations

- Book Chapters (6th Edition)
  - Sections 7.6, 7.8.6

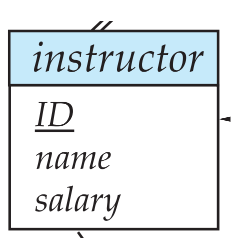
- Key Topics
  - Creating Relational Schema from an E/R Model
  - Mapping Entities and Relationships to Relations
  - Weak Entity Sets to Relations
  - Other E/R Constructs

E/R Diagrams → Relations

- Convert entity sets into a relational schema with the same set of attributes



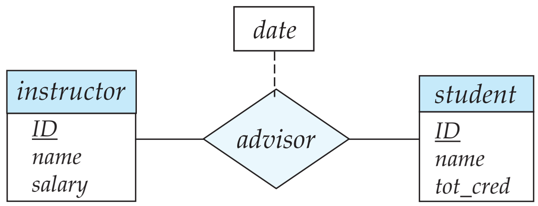
⇨ Student (<u>ID</u>, name, tot_cred)



⇨ Instructor(<u>ID</u>, name, salary)

E/R Diagrams → Relations

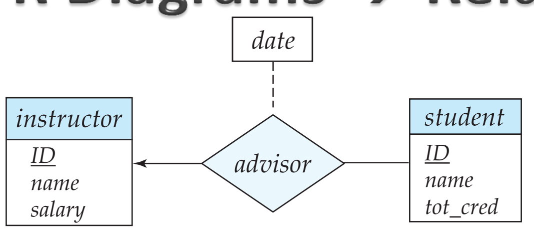
- Convert relationship sets *also* into a relational schema
- Remember: A relationship is completely described by primary keys of associated entities and its own attributes



We can do better for many-to-one or one-to-one

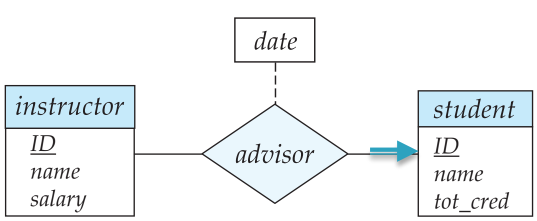
Advisor (student_ID, instructor_ID, date)

E/R Diagrams → Relations



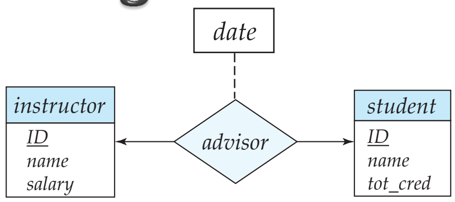
Foreign key into Instructor relation

Fold into Student:
Student(ID, name, tot_credits, advisor_ID, date)



Fold into Instructor:
Instructor(ID, name, salary, advisee_ID, date)

E/R Diagrams → Relations



Fold into Student:
Student(ID, name, tot_credits, advisor_ID)

OR

Fold into Instructor:
Instructor(ID, name, salary, advisee_ID)

Weak Entity Sets

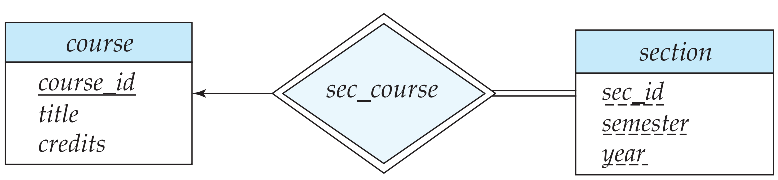


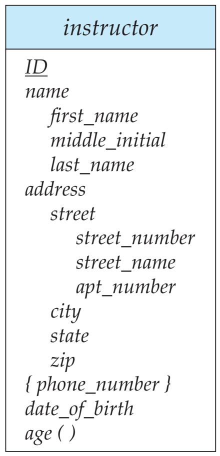
Figure 7.14 E-R diagram with a weak entity set.

Need to copy the primary key from the strong entity set:

Section(course_id, sec_id, semester, year)

Primary key for section = Primary key for course + Discriminator Attributes

Multi-valued Attributes



instructor (*ID*, *first_name*, *middle_name*, *last_name*, *street_number*, *street_name*, *apt_number*, *city*, *state*, *zip_code*, *date_of_birth*)

BUT

Phone_number needs to be split out into a separate table

Instructor_Phone(<u>Instructor_ID, phone_number)</u>

Specialization and Generalization

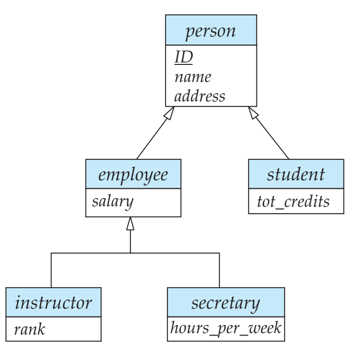


Figure 7.21 Specialization and generalization.

A few different ways to handle it

1. Common table for common information and separate tables for additional information

   person (<u>*ID*</u>, *name*, *street*, *city*)
   employee (<u>*ID*</u>, *salary*)
   student (<u>*ID*</u>, *tot_cred*)

2. Separate tables altogether – good idea if an employee can't be a student also – querying becomes harder (have to do unions for queries across all "persons")

   employee (<u>*ID*</u>, *name*, *street*, *city*, *salary*)
   student (<u>*ID*</u>, *name*, *street*, *city*, *tot_cred*)

CMSC424: Database Design

Module: Design: E/R Models and Normalization

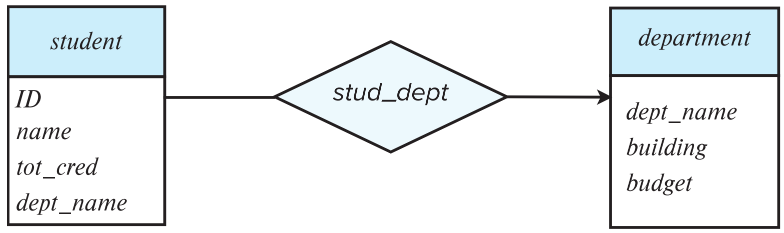
Design Issues;
Alternate Notations

Instructor: Amol Deshpande
amol@umd.edu

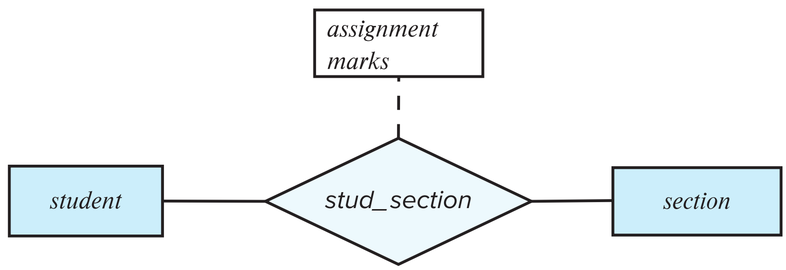
Design Issues; Alternate Notations

- Book Chapters (6th Edition)
  - Sections 7.7, 7.9 (briefly)
- Key Topics
  - Some Common Mistakes
  - Choosing between different ways to do the same thing
  - Alternate notations commonly used (including UML)
  - Recap

Some Common Mistakes

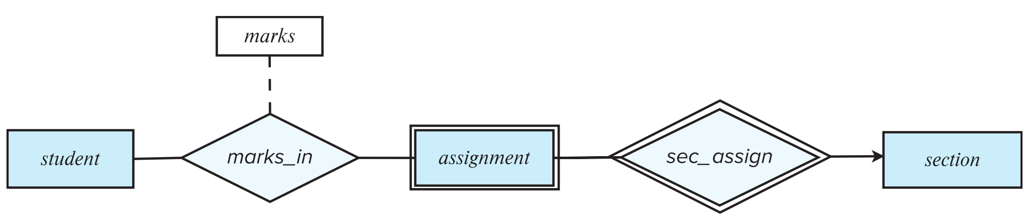


(a) Incorrect use of attribute

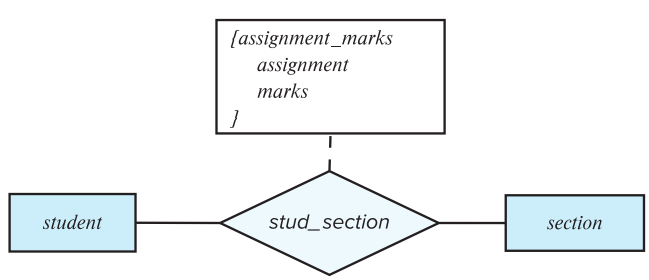


(b) Erroneous use of relationship attributes

Some Common Mistakes



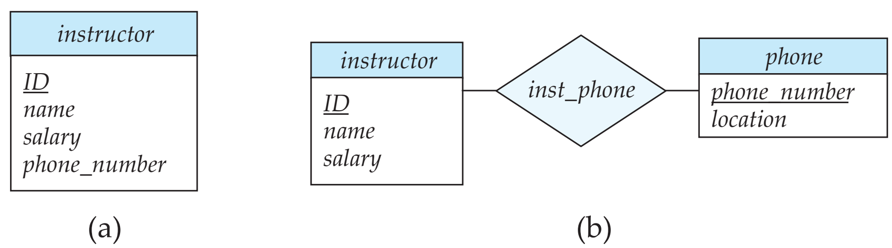
(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Design Issues

- Entity sets vs attributes
  - Depends on the semantics of the application
  - Consider *telephone*



Design Issues

- Entity sets vs Relationsihp sets
  - Consider *takes*

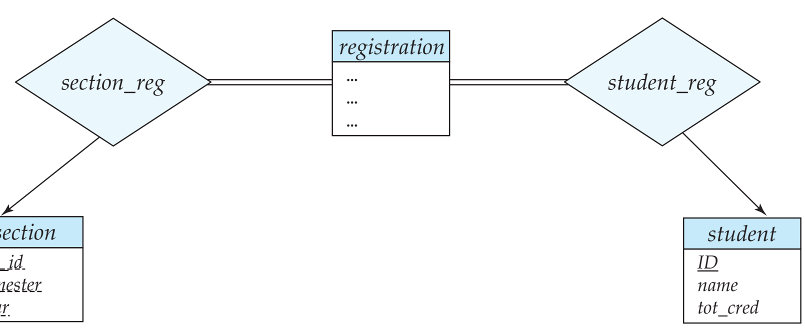


Figure 7.18 Replacement of *takes* by *registration* and two relationship sets

Design Issues

- N-ary vs binary relationships
  - Possible to avoid n-ary relationships, but there are some cases where it is advantageous to use them

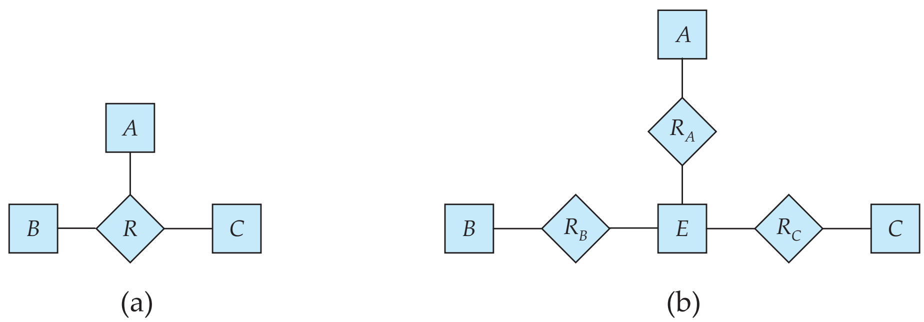


Figure 7.19 Ternary relationship versus three binary relationships.

Alternate Notations

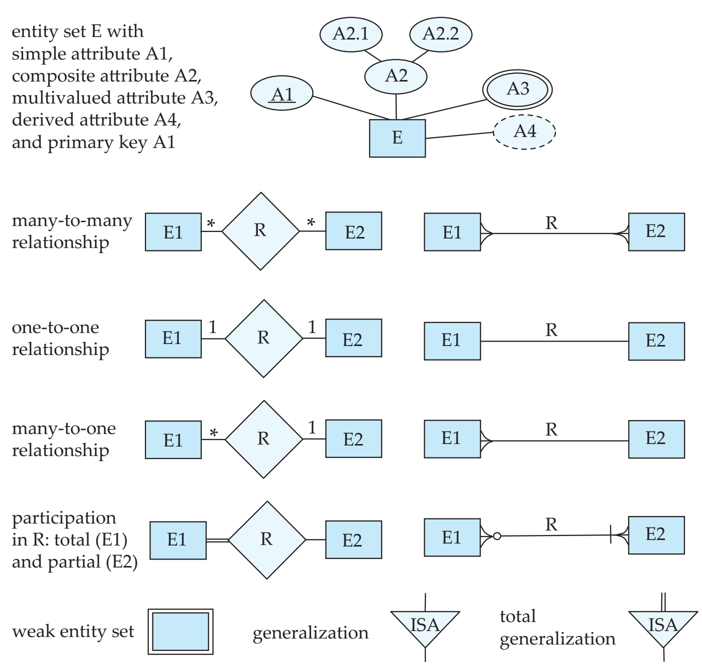
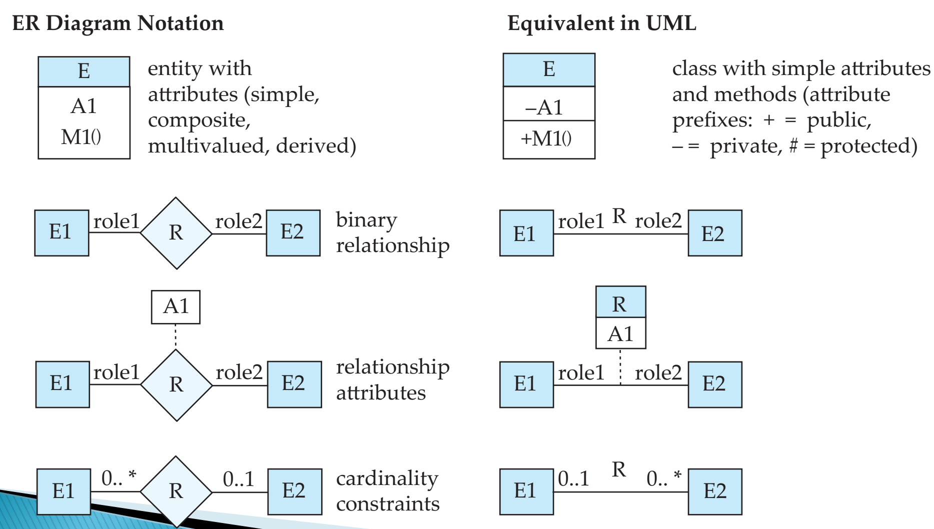


Figure 7.25 Alternative E-R notations.

Unified Modeling Language (UML)

- More comprehensive – covers use cases, flow of tasks between components, implementation diagrams, etc., in addition to data representation



Thoughts…

- Nothing about actual data
  - How is it stored ?
- No talk about the query languages
  - How do we access the data ?
- Semantic vs Syntactic Data Models
  - Remember: E/R Model is used for conceptual modeling
  - Many conceptual models have the same properties
- They are much more about representing the knowledge than about database storage/querying

Thoughts…

- Basic design principles
  - Faithful
    - Must make sense
  - Satisfies the application requirements
  - Models the requisite domain knowledge
    - If not modeled, lost afterwards
  - Avoid redundancy
    - Potential for inconsistencies
  - Go for simplicity
- Typically an iterative process that goes back and forth

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Normalization

Instructor: Amol Deshpande
amol@umd.edu

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Normalization: Basics

Instructor: Amol Deshpande
amol@umd.edu

Relational Database Design

- Where did we come up with the schema that we used ?
  - E.g. why not store the student course titles with their names ?
- If from an E-R diagram, then:
  - Did we make the right decisions with the E-R diagram ?
- Goals:
  - Formal definition of what it means to be a "good" schema.
  - How to achieve it.
- More abstract and formal than most other topics we will study

Normalization

- Book Chapters (6th Edition)
  - Section 8.1, 8.2
- Key Topics
  - What makes a ”good” schema
  - Problems with small schemas
  - Problems with large schemas
  - Atomic domains and First Normal Form

Simplified University Database Schema

Student(*student_id*, name, tot_cred)

Student_Dept(*student_id, dept_name*)

Department(*dept_name*, building, budget)

Course(*course_id*, title, dept_name, credits)

Takes(*course_id, student_id, semester, year*)

Changed to:

Student_Dept(*student_id, dept_name*, name, tot_cred, building, budget)

<Student, Student_Dept, and Department Merged Together>

Course(*course_id*, title, dept_name, credits)

Takes(*course_id, student_id*, semester, year)

Is this a good schema ???

Student_Dept(*student_id, dept_name*, name, tot_cred, building, budget)

student_id	dept_name	name	tot_cred	building	budget
s1	Comp. Sci.	John	30	Iribe Center	10 M
s2	Comp. Sci.	Alice	20	Iribe Center	10 M
s2	Math	Alice	20	Kirwan Hall	10 M
s3	Comp. Sci.	Mike	30	Iribe Center	10 M
s3	Math	Mike	30	Kirwan Hall	10 M

Issues:

1. Redundancy ➔ higher storage, inconsistencies (“anomalies”)

   update anomalies, insertion anamolies

2. Need nulls

   Unable to represent some information without using nulls

   How to store depts w/o students, or vice versa ?

   Can’t have NULLs in primary keys

Student_Dept(student_ids, *dept_name*, names, tot_creds, building, budget)

student_ids	dept_name	names	tot_creds	building	budget
{s1, s2, s3}	Comp. Sci.	{John, Alice, Mike}	{30, 20, 30}	Iribe Center	10 M
{s2, s3}	Math	{Alice, Mike}	{20, 30}	Kirwan Hall	10 M

Issues:

3. Avoid sets

   - Hard to represent
   - Hard to query
   - In this case, too many issues

Smaller schemas always good ????

Split Course(course_id, title, dept_name, credits) into:

Course1 (course_id, title, dept_name)   Course2(course_id, credits)???

course_id	title	dept_name
c1	"Intro to.."	Comp. Sci.
c2	"Discrete Structures"	Comp. Sci.
c3	"Database Design"	Comp. Sci.

course_id	credits
c1	3
c2	3
c3	3

This process is also called *"decomposition"*

Issues:

4. Requires more joins (w/o any obvious benefits)
5. Hard to check for some dependencies

What if the "credits" depend on the "dept_name" (e.g., all CS courses must be 3 credits)?

No easy way to ensure that constraint (w/o a join)

Smaller schemas always good ????

Decompose Takes(course_id, student_id, semester, year) into:

course_id	student_id	semester	year
c1	s1	Fall	2020
c1	s2	Spring	2020
c2	s1	Spring	2020

Takes1(course_id, semester, year)

course_id	semester	year
c1	Fall	2020
c1	Spring	2020
c2	Spring	2020

Takes2(course_id, student_id)

course_id	student_id
c1	s1
c1	s2
c2	s1

Issues:

6. "joining" them back (on course_id) results in more tuples than what we started with

(c1, s1, Spring 2020) & (c1, s2, Fall 2020)

This is a "lossy" decomposition

We lost some constraints/information

The previous example was a "lossless" decomposition.

Desiderata

- No sets
- Correct and faithful to the original design
  - Must avoid lossy decompositions
- As little redundancy as possible
  - To avoid potential anomalies
- No "inability to represent information"
  - Nulls shouldn't be required to store information
- Dependency preservation
  - Should be possible to check for constraints

Not always possible.
We sometimes relax these for:
simpler schemas, and *fewer joins during queries.*

Overall Approach

1. We will encode and list all our knowledge about the schema
   - e.g., Functional dependencies (FDs)

     SSN → name (means: SSN "implies" length)

     - If two tuples have the same "SSN", they must have the same "name"

       movietitle → length ???? Not true.

     - But, (movietitle, movieYear) → length --- True.
2. We will define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

Atomic Domains and 1st Normal Form

- A domain is called "atomic" if the elements can be considered indivisible
  - i.e., not composite or sets
  - Somewhat subjective and depends on how it is being used
- What about CMSC424?
  - A natural split into "CMSC" and "424".
  - Technically not atomic since programs/analysis often split it
  - Often treated as atomic, but better to keep as separate columns
- As long as all attributes are atomic → 1st Normal Form

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Functional Dependencies

Instructor: Amol Deshpande
amol@cs.umd.edu

Functional Dependencies

- Book Chapters (6th Edition)
  - Section 8.3.1
- Key Topics
  - Definition of a FD
  - Examples
  - Holding on an instance vs on all "legal" instances
  - FDs and Redundancies

Functional Dependencies

- On a relational schema: R(A, B, C, …)

  A → B (A "implies" B)

  means that if two tuples have the same value for A, they have the same value for B

- *A way to reason about duplication in a relational schema*

FDs: Example 1

student_id	dept_name	name	tot_cred	building	budget
s1	Comp. Sci.	John	30	Iribe Center	10 M
s2	Comp. Sci.	Alice	20	Iribe Center	10 M
s2	Math	Alice	20	Kirwan Hall	10 M
s3	Comp. Sci.	Mike	30	Iribe Center	10 M
s3	Math	Mike	30	Kirwan Hall	10 M

student_id → name

student_id → name, tot_cred

dept_name → building

dept_name → building, budget

FDs: Example 2

State Name	State Code	State Population	County Name	County Population	Senator Name	Senator Elected	Senator Born	Senator Affiliation
Alabama	AL	4779736	Autauga	54571	Jeff Sessions	1997	1946	'R'
Alabama	AL	4779736	Baldwin	182265	Jeff Sessions	1997	1946	'R'
Alabama	AL	4779736	Barbour	27457	Jeff Sessions	1997	1946	'R'
Alabama	AL	4779736	Autauga	54571	Richard Shelby	1987	1934	'R'
Alabama	AL	4779736	Baldwin	182265	Richard Shelby	1987	1934	'R'
Alabama	AL	4779736	Barbour	27457	Richard Shelby	1987	1934	'R'

State Name → State Code
State Code → State Name
Senator Name → Senator Born

FDs: Example 3

Course ID	Course Name	Dept Name	Credits	Section ID	Semester	Year	Building	Room No.	Capacity	Time Slot ID

Functional dependencies

course_id → title, dept_name, credits
building, room_number → capacity
course_id, section_id, semester, year → building, room_number, time_slot_id

Functional Dependencies

- Let *R* be a relation schema and

  $\alpha \subseteq R$ and $\beta \subseteq R$

- The *functional dependency*

  $\alpha \rightarrow \beta$

  holds on *R* iff for any *legal* relations *r*(R), whenever two tuples t_1 and t_2 of *r* have same values for α, they have same values for β.

  $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$

- Example:

A	B
1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

Functional Dependencies

<u>Difference between holding on an *instance* and holding on *all legal relation*</u>

student_id	dept_name	name	tot_cred	building	budget
s1	Comp. Sci.	John	30	Iribe Center	10 M
s2	Comp. Sci.	Alice	20	Iribe Center	10 M
s2	Math	Alice	20	Kirwan Hall	10 M
s3	Comp. Sci.	Mike	30	Iribe Center	10 M
s3	Math	Mike	30	Kirwan Hall	10 M

Name → Tot_Cred *holds on this instance*

Is this a true functional dependency ? ***No****.*

Two students with the same name can have the different credits.

Can't draw conclusions based on a *single instance*

Need to use domain knowledge to decide which FDs hold

FDs and Redundancy

- Consider a table: R(<u>A</u>, B, C):
  - With FDs: B → C, and A → BC
  - So "A" is a Key, but "B" is not
- So: there is a FD whose left hand side is not a key
  - **Leads to redundancy**

Since B is not unique, it may be duplicated
Every time B is duplicated, so is C

Not a problem with A → BC
A can never be duplicated

A	B	C
a1	b1	c1
a2	b1	c1
a3	b1	c1
a4	b2	c2
a5	b2	c2
a6	b3	c3
a7	b4	c1

Not a duplication → Two different tuples just happen to have the same value for C

FDs and Redundancy

- Better to split it up

A	B
a1	b1
a2	b1
a3	b1
a4	b2
a5	b2
a6	b3
a7	b4

B	C
b1	c1
b2	c2
b3	c3
b4	c1

Not a duplication → Two different tuples just happen to have the same value for C

Functional Dependencies

- Functional dependencies and *keys*
  - A *key* constraint is a specific form of a FD.
  - E.g. if *A* is a superkey for *R*, then:

    $A \rightarrow R$
  - Similarly for *candidate keys and primary keys.*
- Deriving FDs
  - A set of FDs may imply other FDs
  - *e.g. If A → B, and B → C, then clearly A → C*
  - *We will see a formal method for inferring this later*

Definitions

1. A relation instance *r* *satisfies* a set of functional dependencies, *F*, if the FDs hold on the relation

2. F holds on a relation schema *R* if no legal (allowable) relation instance of *R* violates it

3. A functional dependency, *A → B*, is called *trivial* if:
   - B is a subset of A
   - e.g. Movieyear, length → length

4. Given a set of functional dependencies, *F*, its *closure*, F^+, is all the FDs that are implied by FDs in *F*.

CMSC424: Database Design

Module: Design: E/R Models and Normalization

FDs: Armstrong Axioms, etc.

Instructor: Amol Deshpande
amol@umd.edu

Working with Functional Dependencies

- Book Chapters (6th Edition)
  - Section 8.4.1, 8.4.2, 8.4.3
- Key Topics
  - Closure of an attribute and attribute set
  - Armstrong Axioms
  - Extraneous Attributes
  - Canonical Cover
- Sufficient to get a high-level idea of these – don't need to understand the entire theory to follow rest of this

1. Closure

- Given a set of functional dependencies, *F,* its *closure, F^+* , is all FDs that are implied by FDs in *F.*
  - *e.g. If A → B, and B → C, then clearly A → C*
- We can find F+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
  - sound (generate only functional dependencies that actually hold)
  - complete (generate all functional dependencies that hold)

Additional rules

- If $\alpha \rightarrow \beta$ *and* $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$ **(union)**
- If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ **(decomposition)**
- If $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$ **(pseudotransitivity)**

- The above rules can be inferred from Armstrong's axioms.

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,
      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,
      and then transitivity

2. Closure of an attribute set

- Given a set of attributes *A* and a set of FDs *F*, *closure of A under F* is the set of all attributes implied by *A*
- In other words, the largest *B* such that: $A \rightarrow B$
- Redefining *super keys:*
  - *The closure of a super key is the entire relation schema*
- Redefining *candidate keys:*
  1. It is a super key
  2. No subset of it is a super key

Computing the closure for *A*

- Simple algorithm
- 1. Start with *B = A.*
- 2. Go over all functional dependencies, $\beta \rightarrow \gamma$, in F^+
- 3. If $\beta \subseteq$ *B, then*
  Add γ to *B*
- 4. Repeat till *B* changes

Example

- *R = (A, B, C, G, H, I)*
  F = { $A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H$}

- $(AG)^+$?
  - *1.* result = AG
  - 2.result = ABCG ($A \rightarrow C$ and $A \rightarrow B$)
  - 3.result = ABCGH ($CG \rightarrow H$ and $CG \subseteq AGBC$)
  - 4.result = ABCGHI ($CG \rightarrow I$ and $CG \subseteq AGBCH$

- Is (AG) a candidate key ?
  1. It is a super key.
  2. (A+) = ABCH, (G+) = G.

  YES.

Uses of attribute set closures

- Determining *superkeys and candidate keys*

- Determining if *A* → *B* is a valid FD
  - Check if A+ contains B

- Can be used to compute *F+*

3. Extraneous Attributes

- Consider *F,* and a functional dependency, $A \rightarrow B$.
- "Extraneous": Are there any attributes in *A or B* that can be safely removed ?

  Without changing the constraints implied by F

- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - C is extraneous in AB $\rightarrow$ CD since AB $\rightarrow$ C can be inferred even after deleting C
  - ie., given: A $\rightarrow$ C, and AB $\rightarrow$ D, we can use Armstrong Axioms to infer $AB \rightarrow CD$

4. Canonical Cover

- A *canonical cover* for *F* is a set of dependencies F_c such that
  - F logically implies all dependencies in F_c, and
  - F_c logically implies all dependencies in F, and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique
- In some (vague) sense, it is a *minimal* version of *F*
- Read up algorithms to compute F_c

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Decompositions

Instructor: Amol Deshpande
amol@cs.umd.edu

Lossless and Lossy Decompositions

- Book Chapters (6th Edition)
  - Section 8.4.4
- Key Topics
  - How to decompose a schema in a lossless manner
  - Dependency preserving decompositions

Decompositions

- Splitting a relational schema *R* into two relations *R1, R2,* typically for normalization
- e.g., R(A, B, C, D, E) can be decomposed into:
  - R1(A, B, C), R2(D, E)
  - R1(A, B, C, D), R2(D, E)
  - …
- When is this okay to do?
  - The two resulting relations must be equivalent to the original relation… always
- Otherwise, it is a "lossy" decomposition, and not allowed

Loss-less Decompositions

- Definition: A decomposition of *R* into *(R1, R2)* is called *lossless* if, for all legal instances of *r(R):*

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- In other words, projecting on *R1 and R2,* and *joining back,* results in the relation you started with
- Rule: A decomposition of *R* into *(R1, R2)* is *lossless,* iff:

$$R1 \cap R2 \rightarrow R1 \quad or \quad R1 \cap R2 \rightarrow R2$$

  in *F+.*

- Why? The join attributes then form a key for one of the relations
  - Each tuple from the other relation joins with exactly one from that relation

Loss-less Decompositions

- Example: R(A, B, C), FDs: A → B
  - Decomposition into R1(A, B) and R2(A, C) is lossless
    - (R1 ∩ R2 =) A → (R1 =) AB
  - Decomposition into R1(A, B) and R2(B, C) is not lossless

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c3
a3	b1	c4

→

A	B
a1	b1
a2	b1
a3	b1

⋈

B	C
b1	c1
b1	c2
b1	c3
b1	c4

=

A	B	C
a1	b1	c1
a1	b1	c2
a1	b1	c3
a1	b1	c4
a2	b1	c1
a2	b1	c2
a2	b1	c3
a2	b1	c4
a3	b1	c1
a3	b1	c2
a3	b1	c3
a3	b1	c4

Dependency-preserving Decompositions

Is it easy to check if the dependencies in *F* hold ?

Okay as long as the dependencies can be checked in the same table.

Consider *R = (A, B, C)*, and *F ={A → B, B → C}*

1. Decompose into R1 = (A, B), and R2 = (A, C)

Lossless ? Yes.

But, makes it hard to check for *B → C*

The data is in multiple tables.

2. On the other hand, *R1 = (A, B), and R2 = (B, C),*

is both lossless and dependency-preserving

Really ? What about *A → C ?*

If we can check *A → B*, and *B → C, A → C* is implied.

Dependency-preserving Decompositions

- Definition:
  - Consider decomposition of *R into R1, …, Rn.*
  - Let F_i be the set of dependencies F^+ that include only attributes in R_i.

- The decomposition is dependency preserving, if

$$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Boyce-Codd Normal Form

Instructor: Amol Deshpande
amol@umd.edu

Boyce Codd Normal Form

- Book Chapters (6th Edition)
  - Section 8.3.2
- Key Topics
  - Definition
  - How BCNF helps avoid redundancy
  - How to decompose a schema into BCNF

Approach

1. We will encode and list all our knowledge about the schema
   - Functional dependencies (FDs)
   - Also:
     - Multi-valued dependencies (briefly discuss later)
     - Join dependencies etc…
2. We will define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

BCNF: Boyce-Codd Normal Form

- A relation schema *R* is "in BCNF" if:
  - Every functional dependency *A* → *B* that holds on it is *EITHER*:
    1. Trivial *OR*
    2. *A* is a *superkey* of *R*
- *Why is BCNF good ?*
  - Guarantees that there can be no redundancy because of a functional dependency
  - Consider a relation *r(A, B, C, D)* with functional dependency *A* → *B* and two tuples: (a1, b1, c1, d1), and (a1, b1, c2, d2)
    - *b1* is repeated because of the functional dependency
    - BUT this relation is not in BCNF
      - *A* → *B* is neither trivial nor is *A* a superkey for the relation

BCNF and Redundancy

- *Why does redundancy arise ?*
  - Given a FD, A → B, if A is repeated (B – A) has to be repeated
  1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
  2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either
- Hence no redundancy because of FDs
  - Redundancy may exist because of other types of dependencies
    - Higher normal forms used for that (specifically, 4NF)
  - Data may naturally have duplicated/redundant data
    - We can't control that unless a FD or some other dependency is defined

BCNF

- What if the schema is not in BCNF ?
  - *Decompose (split) the schema into two pieces.*
- From the previous example: split the schema into:
  - *r1(A, B), r2(A, C, D)*
  - The first schema is in BCNF, the second one may not be (and may require further decomposition)
  - No repetition now: *r1* contains *(a1, b1)*, but *b1* will not be repeated
- Careful: you want the decomposition to be lossless
  - *No information should be lost*
    - The above decomposition is lossless

Achieving BCNF Schemas

For all dependencies *A* → *B* in *F+,* check if *A* is a superkey

By using attribute closure

If not, then

Choose a dependency in F+ that breaks the BCNF rules, say A → B

Create R1 = A B

Create R2 = A (R – B – A)

Note that: R1 ∩ R2 = A and A → AB (= R1), so this is lossless decomposition

Repeat for *R1, and R2*

By defining F1+ to be all dependencies in F that contain only attributes in R1

Similarly F2+

Example 1

R = (A, B, C)

F = {A → B, B → C}

Candidate keys = {A}

BCNF = No. B → C violates.

B → C

R1 = (B, C)
F1 = {B → C}
Candidate keys = {B}
BCNF = true

R2 = (A, B)
F2 = {A → B}
Candidate keys = {A}
BCNF = true

Example 2-1

R = (A, B, C, D, E)

F = {A → B, BC → D}

Candidate keys = {ACE}

BCNF = Violated by {A → B, BC → D} etc…

A → B

From A → B and BC → D by pseudo-transitivity

R1 = (A, B)
F1 = {A → B}
Candidate keys = {A}
BCNF = true

R2 = (A, C, D, E)
F2 = {AC → D}
Candidate keys = {ACE}
BCNF = false (AC → D)

AC → D

R3 = (A, C, D)
F3 = {AC → D}
Candidate keys = {AC}
BCNF = true

R4 = (A, C, E)
F4 = {} [[only trivial]]
Candidate keys = {ACE}
BCNF = true

Dependency preservation ???
We can check:
A → B (R1), AC → D (R3),
but we lost BC → D
So this is not a dependency
-preserving decomposition

Example 2-2

R = (A, B, C, D, E)

F = {A → B, BC → D}

Candidate keys = {ACE}

BCNF = Violated by {A → B, BC → D} etc…

BC → D

R1 = (B, C, D)
F1 = {BC → D}
Candidate keys = {BC}
BCNF = true

R2 = (B, C, A, E)
F2 = {A → B}
Candidate keys = {ACE}
BCNF = false (A → B)

A → B

R3 = (A, B)
F3 = {A → B}
Candidate keys = {A}
BCNF = true

R4 = (A, C, E)
F4 = {} [[only trivial]]
Candidate keys = {ACE}
BCNF = true

Dependency preservation ???
We can check:
BC → D (R1), A → B (R3),
Dependency-preserving decomposition

Example 3

R = (A, B, C, D, E, H)

F = {A → BC, E → HA}

Candidate keys = {DE}

BCNF = Violated by {A → BC} etc…

A → BC

R1 = (A, B, C)
F1 = {A → BC}
Candidate keys = {A}
BCNF = true

R2 = (A, D, E, H)
F2 = {E → HA}
Candidate keys = {DE}
BCNF = false (E → HA)

E → HA

R3 = (E, H, A)
F3 = {E → HA}
Candidate keys = {E}
BCNF = true

R4 = (ED)
F4 = {} [[only trivial]]
Candidate keys = {DE}
BCNF = true

Dependency preservation ???
We can check:
A → BC (R1), E → HA (R3),
Dependency-preserving decomposition

CMSC424: Database Design

Module: Design: E/R Models and Normalization

3NF, 4NF, and Other Issues

Instructor: Amol Deshpande
amol@umd.edu

3rd and 4th Normal Forms

- Book Chapters (6th Edition)
  - Section 8.3.4, 8.3.5, 8.5.2, 8.6 (at a high level)
- Key Topics
  - BCNF can't always preserve dependencies
  - How 3NF fixes that
  - BCNF causes redundancy because of "multi-valued dependencies"
  - How 4NF fixes that

Issue 1: BCNF may not preserve dependencies

- $R = (J, K, L)$
- $F = \{JK \rightarrow L, L \rightarrow K\}$
- Two candidate keys = JK and JL
- R is not in BCNF
- Any decomposition of R will fail to preserve

  $$JK \rightarrow L$$
- This implies that testing for $JK \rightarrow L$ requires a join

Issue 1: BCNF may not preserve dependencies

- Not always possible to find a dependency-preserving decomposition that is in BCNF.
- PTIME to determine if there exists a dependency-preserving decomposition in BCNF
  - in size of F
- NP-Hard to find one if it exists
- Better results exist if F satisfies certain properties

3NF (3rd Normal Form)

- Definition: *Prime* attributes
  An attribute that is contained in a candidate key for R

- Example 1:
  - R = (A, B, C, D, E, H}, F = {A → BC, E → HA},
  - Candidate keys = {ED}
  - Prime attributes: D, E

- Example 2:
  - R = (J, K, L), F = {JK → L, L → K},
  - Candidate keys = {JL, JK}
  - Prime attributes: J, K, L

- Observation/Intuition:
  1. A *key* has no redundancy (is not repeated in a relation)
  2. A *prime attribute* has limited redundancy

3NF (3rd Normal Form)

- Given a relation schema *R,* and a set of functional dependencies *F,* if every FD, *A → B*, is either:
  1. Trivial, or
  2. *A* is a *superkey* of *R, or*
  3. *All attributes in (B – A) are prime*

  Then, *R* is in *3NF (3rd Normal Form)*

- *Why is 3NF good ?*

3NF and Redundancy

- *Why does redundancy arise ?*
  - Given a FD, A → B, if A is repeated (B – A) has to be repeated
  1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
  2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either
  3. If not, rule 3 says (B – A) must contain only *prime attributes*
     This limits the redundancy somewhat.

- So 3NF relaxes BCNF somewhat by allowing for some (hopefully limited) redundancy
- Why ?
  - *There always exists a dependency-preserving lossless decomposition in 3NF.*

Decomposing into 3NF

- A *synthesis* algorithm
- Start with the canonical cover, and construct the 3NF schema directly
- Homework assignment.

Issue 2: BCNF and redundancy

MovieTitle	MovieYear	StarName	Address
Star wars	1977	Harrison Ford	Address 1, LA
Star wars	1977	Harrison Ford	Address 2, FL
Indiana Jones	198x	Harrison Ford	Address 1, LA
Indiana Jones	198x	Harrison Ford	Address 2, FL
Witness	19xx	Harrison Ford	Address 1, LA
Witness	19xx	Harrison Ford	Address 2, FL
…	…	…	…

Lot of redundancy

FDs ? No non-trivial FDs.

So the schema is trivially in BCNF (and 3NF)

What went wrong ?

Multi-valued Dependencies

- The redundancy is because of *multi-valued dependencies*
- *Denoted:*

  starname $\rightarrow\rightarrow$ *address*

  starname $\rightarrow\rightarrow$ *movietitle, movieyear*

- Should not happen if the schema is constructed from an E/R diagram
- Functional dependencies are a special case of multi-valued dependencies

4NF

- Similar to BCNF, except with MVDs instead of FDs.

- Given a relation schema *R,* and a set of multi-valued dependencies *F,* if every MVD, *A* →→ *B*, is either:
  1. Trivial, or
  2. *A* is a *superkey* of *R*

  Then, *R* is in *4NF (4th Normal Form)*

- 4NF → BCNF → 3NF → 2NF → 1NF:
  - If a schema is in 4NF, it is in BCNF.
  - If a schema is in BCNF, it is in 3NF.
- Other way round is untrue.

Comparing the normal forms

	3NF	**BCNF**	**4NF**
Eliminates redundancy because of FD's	Mostly	Yes	Yes
Eliminates redundancy because of MVD's	No	No	Yes
Preserves FDs	Yes.	Maybe	Maybe
Preserves MVDs	Maybe	Maybe	Maybe

4NF is typically desired and achieved.

A good E/R diagram won't generate non-4NF relations at all

Choice between 3NF and BCNF is up to the designer

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Recap and Other Issues

Instructor: Amol Deshpande
amol@umd.edu

Recap and Other Issues

- Book Chapters (6th Edition)
  - Section 8.8
- Key Topics
  - Database design process
  - Denormalization
  - Other normal forms
  - Recap

Database design process

- Three ways to come up with a schema

1. Using E/R diagram
   - If good, then little normalization is needed
   - Tends to generate 4NF designs
2. A universal relation *R* that contains all attributes.
   - Called universal relation approach
   - Note that MVDs will be needed in this case
3. An *ad hoc* schema that is then normalized
   - MVDs may be needed in this case

Recap

- What about 1^{st} and 2^{nd} normal forms ?
- 1NF:
  - Essentially says that no set-valued attributes allowed
  - Formally, a domain is called *atomic* if the elements of the domain are considered indivisible
  - A schema is in 1NF if the domains of all attributes are atomic
  - We assumed 1NF throughout the discussion
    - Non 1NF is just not a good idea
- 2NF:
  - Mainly historic interest
  - See Exercise 7.15 in the book

Recap

- We would like our relation schemas to:
  - Not allow potential redundancy because of FDs or MVDs
  - Be *dependency-preserving:*
    - Make it easy to check for dependencies
    - Since they are a form of integrity constraints
- Functional Dependencies/Multi-valued Dependencies
  - Domain knowledge about the data properties
- Normal forms
  - Defines the rules that schemas must follow
  - 4NF is preferred, but 3NF is sometimes used instead

Recap

- Denormalization
  - After doing the normalization, we may have too many tables
  - We may *denormalize* for performance reasons
    - Too many tables → too many joins during queries
  - A better option is to use *views* instead
    - So if a specific set of tables is joined often, create a view on the join
- More advanced normal forms
  - project-join normal form (PJNF or 5NF)
  - domain-key normal form
  - Rarely used in practice

CMSC424: Database Design

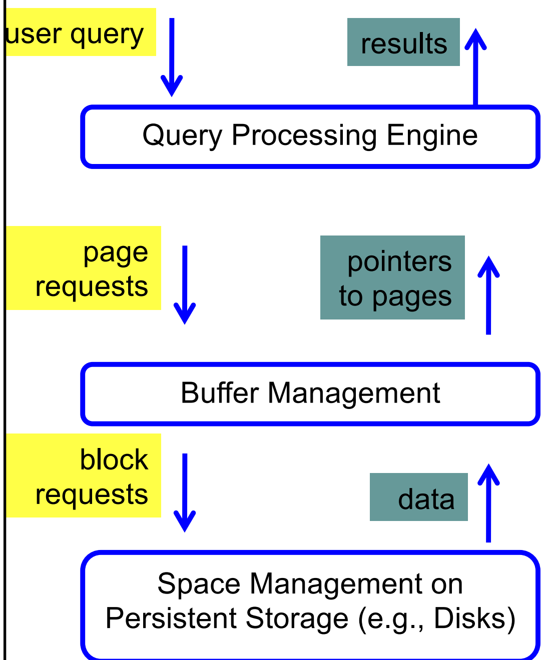
Module: Database Implementation

Instructor: Amol Deshpande
amol@cs.umd.edu

Database Implementation

- Shifting into discussing the internals of a DBMS
  - How data stored? How queries/transactions executed?
- Topics:
  - Storage: How is data stored? Important features of the storage devices (RAM, Disks, SSDs, etc)
  - File Organization: How are tuples mapped to blocks
  - Indexes: How to quickly find specific tuples of interest (e.g., all 'friends' of 'user0')
  - Query processing: How to execute different relational operations? How to combine them to execute an SQL query?
  - Query optimization: How to choose the best way to execute a query?

Query Processing/Storage



- Given an input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

CMSC424: Database Design

Module: Database Implementation

Storage: Storage Hierarchy

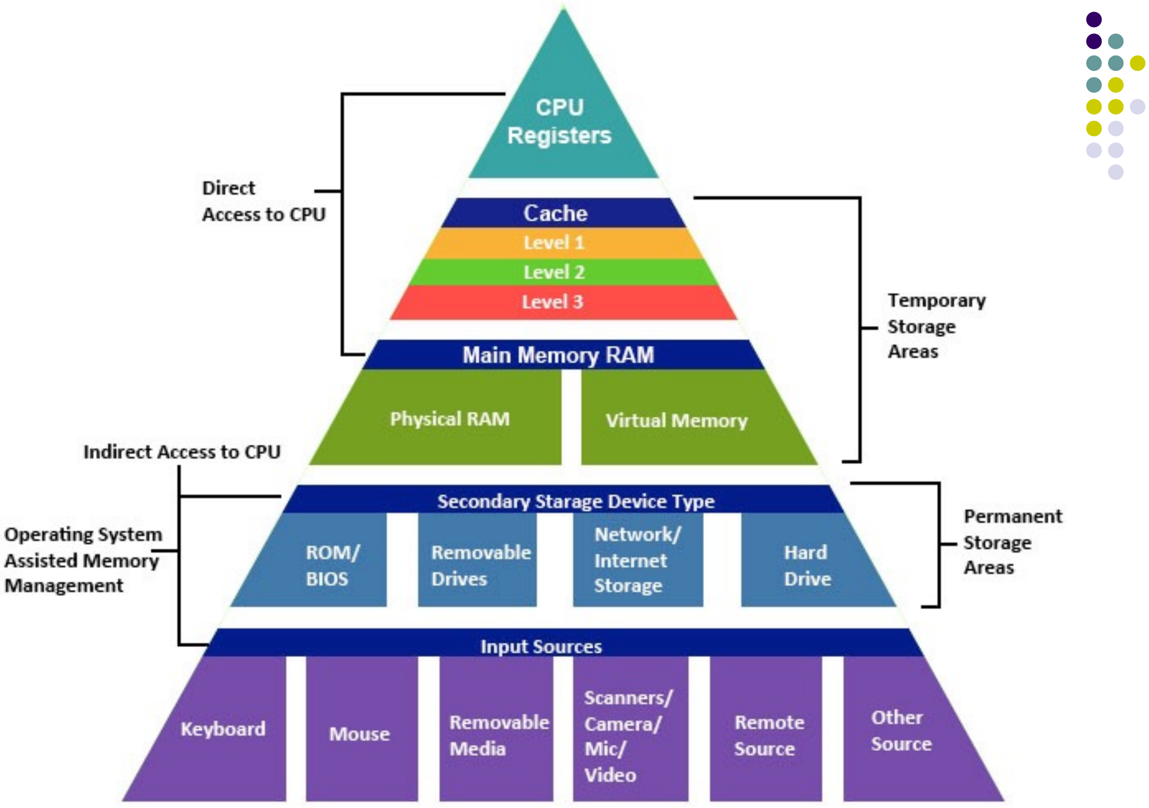
Instructor: Amol Deshpande
amol@cs.umd.edu

Storage Hierarchy

- Book Chapters
  - 10.1 (and some other online resources)
- Key topics:
  - Differences between storage media
  - Storage hierarchy
  - Caches

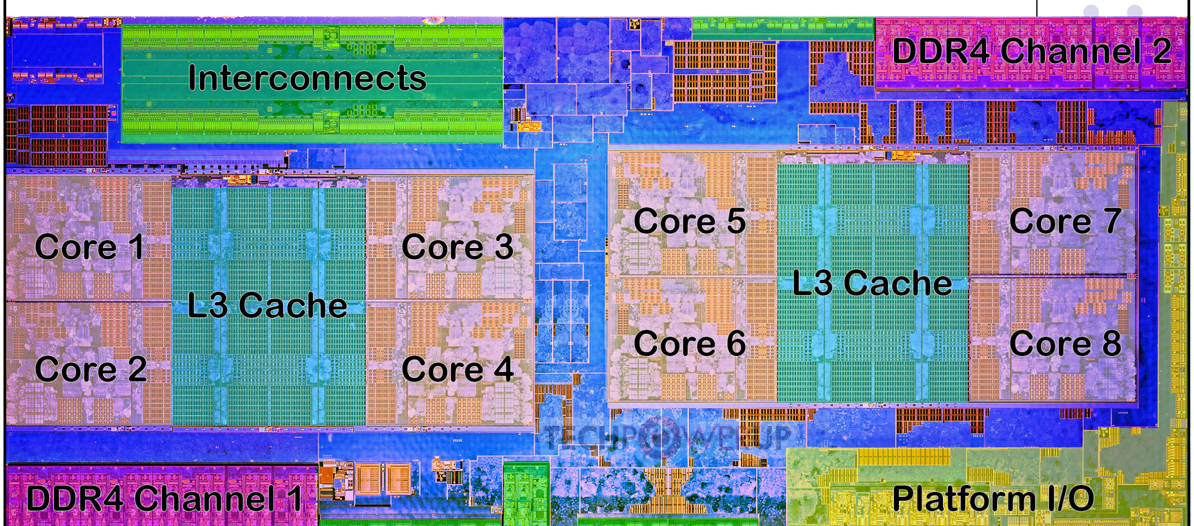
Storage Options

- At various points, data stored in different storage hardware
  - Memory, Disks, SSDs, Tapes, Cache
  - Tradeoffs between speed and cost of access
  - CPU needs the data in memory and cache to operate on it
- Volatile vs nonvolatile
  - Volatile: Loses contents when power switched off
- Sequential vs random access
  - Sequential: read the data contiguously
    - select * from employee
  - Random: read the data from anywhere at any time
    - select * from employee where name like '__a__b'



source: http://cse1.net/recaps/4-memory.html

AMD Ryzen CPU Architecture



Die shot overlaid with functional units

https://www.techpowerup.com/review/amd-ryzen-5-1600/3.html

Storage Hierarchy

Storage type	Access time	Relative access time
L1 cache	0.5 ns	Blink of an eye
L2 cache	7 ns	4 seconds
1MB from RAM	0.25 ms	5 days
1MB from SSD	1 ms	23 days
HDD seek	10 ms	231 days
1MB from HDD	20 ms	1.25 years

source: http://cse1.net/recaps/4-memory.html

Storage Options

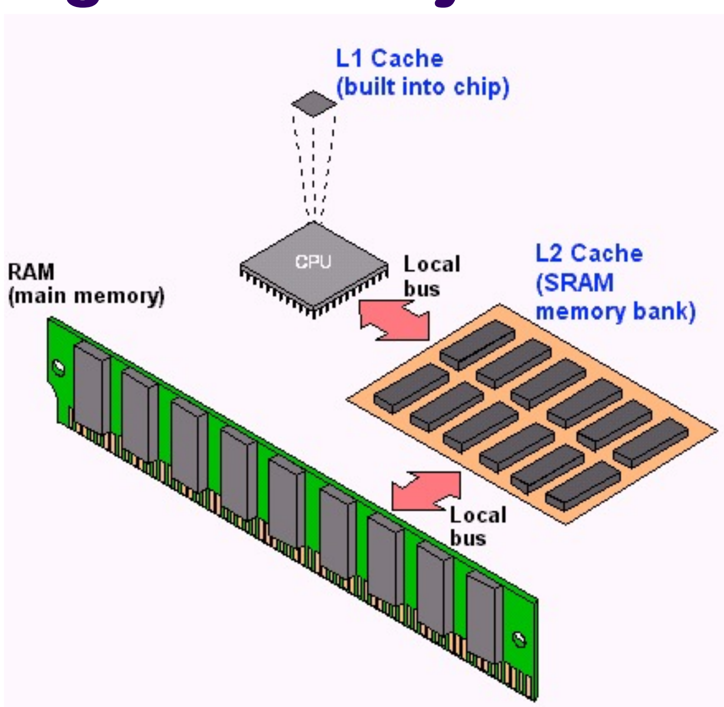
- Primary
  - e.g. Main memory, cache; typically volatile, fast
- Secondary
  - e.g. Disks; Solid State Drives (SSD); non-volatile
- Tertiary
  - e.g. Tapes; Non-volatile, super cheap, slow

- Each storage media has different performance characteristics
  - Important to understand in order to write systems or optimize queries or tasks

Storage Hierarchy: Cache

- Cache
  - Super fast; volatile; Typically on chip
  - L1 vs L2 vs L3 caches ???
    - L1 about 64KB or so; L2 about 1MB; L3 8MB (on chip) to 256MB (off chip)
    - Huge L3 caches available now-a-days
  - Becoming more and more important to care about this
    - Cache misses are expensive
  - Similar tradeoffs as were seen between main memory and disks

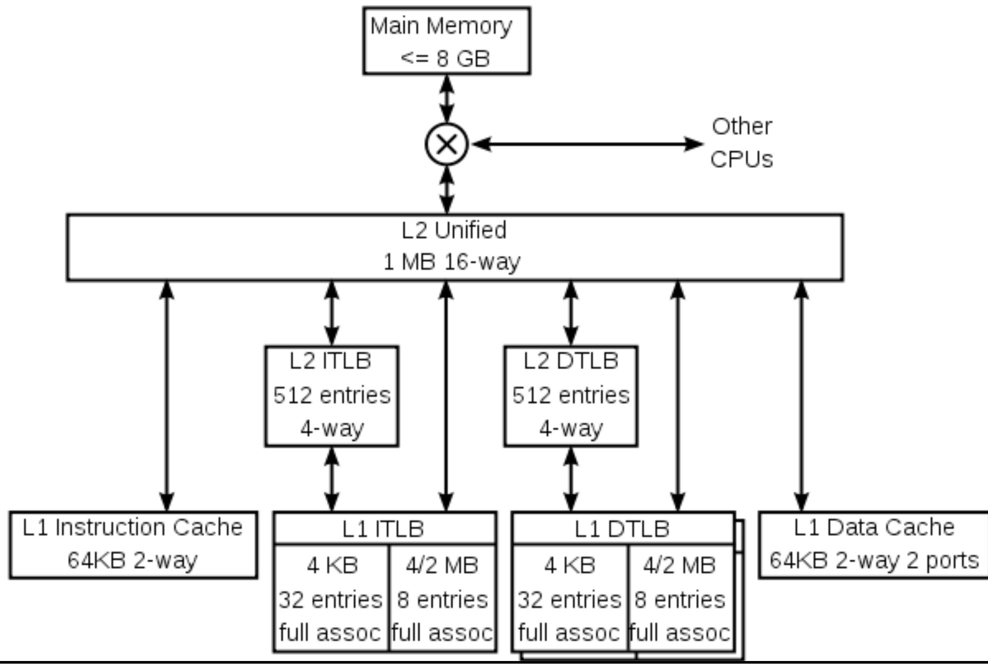
Storage Hierarchy: Cache



source: http://cse1.net/recaps/4-memory.html

Storage Hierarchy: Cache

K8 core in the AMD Athlon 64 CPU



Storage Hierarchy: Main Memory

- Data must be brought from disks/SSDs into Memory (and then into Caches) for the CPU to access it
  - CPU has no "direct" connection to the disks
- 10s or 100s of ns; Volatile (so will not survive a power failure)
- Pretty cheap and dropping: 1GByte < $10 today
- Main memory databases very common now-a-days
  - Dramatically changes the tradeoffs
  - Don't need to worry about the disks or SSDs as much

Storage Hierarchy

- Magnetic Disk (Hard Drive)
  - Non-volatile
  - Sequential access much much faster than random access
  - Discuss in more detail later
- Optical Storage - CDs/DVDs; Jukeboxes
  - Used more as backups… Why ?
  - Very slow to write (if possible at all)
- Tape storage
  - Backups; super-cheap; painful to access
  - IBM just released a secure tape drive storage solution

How important is this today?

- Trade-offs shifted drastically over last 10-15 years
  - Especially with fast network, SSDs, and high memories
  - However, the volume of data is also growing quite rapidly

- Some observations:
  - Cheaper to access another computer's memory than accessing your own disk
  - Cache is playing more and more important role
  - Enough memory around that data often fits in memory of a single machine, or a cluster of machines
  - "Disk" considerations less important
    - Still: Disks are where most of the data lives today
  - Similar reasoning/algorithms required though

CMSC424: Database Design

Module: Database Implementation

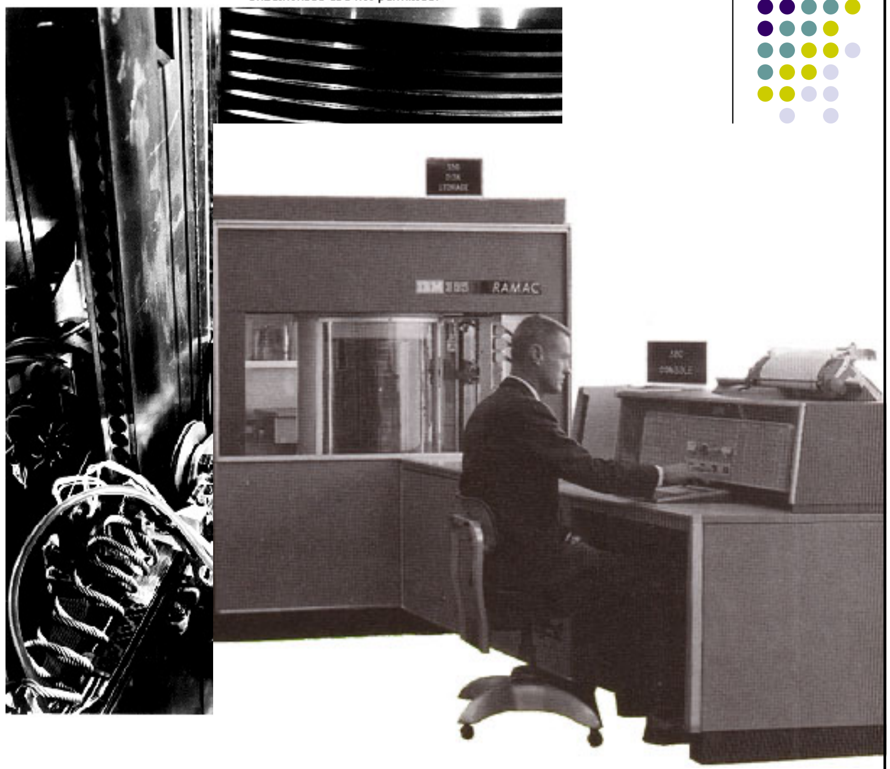
Storage: Disks and SSDs

Instructor: Amol Deshpande
amol@cs.umd.edu

Disks and SSDs

- Book Chapters
  - 10.2
- Key topics:
  - Key components
  - Characteristics
  - Solid State Drives

1956
IBM RAMAC
24” platters
100,000 characters each
5 million characters



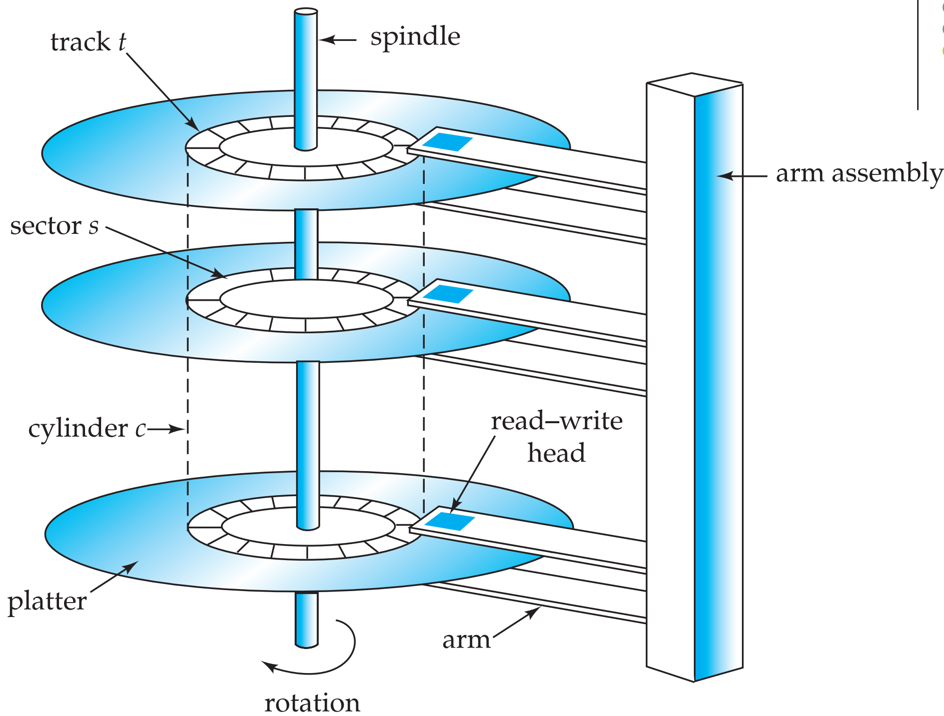
373

1979
SEAGATE
5MB

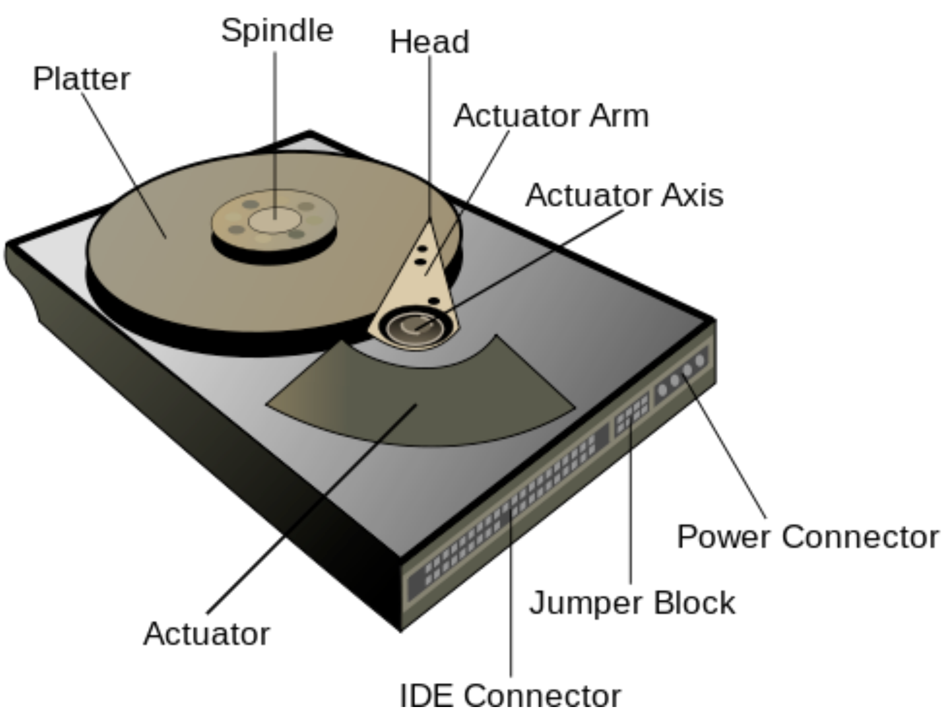
1998
SEAGATE
47GB

2006
Western Digital
500GB
Weight (max. g): 600g

374


track t
spindle
arm assembly
sector s
cylinder c
read–write head
platter
arm
rotation

375


Spindle
Head
Platter
Actuator Arm
Actuator Axis
Power Connector
Jumper Block
Actuator
IDE Connector

376

"Typical" Values

Diameter:	1 inch → 15 inches
Cylinders:	100 → 2000
Surfaces:	1 or 2
(Tracks/cyl)	2 (floppies) → 30
Sector Size:	512B → 50K
Capacity →	360 KB to 2TB (as of Feb 2010)
Rotations per minute (rpm) →	5400 to 15000

Accessing Data

- Accessing a sector
  - Time to *seek* to the track (seek time)
    - average 4 to 10ms
  - + Waiting for the sector to get under the head (rotational latency)
    - average 4 to 11ms
  - + Time to transfer the data (transfer time)
    - very low
  - About 10ms per access
    - So if randomly accessed blocks, can only do 100 block transfers
    - 100 x 512bytes = 50 KB/s
- Data transfer rates
  - Rate at which data can be transferred (w/o any seeks)
  - 30-50MB/s to up to 200MB/s (Compare to above)
    - Seeks are bad !

Reliability

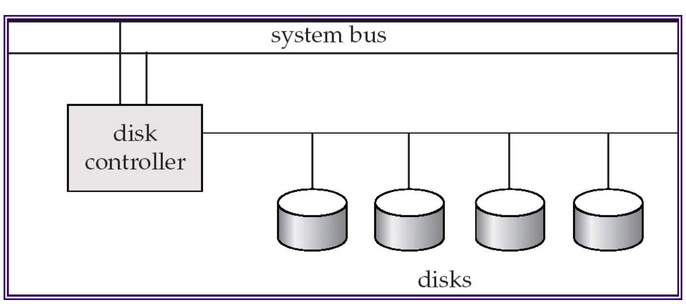
- Mean time to/between failure (MTTF/MTBF):
  - 57 to 136 years

- Consider:
  - 1000 new disks
  - 1,200,000 hours of MTTF each
  - On average, one will fail 1200 hours = 50 days !

- Need to assume disk failures are common
  - Handled today through keeping data in duplicate, or triplicate
  - If a disk fails, replace with a new disk and copy data over

Disk Controller

- Interface between the disk and the CPU
- Accepts the commands
- *checksums* to verify correctness
- Remaps bad sectors

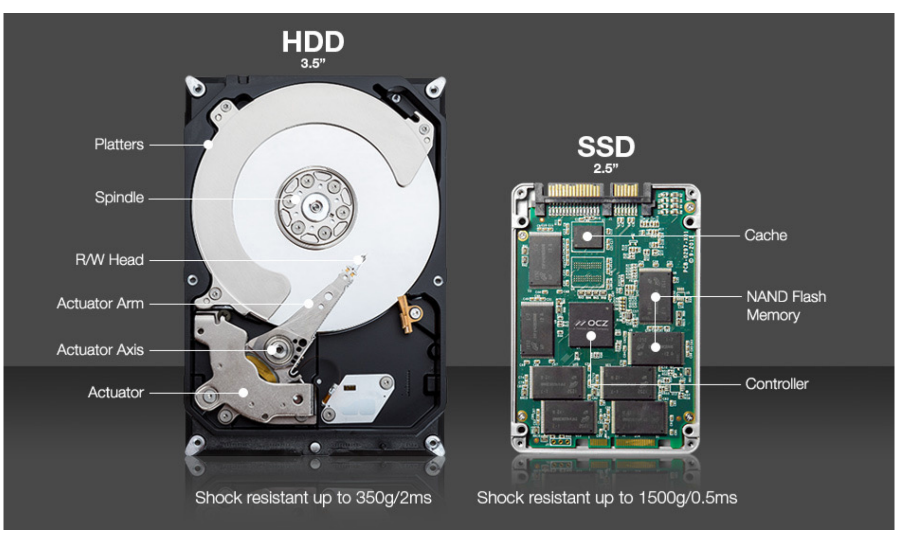


Optimizing block accesses

- Typically sectors too small
- Block: A contiguous sequence of sectors
  - 512 bytes to several Kbytes
  - All data transfers done in units of blocks
- Scheduling of block access requests ?
  - Considerations: *performance* and *fairness*
  - *Elevator algorithm*

Solid State Drives

- Essentially flash that emulates hard disk interfaces



https://blogs.umass.edu/Techbytes/2018/02/23/types-of-ssds-and-which-ones-to-buy/

Solid State Drives

- Still support the same “block-oriented” interface
  - So reads/writes happen in units of blocks
- No seeks → Much better random reads performance
- Writes are more complicated
  - Must write an entire block at a time, after first “erasing” it
  - Limit on how many times you can erase a block
- Wear leveling
  - Distributes writes across the SSD for uniform wearing out
- Flash Translation Layer (FTL) takes care of these issues
- About a factor of 5-10 more expensive right now

CMSC424: Database Design

Module: Database Implementation

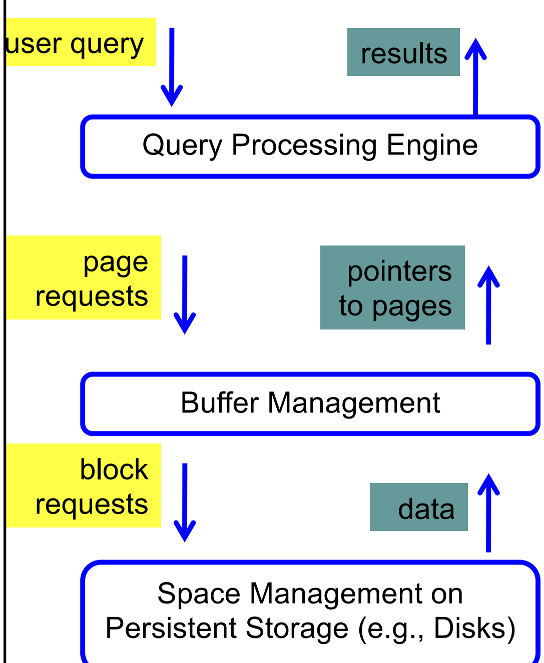
Virtual Memory and Buffer Manager

Instructor: Amol Deshpande
amol@umd.edu

Virtual memory and buffer manager

- Book Chapters
  - 10.7 and other resources (VM not covered in book)
- Key topics:
  - Role of a Buffer Manager
  - Buffer Manager Replacement Policies
  - Key requirements and definitions for Buffer Manager
  - Brief recap of Virtual Memory and Why it matters in practice

Query Processing/Storage



- Given an input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

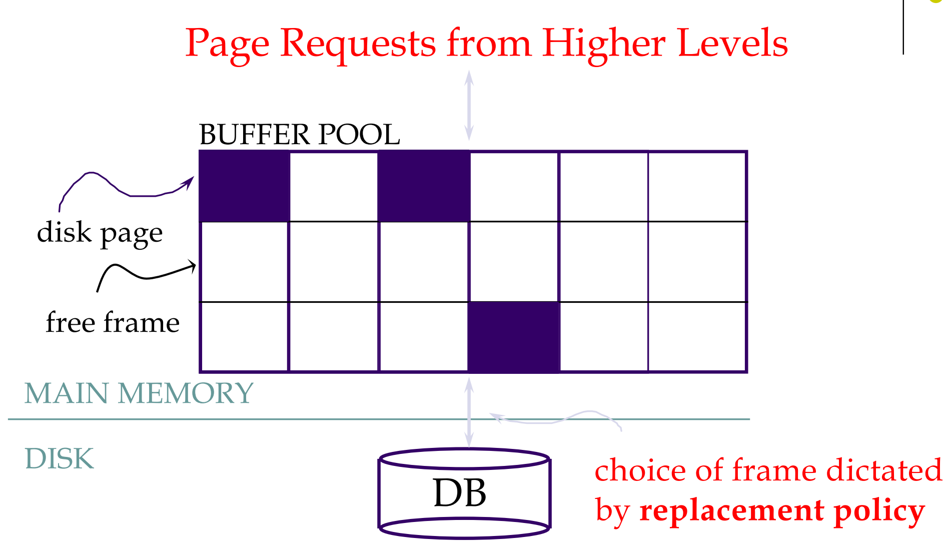
- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Buffer Manager

- When the QP wants a block, it asks the "buffer manager"
  - The block must be in memory to operate upon
- Buffer manager:
  - If block already in memory: return a pointer to it
  - If not:
    - Evict a current page
      - Either write it to temporary storage,
      - or write it back to its original location,
      - or just throw it away (if it was read from disk, and not modified)
    - and make a request to the storage subsystem to fetch it

Buffer Manager



Buffer Manager

- Similar to *virtual memory manager*
- Buffer replacement policies
  - What page to evict ?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - Why ? Works better for database “scan” operations
  - LRU-k
    - Look at the penultimate access rather than the last access
    - Does as well as MRU for scans

Replacement Policy: Example

- Say Buffer can hold 3 pages, and pages are: A, B, C, D, E, F
- For LRU-2: we look at the second-last access
  - If no second-last access, then treat it as: - ∞
  - Break ties based on last access
  - Once a page goes to disk, the accesses reset

Page Request	LRU State	MRU State	LRU-2 State
A	A	A	A
B	A, B	B, A	A, B
C	A, B, C	C, B, A	A, B, C
D	B, C, D	D, B, A	B, C, D
A	C, D, A	A, D, B	C, D, A
C	D, A, C	C, D, B	D, A, C
B	A, C, B	B, C, D	A, B, C

Order of eviction
i.e., A will be evicted first

Different from LRU – B will be evicted earlier
Penultimate access for C is earlier than B (- infinity for B)

Buffer Manager

- *Pinning* a block
  - Not allowed to write back to the disk
- *Force-output (force-write)*
  - Force the contents of a block to be written to disk
- *Order the writes*
  - This block must be written to disk before this block
- Critical for fault tolerant guarantees
  - Otherwise the database has no control over whats on disk and whats not on disk

Reality Check…

- Most operating systems don't provide user programs with direct access to memory
  - Some DBs built their own OSs because of this in the early days
- Most databases today run on top of your OSes
  - Including our PostgreSQL
- Causes several problems
  - OS Buffer Manager doesn't provide the required functionality
  - No real control over when pages are written back
  - Can't "pin" pages, or "force-write"

Reality Check...

- Memory-mapped files help with many of these issues
  - Allow mapping a disk file directly to virtual memory
  - More efficient than going through the OS
- With increasing memory sizes, most databases now-a-days fit in memory
  - Many newer database systems redesigned to exploit this
  - Issues of cache/memory, how memory is managed, etc. becoming increasingly important
  - Distributed/parallel architectures also add more complexity to this

CMSC424: Database Design

Module: File Organization and Indexes

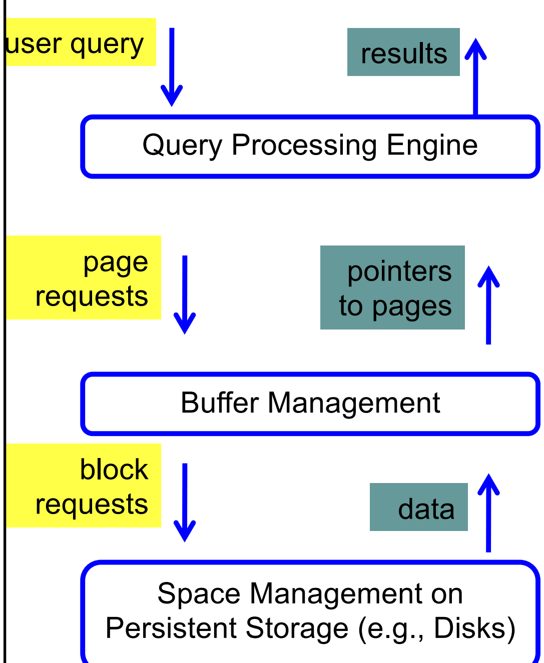
File Organization

Instructor: Amol Deshpande
amol@umd.edu

File Organization & Indexes Overview

- Book Chapters
  - 10.5, 10.6
- Key topics:
  - Different ways the tuples mapped to disk blocks
  - Pros and cons of the different approaches

Review: Query Processing/Storage

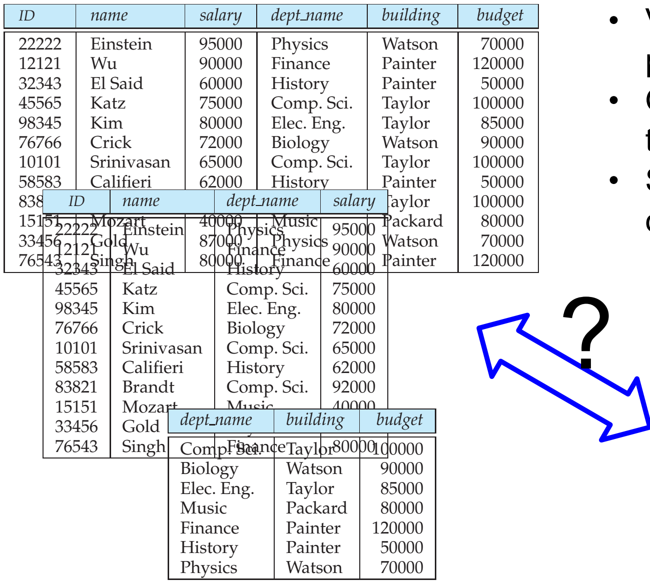


- Given a input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

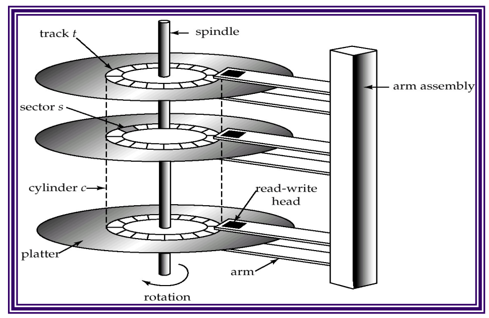
- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Mapping Tuples to Disk Blocks



- Very important implications on performance
- Quite a few different ways to do this
- Similar issues even if not using disks as the primary storage



File Organization

- Requirements and Performance Goals:
  - Allow insertion/deletions of tuples/records in relations
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
  - Fetch all tuples from a specific relation (scans)
    - Faster if they are all sequential/in contiguous blocks
  - Allow building of "indexes"
    - Auxiliary data structures maintained on disks and in memory for faster retrieval
  - And so on…

File System or Not

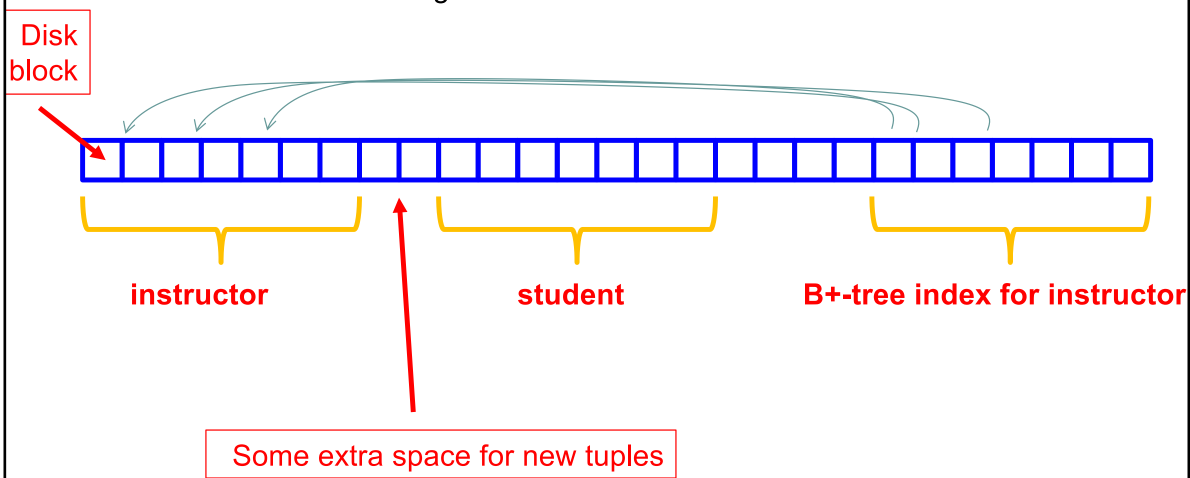
- Option 1: Use OS File System
  - File systems are a standard abstraction provided by Operating Systems (OS) for managing data
  - Major Con: Databases don't have as much control over the physical placement anymore --- OS controls that
    - E.g., Say DBMS maps a relation to a "file"
    - No guarantee that the file will be "contiguous" on the disk
    - OS may spread it across the disk, and won't even tell the DBMS
- Option 2: DBMS directly works with the disk or uses a lightweight/custom OS
  - Increasingly uncommon – most DBMSs today run on top of OSes (e.g., PostgreSQL on your laptop, or on linux VMs in the cloud, or on a distributed HDFS)

Through a File System

- Option 1: Allocate a single "file" on the disk, and treat it as a contiguous sequence of blocks
  - This is what PostgreSQL does
  - The blocks may not actually be contiguous on disk
- Option 2: A different file per relation
  - Some of the simpler DBMS use this approach
- Either way: we have a set of relations mapped to a set of blocks on disk

Assumptions for Now

- Each relation stored separately on a separate set of blocks
  - Assumed to be contiguous
- Each "index" maintained in a separate set of blocks
  - Assumed to be contiguous



Within block: Fixed Length Records

- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: Rearrange
  - Option 2: Keep a *free list* and use for next insert

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

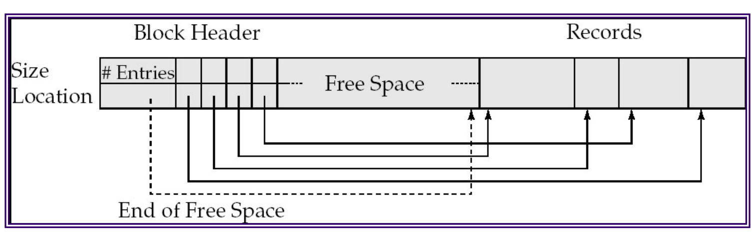
Within block: Fixed Length Records

- Deleting: using "free lists"

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Within block: Variable-length Records

Slotted page/block structure



- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

Across Blocks of a Relation

- Which block should a record go to ?
  - Anywhere ?
    - How to search for "SSN = 123" ?
    - Called "heap" organization
  - Sorted by SSN ?
    - Called "sequential" organization
    - Keeping it sorted would be painful
    - How would you search ?
  - Based on a "hash" key
    - Called "hashing" organization
    - Store the record with SSN = x in the block number x%1000
    - Why ?

Across Blocks: Sequential File Organization

- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become *fragmented*
  - Must reorganize once in a while

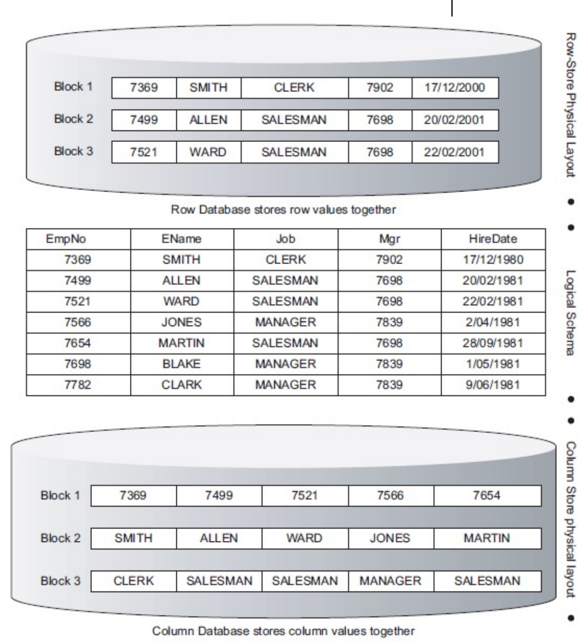
Across Blocks: Sequential File Organization

- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses
  - Too much
    - n = 1,000,000,000 -- log(n) = 30
    - Recall each random access approx 10 ms
    - 300 ms to find just one account information
    - < 4 requests satisfied per second

Indexes – next topic

Advanced Topics

- Row vs columnar representation:
  - We are largely focused on row representation
  - Column-based organization much more efficient for queries
    - But are not as efficient to update
  - Used by most modern warehouses



Advanced Topics

- Data Storage Formats used in ”big data” world
  - Parquet, Avro, and many others
- Sophisticated on-disk and in-memory representations for maintaining very large volumes of data as ”files”
  - That can be emailed, shared, interpreted by many different programs
- Typically tend to be ”column-oriented”
  - Are not designed to be easy to update (by and large)
- Lot of work in recent years on this

CMSC424: Database Design

Module: File Organization and Indexes

Indexes Overview

Instructor: Amol Deshpande
amol@cs.umd.edu

File Organization & Indexes Overview

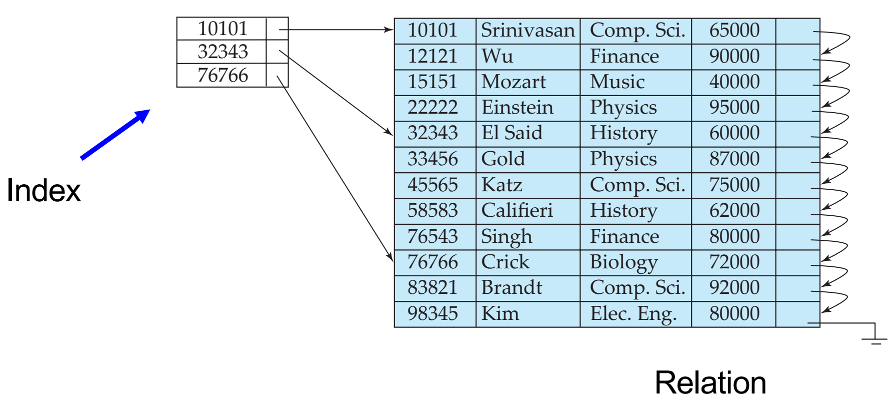
- Book Chapters
  - 11.1, 11.2
- Key topics:
  - How an “index” helps efficiently find tuples that satisfy a condition?
  - What are key characteristics of indexes?

Index

- A data structure for efficient search through large databaess
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
- Search key:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
- Two types of indexes
  - Ordered indexes
  - Hash-based indexes
- Think library index/catalogue

Ordered Indexes

- Primary index
  - The relation is sorted on the search key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation



10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Primary *Sparse* Index

- Every key doesn't have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoff: Must access the relation file even if the record is not present

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Primary *Dense* Index

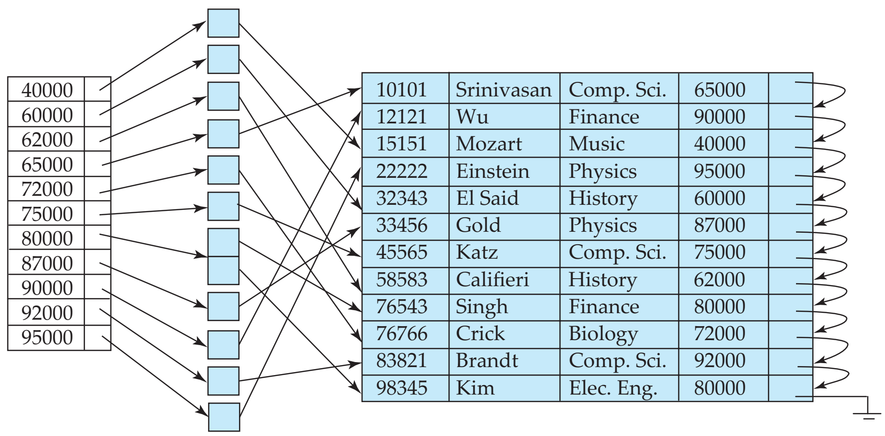
- Every key must appear in the index
- Index becomes pretty large, but can often avoid having to go to the relation
  - E.g., select * from instructor where ID = 10000
    - Not found in the index, so can return immediately

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

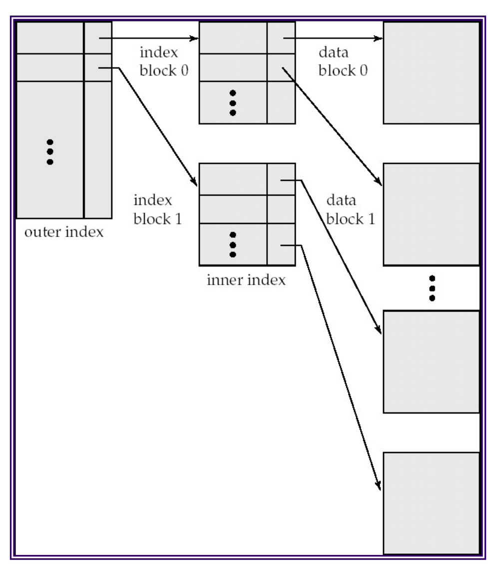
Secondary Index

- Relation sorted on *ID*
- But we want an index on *salary*
- Must be dense
  - Every search key must appear in the index



Multi-level Indexes

- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution
  - Build an index on the index itself



Multi-level Indexes

- How do you search through a multi-level index ?
- What about keeping the index up-to-date ?
  - Tuple insertions and deletions
    - This is a static structure
    - Need overflow pages to deal with insertions
  - Works well if no inserts/deletes
  - Not so good when inserts and deletes are common

CMSC424: Database Design

Module: File Organization and Indexes

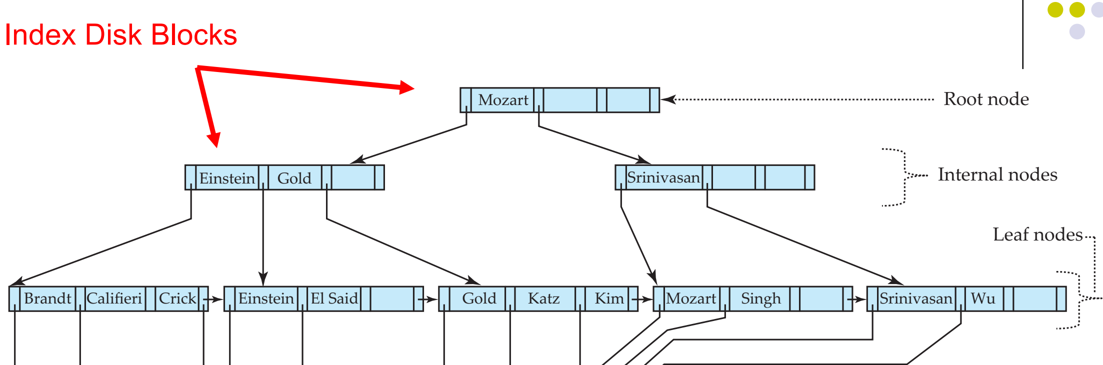
B+-Trees: Basics

Instructor: Amol Deshpande
amol@cs.umd.edu

B+-Trees

- Book Chapters
  - 11.3
- Key topics:
  - B+-Trees as a multi-level index, and basic properties
  - How to search in a B+-Tree?

Example B+-Tree Index



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B⁺-Tree Node Structure

- Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Properties of B+-Trees

- It is balanced
  - Every path from the root to a leaf is same length
- Leaf nodes (at the bottom)
  - *P1* contains the pointers to tuple(s) with key *K1*
  - …
  - *Pn* is a pointer to the *next* leaf node
  - Must contain at least n/2 entries

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

Properties

- Interior nodes

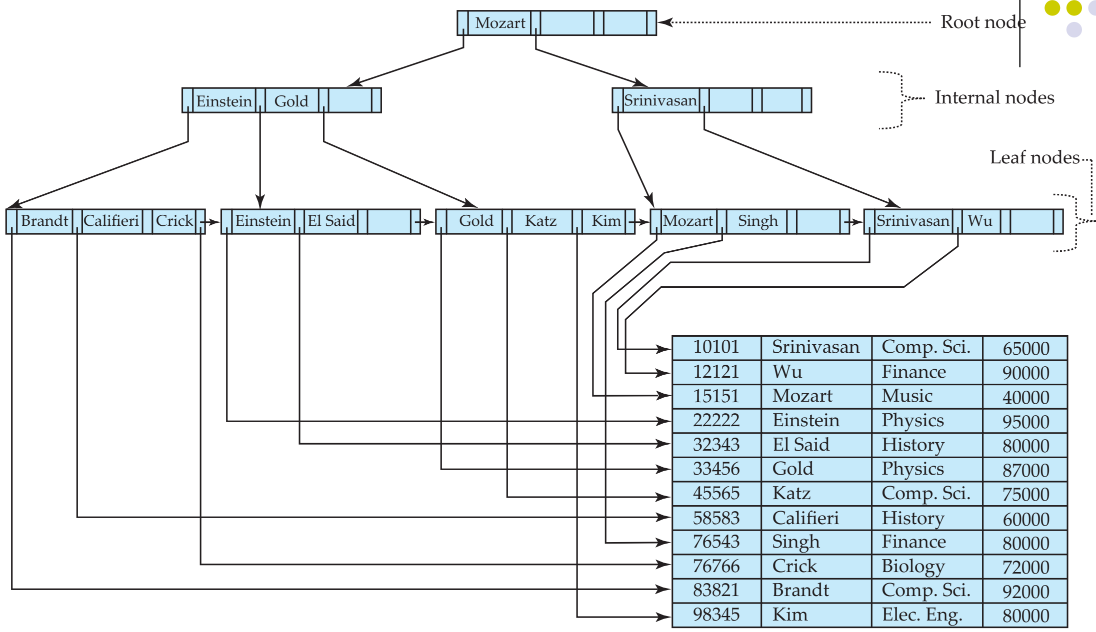
P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

  - All tuples in the subtree pointed to by *P1,* have search key < *K1*
  - To find a tuple with key *K1' < K1,* follow *P1*
  - …
  - Finally, search keys in the tuples contained in the subtree pointed to by *Pn,* are all larger than *Kn-1*
  - Must contain at least n/2 entries (unless root)

B+-Trees - Searching

- How to search ?
  - Follow the pointers
- Logarithmic
  - $log_{B/2}(N)$*,* where *B = Number of entries per block*
  - *B* is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000,000 entries, takes only 4 random accesses
- The top levels are typically in memory
  - So only requires 1 or 2 random accesses per request

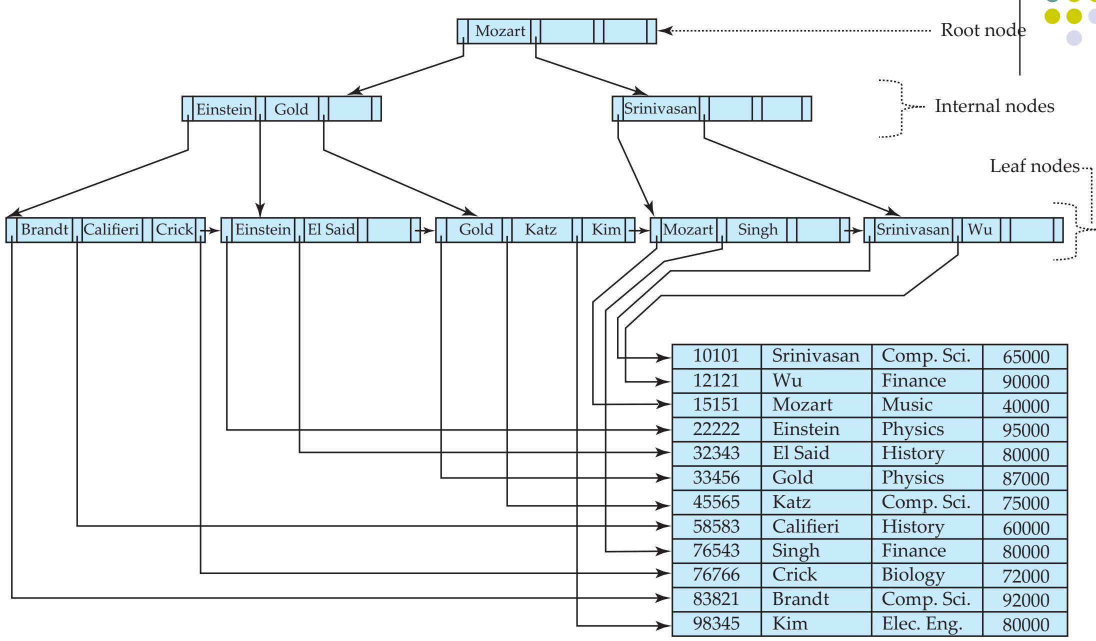
Example B+-Tree Index



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

427

Example B+-Tree Index



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

If this were a “primary” index, then not all ”keys” are present in the index

428

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

CMSC424: Database Design

Module: File Organization and Indexes

B+-Trees: Inserts

Instructor: Amol Deshpande
amol@cs.umd.edu

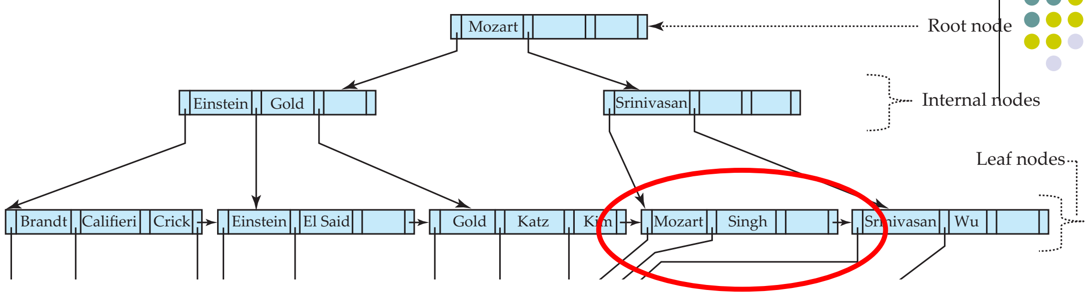
B+-Trees: Inserts

- Book Chapters
  - 11.3.3.1
- Key topics:
  - How to insert a new entry in the index while keeping it balanced and satisfying half-full guarantees

Tuple Insertion

- Find the leaf node where the search key should go
- If already present
  - Insert record in the file. Update the bucket if necessary
    - This would be needed for secondary indexes
- If not present
  - Insert the record in the file
  - Adjust the index
    - Add a new *(Ki, Pi)* pair to the leaf node
    - Recall the keys in the nodes are sorted
  - What if there is no space ?

B+-Trees: Insertion



Tuple Insertion

- Splitting a node
  - Node has too many key-pointer pairs
    - Needs to store *n,* only has space for *n-1*
  - Split the node into two nodes
    - Put about half in each
  - Recursively go up the tree
    - May result in splitting all the way to the root
    - In fact, may end up adding a *level* to the tree
  - Pseudocode in the book !!

B+-Trees: Insertion

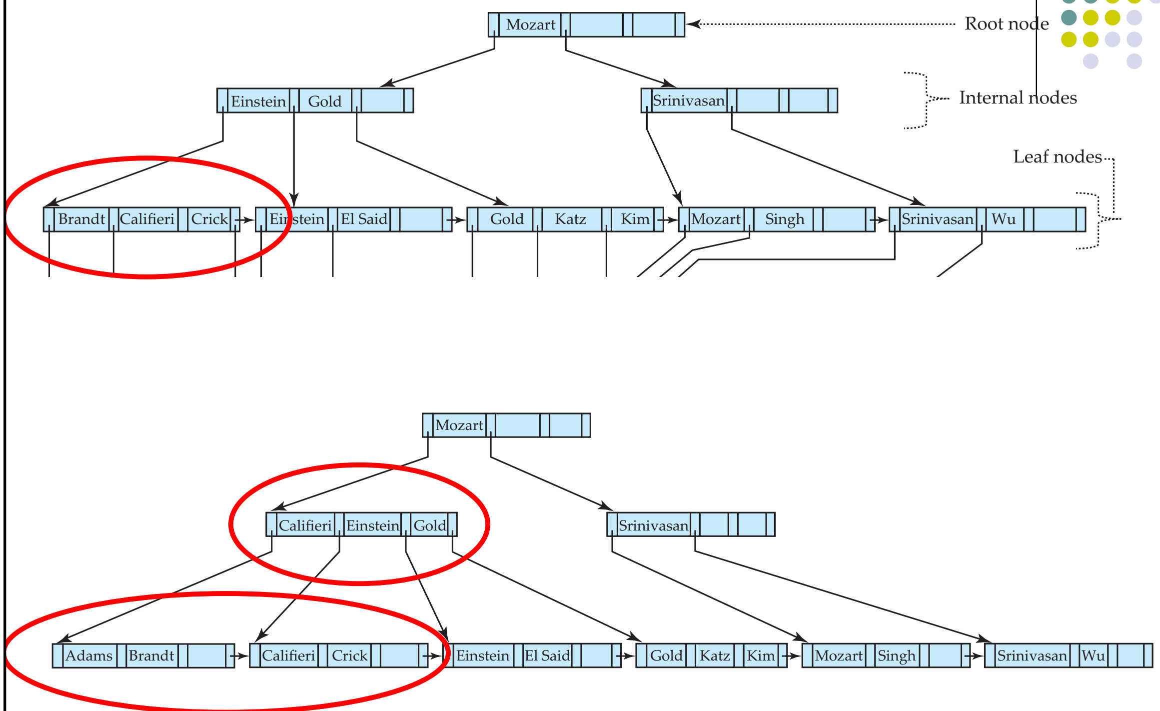


Figure 11.13 Insertion of "Adams" into the B+-tree of Figure 11.9.

B+-Trees: Insertion

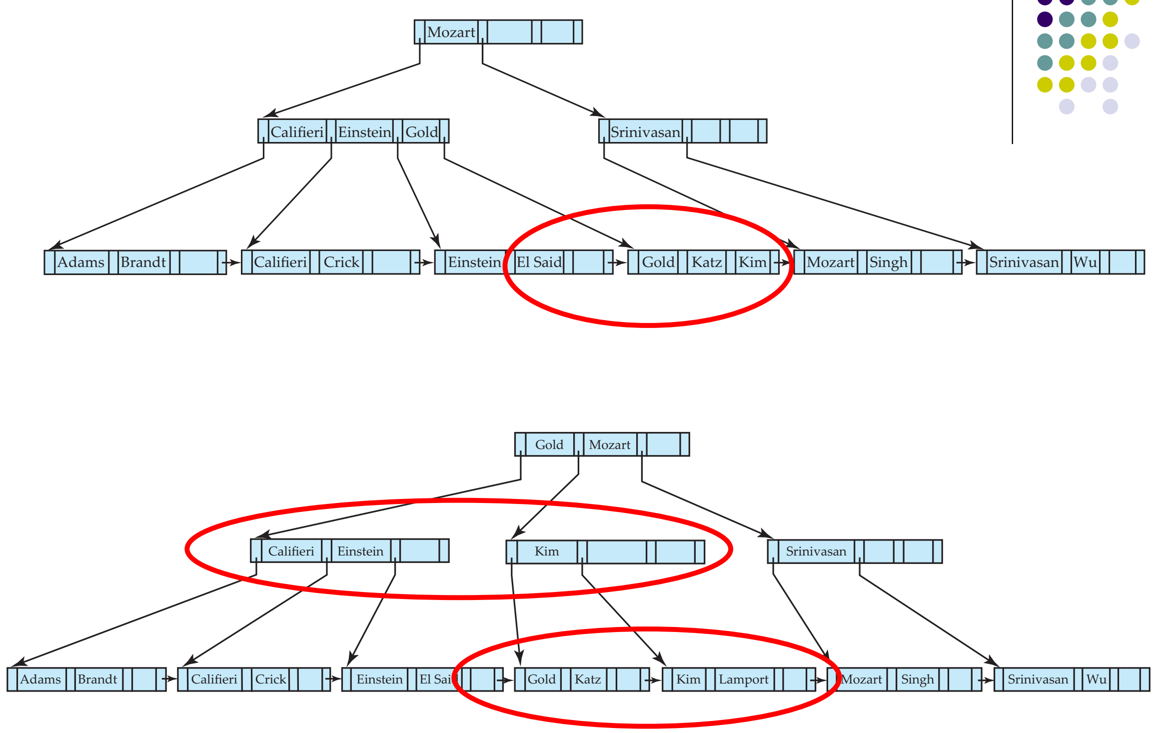
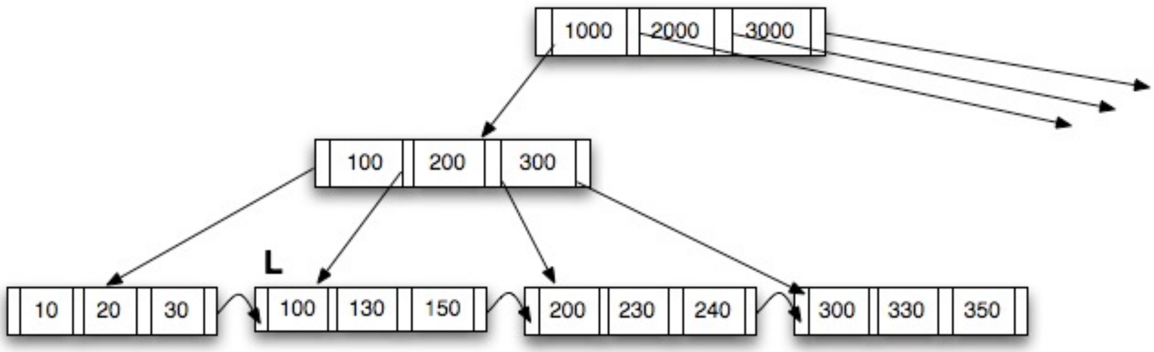


Figure 11.14 Insertion of "Lamport" into the B+-tree of Figure 11.13.

Another B+Tree Insertion Example

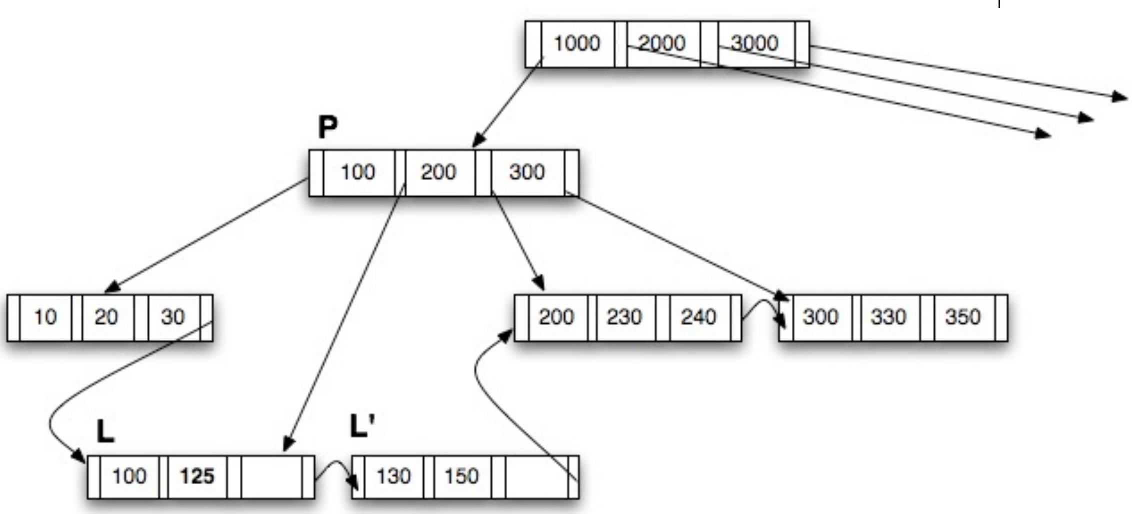
INITIAL TREE



Next slides show the insertion of (125) into this tree
According to the Algorithm in Figure 12.13, Page 495

Another Example: INSERT (125)

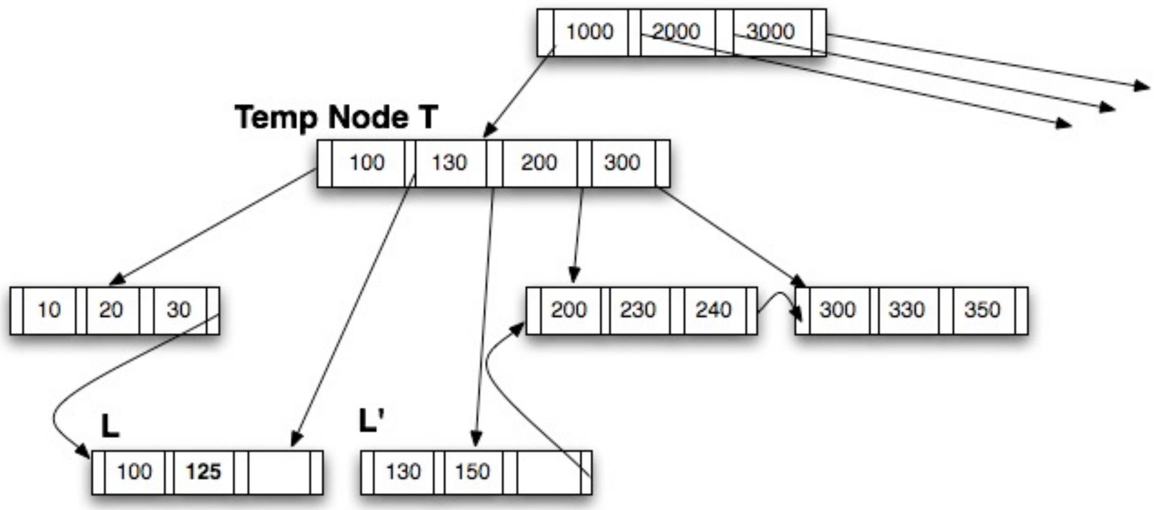
Step 1: Split L to create L'



Insert the lowest value in L' (130) upward into the parent P

Another Example: INSERT (125)

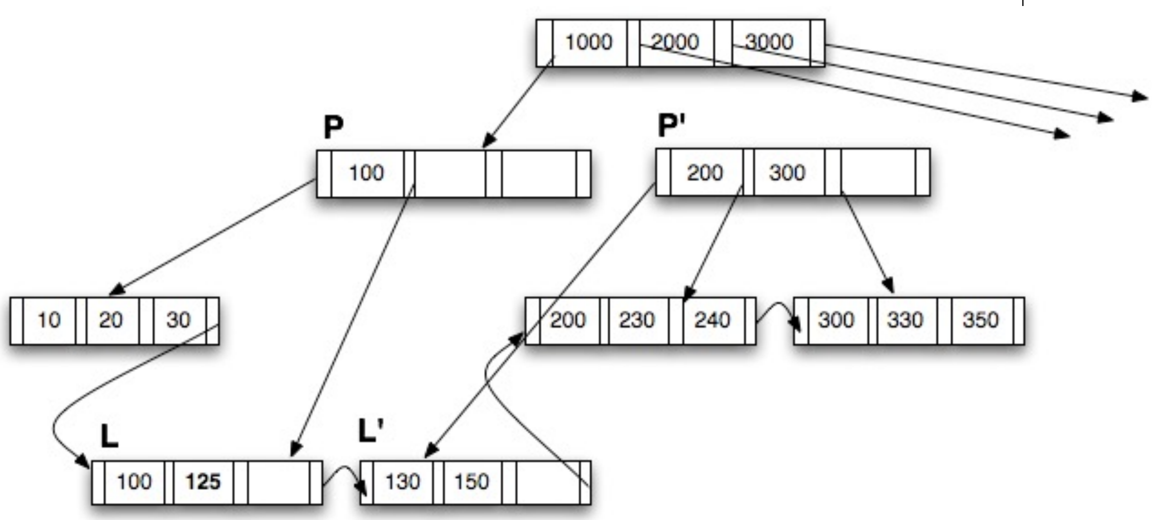
Step 2: Insert (130) into P by creating a temp node T



439

Another Example: INSERT (125)

Step 3: Create P'; distribute from T into P and P'

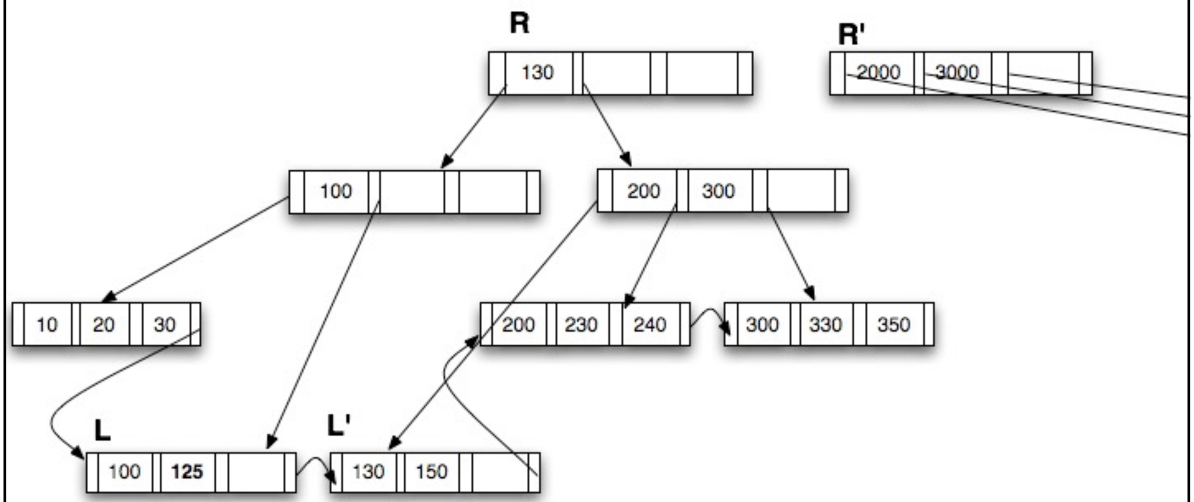


New P has only 1 key, but two pointers so it is OKAY.
This follows the last 4 lines of Figure 12.13 (note that "n" = 4)
K" = 130. Insert upward into the root

440

Another Example: INSERT (125)

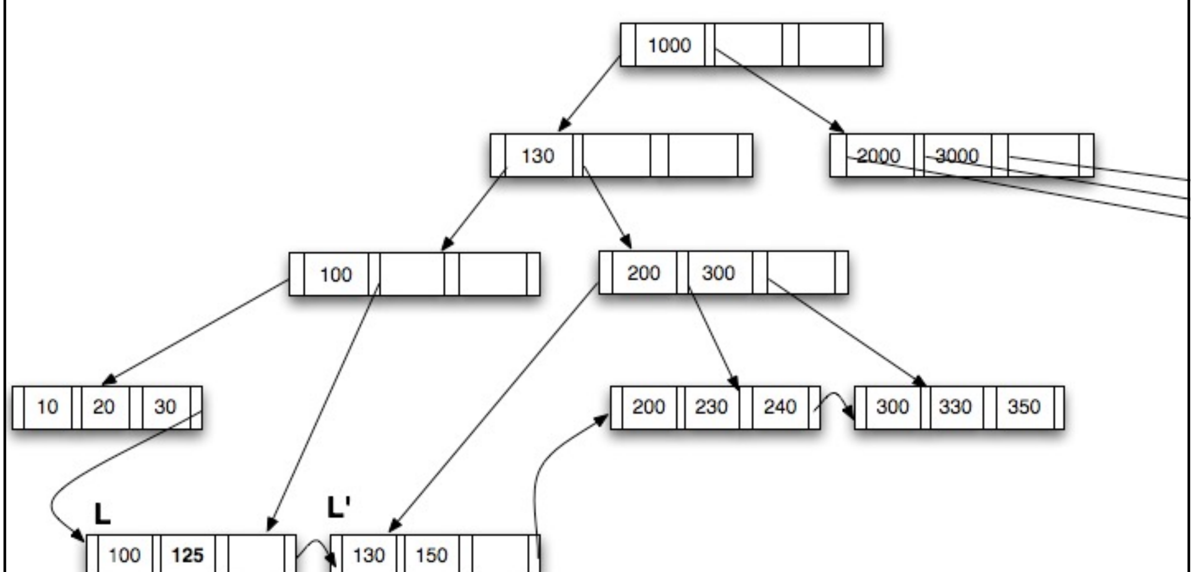
Step 4: Insert (130) into the parent (R); create R'



Once again following the insert_in_parent() procedure, K" = 1000

Another Example: INSERT (125)

Step 5: Create a new root



CMSC424: Database Design

Module: File Organization and Indexes

B+-Trees: Deletions

Instructor: Amol Deshpande
amol@cs.umd.edu

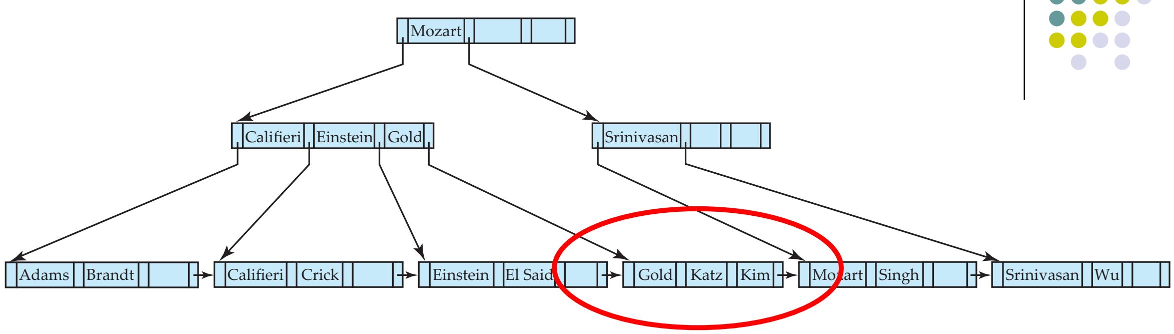
B+-Trees: Deletions

- Book Chapters
  - 11.3.3.2
- Key topics:
  - How to delete an existing entry in the index while keeping it balanced and satisfying half-full guarantees

Updates on B+-Trees: Deletion

- Find the record, delete it.
- Remove the corresponding (search-key, pointer) pair from a leaf node
  - Note that there might be another tuple with the same search-key
  - In that case, this is not needed
- Issue:
  - The leaf node now may contain too few entries
    - Why do we care ?
  - Solution:
    1. See if you can borrow some entries from a sibling
    2. If all the siblings are also just barely full, then *merge (opposite of split)*
  - May end up merging all the way to the root
  - In fact, may reduce the height of the tree by one

Examples of B+-Tree Deletion



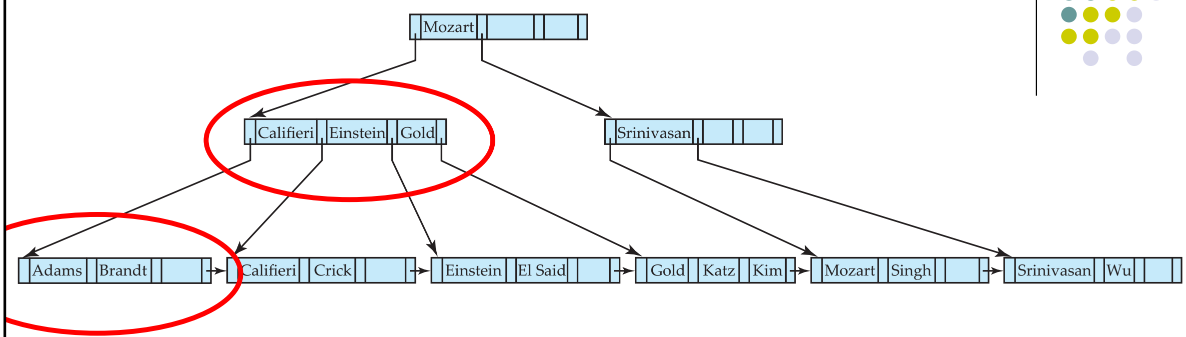
Deleting "Katz" – No issues

Deleting "Gold" – Just delete from the leaf

Gold can stay in the "interior" node – no need to delete it

The purpose of the search keys in the interior nodes is to "direct" searches

Examples of B⁺-Tree Deletion



Deleting "Brandt"

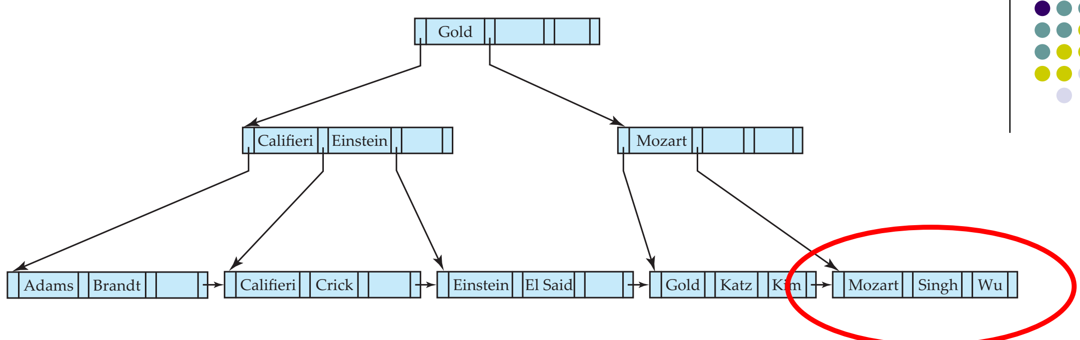
The first leaf node becomes underful

Merge with the next node, and modify Parent appropriately.

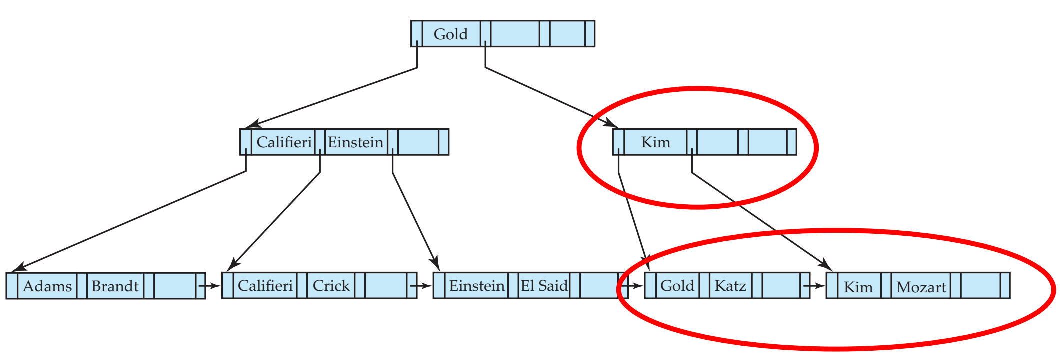
Merged leaf node: Adams, Califieri, Crick

Updated parent node: Einstein, Gold (one fewer entry)

Examples of B⁺-Tree Deletion



Before and after deleting "Singh" and "Wu"



Examples of B⁺-Tree Deletion

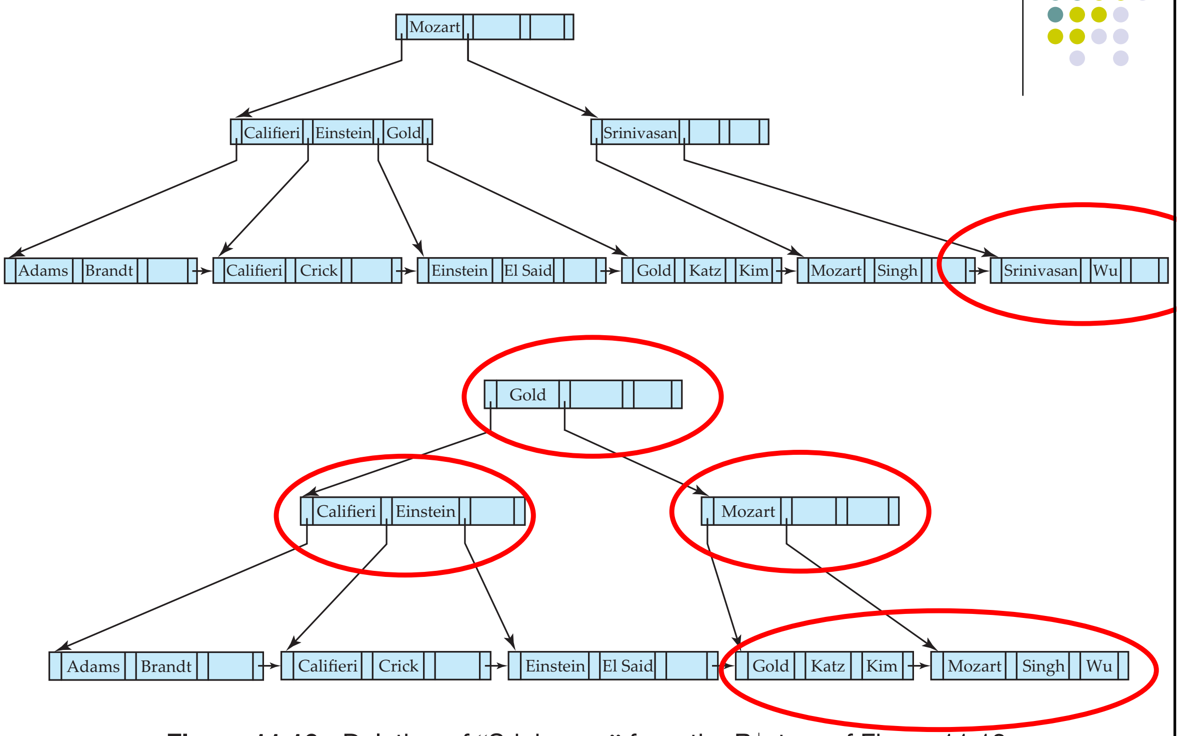
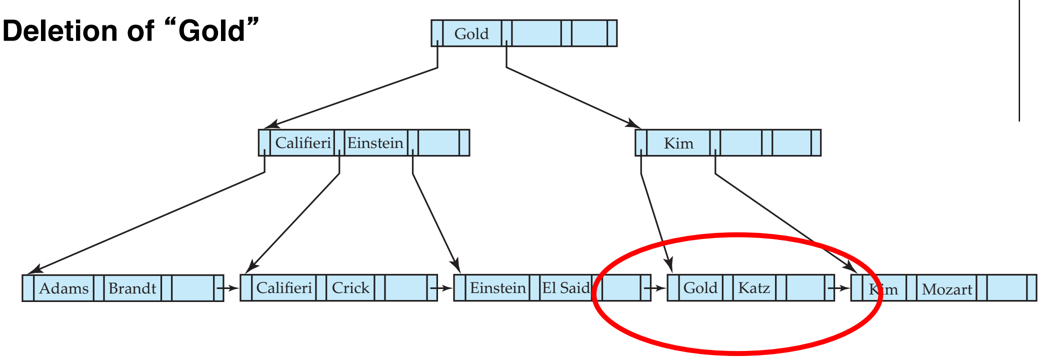


Figure 11.16 Deletion of "Srinivasan" from the B⁺-tree of Figure 11.13.

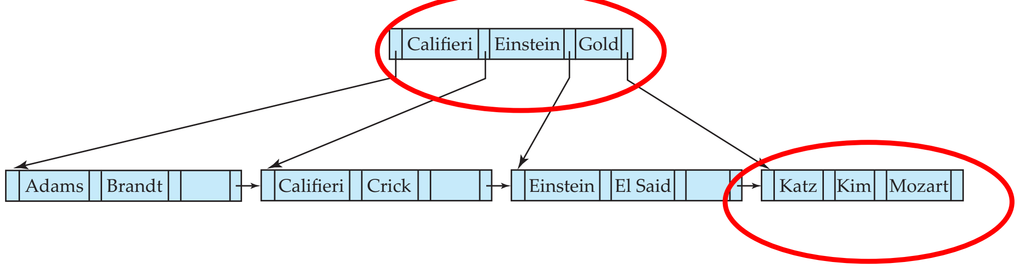
449

Examples of B⁺-Tree Deletion

Deletion of "Gold"



- Rightmost two leaves merged into a single one: (Katz, Kim, Mozart)
- Need to remove a pointer from parent node (Kim), which also becomes underful and merged with its sibling (Califieri, Einstein) ➔ New root



450

CMSC424: Database Design

Module: File Organization and Indexes

Hash Indexes; Miscellaneous

Instructor: Amol Deshpande
amol@cs.umd.edu

Hash Indexes

- Book Chapters
  - 11.6, 11.7 (at a high level), 11.4.1, 11.4.5, 11.5, 11.9 (briefly)
- Key topics:
  - Hash-based file organization
  - Static hashing-based indexes
  - Handling of bucket overflows
  - B-Tree Indexes, B+-Tree File Organization
  - Multi-key indexes, Bitmap indexes, R-Trees

Hash-based File Organization

Store record with search key *k* in block number *h(k)*

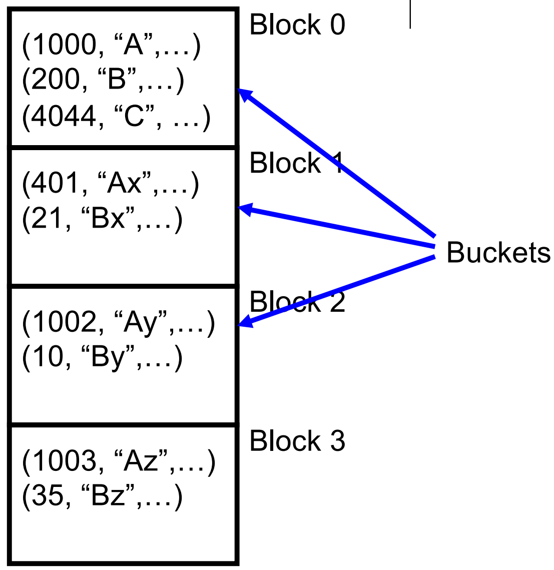
e.g. for a person file,
h(SSN) = SSN % 4

Blocks called "*buckets*"

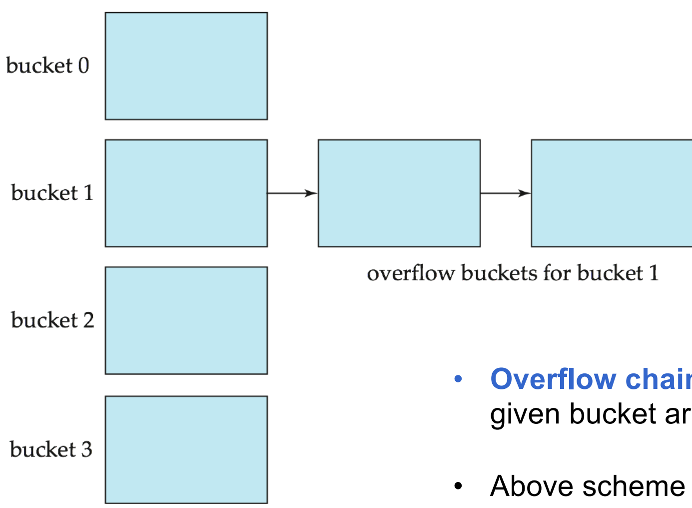
What if the block becomes full ?
Overflow pages

Uniformity property:
Don't want all tuples to map to the same bucket
h(SSN) = SSN % 2 would be bad

Hash functions should also be **random**
Should handle different real datasets



Overflow Pages



- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.

Hash-based File Organization

Hashed on "branch-name"

Hash function:

a = 1, b = 2, .., z = 26

h(abz)

= (1 + 2 + 26) % 10

= 9

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash Indexes

Extends the basic idea

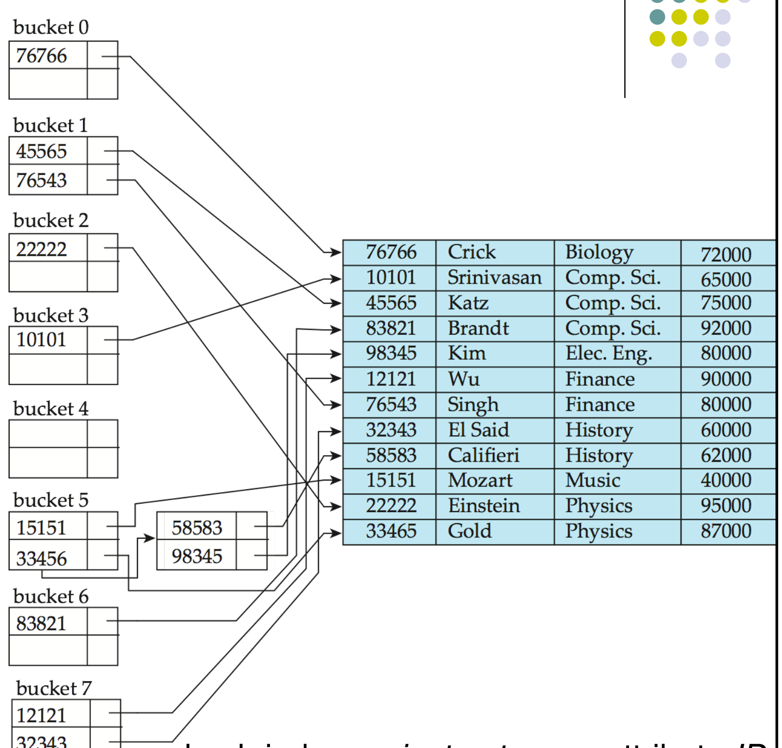
Search:

Find the block with search key

Follow the pointer

Range search ?

a < X < b ?



hash index on *instructor*, on attribute *ID*

Hash Indexes

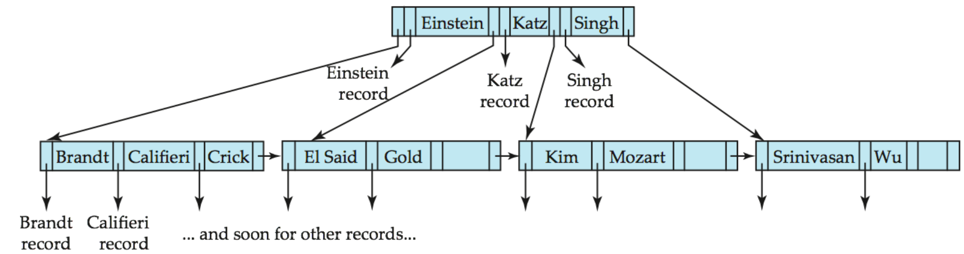
- Very fast search on equality
- Can't search for "ranges" at all
  - Must scan the file
- Inserts/Deletes
  - Overflow pages can degrade the performance
  - Can do periodic reorganization (by modifying hash functions)
- A better approach is to use "dynamic hashing"
  - Allow use of a hash function that can be modified
  - e.g., Extendable Hashing, or Linear Hashing

Comparison of Ordered Indexing and Hashing

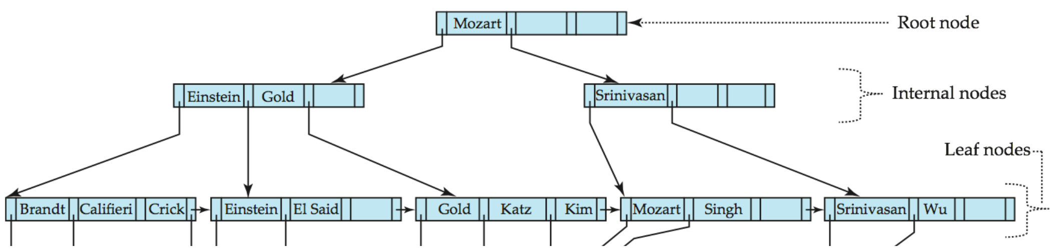
- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

- Hashing very common in distributed settings (e.g., in key-value stores)

B-Tree Index Example



B-tree (above) and B+-tree (below) on same data – B-Trees have "record pointers" at interior nodes

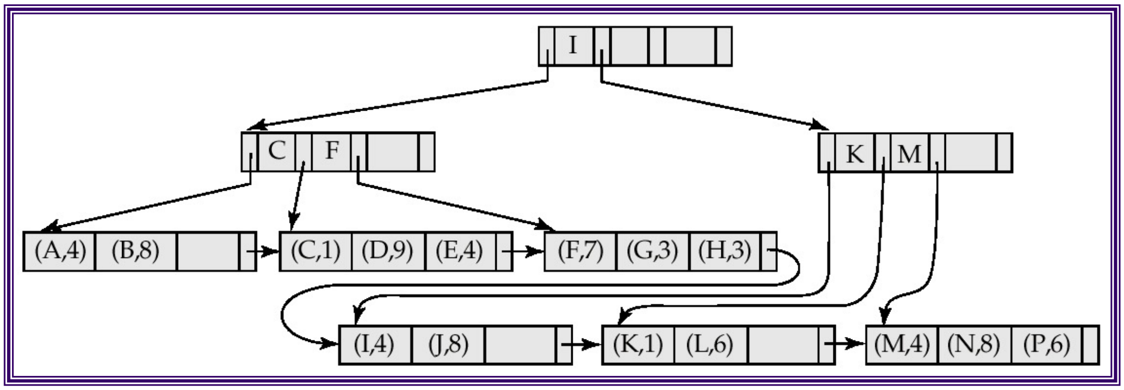


B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

B+-Tree File Organization

- Store the records at the leaves
- Sorted order etc..



Multiple-Key Access

select *ID*
from *instructor*
where *dept_name* = "Finance" **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  - Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  - Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
  - Use *dept_name* index to find pointers to all records pertaining to the "Finance" department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.
    - Called "INDEX-ANDING"

Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1=b_1$ and $a_2 < b_2$

- Ideal for something like:

  where *dept_name* = "Finance" **and** *salary* = 80000
- Can also efficiently handle

  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle

  where *dept_name* < "Finance" **and** *balance* = 80000

Bitmap Indices

- Specialized indexes used in data warehouses

- Assume records numbered sequentially from 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size

- Best for attributes that with a small domain
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)

- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- Bitmap index on an attribute has one bitmap **for each value of the attribute**
  - Bitmap has as many bits as records
  - Keeps track of whether a record has that value for the attr

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

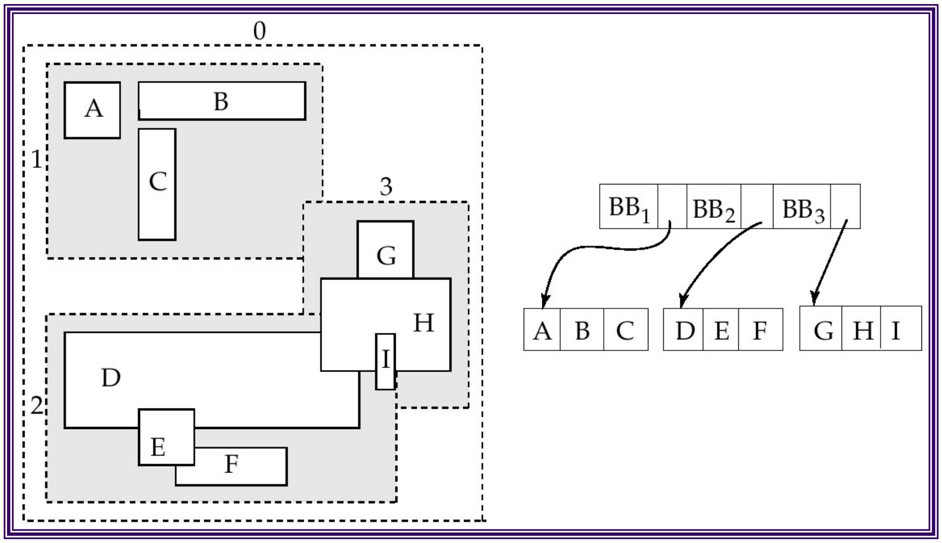
L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Not particularly useful for single attribute queries
- But consider a query: gender = m and income_level = L1
  - Retrieve individual bitmaps for those two
  - Do an AND to find all records that satisfy both conditions
  - Retrieve only those records
- Can also be used for gender = m or income_level = L1
- Really useful when queries have many predicates, and relations are large (i.e., a data warehouse)
- Updating bitmap indexes is very expensive

R-Trees

For spatial data (e.g. maps, rectangles, GPS data etc)



Conclusions

- Indexing Goal: "Quickly find the tuples that match certain conditions"
- Equality and range queries most common
  - Hence B+-Trees the predominant structure for on-disk representation
  - Hashing is used more commonly for in-memory operations
- Many many more types of indexing structures exist
  - For different types of data
  - For different types of queries
    - E.g. "nearest-neighbor" queries

CMSC424: Database Design

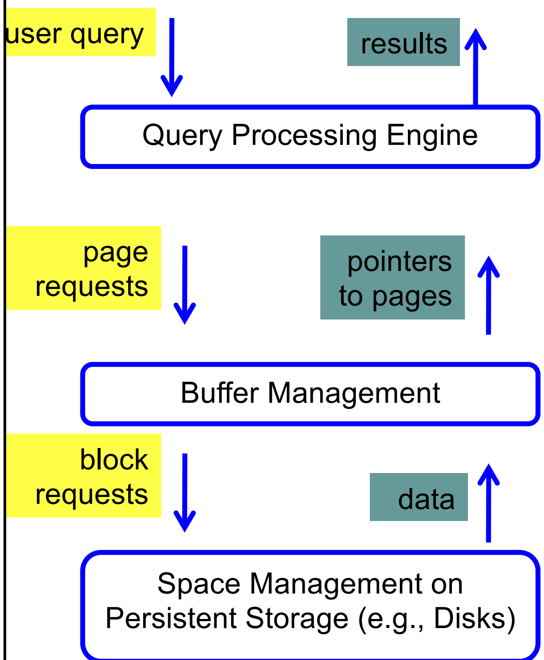
Module: Database Implementation

Instructor: Amol Deshpande
amol@cs.umd.edu

Database Implementation

- Shifting into discussing the internals of a DBMS
  - How data stored? How queries/transactions executed?
- Topics:
  - Storage: How is data stored? Important features of the storage devices (RAM, Disks, SSDs, etc)
  - File Organization: How are tuples mapped to blocks
  - Indexes: How to quickly find specific tuples of interest (e.g., all 'friends' of 'user0')
  - Query processing: How to execute different relational operations? How to combine them to execute an SQL query?
  - Query optimization: How to choose the best way to execute a query?

Query Processing/Storage



- Given an input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

CMSC424: Database Design

Module: Database Implementation

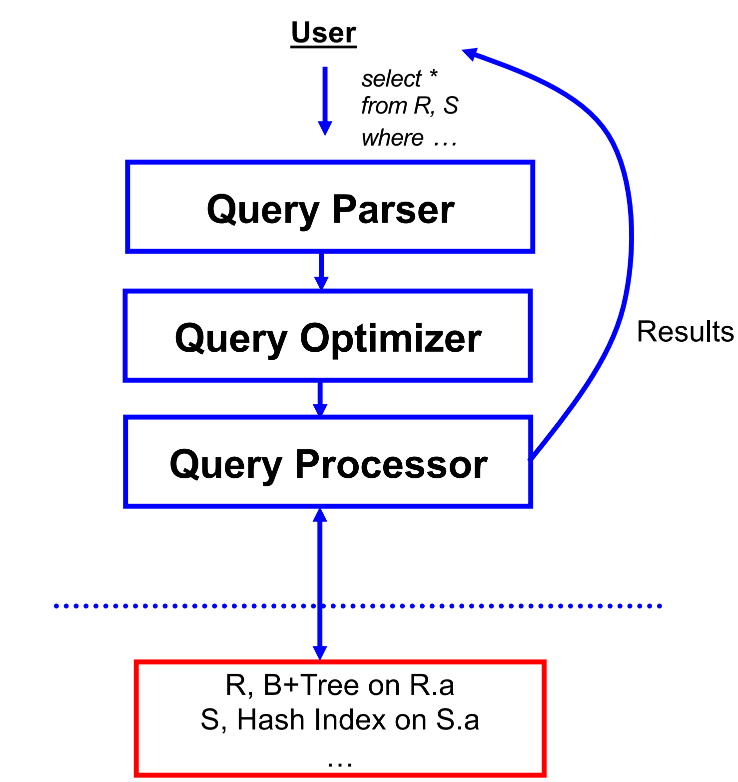
Query Processing: Overview, and Cost Measures

Instructor: Amol Deshpande
amol@umd.edu

Overview and Cost Measures

- Book Chapters
  - 12.1, 12.2
- Key topics:
  - Main steps in Query Processing
  - How to measure the ”cost” of an operation so we can compare alternatives?

Getting Deeper into Query Processing

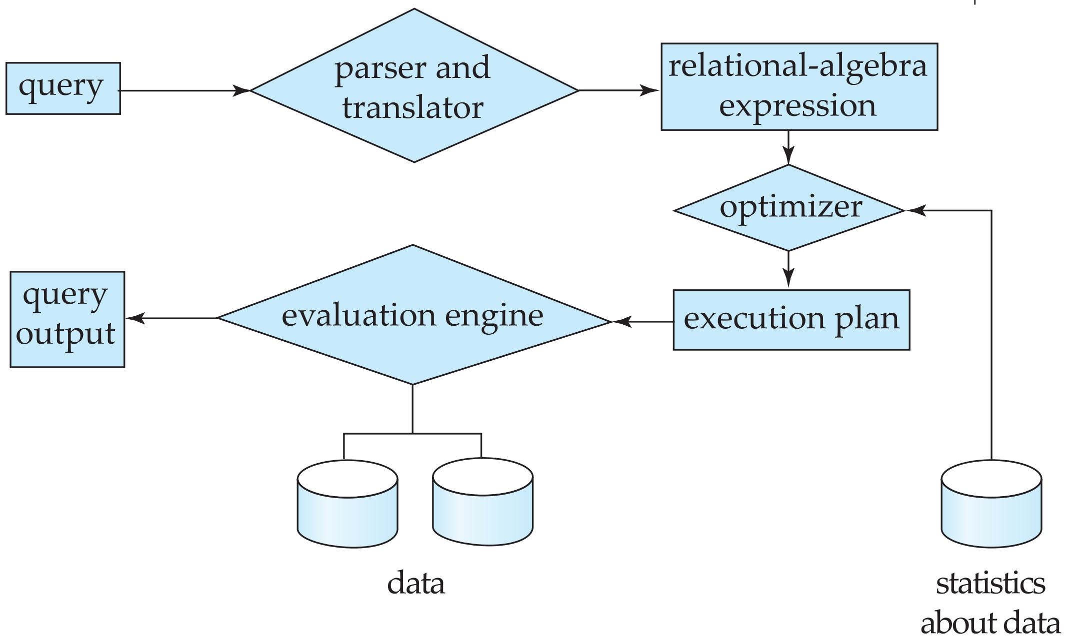


Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?
…

Read the data from the files
Do the query processing
joins, selections, aggregates
…

Getting Deeper into Query Processing



"Cost"

- Complicated to compute, but very important to decide early on
  - Need to know what you are "optimizing" for
- Many competing factors in today's computing environment
  - CPU Instructions
  - Disk I/Os
  - Network Usage – either peak or average (for distributed settings)
  - Memory Usage
  - Cache Misses
  - … and so on
- Want to pick the one (or combination) that's actually a bottleneck
  - No sense in optimizing for "memory usage" if you have a TB of memory and a single disk
  - Can do combinations by doing a weighted sum: e.g., 10 * Memory + 50 * Disk I/Os

"Cost"

- We will focus on disk for simplicity:
  - Number of I/Os ?
    - Not sufficient
    - Number of seeks matters a lot… why ?
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for *b* block transfers plus *S* seeks

    $b * t_T + S * t_S$
  - Measured in *seconds*
- Real systems do take CPU cost into account

"Cost" Example

- t_S = 10 ms (seek time)
- t_T = ?
  - Typical block size = 4kB
  - Say transfer rate = 200MB/s → 200kB/ms → 0.02ms per 4kB
- If a plan makes 100 seeks, and transfer 100 blocks:
  - Cost = 100 * 10 + 0.02 * 100 = 1002ms
- If a plan makes 1 seek, and transfer 5000 blocks:
  - Cost = 10 + 0.02 * 5000 = 110ms
- Transfer rates keep going up (through better hardware and parallelization), but seek times are constant
  - The gap keeps increasing

Next…

- For each relational operation, we will discuss different techniques for doing them
  - The basic technique usually straightforward, adaptations more complex
- For each technique, we will try to figure out roughly the number of seeks and I/Os
- Try to focus on the abstract principles involved, and not the details
- Very similar techniques used in data processing in other systems like Apache Spark, Hadoop, Python Pandas, etc.

CMSC424: Database Design

Module: Query Processing

Selection Operation

Instructor: Amol Deshpande
amol@umd.edu

Selections

- Book Chapters
  - 12.3
- Key topics:
  - Different ways to do a ”selection” operation (“where” clause) based on the properties of the predicates and the availability of indexes

Selection Operation

- select * from person where SSN = “123”
- Option 1: <u>Sequential Scan</u>
  - Read the relation start to end and look for “123”
    - Can always be used (not true for the other options)
  - Cost ?
    - *Let b_r = Number of relation blocks*
    - Then:
      - 1 seek and b_r block transfers
    - So:
      - $t_S + b_r * t_T$ *sec*
    - *Improvements:*
      - *If SSN is a key, then can stop when found*
        - *So on average, $b_r/2$ blocks accessed*

Selection Operation

- select * from person where SSN = "123"
- Option 2 : <u>Use Index</u>
  - Pre-condition:
    - *An appropriate index must exist*
  - Use the index
    - Find the first leaf page that contains the search key
    - Retrieve all the tuples that match by following the pointers
      - If primary index, the relation is sorted by the search key
        - Go to the relation and read blocks sequentially
      - If secondary index, must follow all pointers using the index

Selection w/ B+-Tree Indexes

	cost of finding the first leaf	cost of retrieving the tuples
primary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
primary index, not a key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S) + (b - 1) * t_T$ *Note: primary == sorted* *b = number of pages that contain the matches*
secondary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
secondary index, not a key, equality	$h_i * (t_T + t_S)$	$n * (t_T + t_S)$ *n = number of records that match* This can be bad

h_i = *height of the index*

Selection Operation

- Selections involving ranges
  - *select * from accounts where balance > 100000*
  - *select * from matches where matchdate between '10/20/06' and '10/30/06'*
  - Option 1: Sequential scan
  - Option 2: Using an appropriate index
    - Can't use hash indexes for this purpose
    - Cost formulas:
      - Range queries == "equality" on "non-key" attributes
      - So rows 3 and 5 in the preceding page

Selection Operation

- Complex selections
  - Conjunctive: *select * from accounts where balance > 100000* and *SSN = "123"*
  - Disjunctive: *select * from accounts where balance > 100000* or *SSN = "123"*
  - Option 1: Sequential scan
  - Option 2 *(Conjunctive only)*: Using an appropriate index *on one of the conditions*
    - E.g. Use SSN index to evaluate SSN = "123". Apply the second condtion to the tuples that match
    - Or do the other way around (if index on balance exists)
    - Which is better ?
  - Option 3 *(Conjunctive only)* : Choose a multi-key index
    - Not commonly available

Selection Operation

- Complex selections
  - <u>*Conjunctive*</u>*: select * from accounts where balance > 100000 and SSN = "123"*
  - <u>Disjunctive</u>: *select * from accounts where balance > 100000 or SSN = "123"*
  - Option 4: Conjunction or disjunction of *record identifiers*
    - Use indexes to find all RIDs that match each of the conditions
    - Do an intersection (for conjunction) or a union (for disjunction)
    - Sort the records and fetch them in one shot
    - Called "Index-ANDing" or "Index-ORing"
  - Heavily used in commercial systems

CMSC424: Database Design

Module: <u>Query Processing</u>

Joins

Instructor: Amol Deshpande
amol@umd.edu

Joins

- Book Chapters
  - 12.5.1, 12.5.2, 12.5.3, 12.5.5
- Key topics:
  - Simplest way to do a join as a nested for loop
  - Block Nested Loops Joins
  - Using "indexes" for more efficient joins
  - Hash Joins
  - Sort-merge Joins

Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *select * from R, S where |R.a – S.a | < 0.5*
  - *Not an "equi-join"*
- *Goal: For each tuple r in R, find all "matching" tuples in S (or vice versa)*
- Simplest Algorithm ("nested loops" join)

  for each tuple r in R

      for each tuple s in S

          check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Complexity too high– also not disk efficient
  - e.g., imagine if |R| and |S| both in millions of tuples

Block Nested-loops Join

- Simple modification to the basic "nested-loops join" that is disk efficient
- Read a chunk of blocks of R from disk at a time; go through S for each chunk

  for each k blocks of R

  for each block B_s of S

  for each tuple r in those k blocks of R

  for each tuple s in Bs

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Cost?
  - Blocks Read of R: |Br| (every block read exactly once)
  - Blocks Read of S: |Bs| * |Br|/k (every block of S read |Br|/k times)
  - Seeks: 2 * |Br|/k
- Choose *k* to be as large as possible (but can't be more than M)
- However: We are still comparing every R tuple with every S tuple → high CPU cost

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join"*
- Let's say there is an "index" on S.a

  for each tuple r in R

  use the index to find S tuples with S.a = r.a

- Blocks read of R: Br
- Blocks read of S: depends on the index (see previous formulas)
- Seeks: Br for R, but seeks for S depend on the index

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about $|R.a - S.a| < 5$?
- Great for queries with joins and selections

  *select **

  from accounts, customers

  where accounts.customer-SSN = customers.customer-SSN and

  accounts.acct-number = "A-101"
- Only need to access one SSN from the other relation

Hash Join

- Case 1: Smaller relation *(S)* fits in memory

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a
- Cost: $b_r + b_s$ transfers, 2 seeks
- Why good ?
  - CPU cost is much better (even though we technically don't care about it in our cost function, in reality, it matters a lot)
  - Performs much better than nested-loops join when *S* doesn't fit in memory (next)

Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, …, Rk
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement:
    - Each S partition fits in memory
    - Requires SQRT(Bs) Memory
      - Can do "recursive" partitioning if not enough memory – rarely the case today

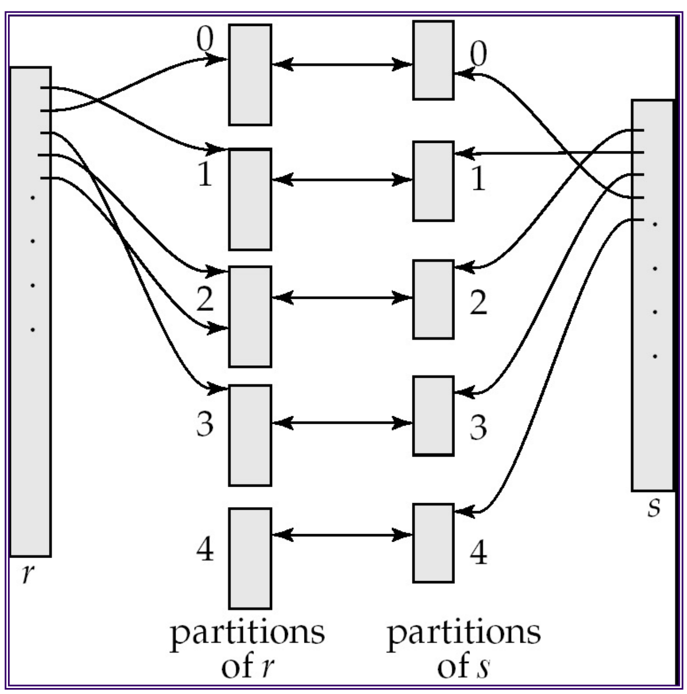
Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 2:
  - Read S1 into memory, and bulid a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

Hash Join

- <u>Case 2: Smaller relation *(S)* doesn't fit in memory</u>
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h_1(a)$
- Phase 2:
  - Read S_i into memory, and bulid a hash index on it (S_i fits in memory)
  - Read R_i block by block, and use the hash index to find matches.
- Cost ?
  - $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
    - Where b_b is the size of each output buffer
  - Much better than Nested-loops join under the same conditions

Hash Join



Hash Join: Issues

- How to guarantee that the partitions of *S* all fit in memory ?
  - Say S = 10000 blocks, Memory = M = 100 blocks
  - Use a hash function that hashes to 100 different values ?
    - Eg. *h1(a) = a % 100 ?*
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to *100*f* different values
    - *f* is called fudge factor, typically around 1.2
    - So we may consider *h1(a) = a % 120.*
    - This is okay IF *a* is uniformly distributed
  - What if the hash function turns out to be bad ?
    - Repartition using a different hash function (at run time)

CMSC424: Database Design

Module: Query Processing

Aggregates

Instructor: Amol Deshpande
amol@umd.edu

Group By and Aggregation

select a, count(b)
from R
group by a;

- Hash-based algorithm
- Steps:
  - Create a hash table on *a,* and keep the *count(b) so far*
  - Read *R* tuples one by one
  - For a new *R tuple, "r"*
    - Check if *r.a* exists in the hash table
    - If yes, increment the count
    - If not, insert a new value

Group By and Aggregation

select a, count(b)
from R
group by a;

- Sort-based algorithm
- Steps:
  - Sort *R* on *a*
  - Now all tuples in a single group are contigous
  - Read tuples of *R (sorted)* one by one and compute the aggregates

Group By and Aggregation

select a, AGGR(b) from R group by a;

- sum(), count(), min(), max(): only need to maintain one value per group
  - Called “distributive”
- average() : need to maintain the “sum” and “count” per group
  - Called “algebraic”
- stddev(): algebraic, but need to maintain some more state
- median(): can do efficiently with sort, but need two passes (called “holistic”)
  - First to find the number of tuples in each group, and then to find the median tuple in each group
- count(distinct b): must do duplicate elimination before the count

CMSC424: Database Design

Module: Query Processing

Sorting and Merge Joins; Some Other Operators

Instructor: Amol Deshpande
amol@umd.edu

Sorting; Merge Joins

- Book Chapters
  - 12.4, 12.5.4, 12.6
- Key topics:
  - How to sort when data doesn't fit in memory
  - Using sorting for joins
  - Duplicate elimination
  - Set operations
  - Outerjoins

Sorting

- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of the index
    - May be enough in many cases
  - But if relation not sorted, this leads to too many random accesses
- If relation small enough…
  - Read in memory, use quick sort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

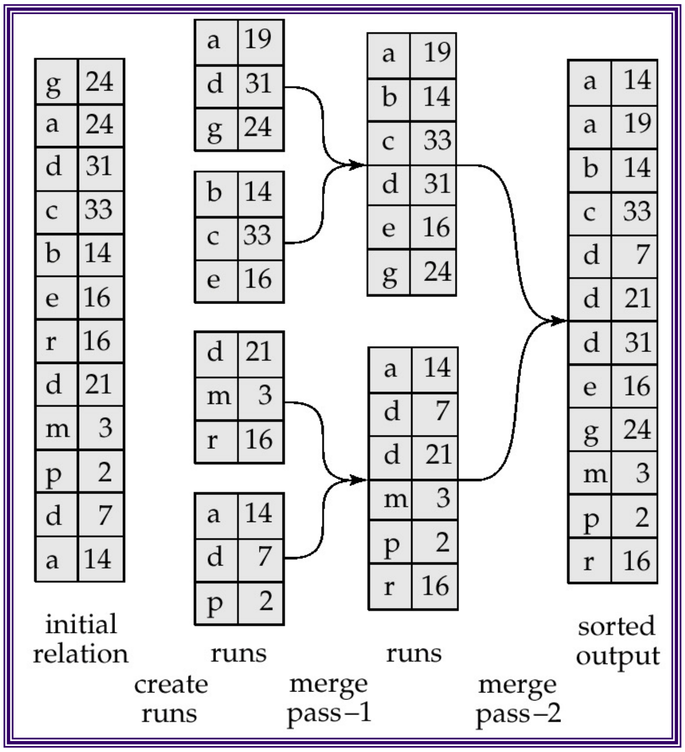
External sort-merge

- Divide and Conquer !!
- Let *M* denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk …
  - Say we have to do this "N" times
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*

External sort-merge

- Phase 1:
  - Create *sorted runs of size M* each
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
- What if *N* > *M ?*
  - Do it recursively
  - Not expected to happen
  - If *M* = 1000 blocks = 4MB (assuming blocks of 4KB each)
    - Can sort: 4000MB = 4GB of data

Example: External Sorting Using Sort-Merge



External Merge Sort (Cont.)

- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Disk accesses for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

  Thus total number of disk accesses for external sorting:

  $$b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$

- What about seeks?
  - More complicated

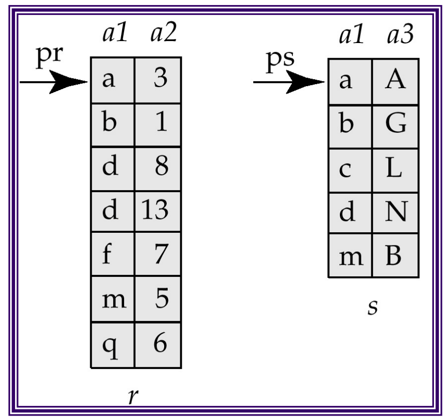
Merge-Join (Sort-merge join)

- Pre-condition:
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this algorithms
- Called “sort-merge join” sometimes

*select **
from r, s
where r.a1 = s.a1

Step:

1. Compare the tuples at pr and ps
2. Move pointers down the list
- Depending on the join condition
3. Repeat



Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted ?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g. duplicate elimination

Joins: Summary

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
- Hash joins – only for equi-joins
  - Join algorithm of choice when the relations are large
- Hybrid hash join
  - An optimization on hash join that is always implemented
- Sort-merge join
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

Duplicate Elimination

select distinct a
from R ;

- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation *R*
  - Read tuples of *R* in sorted order
  - prev = null;
  - for each tuple *r* in *R (sorted)*
    - if *r != prev* then
      - Output *r*
      - *prev* = *r*
    - else
      - Skip *r*

Set operations

*(select * from R) union (select * from S) ;*
*(select * from R) intersect (select * from S) ;*
*(select * from R) union all (select * from S) ;*
*(select * from R) intersect all (select * from S) ;*

- Remember the rules about duplicates
- "union all": just append the tuples of *R and S*
- *"*union": append the tuples of *R and S, and do duplicate elimination*
- *"intersection"*: similar to joins
  - Find tuples of R and S that are identical on all attributes
  - Can use *hash-based or sort-based algorithm*

Outer Joins

- Say: R FULL OUTER JOIN S, on R.a = S.a
- Need to keep track of which tuples of R "do not match" any tuples from S, and vice versa

- Hash-based, with a hash index on S:
  - For a tuple r in R, if the probe returns NULL, output r padded with NULLs
  - For each tuple *s* in *S*, maintain a Boolean variable (in the hash table) to track whether *s* was returned for any probes
  - At the end, go through the hash table, and look for *S* tuples that did not match anything

- Merge join can also be adapted in a similar way

CMSC424: Database Design

Module: Query Processing

Putting it All Together

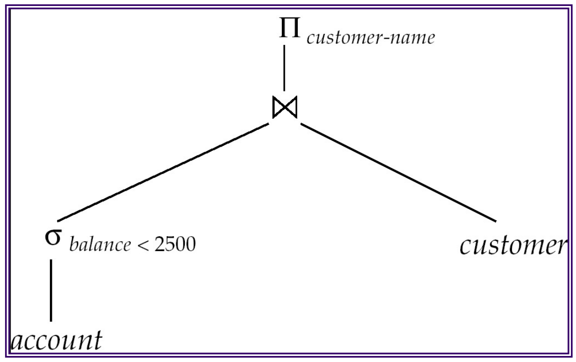
Instructor: Amol Deshpande
amol@cs.umd.edu

Putting it all together

- Book Chapters
  - 12.7
- Key topics:
  - How to put it all together in a query plan
  - Pipelining vs Materialization
  - Iterator Interface

Evaluation of Expressions

select customer-name
from account a, customer c
where a.SSN = c.SSN and
a.balance < 2500



- Two options:
  - Materialization
  - Pipelining

Evaluation of Expressions

- Materialization
  - Evaluate each expression separately
    - Store its result on disk in *temporary relations*
    - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join
  - Harder to reason about

Materialization

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- Evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires higher amount of memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- A "blocking" operation: An operation that has to consume entire input before it starts producing output tuples

Pipelining

- Need operators that generate output tuples while receiving tuples from their inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no. Need to wait for the entire input before producing output
    - However, there are tricks you can play here
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: see duplicate elimination

Pipelining: Demand-driven

- Iterator Interface
  - Each operator implements:
    - *init(): Initialize the state (sometimes called open())*
    - *get_next(): get the next tuple from the operator*
    - *close(): Finish and clean up*
  - Sequential Scan:
    - *init():* open the file
    - *get_next():* get the next tuple from file
    - *close():* close the file
- Execute by repeatadly calling *get_next()* at the root
  - root calls get_next() on its children, the children call get_next() on their children etc…
- The operators need to maintain internal state so they know what to do when the parent calls get_next()

Hash-Join Iterator Interface

- open():
  - Call open() on the left and the right children
  - Decide if partitioning is needed (if size of smaller relation > allotted memory)
  - Create a hash table
- get_next(): (((assuming no partitioning needed)))
  - First call:
    - Get all tuples from the right child one by one (using get_next()), and insert them into the hash table
    - Read the first tuple from the left child (using get_next())
  - All calls:
    - Probe into the hash table using the “current” tuple from the left child
      - Read a new tuple from left child if needed
    - Return exactly “one result”
      - Must keep track if more results need to be returned for that tuple

Hash-Join Iterator Interface

- close():
  - Call close() on the left and the right children
  - Delete the hash table, other intermediate state etc…
- get_next(): (((partitioning needed)))
  - First call:
    - Get all tuples from both children and create the partitions on disk
    - Read the first partition for the right child and populate the hash table
    - Read the first tuple from the left child from appropriate partition
  - All calls:
    - Once a partition is finished, clear the hash table, read in a new partition from the right child, and re-populate the hash table
  - Not that much more complicated
- Take a look at the postgreSQL codebase

Pipelining (Cont.)

- In produce-driven or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples

Recap: Query Processing

- Many, many ways to implement the relational operations
  - Numerous more used in practice
  - Especially in data warehouses which handles TBs (even PBs) of data
- However, consider how complex SQL is and how much you can do
  - Compared to that, this isn't much
- Most of it is very nicely modular
  - Especially through use of the *iterator()* interface
  - Can plug in new operators quite easily
  - PostgreSQL query processing codebase very easy to read and modify
- Having so many operators does complicate the codebase and the query optimizer though
  - But needed for performance

CMSC424: Database Design

Module: Query Processing

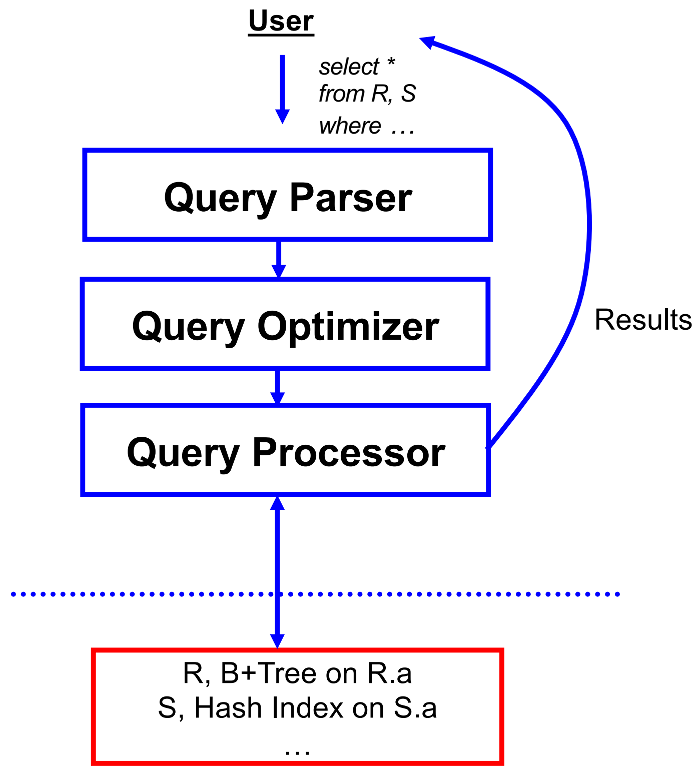
Query Optimization

Instructor: Amol Deshpande
amol@umd.edu

Query Optimization: Overview

- Key topics:
  - Why query optimization is so important?
  - Key steps in query optimization
  - High-level concepts

Getting Deeper into Query Processing



Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?
…

Read the data from the files
Do the query processing
joins, selections, aggregates
…

Query Optimization

- Why ?
  - Many different ways of executing a given query
  - Huge differences in cost
- Example:
  - select * from person where ssn = “123”
  - Size of *person* = 1GB
  - Sequential Scan:
    - Takes 1GB / (20MB/s) = 50s
  - Use an index on SSN (assuming one exists):
    - Approx 4 Random I/Os = 40ms

Query Optimization

- Many choices
  - Using indexes or not, which join method (hash, vs merge, vs NL)
  - What join order ?
    - Given a join query on R, S, T, should I join R with S first, or S with T first ?
- This is an optimization problem
  - Similar to say *traveling salesman problem*
  - Number of different choices is very very large
  - Step 1: Figuring out the *solution space*
  - Step 2: Finding algorithms/heuristics to search through the solution space

Query Optimization: Goal

- Find the best (or a good enough) execution plan
- Execution plans = Evaluation expressions annotated with the methods used

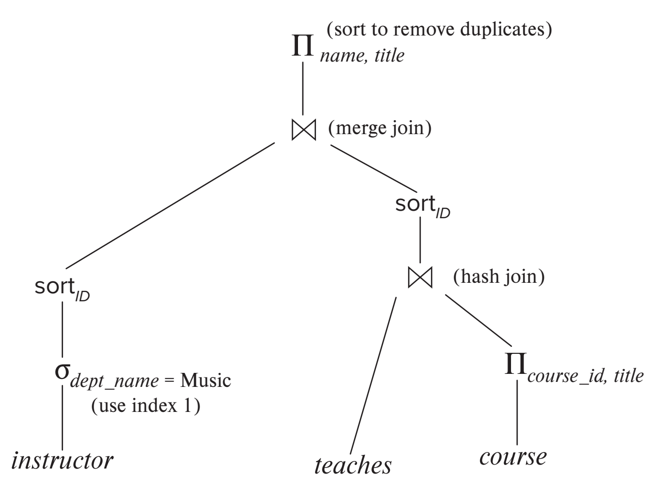


Figure 16.2 An evaluation plan.

Query Optimization

- Steps:
  - Generate all possible execution plans for the query
  - Figure out the cost for each of them
  - Choose the best
- Not done exactly as listed above
  - Too many different execution plans for that
  - Typically interleave all of these into a single efficient search algorithm

Equivalence of Expressions

- Equivalent relational expressions
  - Drawn as a tree
  - List the operations and the order

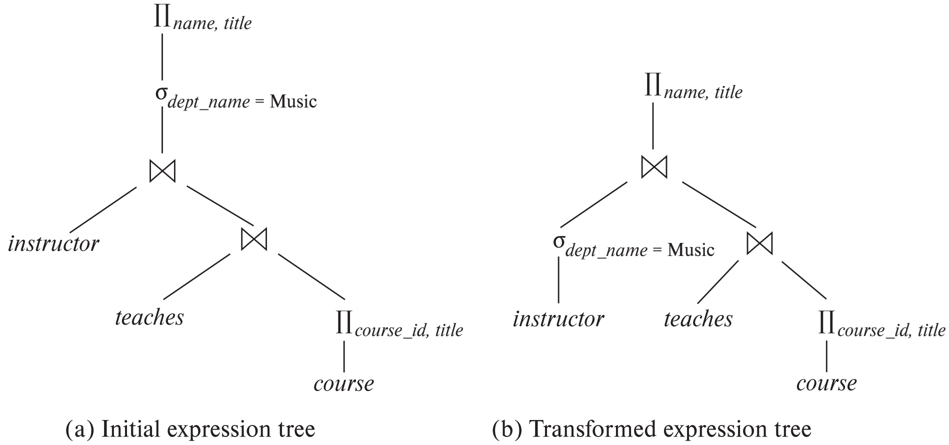


Figure 16.1 Equivalent expressions.

Equivalence of Expressions

- Two relational expressions equivalent iff:
  - Their result is identical on all legal databases
- Equivalence rules:
  - Allow replacing one expression with another
- Examples:

1. $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$

2. Selections are commutative

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

Equivalence Rules

- Examples:

3. $\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$

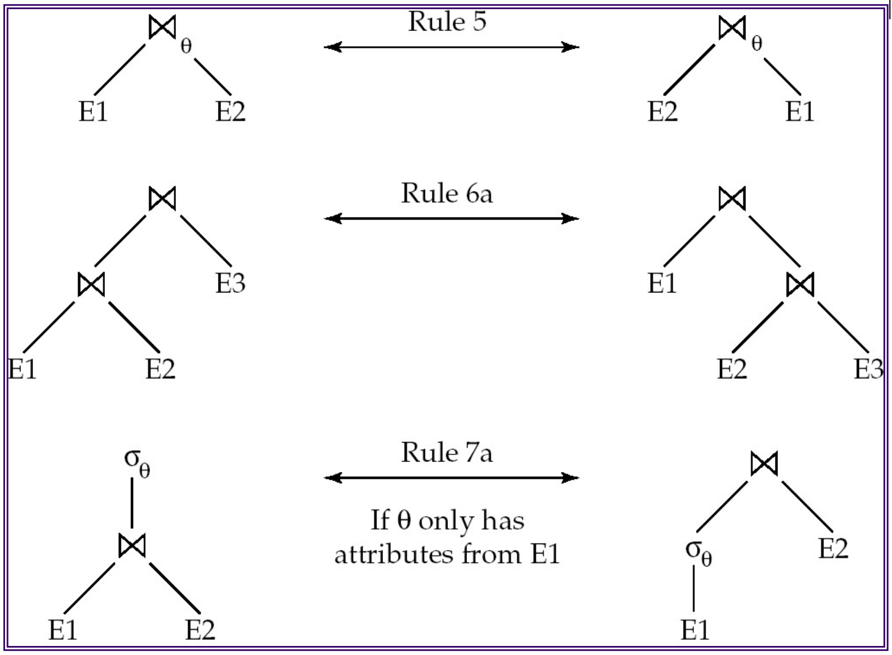
5. $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

7(a). If *θ_0 only involves attributes from E_1*

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

- And so on…
  - Many rules of this type

Pictorial Depiction



Equivalence of Expressions

- The rules give us a way to enumerate all equivalent expressions
  - Note that the expressions don't contain physical access methods, join methods etc…
- Simple Algorithm:
  - Start with the original expression
  - Apply all possible applicable rules to get a new set of expressions
  - Repeat with this new set of expressions
  - Till no new expressions are generated

Equivalence of Expressions

- Works, but is not feasible
- Consider a simple case:
  - *R1 ⋈ (R2 ⋈ (R3 ⋈ (… ⋈ Rn)))….)*
- Just join commutativity and associativity will give us:
  - At least:
    - n^2 * 2^n
  - At worst:
    - n! * 2^n
- Typically the process of enumeration is combined with the search process

Evaluation Plans

- We still need to choose the join methods etc..
  - Option 1: Choose for each operation separately
    - Usually okay, but sometimes the operators interact
    - Consider joining three relations on the same attribute:
      - $R1 \bowtie_a (R2 \bowtie_a R3)$
    - Best option for R2 join R3 might be hash-join
      - But if *R1* is sorted on *a,* then *sort-merge join* is preferable
      - Because it produces the result in sorted order by *a*
- Also, we need to decide whether to use pipelining or materialization
- Such issues are typically taken into account when doing the optimization

Query Optimization

- Steps:
  - Generate all possible execution plans for the query
    - First generate all equivalent expressions
    - Then consider all annotations for the operations
  - Figure out the cost for each of them
    - Compute cost for each operation
      - Using the formulas discussed before
      - One problem: How do we know the number of result tuples for, say, $\sigma_{balance<2500}(account)$
    - Add them !
  - Choose the best

Cost estimation

- Computing operator costs requires information like:
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - RAID ?? Which one ?
    - Read/write costs are quite different
  - How many tuples match a predicate like “age > 40” ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
    - E.g. (R JOIN S) is input to another join operation – need to know if it fits in memory
  - And so on…

Cost estimation

- Some information is static and is maintained in the metadata
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - RAID ?? Which one ?
    - Read/write costs are quite different
- Typically kept in some tables in the database
  - "all_tab_columns" in Oracle
- Most systems have commands for updating them

Cost estimation

- However, others need to be estimated somehow
  - How many tuples match a predicate like "age > 40" ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
- The problem variously called:
  - "intermediate result size estimation"
  - "selectivity estimation"
- Very important to estimate reasonably well
  - e.g. consider "select * from R where zipcode = 20742"
  - We estimate that there are 10 matches, and choose to use a secondary index (remember: random I/Os)
  - Turns out there are 10000 matches
  - Using a secondary index very bad idea
  - Optimizer also often choose Nested-loop joins if one relation very small… underestimation can result in very bad

Selectivity Estimation

- Basic idea:
  - Maintain some information about the tables
    - More information → more accurate estimation
    - More information → higher storage cost, higher update cost
  - Make uniformity and randomness assumptions to fill in the gaps
- Example:
  - For a relation “people”, we keep:
    - Total number of tuples = 100,000
    - Distinct “zipcode” values that appear in it = 100
  - Given a query: “zipcode = 20742”
    - We estimated the number of matching tuples as: 100,000/100 = 1000
  - What if I wanted more accurate information ?
    - Keep better statistics/summaries…

Examples

- Consider a range query: $x < R.a < y$
  - *Let Max(a, R) = maximum value of a in R*
  - *Let Min(a, R) = minimum value of a in R*
  - *Then: fraction of tuples that satisfy = (y – x) / (Max – Min)*
    - *Assuming all tuples are distributed uniformly and randomly*
    - *If y > Max or x < Min → adjust accordingly*
  - Better summary statistics (like histograms) can help with refining these estimates

Example: Joins

- R JOIN S: R.a = S.a
  - |R| = 10,000; |S| = 5000
- CASE 1: *a* is key for S
  - *Each tuple of R joins with exactly one tuple of S*
  - So: |R JOIN S| = |R| = 10,000
  - Assumption: Referential integrity holds
  - What if there is a selection on R or S
    - Adjust accordingly
    - Say: S.b = 100, with selectivity 0.1
    - THEN: |R JOIN S| = |R| * 0.1 = 100
- CASE 2: *a* is key for R
  - Similar

Joins

- R JOIN S: R.a = S.a
  - |R| = 10,000; |S| = 5000
- CASE 3: *a* is not a key for either
  - Reason with the distributions on *a*
  - Say: the domain of *a: V(A, R)* = 100 (the number of distinct values *a* can take)
  - THEN, *assuming uniformity*
    - For each value of *a*
      - We have 10,000/100 = 100 tuples of R with that value of a
      - We have 5000/100 = 50 tuples of S with that value of a
      - All of these will join with each other, and produce 100 *50 = 5000
    - So total number of results in the join:
      - 5000 * 100 = 500000
  - We can improve the accuracy if we know the distributions on *a* better
    - Say using a histogram

Query Optimization

- Steps:
  - Generate all possible execution plans for the query
    - First generate all equivalent expressions
    - Then consider all annotations for the operations
  - Figure out the cost for each of them
    - Compute cost for each operation
      - Using the formulas discussed before
      - One problem: How do we know the number of result tuples for, say, $\sigma_{balance<2500}(account)$
    - Add them !
  - Choose the best

Optimization Algorithms

- Two types:
  - Exhaustive: That attempt to find the best plan
  - Heuristical: That are simpler, but are not guaranteed to find the optimal plan

- Consider a simple case
  - Join of the relations *R1, …, Rn*
  - No selections, no projections

- Still very large plan space

Searching for the best plan

- <u>Option 1:</u>
  - Enumerate all equivalent expressions for the original query expression
    - Using the rules outlined earlier
  - Estimate cost for each and choose the lowest
- Too expensive !
  - Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
  - There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!

Searching for the best plan

- <u>Option 2:</u>
  - Dynamic programming
    - There is too much commonality between the plans
    - Also, costs are additive
      - Caveat: Sort orders (also called "interesting orders")
  - Reduces the cost down to O(n3^n) or O(n2^n) in most cases
    - Interesting orders increase this a little bit
  - Considered acceptable
    - Typically n < 10.
  - Switch to heuristic if not acceptable

Heuristic Optimization

- Dynamic programming is expensive
- Use *heuristics* to reduce the number of choices
- Typically rule-based:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Summary

- Integral component of query processing
  - Why ?
- One of the most complex pieces of code in a database system
- Active area of research
  - E.g. XML Query Optimization ?
  - What if you don't know anything about the statistics
  - Better statistics
  - Etc …

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Overview; Parallel/Distributed Architectures

Instructor: Amol Deshpande
amol@umd.edu

NoSQL and Big Data Systems: Motivation

- Book Chapters
  - 10.1, 10.2 **(7TH EDITION)**
- Key topics:
  - Big data motivating scenarios
  - Why systems so far (relational databases, data warehouses, parallel databases) don't work

RDBMS: Application Scenarios

- Online Transaction Processing (OLTP)
  - E-commerce, Airline Reservations, Class registrations, etc.
  - Simple queries (get all orders for a customer)
  - Many updates (inserts, updates, deletes)
  - Need ACID properties (consistency, etc.)
- Online Analytical Processing (OLAP)
  - Decision-support, data mining, ML (today), etc.
  - Huge volumes of data, but not updated
  - Complex, but read-only queries (many joins, group-by's)

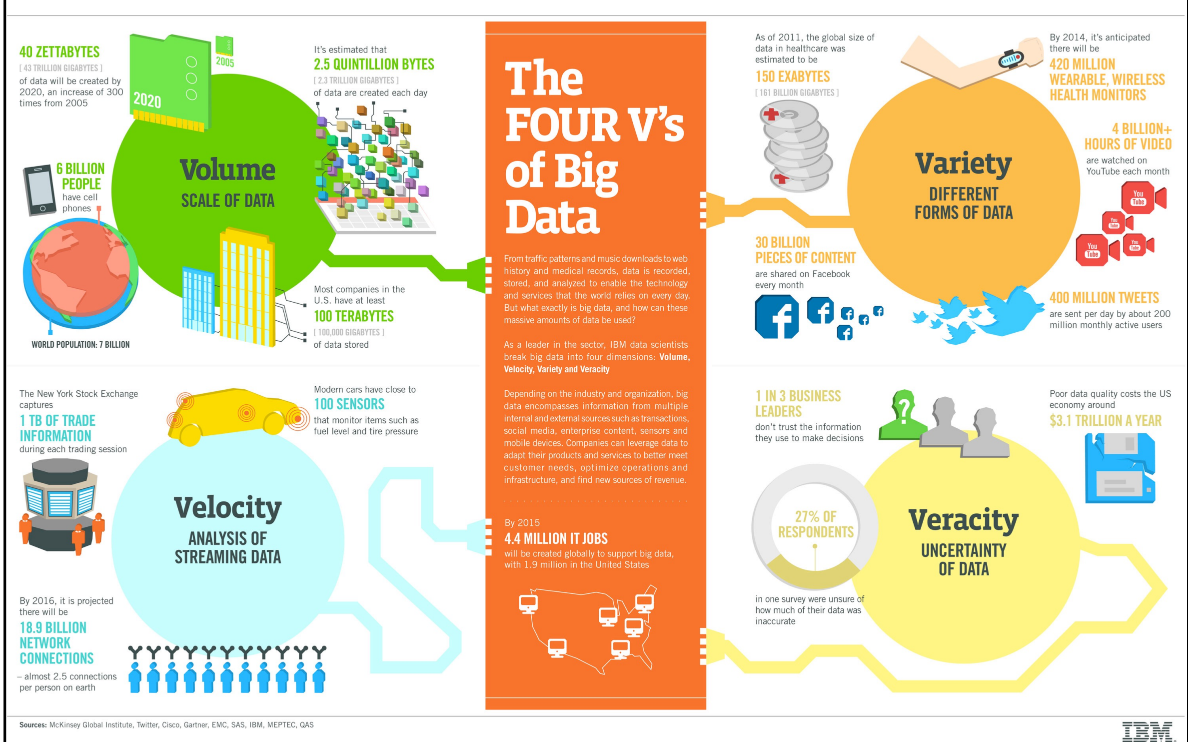
RDBMS Evolution

- Original database systems aimed to support both use cases
- Slowly, specialized systems were built, starting late 80's-early 90's, especially for decision support (Data Warehouses)
- Today, different RDBMSs systems for different use cases, e.g.,:
  - VoltDB for OLTP – fully in-memory, very fast transactions, but no complex queries
  - Teradata, Aster Data, Snowflake, AWS Redshift – handle PBs of data, but batch updates only – many indexes and summary structures (cubes) for queries – typically "parallel" (i.e., use many machines)
- Fundamental and wide differences in the technology
- But both still support SQL as the primary interface (with visualizations, exploration, and other tools on top)

NoSQL + Big Data Systems: Motivation

- Very large volumes of data being collected
  - Driven by growth of web, social media, and more recently internet-of-things
  - Web logs were an early source of data
    - Analytics on web logs has great value for advertisements, web site structuring, what posts to show to a user, etc

- Big Data: differentiated from data handled by earlier generation databases
  - **Volume**: much larger amounts of data stored
  - **Velocity**: much higher rates of insertions
  - **Variety**: many types of data, beyond relational data



Some motivating scenarios

- Deciding what to show a user in a social network, or news aggregator
- Advertising on the Web or Mobile
- Analyzing user behavior on web sites to optimize or increase engagement
- Analyzing large numbers of images and building search indexes on them
- Text analytics for topic modelling, summarization, …
- Internet of things…
- And many many others…

Two Primary Use Cases

- OLTP-like
  - Simple queries, but lots of updates
  - Need to support distributed users
  - Need to support non-relational data (e.g., graphs, JSONs)
  - Need to scale fast (10 users to 10s of Millions of Users)
  - Need to work well in 3-tier Web Apps
  - Need to support fast schema changes
- OLAP-like
  - Complex analysis on large volumes of data
  - Often no "real-time" component, and no updates
  - Mostly non-relational data (images, webpages, text, etc)
  - Tasks often procedural in nature (analyse webpages for searching, data cleaning, ML)

Why (Parallel) Databases Don't Work

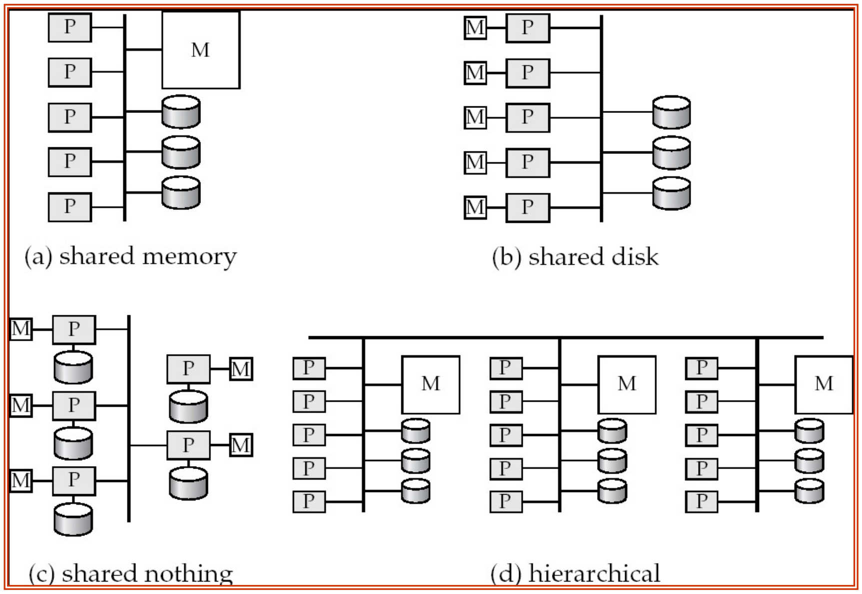
- The data is often not relational in nature
  - E.g., images, text, graphs
- The analysis/queries are not relational in nature
  - E.g., Image Analysis, Text Analytics, Natural Language Processing, Web Analytics, Social Network Analysis, Machine Learning, etc.
  - Databases don't really have constructs to support this
    - User-defined functions can help to some extent
  - Need to interleave relational-like operations with non-relational (e.g., data cleaning, etc.)
  - Domain users are more used to procedural languages
- The operations are often one-time
  - Only need to analyse images once in a while to create a "deep learning" model
  - Databases are really better suited for repeated analysis of the data
- Much of the analysis not time-sensitive
- Parallel databases too expensive given the data volumes
  - Were designed for large enterprises, with typically big budgets

Parallel and Distributed Architectures

- Ability to scale "up" a computer is limited ➔ Use many computers together
  - Called cluster or network of computers (and today, just a "data center")
- Also need to "meet" where the users are
  - To minimize interactive latencies (e.g., social networks)
- Has made parallel and distributed architectures very common today

Parallel Architectures

- Shared-nothing vs. shared-memory vs. shared-disk



Parallel Architectures

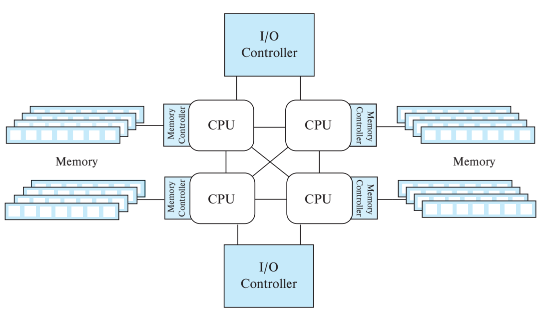


Figure 20.6 Architecture of a modern shared-memory system.

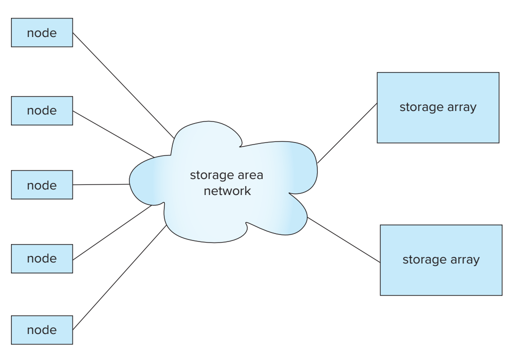


Figure 20.8 Storage-area network.

Parallel Architectures

	Shared Memory	Shared Disk	Shared Nothing
Communication between processors	Extremely fast	Disk interconnect is very fast	Over a LAN, so slowest
Scalability ?	Not beyond 32 or 64 or so (memory bus is the bottleneck)	Not very scalable (disk interconnect is the bottleneck)	Very very scalable
Notes	Cache-coherency an issue	Transactions complicated; natural fault-tolerance.	Distributed transactions are complicated (deadlock detection etc);
Main use	Low degrees of parallelism	Not used very often	Everywhere

Parallel Systems

- A **coarse-grain parallel** machine → a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine → thousands of smaller processors.
- We see a variety of mixes of these today, especially with the rise of multi-core machines

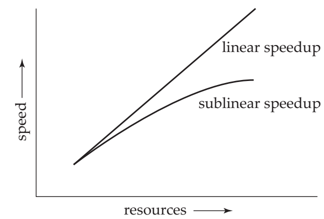
- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted

Speed-Up and Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is *N*-times larger.
  - Measured by:

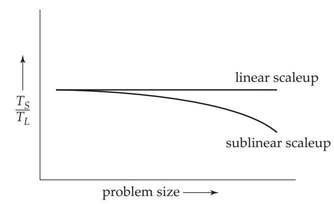
    speedup = small system elapsed time / large system elapsed time

    $$speedup = \frac{\text{small system elapsed time}}{\text{large system elapsed time}}$$
  - Speedup is **linear** if equation equals N.



- **Scaleup**: increase the size of both the problem and the system
  - *N*-times larger system used to perform *N*-times larger job
  - Measured by:

    $$scaleup = \frac{\text{small system small problem elapsed time}}{\text{big system big problem elapsed time}}$$
  - Scale up is **linear** if equation equals 1.



Factors Limiting Speedup and Scaleup

- **Sequential computation:** Some parts may not be parallalelizable
  - **Amdahl's Law**: If "p" is the fraction of the task that can be parallelized, then the best speedup you can get is: $\frac{1}{(1-p)+(p/n)}$.
  - If "p" is 0.9, the best speedup is 10
- **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, thus spending time waiting on other processes, rather than performing useful work.
- **Skew**: Increasing the degree of parallelism increases the variance in service times of parallely executing tasks. Overall execution time determined by **slowest** of parallely executing tasks.

What about "Distributed" Systems?

- Over a wide area network
- Typically not done for *performance reasons*
  - For that, use a parallel system
- Done because of necessity
  - Imagine a large corporation with offices all over the world
  - Or users distributed across the globe
  - Also, for redundancy and for disaster recovery reasons

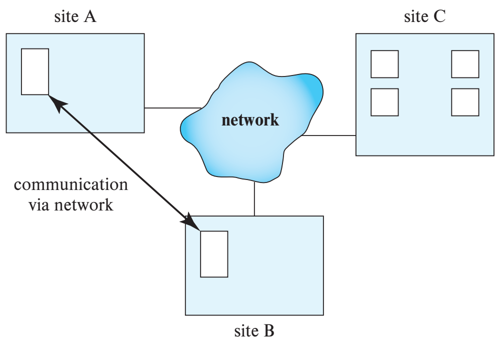
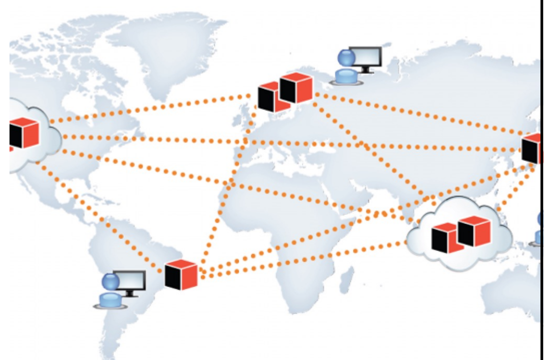


Figure 20.9 A distributed system.



CMSC424: Database Design

Module: NoSQL; Big Data Systems

Data Replication; Sharding; Failures

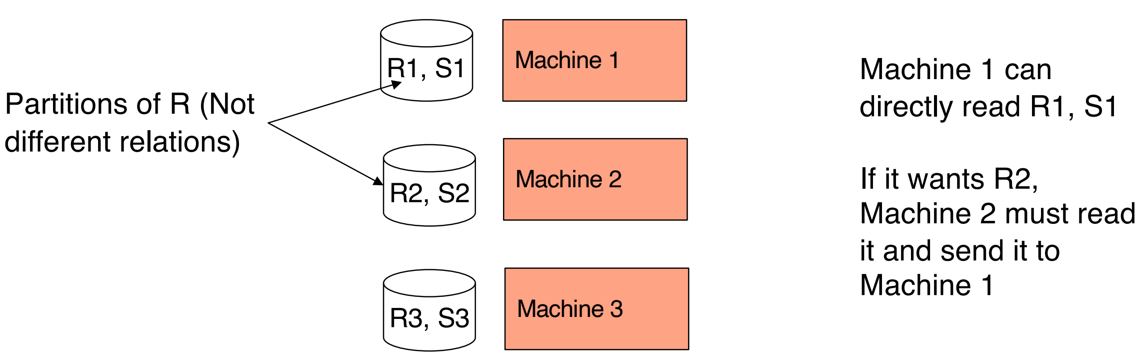
Instructor: Amol Deshpande
amol@umd.edu

Parallel or Distributed Systems

- Key Questions from Data Management Perspective:
  - How to partition (or "shard") data across a collection of storage devices/machines
  - How to execute an "operation" across a group of computers
    - In different configurations (shared-memory vs shared-disk vs shared-nothing vs NUMA)
    - Trade-offs and bottlenecks can be vastly different
  - How to execute an "update" across a group of computers
    - Need to ensure consistency
  - How to deal with "failures"

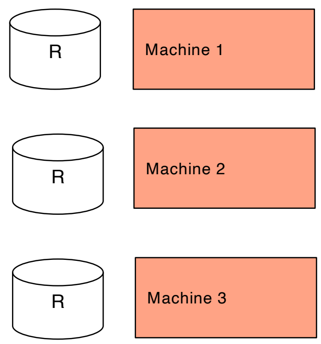
Data Partitioning

- Partition a relation or a dataset across machines
  - Typically through "hashing"
- Advantages:
  - **In-memory computation**: data fits in memory across machines
  - **Parallelism**: simple read/write queries can be distributed across machines
- Disadvantages:
  - **Complex queries:** require combining data across all partitions, especially "joins" are tricky



Data Replication

- A data item (file, relation, relation fragment, object, tuple) is **replicated** if it is stored redundantly in two or more sites
- Advantages:
  - **Availability**: failures can be handled through replicas
  - **Parallelism**: queries can be run on any replica
  - **Reduced data transfer**: queries can go to the "closest" replica
- Disadvantages:
  - **Increased cost of updates:** both computation as well as latency
  - **Increased complexity of concurrency control:** need to update all copies of a data item/tuple



Read queries can go to any machine

Write queries must go to "all" machines (if we want consistency)

e.g., what if Application 1 writes to Machine 1, and Application 2 sends its write to Machine 3
-- May result in an inconsistent state

Data Sharding + Replication

- Many data management systems today combine both
  - Partition a dataset/file/relation into smaller pieces and distributed it across machines
  - Replicate each of the pieces multiple times
- This may be done:
  - In a data center with very fast networks, or
  - In a wide-area setting with slower networks and higher latencies
- So need to worry about:
  - Efficient execution of complex queries
  - Consistency for updates
  - Recovery from failures

Failures

- Need to consider:
  - Disk failures: one of the disks (hard drives or SSDs) fails
    - Not uncommon with 10's of thousands of disks
  - Network failures: machines may not be able to talk to each other
  - Machine failure: a machine crashes during the execution of a query or a transaction
- Required guarantees:
  - Shouldn't lose any data if a disk fails
  - Consistency (when making updates) shouldn't be affected if one of the involved machines fails
    - Or if machines are not able to talk to each other
  - Shouldn't have to restart a complex analytics task entirely if one of the involved machines fails

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Overview (Cntd)

Instructor: Amol Deshpande
amol@umd.edu

Two Primary Use Cases

- OLTP-like
  - Simple queries, but lots of updates
  - Need to support distributed users
  - Need to support non-relational data (e.g., graphs, JSONs)
  - Need to scale fast (10 users to 10s of Millions of Users)
  - Need to work well in 3-tier Web Apps
  - Need to support fast schema changes

NoSQL Storage Systems: HDFS, Cassandra, MongoDB, Neo4j, AWS DynamoDB, and many many others

- OLAP-like
  - Complex analysis on large volumes of data
  - Often no "real-time" component, and no updates
  - Mostly non-relational data (images, webpages, text, etc)
  - Tasks often procedural in nature (analyse webpages for searching, data cleaning, ML)

Big data frameworks: Hadoop MapReduce, Flink, Spark, and many others

Examples of Systems

- Too much variety in the systems out there today
  - different types of data models supported
    - Files/Objects (HDFS, AWS S3), Document (MongoDB), Graph (Neo4j), Wide-table (Cassandra, DynamoDB), Multi-Model (Azure CosmosDB)
  - different types of query languages or frameworks or workloads
    - SQL (Snowflake, Redshift, …), MongoQL, Cassandra QL, DataFrames (Spark), MapReduce (Hadoop), TensorFlow for ML, …
  - different environmental assumptions
    - Distributed vs parallel, disks or in-memory only, single-machine or not, streaming or static, etc.
  - different performance focus and/or guarantees
    - e.g., consistency guarantees in a distributed setting differ quite a bit
- Many of these systems work with each
  - e.g., Spark can read data from most of the storage systems
  - Interoperability increasing a requirement

What We Will Cover

- Apache Spark
  - Current leader in big data (OLAP-style) frameworks
  - Supports many query/analysis models, including a light version of SQL
- MongoDB
  - Perhaps the most popular NoSQL system, uses a "document" (JSON) data model
  - Focus primarily on OLTP
  - Doesn't really support joins (some limited ability today) – have to do that in the app
- How to "Parallelize" Operations
  - Useful to understand how Spark and other systems actually work
  - Often times you have to build these in the application layer
  - The original MapReduce framework
    - Led to development of much work on large-scale data analysis (OLAP-style)
    - Basically a way to execute a group-by at scale on non-relational data
- Hadoop Distributed File System (briefly)
  - A key infrastructure piece, with no real alternative
  - Basic file system interface, with replication and redundancy built in for failures
- Quick overview of other NoSQL data models

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Apache Spark

Instructor: Amol Deshpande
amol@umd.edu

Apache Spark

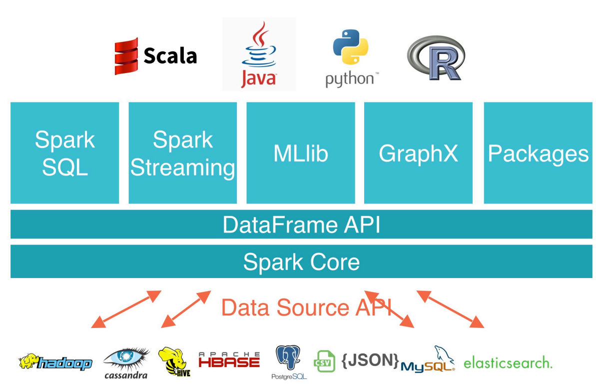
- Book Chapters
  - 10.4 **(7TH EDITION)** covers this topic, but Spark programming guide is a better resource
  - Assignments will refer to the programming guide

- Key topics:
  - A Resilient Distributed Dataset (RDD)
  - Operations on RDDs

585

Spark

- Open-source, distributed cluster computing framework
- Much better performance than Hadoop MapReduce through in-memory caching and pipelining
- Originally provided a low-level RDD-centric API, but today, most of the use is through the "Dataframes" (i.e., relations) API
  - Dataframes support relational operations like Joins, Aggregates, etc.

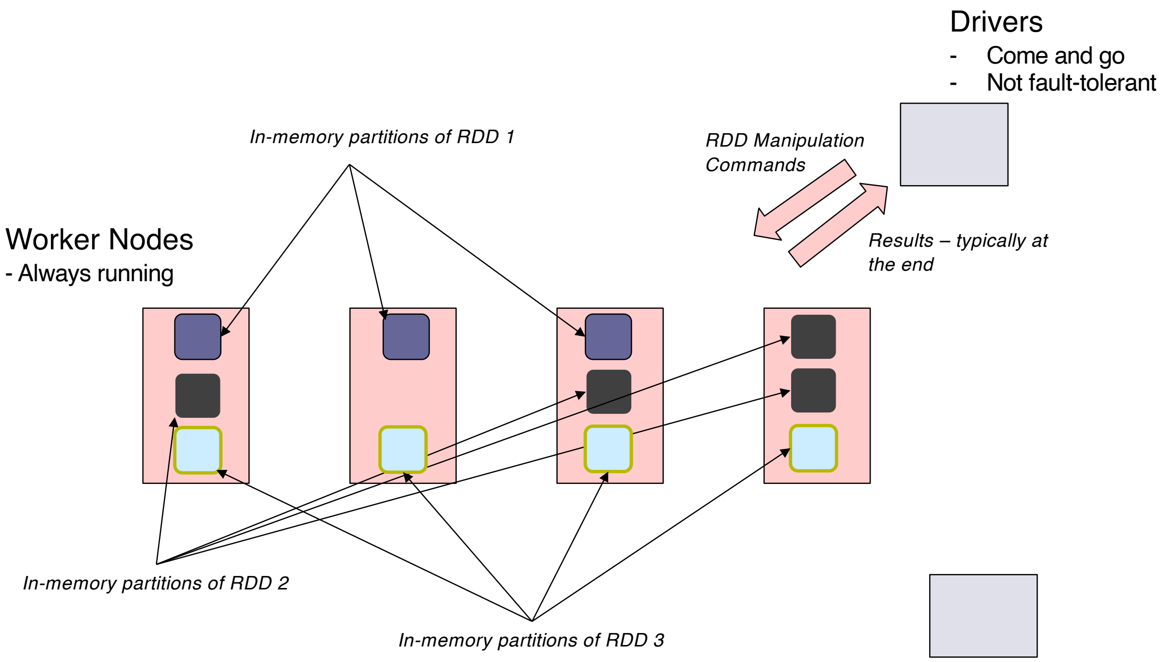


databricks

586

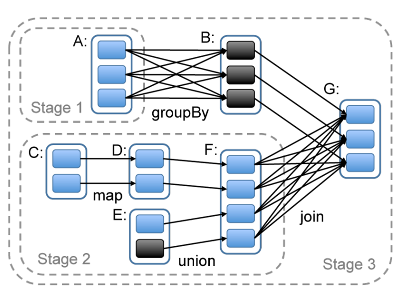
Resilient Distributed Dataset (RDD)

- **RDD =** Collection of records stored across multiple machines in-memory



Spark

- Why "Resilient"?
  - Can survive the failure of a worker node
  - Spark maintains a "lineage graph" of how each RDD partition was created
  - If a worker node fails, the partitions are recreated from its inputs
  - Only a small set of well-defined operations are permitted on the RDDs
    - But the operations usually take in arbitrary "map" and "reduce" functions



- Fault tolerance for the "driver" is trickier
  - Drivers have arbitrary logic (cf., the programs you are writing)
  - In some cases (e.g., Spark Streaming), you can do fault tolerance
  - But in general, driver failure requires a restart

Example Spark Program

Initialize RDD by reading the textFile and partitioning
If textFile stored on HDFS, it is already partitioned – just read each partition as a separate RDD partition

Split each line into words, creating an RDD of words

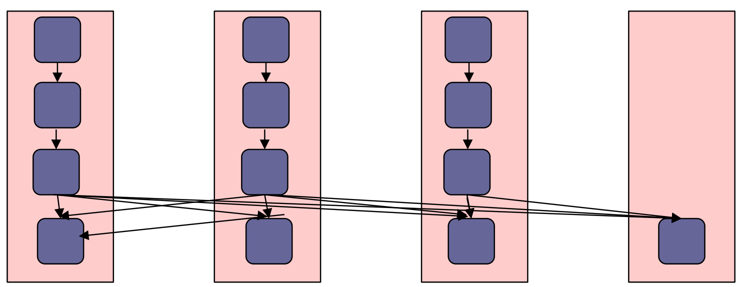
For each word, output (word, 1), creating a new RDD

Do a group-by SUM aggregate to count the number of times each word appears

Driver

```
from pyspark import SparkContext

sc = SparkContext("local", "Simple App")

textFile = sc.textFile("README.md")

counts = textFile
           .flatMap(lambda line: line.split(" "))
           .map(lambda word: (word, 1))
           .reduceByKey(lambda a, b: a + b)

print(counts.take(100))
```

Retrieve 100 of the values in the final RDD



Spark

- Operations often take in a ”function” as input
- Using the inline “lambda” functionality

```
flatMap(lambda line: line.split(" "))
```

- Or a more explicit function declaration

```
def split(line):
      return line.split(" ")

flatMap(split)
```

- Similarly ”reduce” functions essentially tell Spark how to do pairwise aggregation

```
reduceByKey(lambda a, b: a + b)
```

  - Spark will apply this to the dataset pair of values at a time
  - Difficult to do something like “median”

Spark: Map

x1, x2, … can be anything, including documents, images, text files, tuples, dicts, etc.

InputRDD: [x1, x2, …., x_n]

map(lambda x: x + 1)

def fn(x):
    return x+1
map(fn)

outputRDD: [x1+1, x2+1, …., x_n+1]

InputRDD: [x1, x2, …, x_n]

map(fn)

outputRDD: [fn(x1), fn(x2), …, fn(x_n)]

Spark: flatMap

InputRDD: [(a1, b1), (a2, b2), ….]

flatMap(lambda x: [x[0], x[1]])

outputRDD: [a1, b1, a2, b2, …]

InputRDD: ['the little brown fox…', …]

flatMap(lambda x: x.split())

outputRDD: ['the', 'little', 'brown', …]

Spark: groupByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs
Value can be anything (e.g., dicts, tuples, bytes)

groupByKey()

outputRDD: [(a1, [b1, b3, b4, …]), (a2, [b3, b5,…]), …]

Spark: reduceByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs

reduceByKey(func)

```
def func(V1, V2):
    return V3
```

All of V1, V2, and V3 be of the same type

outputRDD: [(a1, …func(func(b1, b3), b4)…),
(a2, …func(func(b2, b5), …)…),]

”func” executed in parallel in a pairwise fashion

Spark: join

InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]
InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples

inputRDD1.join(inputRDD2)

outputRDD: [(a1, (b1, c1)),
(a1, (b1, c3)),
(a1, (b1, c4)),
….]

Spark: cogroup

InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]
InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples

inputRDD1.cogroup(inputRDD2)

outputRDD: [(a1, ([b1, b3, b4, …], [c1, c3, c4, …]),
(a2, ([b2, b5, …], [c2, c5, …]), …
]

RDD Operations

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

Dataframes Example

```
def basic_df_example(spark):
    # $example on:create_df$
    # spark is an existing SparkSession
    df = spark.read.json("examples/src/main/resources/people.json")
    # Displays the content of the DataFrame to stdout
    df.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:create_df$


    # $example on:untyped_ops$
    # spark, df are from the previous example
    # Print the schema in a tree format
    df.printSchema()
    # root
    # |-- age: long (nullable = true)
    # |-- name: string (nullable = true)


    # Select only the "name" column
    df.select("name").show()
    # +-------+
    # |   name|
    # +-------+
    # |Michael|
    # |   Andy|
    # | Justin|
    # +-------+


    # Select everybody, but increment the age by 1
    df.select(df['name'], df['age'] + 1).show()
    # +-------+---------+
    # |   name|(age + 1)|
    # +-------+---------+
    # |Michael|     null|
    # |   Andy|       31|
    # | Justin|       20|
    # +-------+---------+
```

```
# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+----+
# |age|name|
# +---+----+
# | 30|Andy|
# +---+----+


# Count people by age
df.groupBy("age").count().show()
# +----+-----+
# | age|count|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
# $example off:untyped_ops$

 sqlDF = spark.sql("SELECT * FROM people")
 sqlDF.show()
 # +----+-------+
 # | age|   name|
 # +----+-------+
 # |null|Michael|
 # |  30|   Andy|
 # |  19| Justin|
 # +----+-------+
 # $example off:run_sql$


 # $example on:global_temp_view$
 # Register the DataFrame as a global temporary view
 df.createGlobalTempView("people")


 # Global temporary view is tied to a system preserved database
`global_temp`
 spark.sql("SELECT * FROM global_temp.people").show()
 # +----+-------+
 # | age|   name|
 # +----+-------+
 # |null|Michael|
 # |  30|   Andy|
 # |  19| Justin|
 # +----+-------+
```

Summary

- Spark is a popular and widely used framework for large-scale computing
- Simple programming interface
  - You don't need to typically worry about the parallelization
  - That's handled by Spark transparently
  - In practice, may need to fiddle with number of partitions etc.
- Managed services supported by several vendors including Databricks (started by the authors of Spark), Cloudera, etc.
- Many other concepts that we did not discuss
  - Shared accumulator and broadcast variables
  - Support for Machine Learning, Graph Analytics, Streaming, and other use cases
- Alternatives include: Apache Tez, Flink, and several others

CMSC424: Database Design

Module: NoSQL; Big Data Systems

MongoDB

Instructor: Amol Deshpande
amol@umd.edu

MongoDB: History 1

- A prototypical NoSQL database
- Short for hu**mongo**us
- First version in 2009!
- Still very popular
  - IPO in 2017
  - Now worth >7B in market capital (as of 2020)

Slides adapted from CS186 Slides by:
Alvin Cheung
Aditya Parameswaran

MongoDB: History 1

- A prototypical NoSQL database
- Short for hu**mongo**us
- First version in 2009!
- Still very popular
  - IPO in 2017
  - Now worth >7B in market capital (as of 2020)

MongoDB: History 2

- Internet & social media boom led to a demand for
  - Rapid data model evolution: "a move fast and break things" mentality to system dev
    - E.g., adding a new attrib to a Facebook profile
    - Contrary to DBMS wisdom of declaring schema upfront and changing rarely (costly!)
  - Rapid txn support, even at the cost of losing some updates or non-atomicity
    - Contrary to DBMS wisdom of ACID, esp. with distribution/2PC (costly!)

- Early version centered around storing and querying json documents quickly
- Made several tradeoffs for speed
  - No joins ➔ now support left outer joins
  - Limited query opt ➔ still limited, but many improvements
  - No txn support apart from atomic writes to json docs ➔ limited support for multi-doc txns
  - No checks/schema validation ➔ now support json schema validation (rarely used!)

MongoDB: History 3

https://www.mongodb.com/blog/post/what-about-durability

- Most egregious: no durability or write ahead logging!

We get lots of questions about why MongoDB doesn't have full single server durability, and there are many people that think this is a major problem. We wanted to shed some light on why we haven't done single server durability, what our suggestions are, and our future plans.

Excuse 1: Durability is overrated

damage, fire, some hardware problems, etc… no matter how durable the software is, data can be lost. Yes - there are ways to mitigate some of these, but those add another layer of complexity, that has to be tested, proofed, and adds more variables which can fail.

Excuse 2: It's hard to implement

uses a transaction log for durability, you either have to turn off hardware buffering or have a battery backed RAID controller. Without hardware buffering, transaction logs are very slow. Battery backed raid controllers will work well, but you have to really have one. With the move towards the cloud and outsourced hosting, custom hardware is not always an option.

Sure enough,

MongoDB: History 4

Bottomline:

MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons

We'll focus on two primary design decisions:

- The data model
- The query language

Will discuss these two to start with, then some of the architectural issues

MongoDB Data Model

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {…, field: value, …}

Where value can be:
- Atomic
- A document
- An array of atomic values
- An array of documents

{ qty : 1, status : "D", size : {h : 14, w : 21}, tags : ["a", "b"] },

Can also mix and match, e.g., array of atomics and documents, or array of arrays
[Same as the JSON data model]

Internally stored as BSON = Binary JSON
- Client libraries can directly operate on this natively

MongoDB Data Model 2

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {…, field: value, …}

Can use JSON schema validation

- Some integrity checks, field typing and ensuring the presence of certain fields
- Rarely used, and we'll skip for our discussion

Special field in each document: _id

- Primary key
- Will also be indexed by default
- If it is not present during ingest, it will be added
- Will be first attribute of each doc.
- This field requires special treatment during projections as we will see later

MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Some MQL Principles : Dot (.) Notation

- "." is used to drill deeper into nested docs/arrays
- Recall that a value could be atomic, a nested document, an array of atomics, or an array of nested documents

- Examples:
  - "instock.qty" ➔ qty field within the instock field
  - "instock.1" ➔ second element within the instock array
    - Element could be an atomic value or a nested document
  - "instock.1.qty" ➔ qty field within the second document within the instock array

- Note: such dot expressions need to be in quotes

Some MQL Principles : Dollar ($) Notation

- $ indicates that the string is a special keyword
  - E.g., $gt, $lte, $add, $elemMatch, …
- Used as the "field" part of a "field : value" expression
- So if it is a binary operator, it is *usually* done as:
  - {LOperand : { $keyword : ROperand}}
  - e.g., {qty : {$gt : 30}}
- Alternative: arrays
  - {$keyword : [argument list]}
  - e.g., {$add : [1, 2]}

- Exception: $fieldName, used to refer to a previously defined field on the value side
  - Purpose: disambiguation
  - Only relevant for aggregation pipelines
  - Let's not worry about this for now.

Retrieval Queries Template

db.collection.**find**(<*predicate*>, optional <*projection*>)

returns documents that match <*predicate*>
keep fields as specified in <*projection*>
both <*predicate*> and <*projection*> expressed as documents
in fact, most things are documents!

db.inventory.find({ })

returns all documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Basic Queries


```
> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```


db.collection.find(<predicate>, optional <projection>)

- find({ status : "D"})
  - all documents with status D ➔ paper, planner
- find ({ qty : {$gte : 50} })
  - all documents with qty >= 50 ➔ notebook, paper, planner
- find ({ status : "D", qty : {$gte : 50} })
  - all documents that satisfy both ➔ paper, planner
- find({ $or: [{ status : "D" }, { qty : { $lt : 30 } }] })
  - all documents that satisfy either ➔ journal, paper, planner

```
> db.inventory.find( { $or: [ { status: "D" }, { qty: { $lt: 30 } } ] } )
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Nested Documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

db.collection.find(<predicate>, optional <projection>)

- find({ size: { h: 14, w: 21, uom: "cm" } })
  - exact match of nested document, including ordering of fields! ➔ journal
- find ({ "size.uom" : "cm", "size.h" : {$gt : 14 })
  - querying a nested field ➔ planner
  - Note: when using . notation for sub-fields, expression must be in quotes
  - Also note: binary operator handled via a nested document

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Arrays

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Slightly different example dataset for Arrays and Arrays of Document Examples
db.collection.find(<predicate>, optional <projection>)

- find({ tags: ["red", "blank"] })
  - Exact match of array ➔ notebook
- find({ tags: "red" })
  - If one of the elements matches red ➔ journal, notebook, paper, planner
- find({ tags: "red", tags: "plain" })
  - If one matches red, one matches plain ➔ paper
- find({ dim: { $gt: 15, $lt: 20 } })
  - If one element is >15 and another is <20 ➔ journal, notebook, paper, postcard
- find({ dim: {$elemMatch: { $gt: 15, $lt: 20 } } })
  - If a single element is >15 and <20 ➔ postcard
- find({ "dim.1": { $gt: 25 } })
  - If second item > 25 ➔ planner
  - Notice again that we use quotes to when using . notation

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Arrays of Documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

db.collection.find(<predicate>, optional <projection>)

- find({ instock: { loc: "A", qty: 5 } })
  - Exact match of document [like nested doc/atomic array case] ➔ journal
- find({ "instock.qty": { $gte : 20 } })
  - One nested doc has >= 20 ➔ paper, planner, postcard
- find({ "instock.0.qty": { $gte : 20 } })
  - First nested doc has >= 20 ➔ paper, planner
- find({ "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } })
  - One doc has 20 >= qty >10➔ paper, journal, postcard
- find({ "instock.qty": { $gt: 10, $lte: 20 } })
  - One doc has 20 >= qty, another has qty>10 ➔ paper, journal, postcard, planner

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries Template: Projection


```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```


db.collection.find(<predicate>, optional <projection>)

- Use 1s to indicate fields that you want
  - Exception: _id is always present unless explicitly excluded
- OR Use 0s to indicate fields you don't want
- Mixing 0s and 1s is not allowed for non _id fields

- find({ }, {item: 1})

```
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard" }
```

- find({ }, {item: 1, _id : 0})

```
{ "item" : "journal" }
{ "item" : "notebook" }
{ "item" : "paper" }
{ "item" : "planner" }
{ "item" : "postcard" }
```

- find({},{item : 1, tags: 0, _id : 0})

```
Error: error: {
"ok" : 0,
"errmsg" : "Cannot do exclusion on field tags in inclusion projection",
"code" : 31254,
"codeName" : "Location31254" }
```

- find({},{item : 1, "instock.loc": 1, _id : 0})

```
{ "item" : "journal", "instock" : [ { "loc" : "A" }, { "loc" : "C" } ] }
{ "item" : "notebook", "instock" : [ { "loc" : "C" } ] }
{ "item" : "paper", "instock" : [ { "loc" : "A" }, { "loc" : "B" } ] }
{ "item" : "planner", "instock" : [ { "loc" : "A" }, { "loc" : "B" } ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B" }, { "loc" : "C" } ] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries : Addendum

```
db.inventory.find()
"_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Two additional operations that are useful for retrieval:

- Limit (k) like LIMIT in SQL
  - e.g., db.inventory.find({ }).limit(1)
- Sort ({ }) like ORDER BY in SQL
  - List of fields, -1 indicates decreasing 1 indicates ascending
  - e.g., db.inventory.find({ }, {_id : 0, instock : 0}).sort({ "dim.0": -1, item: 1 })

```
{ "item" : "planner", "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "journal", "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "item" : "notebook", "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "item" : "postcard", "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Summary

find() = SELECT <projection>
FROM Collection
WHERE <predicate>
limit() = LIMIT
sort() = ORDER BY

db.inventory.find(	FROM
{ tags : red },	WHERE
{_id : 0, instock : 0})	SELECT
.sort ({ "dim.0": -1, item: 1 })	ORDER BY
.limit (2)	LIMIT

Syntax somewhat different when called from within Python3 (using pymongo)

What did we not cover?

- The use of regexes for matching
- $all : all entries in an array satisfy a condition
- $in : checking if a value is present in an array of atomic values
- The presence or absence of fields
  - Can use special "null" values
  - {field : null} checks if a field is null or missing
  - $exists : checking the presence/absence of a field

Syntax somewhat different when called from within Python3 (using pymongo)

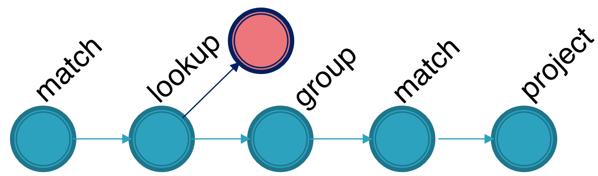
MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ( )
  - project // second arg of find ( ) but more expressiveness
  - sort/limit // same as retrieval
  - group
  - unwind
  - lookup
  - … lots more!!
- Each stage manipulates the existing collection in some way



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] )
```


Next Set of Examples

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
{ "_id" : "01020", "city" : "CHICOPEE", "loc" : [ -72.576142, 42.176443 ], "pop" : 31495, "state" : "MA" }
{ "_id" : "01028", "city" : "EAST LONGMEADOW", "loc" : [ -72.505565, 42.067203 ], "pop" : 13367, "state" : "MA" }
{ "_id" : "01030", "city" : "FEEDING HILLS", "loc" : [ -72.675077, 42.07182 ], "pop" : 11985, "state" : "MA" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [ -72.844092, 42.466234 ], "pop" : 122, "state" : "MA" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [ -72.833309, 42.38167 ], "pop" : 177, "state" : "MA" }
{ "_id" : "01010", "city" : "BRIMFIELD", "loc" : [ -72.188455, 42.116543 ], "pop" : 3706, "state" : "MA" }
{ "_id" : "01034", "city" : "TOLLAND", "loc" : [ -72.908793, 42.070234 ], "pop" : 1652, "state" : "MA" }
{ "_id" : "01035", "city" : "HADLEY", "loc" : [ -72.571499, 42.36062 ], "pop" : 4231, "state" : "MA" }
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA" }
{ "_id" : "01040", "city" : "HOLYOKE", "loc" : [ -72.626193, 42.202007 ], "pop" : 43704, "state" : "MA" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [ -72.936114, 42.182949 ], "pop" : 1240, "state" : "MA" }
{ "_id" : "01050", "city" : "HUNTINGTON", "loc" : [ -72.873341, 42.265301 ], "pop" : 2084, "state" : "MA" }
{ "_id" : "01054", "city" : "LEVERETT", "loc" : [ -72.499334, 42.46823 ], "pop" : 1748, "state" : "MA" }
```

One document per zipcode: 29353 zipcodes

Syntax somewhat different when called from within Python3 (using pymongo)

Grouping (with match/sort) Simple Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

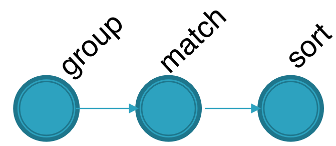
Find states with population > 15M, sort by decending order

```
db.zips.aggregate( [
{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 15000000 } } },
{ $sort : { totalPop : -1 } }
 ] )
```

GROUP BY

AGGS.

match after group = HAVING




```
{ "_id" : "CA", "totalPop" : 29754890 }
{ "_id" : "NY", "totalPop" : 17990402 }
{ "_id" : "TX", "totalPop" : 16984601 }
…
```

Q: what would the SQL query for this be?

SELECT state AS id, **SUM**(pop) **AS** totalPop
FROM zips
GROUP BY state
HAVING totalPop >= 15000000
ORDER BY totalPop **DESCENDING**

Syntax somewhat different when called from within Python3 (using pymongo)

Grouping Syntax

```
$group : {
        _id: <expression>, // Group By Expression
        <field1>: { <aggfunc1> : <expression1> },
        ... }
```

Returns one document per unique group, indexed by _id

Agg.func. can be standard ops like $sum, $avg, $max

Also MQL specific ones:

- **$first** : return the first expression value per group
  - makes sense only if docs are in a specific order [usually done after sort]
- **$push** : create an array of expression values per group
  - didn't make sense in a relational context because values are atomic
- **$addToSet** : like $push, but eliminates duplicates

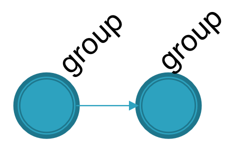
Multiple Attrib. Grouping Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

Group by 2 attribs, giving nested id

aggregate([
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
])

Group by previously def. id.state



Notice use of $ to refer to previously defined fields

Q: Guesses on what this might be doing?
A: Find average city population per state

{ "_id" : "GA", "avgCityPop" : 11547.62210338681 }
{ "_id" : "WI", "avgCityPop" : 7323.00748502994 }
{ "_id" : "FL", "avgCityPop" : 27400.958963282937 }
{ "_id" : "OR", "avgCityPop" : 8262.561046511628 }
{ "_id" : "SD", "avgCityPop" : 1839.6746031746031 }
{ "_id" : "NM", "avgCityPop" : 5872.360465116279 }
{ "_id" : "MD", "avgCityPop" : 12615.775725593667 }
…

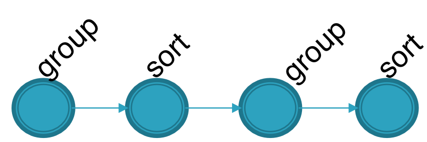
Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

Find, for every state, the biggest city and its population

aggregate([
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
])



Can list multiple aggregations after grouping id

Approach:

- Group by pair of city and state, and compute population per city
- Order by population descending
- Group by state, and find first city and population per group (i.e., the highest population city)
- Order by population descending

{ "_id" : "IL", "bigCity" : "CHICAGO", "bigPop" : 2452177 }
{ "_id" : "NY", "bigCity" : "BROOKLYN", "bigPop" : 2300504 }
{ "_id" : "CA", "bigCity" : "LOS ANGELES", "bigPop" : 2102295 }
{ "_id" : "TX", "bigCity" : "HOUSTON", "bigPop" : 2095918 }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA", "bigPop" : 1610956 }
{ "_id" : "MI", "bigCity" : "DETROIT", "bigPop" : 963243 }
…

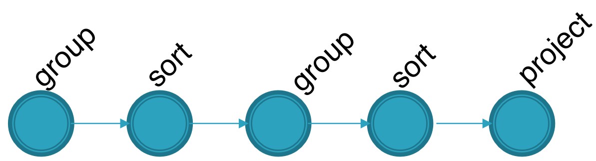
Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. with Vanilla Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

If we only want to keep the state and city …

aggregate([
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
{ $project : {bigPop : 0} }
])



{ "_id" : "IL", "bigCity" : "CHICAGO" }
{ "_id" : "NY", "bigCity" : "BROOKLYN" }
{ "_id" : "CA", "bigCity" : "LOS ANGELES" }
{ "_id" : "TX", "bigCity" : "HOUSTON" }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA" }
…

Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. with Adv. Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

If we wanted to nest the name of the city and population into a nested doc

Can construct new nested documents in output, unlike vanilla projection

aggregate([
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} },
{ $project : { _id : 0, state : "$_id", bigCityDeets: { name: "$bigCity", pop: "$bigPop" } } }
])

{ "state" : "IL", "bigCityDeets" : { "name" : "CHICAGO", "pop" : 2452177 } }
{ "state" : "NY", "bigCityDeets" : { "name" : "BROOKLYN", "pop" : 2300504 } }
{ "state" : "CA", "bigCityDeets" : { "name" : "LOS ANGELES", "pop" : 2102295 } }
{ "state" : "TX", "bigCityDeets" : { "name" : "HOUSTON", "pop" : 2095918 } }
{ "state" : "PA", "bigCityDeets" : { "name" : "PHILADELPHIA", "pop" : 1610956 } }
…

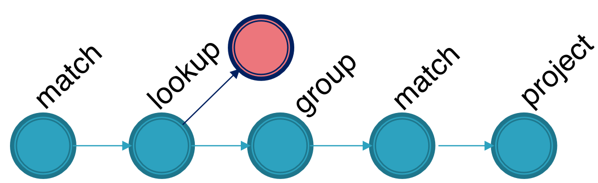
Syntax somewhat different when called from within Python3 (using pymongo)

Advanced Projection vs. Vanilla Projection

- In addition to excluding/including fields like in projection during retrieval (find), projection in the aggregation pipeline allows you to:
  - Rename fields
  - Redefine new fields using complex expressions on old fields
  - Reorganize fields into nestings or unnestings
  - Reorganize fields into arrays or break down arrays
- Try them at home!

Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ( )
  - project // second arg of find ( ) but more expressiveness
  - sort/limit // same
  - group
  - **unwind**
  - **lookup**
  - ... lots more!!
- Each stage manipulates the existing collection in some way



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Unwinding Arrays

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Unwind expands an array by constructing documents one per element of the array

Somewhat like flatMap in Spark

Going back to our old example with an array of tags

Notice no relational analog here: no arrays so no unwinding

[in fact, some RDBMSs do support arrays, but not in the rel. model]

```
aggregate( [
    { $unwind : "$tags" },
    { $project : {_id : 0, instock: 0}}
] )
```

```
{ "item" : "journal", "tags" : "blank", "dim" : [ 14, 21 ] }
{ "item" : "journal", "tags" : "red", "dim" : [ 14, 21 ] }
{ "item" : "notebook", "tags" : "red", "dim" : [ 14, 21 ] }
{ "item" : "notebook", "tags" : "blank", "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : "red", "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : "blank", "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : "plain", "dim" : [ 14, 21 ] }
{ "item" : "planner", "tags" : "blank", "dim" : [ 22.85, 30 ] }
{ "item" : "planner", "tags" : "red", "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "tags" : "blue", "dim" : [ 10, 15.25 ] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Unwind: A Common Template

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Q: Imagine if we want to find sum of qtys across items. How would we do this?

A common recipe in MQL queries is to *unwind* and then *group by*

```
aggregate( [
        { $unwind : "$instock" },
        { $group : {_id : "$item", totalqty : {$sum : "$instock.qty"}}}
] )
```

```
{ "_id" : "notebook", "totalqty" : 5 }
{ "_id" : "postcard", "totalqty" : 50 }
{ "_id" : "journal", "totalqty" : 20 }
{ "_id" : "planner", "totalqty" : 45 }
{ "_id" : "paper", "totalqty" : 75 }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Looking Up Other Collections

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

```
{ $lookup: {
        from: <collection to join>,
        localField: <referencing field>,
        foreignField: <referenced field>,
        as: <output array field>
  } }
```

Conceptually, for each document

- find documents in other coll that join (equijoin)
  - local field must match foreign field
- place each of them in an array

Thus, a left outer equi-join, with the join results stored in an array

Straightforward, but kinda gross. Let's see…

Say, for each item, I want to find other items located in the same location = self-join

```
db.inventory.aggregate( [
    { $lookup : {from : "inventory", localField: "instock.loc",
    foreignField: "instock.loc", as:"otheritems"}},
    { $project : {_id : 0, tags : 0, dim : 0}}
] )
```

```
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty"
: 15 } ], "otheritems" : [
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c24"), "item" : "journal",
    "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ],
    "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] },
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c25"), "item" :
    "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red",
    "blank" ], "dim" : [ 14, 21 ] },
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c26"), "item" : "paper",
    "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ],
    "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] },
    …
    ] }
```

And many other records!

Syntax somewhat different when called from within Python3 (using pymongo)

Lookup… after some more projection


```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```


```
db.inventory.aggregate( [
{ $lookup : {from:"inventory", localField:"instock.loc", foreignField:"instock.loc", as:"otheritems"}},
{$project : {_id : 0, tags :0, dim :0, "otheritems._id":0, "otheritems.tags":0, "otheritems.dim":0,
"otheritems.instock.qty":0}} ] )
```

```
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "otheritems" : [
    { "item" : "journal", "instock" : [ { "loc" : "A" }, { "loc" : "C" } ] },
    { "item" : "notebook", "instock" : [ { "loc" : "C" } ] },
    { "item" : "paper", "instock" : [ { "loc" : "A" }, { "loc" : "B" } ] },
    { "item" : "planner", "instock" : [ { "loc" : "A" }, { "loc" : "B" } ] },
    { "item" : "postcard", "instock" : [ { "loc" : "B" }, { "loc" : "C" } ] } ] }

{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "otheritems" : [
    { "item" : "journal", "instock" : [ { "loc" : "A" }, { "loc" : "C" } ] },
    { "item" : "notebook", "instock" : [ { "loc" : "C" } ] },
    { "item" : "postcard", "instock" : [ { "loc" : "B" }, { "loc" : "C" } ] } ] }

…
```

Syntax somewhat different when called from within Python3 (using pymongo)

Some Rules of Thumb when Writing Queries

- $project is helpful if you want to construct or deconstruct nestings (in addition to removing fields or creating new ones)
- $group is helpful to construct arrays (using $push or $addToSet)
- $unwind is helpful for unwinding arrays
- $lookup is your only hope for joins. Be prepared for a mess. Lots of $project needed

MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Update Queries: InsertMany

[Insert/Delete/Update] [One/Many]
- Many is more general, so we'll discuss that instead

```
db.inventory.insertMany( [
{ item: "journal", instock: [ { loc: "A", qty: 5 }, { loc: "C", qty: 15 } ], tags: ["blank", "red"], dim: [ 14, 21 ] },
{ item: "notebook", instock: [ { loc: "C", qty: 5 } ], tags: ["red", "blank"] , dim: [ 14, 21 ]},
{ item: "paper", instock: [ { loc: "A", qty: 60 }, { loc: "B", qty: 15 } ], tags: ["red", "blank", "plain"] , dim: [ 14, 21 ]},
{ item: "planner", instock: [ { loc: "A", qty: 40 }, { loc: "B", qty: 5 } ], tags: ["blank", "red"], dim: [ 22.85, 30 ] },
{ item: "postcard", instock: [ {loc: "B", qty: 15 }, { loc: "C", qty: 35 } ], tags: ["blue"] , dim: [ 10, 15.25 ] }
] );
```

Several actions will be taken as part of this insert:
- Will create inventory collection if absent [No schema specification/DDL needed!]
- Will add the _id attrib to each document added (since it isn't there)
- _id will be the first field for each document by default

Update Queries: UpdateMany

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Syntax: updateMany ({<condition>}, {<change>})

```
db.inventory.updateMany (
    {"dim.0": { $lt: 15 } },
    { $set: { "dim.0": 15, status: "InvalidWidth"} }
) // if any width <15, set it to 15 and set status to InvalidWidth.
```

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 15, 21 ], "status" : "InvalidWidth" }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 15, 21 ], "status" : "InvalidWidth" }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 15, 21 ], "status" : "InvalidWidth" }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 15, 15.25 ], "status" : "InvalidWidth" }
```

Analogous to: UPDATE R SET <change> WHERE <condition>

Update Queries: UpdateMany 2

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Syntax: updateMany ({<condition>}, {<change>})

db.inventory.updateMany (
{"dim.0": { $lt: 15 } },
{ **$inc: { "dim.0": 5},**
$set: {status: "InvalidWidth"} })

// if any width <15, increment by 5 and set status to InvalidWidth.

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 19, 21 ], "status" : "InvalidWidth" }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 19, 21 ], "status" : "InvalidWidth" }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 19, 21 ], "status" : "InvalidWidth" }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 15, 15.25 ], "status" : "InvalidWidth" }
```

Analogous to: UPDATE R SET <change> WHERE <condition

MongoDB Internals

- MongoDB is a distributed NoSQL database
- Collections are partitioned/sharded based on a field [range-based]
  - Each partition stores a subset of documents
- Each partition is replicated to help with failures
  - The replication is done asynchronously
  - Failures of the main partition that haven't been propagated will be lost
- Limited heuristic-based query optimization (will discuss later)
- Atomic writes to documents within collections by default. Multi-document txns are discouraged (but now supported).

MongoDB Internals

- Weird constraint: intermediate results of aggregations must not be too large (100MB)
  - Else will end up spilling to disk
  - Not clear if they perform any pipelining across aggregation operators
- Optimization heuristics
  - Will use indexes for $match if early in the pipeline [user can explicitly declare]
  - $match will be merged with other $match if possible
    - Selection fusion
  - $match will be moved early in the pipeline sometimes
    - Selection pushdown
    - But: not done always (e.g., not pushed before $lookup)
  - No cost-based optimization as far as one can tell

MongoDB: Summary

Bottomline:

MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons

MongoDB has a flexible data model and a powerful (if confusing) query language.

Many of the internal design decisions as well as the query & data model can be understood when compared with DBMSs

- DBMSs provide a "gold standard" to compare against.
- In the "wild" you'll encounter many more NoSQL systems, and you'll need to do the same thing that we did here!

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Parallelizing Operations

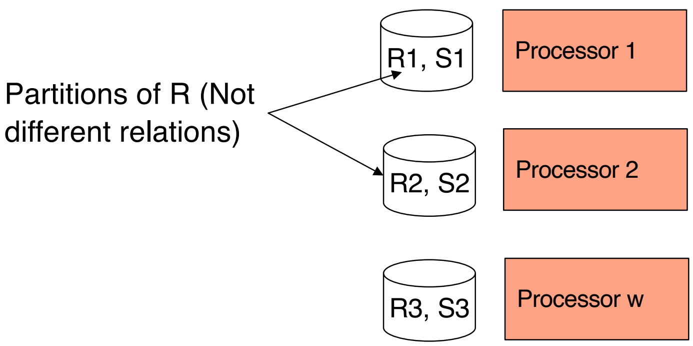
Instructor: Amol Deshpande
amol@umd.edu

Parallelizing Operations

- Book Chapters
  - 18.5, 18.6
- Key topics:
  - Parallelizing a Sort Operation
  - Parallelizing a Join Operation
  - Parallelizing a Group By Operation

Setup

- Assume Shared-Nothing Model
- Relations are already partitioned across a set of machines (will talk about how next video)
- How to execute different operations?



Processor 1 can directly read R1, S1

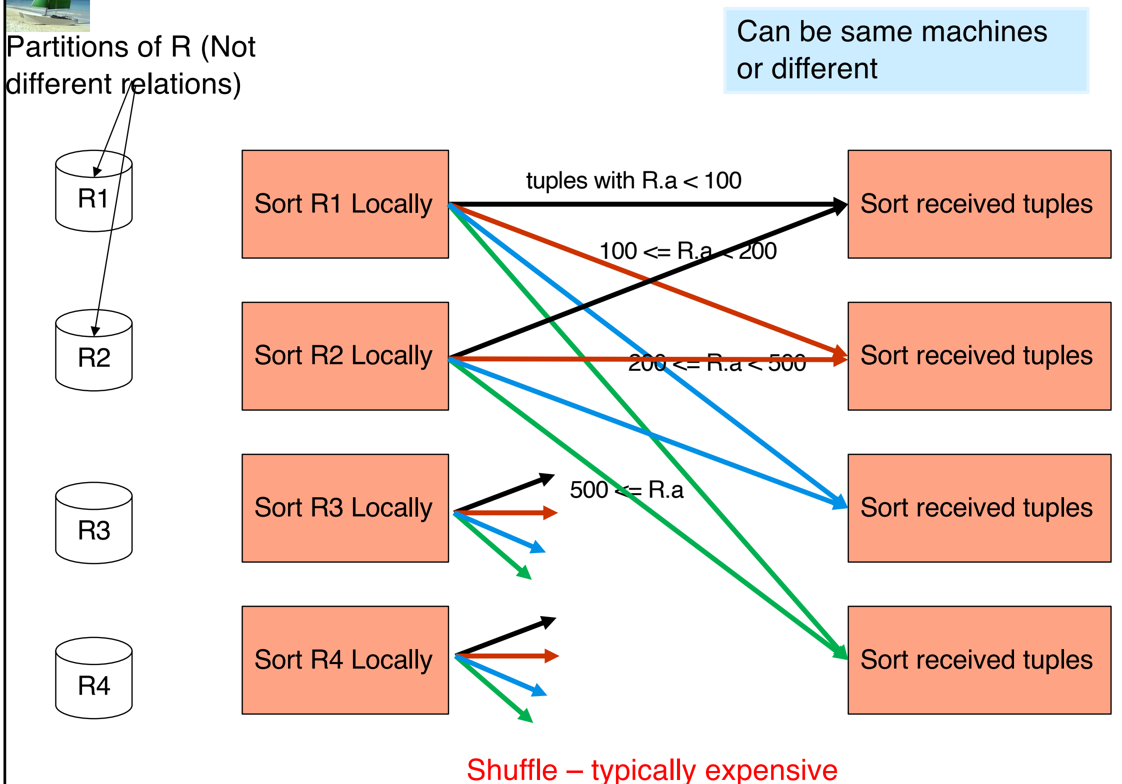
If it wants R2, Processor 2 must read it and send it to Processor 1

Parallel Sort

1. Each processor sorts a portion of the data (e.g., the data on their local disk)
2. If the data is small enough, all the processors can send it to a single machine to do a "merge"
3. If the data is large, then "merge" itself done in parallel through range partitioning
   1. Each processor in the merge phase gets assigned a range of the data
   2. All other processors send the appropriate data based on that range partitioning

- In either phase, the processors work by themselves ("data parallelism") but data must be "shuffled" in between
- Other approaches exist, but basically same steps

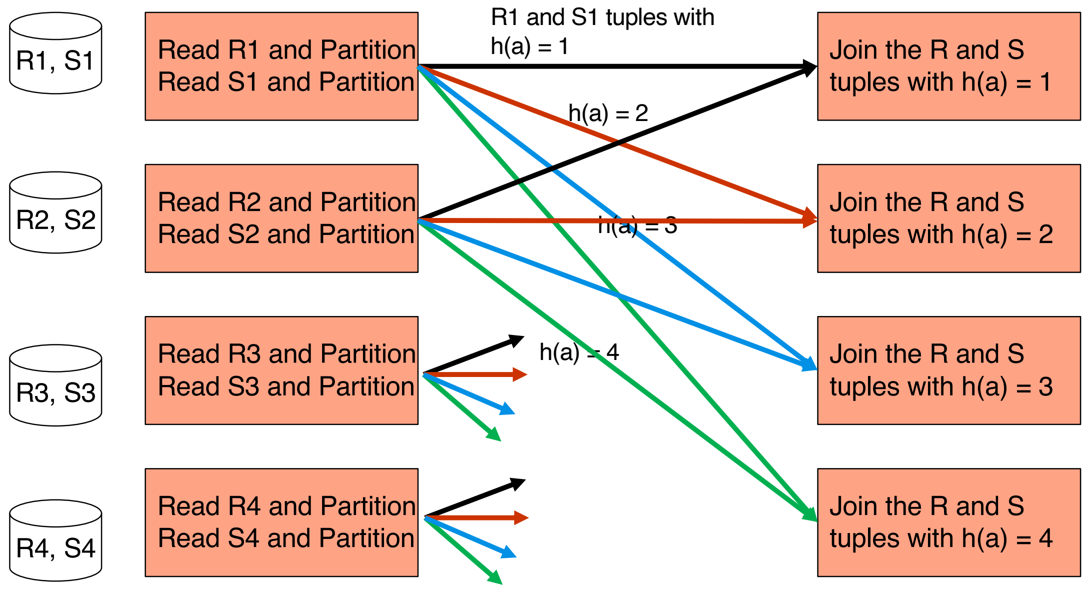
Parallel Sort



Parallel Join

- Hash-based approach
  - Very similar to how partitioning hash join works (i.e., the variant we saw for the case when the relations don't fit in memory)
  - Most common for equi-joins where hashing can be used
  - Easier to guarantee balanced work
- Sort-based approach
  - Similar to the parallel sort approach
  - Both relations sorted using the same key
  - Same processor used for merging in the second phase for both relations

Hash-based Parallel Join

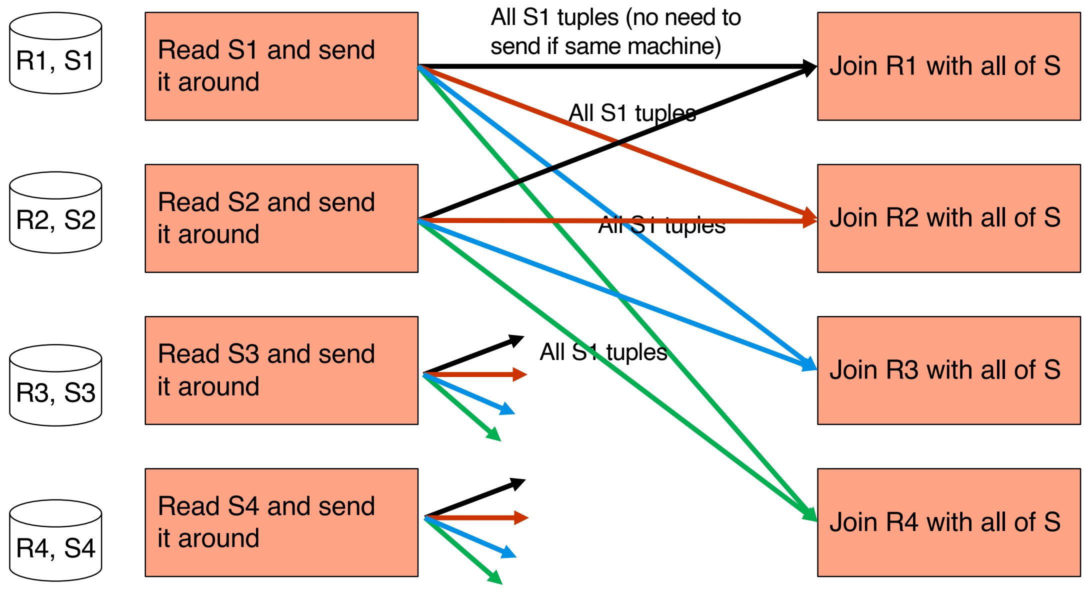


Shuffle – typically expensive

Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.
- For joins were partitioning is not applicable, parallelization can be accomplished by **fragment and replicate** technique
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say *r*, is partitioned; any partitioning technique can be used.
  - The other relation, *s*, is replicated across all the processors.
  - Processor P_i then locally computes the join of r_i with all of s using any join technique.

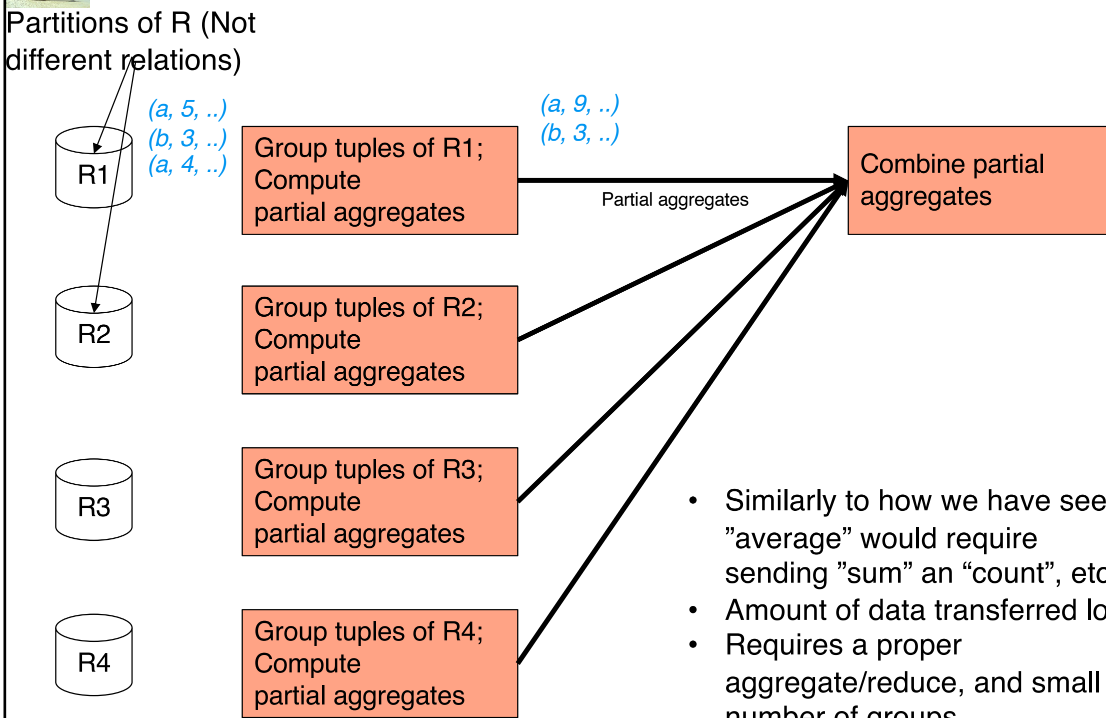
Asymmetric Fragment and Replicate



Grouping/Aggregation

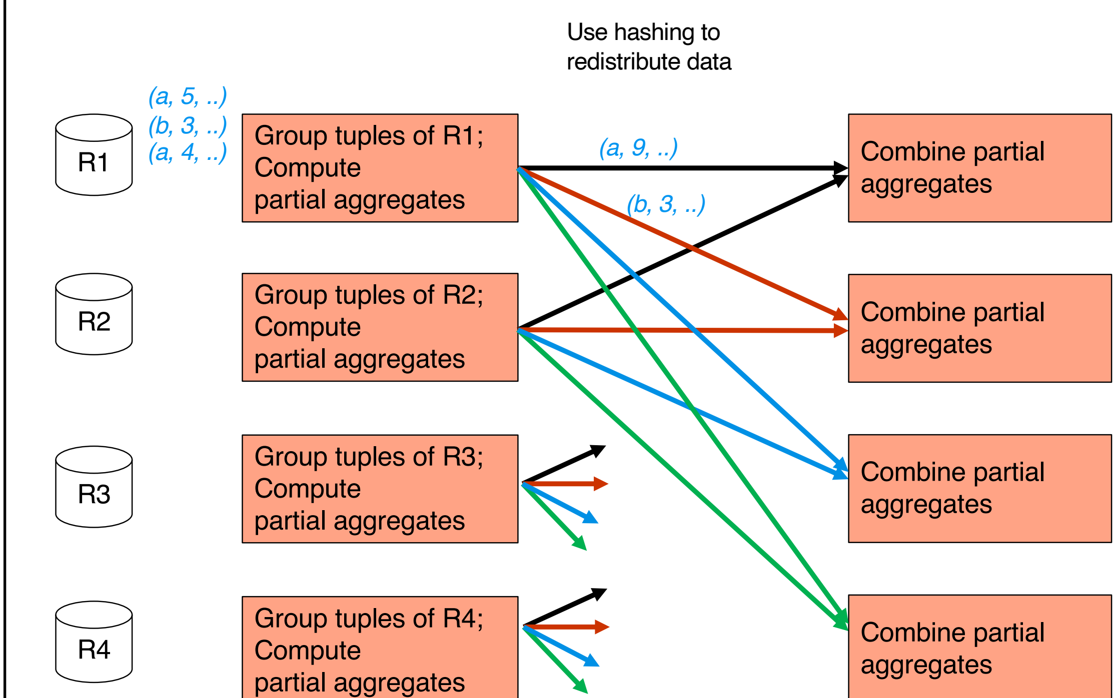
- Very common operation, especially in Map-Reduce applications
  - E.g., grouping by "hostnames" or "words" (as in project 5) or "labels" (in ML context), etc.
  - The idea of distributing data, doing some computations, and collecting results is quite powerful
  - Even "joins" can be seen as a "group by" operation (you can group the tuples of the two relations on the join attribute, and then compute join)
- Need to differentiate between "groupby" and "aggregate" ("reduce")
  - **Groupby:** For every value of "group by attribute" (i.e., "key"), collect all tuples/records with that key on a single machine
  - **Aggregate/Reduce:** Perform some computation on them, typically reducing the size of the data
  - Spark has more granular operations than SQL
- Challenges:
  - Number of keys might be very large
  - Should try to do as much pre-aggregation as possible

Scenario 1: Small # of Groups + Reduce



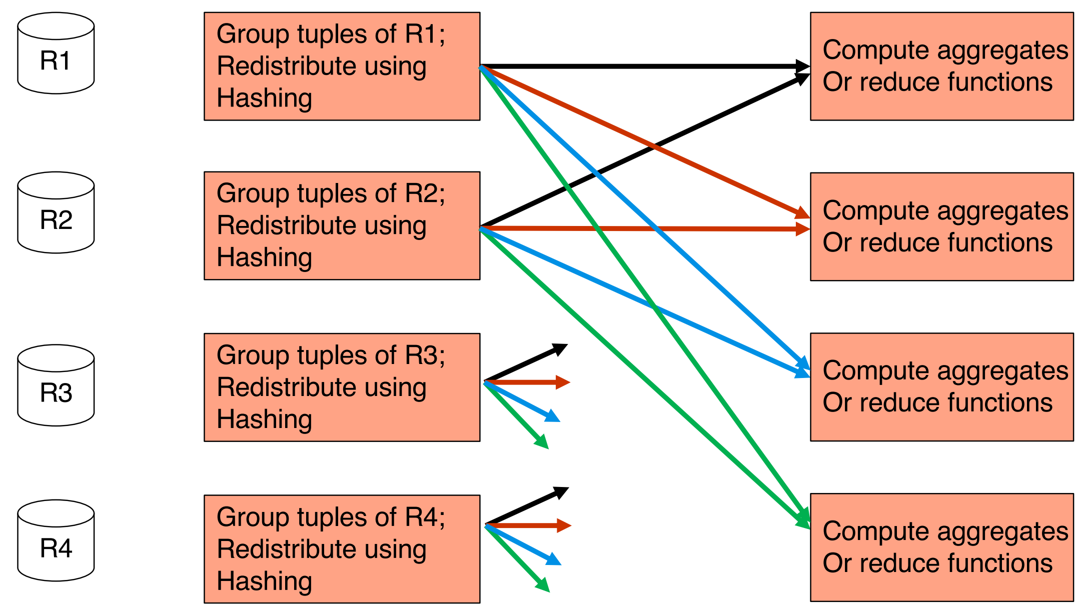
- Similarly to how we have seen, "average" would require sending "sum" an "count", etc
- Amount of data transferred low
- Requires a proper aggregate/reduce, and small number of groups

Scenario 2: Large # of Groups + Reduce



Scenario 3: Large # of Groups + No Reduce

e.g., if we want to compute "median" or some other complex statistics – no "partial aggregation" possible



Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single processor.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each processor whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the processors.

Other Relational Operations (Cont.)

- **Duplicate elimination**
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash-partitioning) and perform duplicate elimination locally at each processor.
- **Projection**
  - Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

CMSC424: Database Design

Module: NoSQL; Big Data Systems

MapReduce Overview

Instructor: Amol Deshpande
amol@umd.edu

Big Data; Storage Systems

- Book Chapters
  - 10.3 **(7TH EDITION)**
- Key topics:
  - Why MapReduce and History
  - Word Count using MapReduce

The MapReduce Paradigm

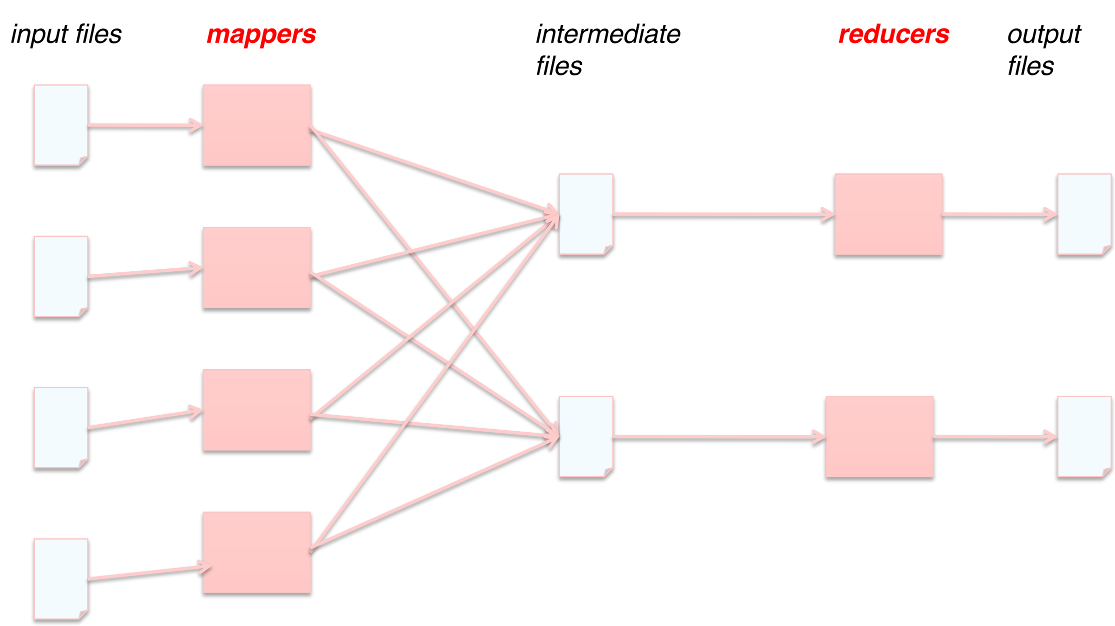
- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer
  - Programmer provides core logic (via map() and reduce() functions)
  - System takes care of parallelization of computation, coordination, etc.
- Paradigm dates back many decades
  - But very large scale implementations running on clusters with 10^3 to 10^4 machines are more recent
  - Google Map Reduce, Hadoop, ..
- Data storage/access typically done using distributed file systems or key-value stores

MapReduce Framework

- Provides a fairly restricted, but still powerful abstraction for programming

- Programmers write a pipeline of functions, called *map* or *reduce*
  - **map programs**
    - inputs: a list of "records" (record defined arbitrarily – could be images, genomes etc…)
    - output: for each record, produce a set of "(key, value)" pairs
  - **reduce programs**
    - input: a list of "(key, {values})" grouped together from the mapper
    - output: whatever
  - Both can do arbitrary computations on the input data as long as the basic structure is followed

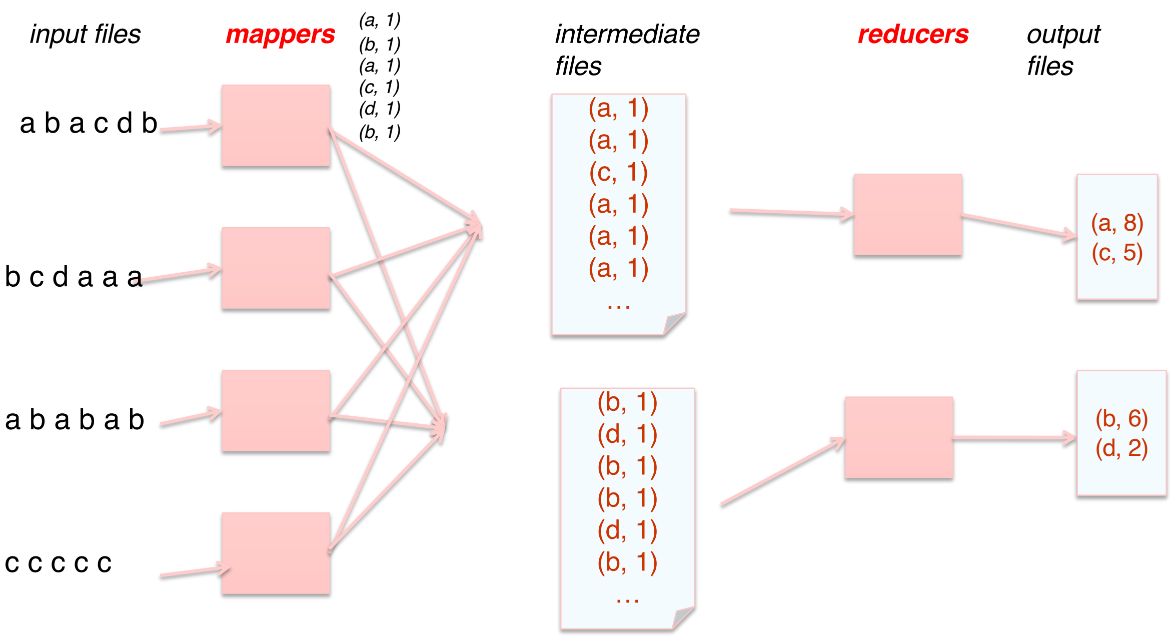
MapReduce Framework



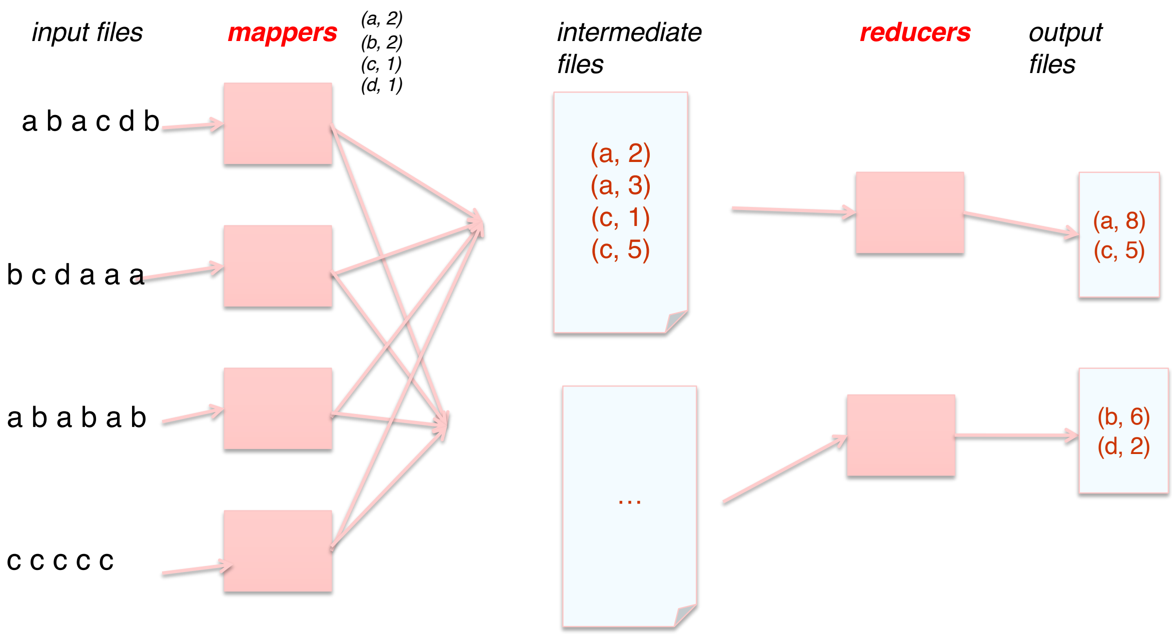
Word Count Example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```


MapReduce Framework: Word Count



More Efficient Word Count



Called "mapper-side" combiner

Hadoop MapReduce

- Google pioneered original map-reduce implementation
  - For building web indexes, text analysis, PageRank, etc.
- Hadoop -- widely used open source implementation in Java
- Huge ecosystem built around Hadoop now, including HDFS, consistency mechanisms, connectors to different systems (e.g., key-value stores, databases), etc.
- Apache Spark a newer implementation of Map-Reduce
  - More user-friendly syntax
  - Significantly faster because of in-memory processing
  - SQL-like in many ways ("DataFrames")

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Other Storage Systems; Wrapup

Instructor: Amol Deshpande
amol@umd.edu

Big Data Storage Options

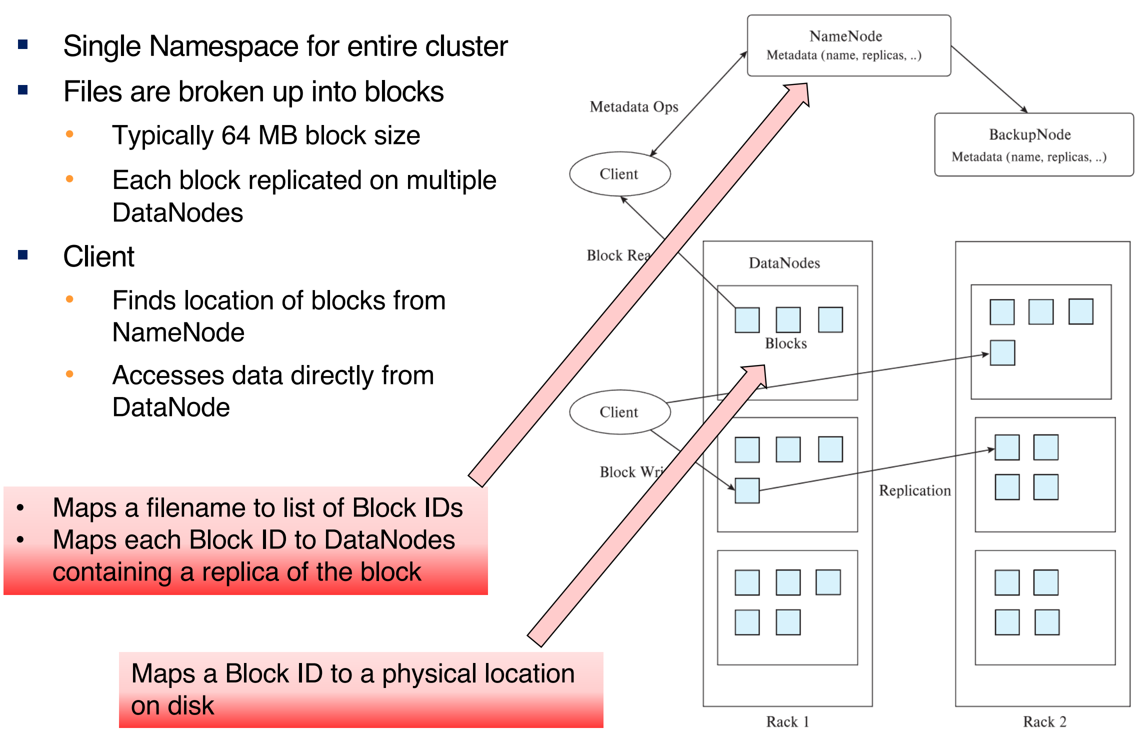
- Parallel or distributed databases
  - Suffer from the issues discussed earlier
- Distributed File Systems
  - Also called object stores
  - A "data lake" is basically a collection of files in a dfs
  - Structured data (relational-like) stored in files (more sophisticated "csv" files)
- Key-value Storage Systems
  - Document stores (MongoDB, etc)
  - Wide column stores (HBase, Cassandra)
  - Graph Stores (Neo4j)
  - And many others…

Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides single file-system view
- Highly scalable distributed file system for large data-intensive applications.
  - E.g., 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amounts of data on cheap and unreliable computers
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- Examples:
  - Google File System (GFS)
  - Hadoop File System (HDFS)

Hadoop File System Architecture

- Single Namespace for entire cluster
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode



Key-Value Storage Systems

- Unlike HDFS, focus here on storing large numbers (billions or even more) of small (KB-MB) sized records
  - **uninterpreted bytes**, with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - **Wide-table** (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- Records **partitioned** across multiple machines
  - Queries are routed by the system to appropriate machine
- Records **replicated** across multiple machines for fault tolerance as well as efficient querying
  - Need to guarantee "consistency" when data is updated
  - **"Distributed Transactions"**

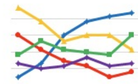
Key-Value Storage Systems

- Key-value stores support
  - ***put***(key, value): used to store values with an associated key,
  - ***get***(key): which retrieves the stored value associated with the specified key
  - ***delete***(key) -- Remove the key and its associated value
- Some support ***range queries*** on key values
- Document stores support richer queries (e.g., MongoDB)
  - Slowly evolving towards the richness of SQL
- Not full database systems (increasingly changing)
  - Have no/limited support for transactional updates
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems, i.e., NoSQL systems

DB-Engines Ranking

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Read more about the method of calculating the scores.



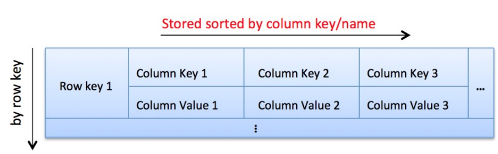
trend chart

360 systems in ranking, November 2020

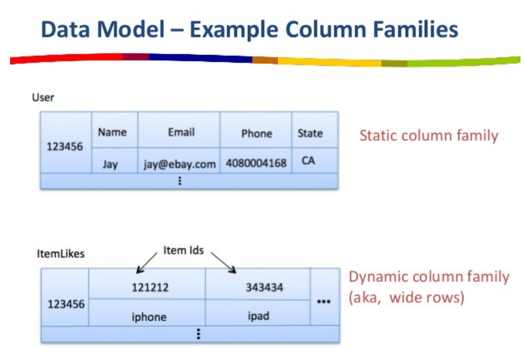
Rank Nov 2020	Rank Oct 2020	Rank Nov 2019	DBMS	Database Model	Score Nov 2020	Score Oct 2020	Score Nov 2019
1.	1.	1.	Oracle	Relational, Multi-model	1345.00	-23.77	+8.93
2.	2.	2.	MySQL	Relational, Multi-model	1241.64	-14.74	-24.64
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1037.64	-5.48	-44.27
4.	4.	4.	PostgreSQL	Relational, Multi-model	555.06	+12.66	+63.99
5.	5.	5.	MongoDB	Document, Multi-model	453.83	+5.81	+40.64
6.	6.	6.	IBM Db2	Relational, Multi-model	161.62	-0.28	-10.98
7.	↑ 8.	↑ 8.	Redis	Key-value, Multi-model	155.42	+2.14	+10.18
8.	↓ 7.	↓ 7.	Elasticsearch	Search engine, Multi-model	151.55	-2.29	+3.15
9.	9.	↑ 11.	SQLite	Relational	123.31	-2.11	+2.29
10.	10.	10.	Cassandra	Wide column	118.75	-0.35	-4.47
11.	11.	↓ 9.	Microsoft Access	Relational	117.23	-1.02	-12.84
12.	12.	↑ 13.	MariaDB	Relational, Multi-model	92.29	+0.52	+6.72
13.	13.	↓ 12.	Splunk	Search engine	89.71	+0.30	+0.64
14.	14.	↑ 15.	Teradata	Relational, Multi-model	75.60	-0.19	-4.75
15.	15.	↓ 14.	Hive	Relational	70.26	+0.71	-13.96
16.	16.	16.	Amazon DynamoDB	Multi-model	68.89	+0.48	+7.52
17.	17.	↑ 25.	Microsoft Azure SQL Database	Relational, Multi-model	66.99	+2.59	+39.37
18.	18.	↑ 19.	SAP Adaptive Server	Relational	55.39	+0.23	+0.10
19.	19.	↑ 20.	SAP HANA	Relational, Multi-model	53.58	-0.66	-1.53
20.	↑ 21.	↑ 22.	Neo4j	Graph	53.53	+2.20	+3.00
21.	↓ 20.	↓ 17.	Solr	Search engine	51.82	-0.66	-5.96
22.	22.	↓ 21.	HBase	Wide column	47.11	-1.25	-6.73
23.	23.	↓ 18.	FileMaker	Relational	46.66	-0.73	-9.07
24.	24.	↑ 27.	Google BigQuery	Relational	35.08	+0.67	+9.64
25.	25.	↓ 24.	Microsoft Azure Cosmos DB	Multi-model	32.50	+0.49	+0.52
26.	26.	↓ 23.	Couchbase	Document, Multi-model	30.55	+0.22	-1.44

Apache Cassandra

- Wide-table key value store



Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

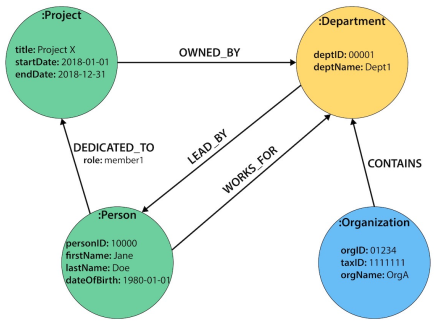



```
CREATE TABLE users_by_username (
    username text PRIMARY KEY,
    email text,
    age int
)

CREATE TABLE users_by_email (
    email text PRIMARY KEY,
    username text,
    age int
)
```

```
cqlsh> select * from University.Student;

 rollno | dept | name    | semester
--------+------+---------+----------
      1 |   CS | Jeegar  |        5
      2 |   CS | Guru99  |        7

(2 rows)
cqlsh>
```

Neo4j

- Graph Database using a Property Graph Model



Cypher

```
MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
LIMIT 10;
```

Cypher   Copy to Clipboard   Run in Neo4j Browser

```
MATCH (p:Product {productName:"Chocolade"})<-[:PRODUCT]-(:Order)<-[:PURCHASED]-(c:Customer)
RETURN distinct c.companyName;
```


Summary

- Traditional databases don't provide the right abstractions for many newer data processing/analytics tasks
- Led to development of NoSQL systems and Map-Reduce (or similar) frameworks
  - Easier to get started
  - Easier to handle ad hoc and arbitrary tasks
  - Not as efficient
- Over the last 10 years, seen increasing convergence
  - NoSQL stores increasingly support SQL constructs like joins and aggregations
  - Map-reduce frameworks also evolved to support joins and SQL more explicitly
  - Databases evolved to support more data types, richer functionality for ad hoc processing
- Think of Map-Reduce systems as another option
  - Appropriate in some cases, not a good fit in other cases

CMSC424: Database Design

Module: Transactions and ACID Properties

Overview

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Overview

- Book Chapters
  - 14.1, 14.2, 14.3, 14.4, 14.5
- Key topics:
  - Transactions and ACID Properties
  - Different states a transaction goes through
  - Notion of a "Schedule"
  - Introduction to Serializability

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $50 from account A to account B:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost
- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

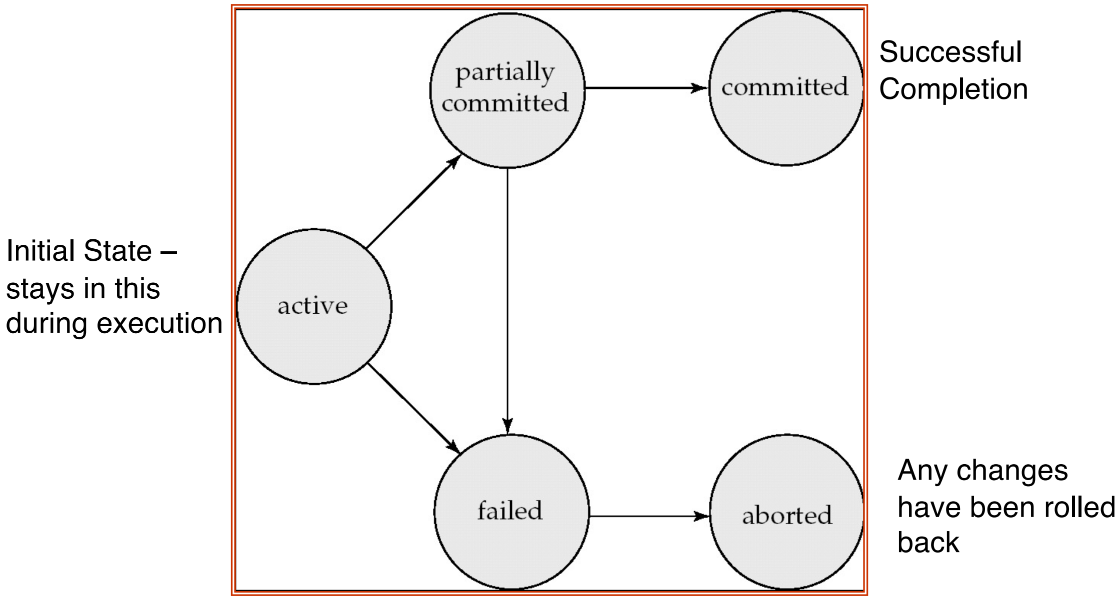
How does..

- .. this relate to *queries* that we discussed ?
  - Queries don't update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing total balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are computing the balance
    - Typically not guaranteed for such long-running queries
- TPC-C vs TPC-H

Assumptions and Goals

- Assumptions:
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won't survive a crash, or power outage
  - BUT… **disks are durable. They might stop, but data is not lost.**
    - For now.
  - Disks only guarantee *atomic sector writes,* nothing more
  - Transactions are by themselves consistent
- Goals:
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance… NO
    - Two transactions updating different account balances… YES

Transaction states



Summary

- Transactions is how we update data in databases
- ACID properties: foundations on which high-performance transaction processing systems are built
  - From the beginning, consistency has been a key requirement
  - Although "relaxed" consistency is acceptable in many cases (originally laid out in 1975)
- NoSQL systems originally eschewed ACID properties
  - MongoDB was famously bad at guaranteeing any of the properties
  - Lot of focus on what's called "eventual consistency"
- Recognition today that strict ACID is more important than that
  - Hard to build any business logic if you have no idea if your transactions are consistent

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency: Basics

Instructor: Amol Deshpande
amol@umd.edu

Concurrency: Basics

- Book Chapters
  - 14.5
- Key topics:
  - Why Concurrency
  - Notion of a "Schedule"
  - Introduction to Serializability

Next…

- Concurrency: Why?
  - Increased processor and disk utilization
  - Reduced average response times
- Concurrency control schemes
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For now, we will assume durability is not a problem
    - So no crashes
    - Though transactions may still abort
- Schedules
- When is concurrency okay ?
  - Serial schedules
  - Serializability

A Schedule

Transactions:

T1: transfers $50 from A to B

T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts)*

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

Each transaction obeys the constraint.

This schedule does too.

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions
- *Serial Schedule:* A schedule in which transaction appear one after the other
  - ie., No interleaving
- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency

Example Schedule

- Another "serial" schedule:

T1	T2
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	

Effect:	Before	After
A	100	40
B	50	110

Consistent ?

Constraint is satisfied.

Since each Xion is consistent, any serial schedule must be consistent

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B + tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Consistent.
So this schedule is okay too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	50
B	50	60

Not consistent

Serializability

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*
- Serializability → schedule is fine and doesn't cause inconsistencies
  - Since serial schedules are fine
- Non-serializable schedules unlikely to result in consistent databases
- We will ensure serializability
  - Typically relaxed in real high-throughput environments
- Not possible to look at all *n!* serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules

Example Schedule with More Transactions

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)				

Summary

- Transactions is how we update data in databases
- ACID properties: foundations on which high-performance transaction processing systems are built
  - From the beginning, consistency has been a key requirement
  - Although “relaxed” consistency is acceptable in many cases (originally laid out in 1975)
- NoSQL systems originally eschewed ACID properties
  - MongoDB was famously bad at guaranteeing any of the properties
  - Lot of focus on what’s called “eventual consistency”
- Recognition today that strict ACID is more important than that
  - Hard to build any business logic if you have no idea if your transactions are consistent

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency: Serializability

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Serializability

- Book Chapters
  - 14.6
- Key topics:
  - Conflict equivalence of schedules
  - Conflict serializability and checking by drawing precedence graphs

An Interleaved schedule

T1	T2
read(A) A = A -50 write(A)	
	read(A) tmp = A*0.1 A = A – tmp write(A)
read(B) B=B+50 write(B)	
	read(B) B = B+ tmp write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1 (T1 before T2)

Called *serializable*

Conflict Serializability

- Two read/write instructions "conflict" if
  - They are by different transactions
  - They operate on the same data item
  - At least one is a "write" instruction
- Why do we care ?
  - If two read/write instructions don't conflict, they can be "swapped" without any change in the final effect
  - However, if they conflict they CAN'T be swapped without changing the final effect

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
read(B)	
	write(A)
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
	read(B)
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

! ==

Effect:	Before	After
A	100	45
B	50	55

Conflict Serializability

- Conflict-equivalent schedules:
  - If S can be transformed into S' through a series of swaps, S and S' are called *conflict-equivalent*
  - *conflict-equivalent guarantees same final effect on the database*
- A schedule S is conflict-serializable if it is conflict-equivalent to a serial schedule

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
read(B)	
B=B+50	
	write(A)
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Example Schedules (Cont.)

A “bad” schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Y: T2's read(A), tmp = A*0.1, A = A – tmp, write(A), read(B)

X: T1's write(A), read(B), B=B+50, write(B)

Can’t move Y below X
read(B) and write(B) conflict

Other options don’t work either

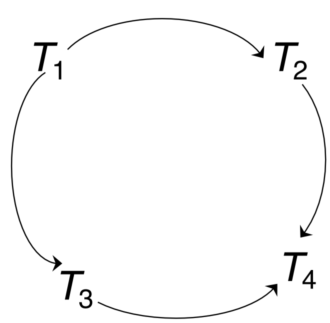
So: Not Conflict Serializable

Testing for conflict-serializability

- Given a schedule, determine if it is conflict-serializable
- Draw a *precedence-graph* over the transactions
  - A directed edge from T1 and T2, if they have conflicting instructions, and T1's conflicting instruction comes first
- If there is a cycle in the graph, not conflict-serializable
  - Can be checked in at most *O(n+e)* time, where *n* is the number of vertices, and *e* is the number of edges
- If there is none, conflict-serializable
- Testing for view-serializability is NP-hard.

Example Schedule (Schedule A) + Precedence Graph

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)				



CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency: View Serializability; Recoverability

Instructor: Amol Deshpande
amol@umd.edu

View Serializability; Recoverability

- Book Chapters
  - 14.6 (last paragraph), 14.7
- Key topics:
  - View serializability
  - Recoverability

Conflict Serializability

- In essence, following set of instructions is not conflict-serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

View-Serializability

- Similarly, following not conflict-serializable

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- BUT, it is serializable
  - Intuitively, this is because the *conflicting write instructions* don't matter
  - The final write is the only one that matters
- View-serializability allows these
  - Read up

Other notions of serializability

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)

- Not conflict-serializable or view-serializable, but serializable
- Mainly because of the +/- only operations
  - Requires analysis of the actual operations, not just read/write operations
- Most high-performance transaction systems will allow these

Recoverability

- Serializability is good for consistency
- But what if transactions fail ?
  - T2 has already committed
    - A user might have been notified
  - Now T1 abort creates a problem
    - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
    - But T2 is *committed*

T1	T2
read(A) A = A -50 write(A)	
	read(A) tmp = A*0.1 A = A – tmp write(A) COMMIT
read(B) B=B+50 write(B) ABORT	

Recoverability

- Recoverable schedule: If T1 has read something T2 has written, T2 must commit before T1
  - Otherwise, if T1 commits, and T2 aborts, we have a problem
- Cascading rollbacks: If T10 aborts, T11 must abort, and hence T12 must abort and so on.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Recoverability

- *Dirty read*: Reading a value written by a transaction that hasn't committed yet
- Cascadeless schedules:
  - A transaction only reads *committed* values.
  - So if T1 has written A, but not committed it, T2 can't read it.
    - *No dirty reads*
- Cascadeless → No cascading rollbacks
  - That's good
  - We will try to guarantee that as well

Recap so far…

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability
  - Recoverability, cascade-less schedules
- We haven't discussed:
  - How to guarantee serializability ?
    - Allowing transactions to run, and then aborting them if the schedules wasn't serializable is clearly not the way to go
  - We instead use schemes to guarantee that the schedule will be conflict-serializable

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency Control: Locking - 1

Instructor: Amol Deshpande
amol@umd.edu

Locking - 1

- Book Chapters
  - 15.1.1-15.1.4
- Key topics:
  - Using locking to guarantee concurrency
  - 2-Phase Locking (2PL)
  - Implementation of locking

Approach, Assumptions etc..

- Approach
  - Guarantee conflict-serializability by allowing certain types of concurrency
    - Lock-based
- Assumptions:
  - Durability is not a problem
    - So no crashes
    - Though transactions may still abort
- Goal:
  - Serializability
  - Minimize the bad effect of aborts (cascade-less schedules only)

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - *Shared* (S) locks (also called *read locks)*
    - Obtained if we want to only read an item – lock-S() instruction
  - *Exclusive* (X) locks (also called *write locks)*
    - Obtained for updating a data item – lock-X() instruction

T1	T2
read(B) B ← B-50 write(B) read(A) A ← A + 50 write(A)	read(A) read(B) display(A+B)

⇨

T1	T2
lock-X(B) read(B) B ← B-50 write(B) unlock(B) lock-X(A) read(A) A ← A + 50 write(A) unlock(A)	lock-S(A) read(A) unlock(A) lock-S(B) read(B) unlock(B) display(A+B)

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it ?
  - Depends

T2 lock type	**T1 lock type**	**Should allow ?**
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible,* grant the lock, otherwise T1 waits in a *queue.*

Lock-based Protocols

- How do we actually use this to guarantee serializability/recoverability ?
  - Not enough just to take locks when you need to read/write something

T1

lock-X(B)
read(B)
B ←B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ←A + 50
write(A)
unlock(A)

lock-X(A), lock-X(B)
TMP = (A + B) * 0.1
A = A - TMP
B = B + TMP
unlock(A), unlock(B)

NOT SERIALIZABLE

2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking phase
  - Transaction may only release locks
- Can be shown that this achieves *conflict-serializability*
  - *lock-point*: the time at which a transaction acquired last lock
  - if *lock-point*(T1) < *lock-point*(T2), there can't be an edge from T2 to T1 in the *precedence graph*

T1

lock-X(B)
read(B)
B ←B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ←A + 50
write(A)
unlock(A)

2 Phase Locking

- Example: T1 in 2PL

	T1
Growing phase	lock-X(B)
	read(B)
	B ← B - 50
	write(B)
	lock-X(A)
	read(A)
	A ← A - 50
	write(A)
Shrinking phase	unlock(B)
	unlock(A)

2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

T1	T2	T3
lock-X(A), lock-S(B)		
read(A)		
read(B)		
write(A)		
unlock(A), unlock(B)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
	Commit	lock-S(A)
		read(A)
		Commit
<xction fails>		

2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability
- Guaranteeing just recoverability:
  - If T2 reads a dirty data of T1 (ie, T1 has not committed), then T2 can't commit unless T1 either commits or aborts
  - If T1 commits, T2 can proceed with committing
  - If T1 aborts, T2 must abort
    - So cascades still happen

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort

T1	T2	T3
lock-X(A), lock-S(B)		
read(A)		
read(B)		
write(A)		
unlock(A), unlock(B)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
	Commit	lock-S(A)
		read(A)
		Commit
<xction fails>		

Strict 2PL will not allow that

Works. Guarantees cascade-less and recoverable schedules.

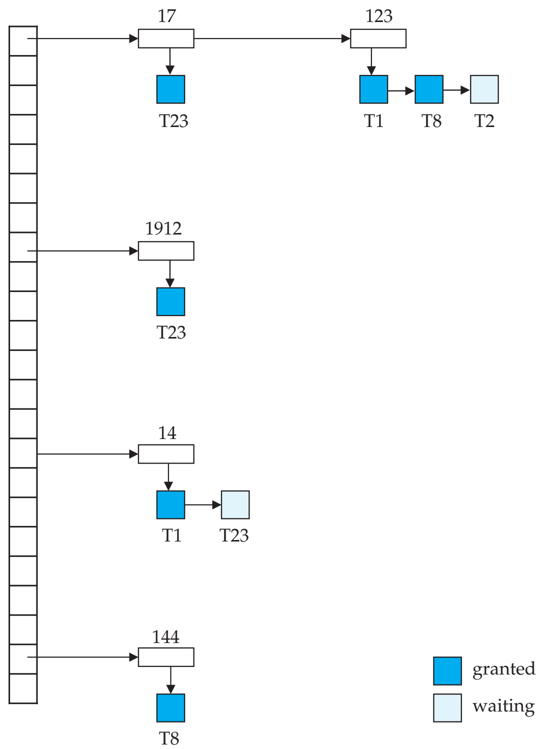
Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
  - Read locks are not important
- Rigorous 2PL: Release both *exclusive and read* locks only at the very end
  - The serializability order === the commit order
  - More intuitive behavior for the users
    - No difference for the system
- Lock conversion:
  - Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
  - Doesn't change any of the other properties of the protocol

Implementation of Locking

- A separate process, or a separate module
- Uses a *lock table* to keep track of currently assigned locks and the requests for locks

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Recap so far…

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc
- Lock-based protocols
  - Use *locks* to prevent multiple transactions accessing the same data items
- 2 Phase Locking
  - Locks acquired during *growing phase,* released during *shrinking phase*
- Strict 2PL, Rigorous 2PL

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency Control:
Locking - 2

Instructor: Amol Deshpande
amol@umd.edu

Locking - 2

- Book Chapters
  - 15.2
- Key topics:
  - Deadlocks and how 2PL doesn't prevent them
  - Deadlock detection through precedence graphs
  - Deadlock avoidance/prevention schemes

More Locking Issues: Deadlocks

- No xction proceeds:

Deadlock

- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

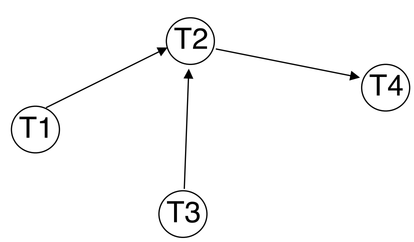
Rollback transactions
Can be costly...

- 2PL does not prevent deadlock
  - Strict doesn't either

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock detection and recovery

- Instead of trying to prevent deadlocks, let them happen and deal with them if they happen
- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - Ti waiting for Tj



T1	T2	T3	T4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	

Suppose T4 requests lock-S(Z)....

Dealing with Deadlocks

- Deadlock detected, now what ?
  - Will need to abort some transaction
  - Prefer to abort the one with the minimum work done so far
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continueing

Preventing deadlocks

- Solution 1: A transaction must acquire all locks before it begins
  - Not acceptable in most cases
  - Still need some way to deal with deadlocks during lock acquisition
- Solution 2: A transaction must acquire locks in a particular order over the data items
  - Also called *graph-based protocols*
  - The particular order used doesn't matter (e.g., based on the value of some unique attribute)
  - Guarantees that there can never be a cycle in the precedence graph

Preventing deadlocks

- Solution 3: Use time-stamps; say T1 is older than T2
  - *wait-die scheme:* T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
    - In the precedence graph, there can be an edge from old transaction to a new transaction, but never the other way
    - So there cannot be a cycle in precedence graph
  - *wound-wait scheme:* T1 will *wound* T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.
    - Similar to above: edges only from newer transactions to older transactions
  - May abort more transactions that needed
- Solution 4: Timeout based
  - Transaction waits a certain time for a lock; aborts if it doesn't get it by then
  - As above, may lead to unnecessary restarts, but very simple to implement

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency Control:
Locking - 3

Instructor: Amol Deshpande
amol@umd.edu

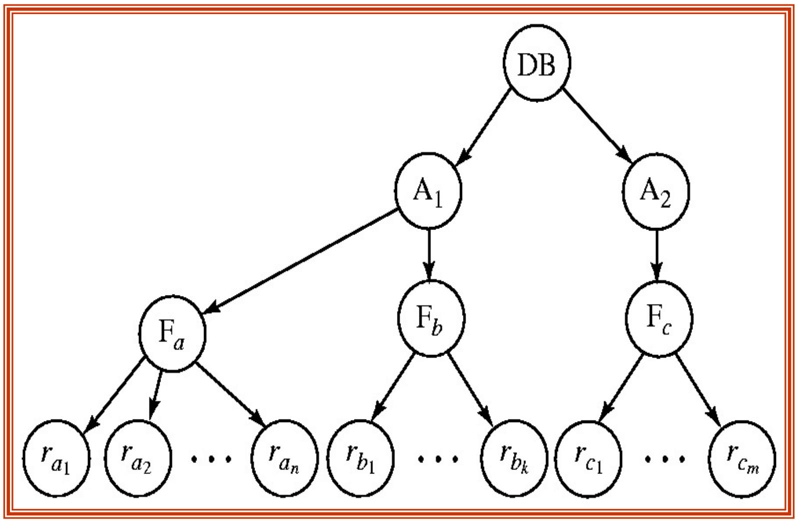
Locking - 3

- Book Chapters
  - 15.3
- Key topics:
  - What are we taking locks on
  - Multi-granularity locking
  - Intentional locks and compatibility

Locking granularity

- Locking granularity
  - What are we taking locks on ? Tables, tuples, attributes ?
- Coarse granularity
  - e.g. take locks on tables
  - less overhead (the number of tables is not that high)
  - very low concurrency
- Fine granularity
  - e.g. take locks on tuples
  - much higher overhead
  - much higher concurrency
  - What if I want to lock 90% of the tuples of a table ?
    - Prefer to lock the whole table in that case

Granularity Hierarchy



The highest level in the example hierarchy is the entire database.

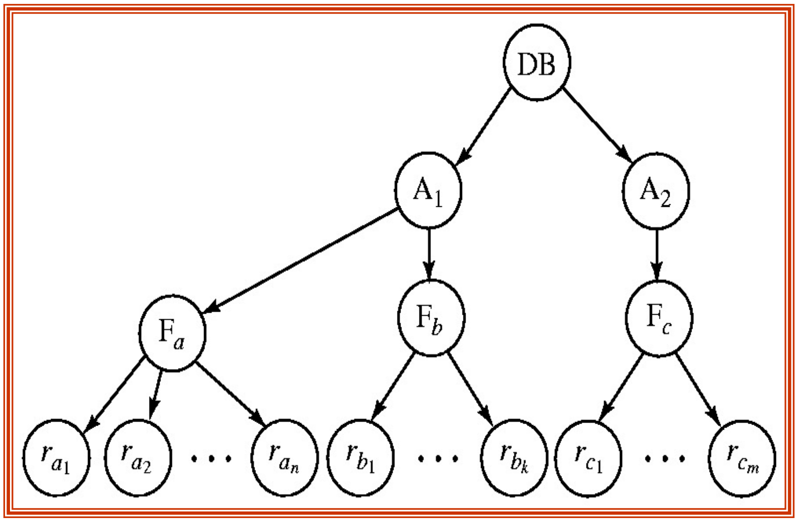
The levels below are of type *area*, *file or relation* and *record* in that order.

Can lock at any level in the hierarchy

Granularity Hierarchy

- New lock mode, called *intentional* locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: *intention shared*
    - The lower levels below may be locked in the shared mode
  - IX: *intention exclusive*
  - *SIX: shared and intention-exclusive*
    - The entire subtree is locked in the shared mode, but I might also want to get exclusive locks on the nodes below
- Protocol:
  - If you want to acquire a lock on a data item, all the ancestors must be locked as well, at least in the intentional mode
  - So you always start at the top *root* node

Granularity Hierarchy



(1) Want to lock *F_a* in shared mode, *DB* and *A1* must be locked in at least IS mode (but IX, SIX, S, X are okay too)

(2) Want to lock *rc1* in exclusive mode, *DB, A2,Fc* must be locked in at least IX mode (SIX, X are okay too)

Multi-granularity Locking

- Rules for Multi-granularity Locking
  - Always start with the root
  - Can lock Q in S or IS, only if parent is locked in IS or IX mode
  - Can lock Q in X, SIX, or IX only if parent is locked in IX or SIX mode
  - Must follow 2-phase locking protocol
  - Unlock Q only if locks on all children (if any) are released
    - i.e., unlock from the bottom up

- However: it is not a problem to lock a child in, say S, if the parent is in SIX
  - It is redundant, but may happen because of “lock upgrades”
  - Depending on implementation, may release the child lock or not

Compatibility Matrix with Intention Lock Modes

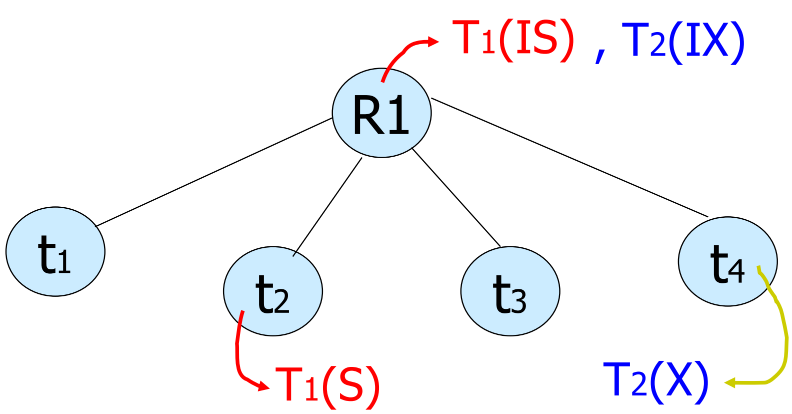
- The compatibility matrix (which locks can be present simultaneously on the same data item) for all lock modes is:

requestor

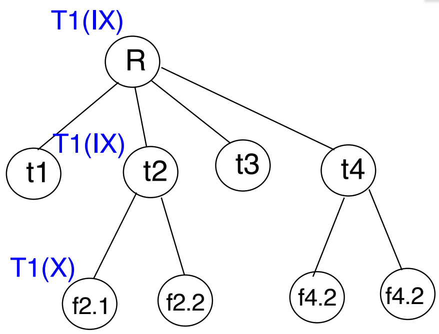
holder

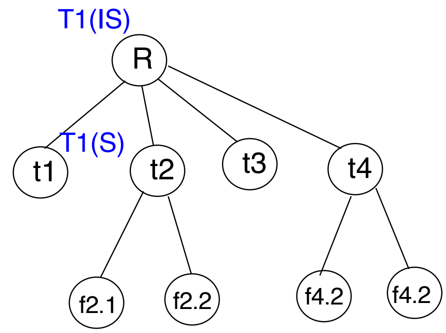
	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Example

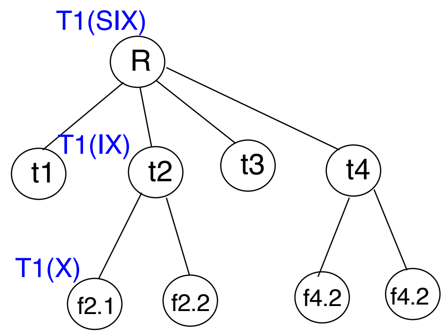


Examples





Can T2 access object f2.2 in X mode?
What locks will T2 get?



Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then occasionally upgrades to X on the specific tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can

  use lock escalation to decide which.

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Recap: Locking-based CC

- Key idea: Take locks as required to ensure conflict serializability
- 2-phase locking, and Strict and Rigorous 2PL
- Deadlocks and how to prevent or detect them
- Multi-granularity locking
- Many commercial databases support locking-based CC, but increasingly multi-version concurrency control more common
  - Locking expensive in comparison, and supports lower concurrency than MVCC techniques (like Snapshot Isolation)

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency Control: Other Schemes

Instructor: Amol Deshpande
amol@umd.edu

1. Time-stamp Based

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - So if T1 entered before T2, then T1 should be before T2 in the serializability order
  - Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

1. Time-stamp Based

- Maintain for each data Q, two timestamps:
  - W-timestamp(Q): largest time-stamp of any transaction that executed Write(Q) successfully
  - R-timestamp(Q): largest time-stamp of any transaction that executed Read(Q) successfully

- Suppose Ti wants to read(Q):
  - If TS(T_i) < W-Timestamp(Q): Reject the operation and roll back T_i
  - Otherwise, allow the operation and modify:
    - R-timestamp(Q) = max(R-timestamp(Q)), TS(T_i))

1. Time-stamp Based

- Maintain for each data Q, two timestamps:
  - W-timestamp(Q): largest time-stamp of any transaction that executed Write(Q) successfully
  - R-timestamp(Q): largest time-stamp of any transaction that executed Read(Q) successfully

- Suppose Ti wants to write(Q):
  1. If TS(T_i) < R-timestamp(Q): reject the write and roll back T_i
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, execute **write**, and W-timestamp(Q) is set to TS(T_i).

1. Example of Schedule Under TSO

- Is this schedule valid under TSO?

  Assume that initially:
  R-TS(A) = W-TS(A) = 0
  R-TS(B) = W-TS(B) = 0
  Assume TS(T_{25}) = 25 and
  TS(T_{26}) = 26

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

- How about this one, where initially R-TS(Q)=W-TS(Q)=0

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

1. Another Example

- Example

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

1. Recoverability and Cascade Freedom

- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2:
  - Limited form of locking: wait for data to be committed before reading it
- Solution 3:
  - Use commit dependencies to ensure recoverability (i.e., require them to commit in some order)

1. Thomas' Write Rule

- Ignore obsolete **write** operations 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}.
  - Rather than rolling back T_i, this {**write**} operation can be ignored.

- Allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

2. Optimistic Concurrency Control

- Optimistic concurrency control
  - Also called validation-based

  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart

  - Optimistic: The hope is that there won't be too many problems/aborts

2. Optimistic Concurrency Control

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

2. Optimistic Concurrency Control

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

2. Optimistic Concurrency Control

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B \quad 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

3. Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol
- A type of "multi-version concurrency control"
  - Also similar to optimistic concurrency control in many ways
- Key idea:
  - For each object, maintain past "versions" of the data along with timestamps
    - Every update to an object causes a new version to be generated

3. Snapshot Isolation

- Read queries:
  - Let "t" be the "time-stamp" of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of "t"
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

3. Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

3. Snapshot Isolation

- Advantages:
  - Read query don't block at all, and run very fast
  - As long as conflicts are rare, update transactions don't abort either
  - Overall better performance than locking-based protocols
- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
    - http://en.wikipedia.org/wiki/Snapshot_isolation

3. Snapshot Isolation

- Example of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

3. SI In Oracle and PostgreSQL

- **Warning**: SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1
  - PostgreSQL's implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
    - Oracle and PostgreSQL < 9.1 do not support true serializable execution
  - PostgreSQL 9.1 introduced new protocol called "Serializable Snapshot Isolation" (SSI)
    - Which guarantees true serializabilty including handling predicate reads (coming up)

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency Control: Phantom Problem; Weak Levels of Isolations

Instructor: Amol Deshpande
amol@umd.edu

Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - **select count**(*)
      from *instructor*
      where *dept_name* = 'Physics'
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - **insert into** *instructor* **values** ('11111', 'Feynman', 'Physics', 94000)

    (conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
  - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction
- Can also occur with updates
  - E.g. update Wu's department from Finance to Physics

Insert/Delete Operations and Predicate Reads

- Another Example Schedule with a problem
  - T1 saw a partial update of T2, but not the full update
  - So not serializable

T1	T2
Read(instructor where dept_name='Physics')	
	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
Read(instructor where dept_name='Comp. Sci.')	

Insert/Delete Operations and Predicate Reads

- **Another Example**: T1 and T2 both find maximum instructor ID in
parallel, and create new instructors with ID = maximum ID + 1
  - Both instructors get same ID, not possible in serializable schedule

Index Locking To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction T_i that inserts, updates or deletes a tuple t_i in a relation *r*
    - Must update all indices to *r*
    - Must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

Weak Levels of Consistency

- **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Guarantees no "dirty reads" (so no recoverability issues)
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency

T_{32}	T_{33}
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

Figure 18.21 Nonserializable schedule with degree-two consistency.

The "Phantom" problem

- An interesting problem that comes up for dynamic databases
- Schema: *accounts(acct_no, balance, zipcode, …)*
- Transaction 1: Find the number of accounts in *zipcode = 20742,* and divide $1,000,000 between them
- Transaction 2: Insert *<acctX, …, 20742, …>*
- Execution sequence:
  - T1 locks all tuples corresponding to "zipcode = 20742", finds the total number of accounts (= num_accounts)
  - T2 does the insert
  - T1 computes 1,000,000/num_accounts
  - When T1 accesses the relation again to update the balances, it finds one new ("phantom") tuples (the new tuple that T2 inserted)
- Not serializable

Weak Levels of Consistency

- **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable:** is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read

- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - **set isolation level serializable**

Summary

- Concurrency control schemes help guarantee isolation while allowing for concurrent transactions

- Many different schemes developed over the years
  - Lock-based, Timestamp-based, Snapshot Isolation, Optimistic

- Lot of new work in the recent years because of shifting hardware trends
  - E.g., locking performance overheads quite significant

- Many NoSQL systems still have limited concurrency

- Important to consider recovery schemes at the same time

CMSC424: Database Design

Module: Transactions and ACID Properties

Recovery: Overview; Terminology; Steal and Force

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Recovery

- Book Chapters
  - 16.1, 16.2, 16.3.2
- Key topics:
  - Challenges in guaranteeing Atomicity and Durability
  - Basics of how disks and memory interact
  - New operations: Output() and Input()
  - STEAL and NO FORCE: Why those are desirable
  - Terminology used in the book: Immediate vs Deferred Modifications

Context

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability ?
    - Atomicity: Two problems
      - Part of the transaction is done, but we want to cancel it
        - ABORT/ROLLBACK
      - System crashes during the transaction. Some changes made it to the disk, some didn't.
    - Durability:
      - Transaction finished. User notified. But changes not sent to disk yet (for performance reasons). System crashed.
- Essentially similar solutions

Reasons for crashes

- Transaction failures
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash
  - Power failures, operating system bugs etc
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure
  - Head crashes; ***for now we will assume***
    - ***STABLE STORAGE: Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data***

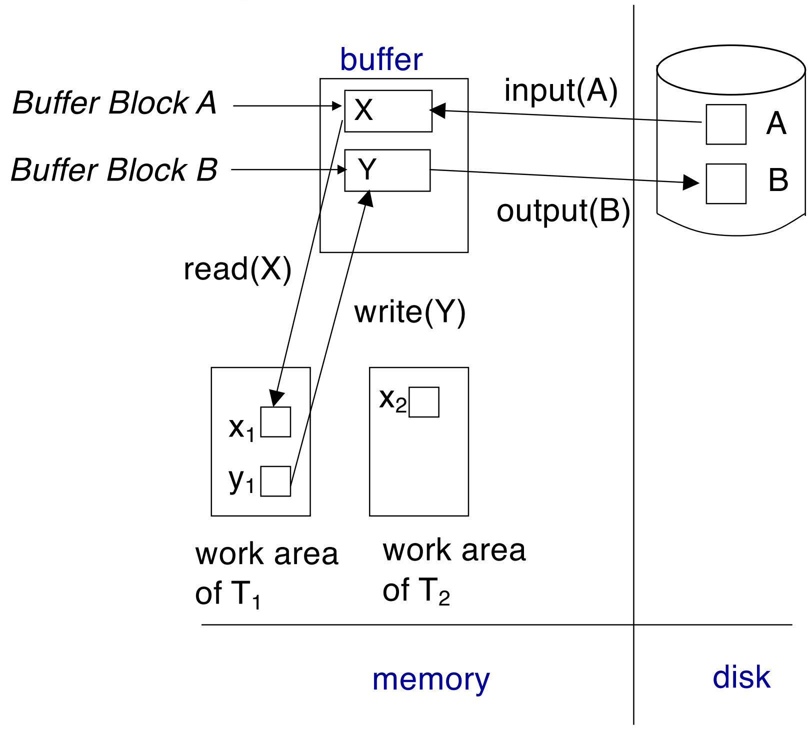
Approach, Assumptions etc..

- Approach:
  - Guarantee A and D:
    - by controlling how the disk and memory interact,
    - by storing enough information during normal processing to recover from failures
    - by developing algorithms to recover the database state
- Assumptions:
  - System may crash, but the *disk is durable*
  - The only *atomicity* guarantee is that *a disk block write* is *atomic*
- Once again, obvious naïve solutions exist that work, but that are too expensive.
  - E.g. The shadow copy solution
    - Make a copy of the database; do the changes on the copy; do an atomic switch of the *dbpointer* at commit time
  - Goal is to do this as efficiently as possible

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, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

STEAL vs NO STEAL, FORCE vs NO FORCE

- STEAL:
  - The buffer manager *can steal* a (memory) page from the database
    - ie., it can write an arbitrary page to the disk and use that page for something else from the disk
    - In other words, the database system doesn't control the buffer replacement policy
  - Why a problem ?
    - The page might contain *dirty writes,* ie., writes/updates by a transaction that hasn't committed
  - But, we must allow *steal* for performance reasons.
- NO STEAL:
  - Not allowed. More control, but less flexibility for the buffer manager.

STEAL vs NO STEAL, FORCE vs NO FORCE

- FORCE:
  - The database system *forces* all the updates of a transaction to disk before committing
  - Why ?
    - To make its updates permanent before committing
  - Why a problem ?
    - Most probably random I/Os, so poor response time and throughput
    - Interferes with the disk controlling policies
- NO FORCE:
  - Don't do the above. Desired.
  - Problem:
    - Guaranteeing durability becomes hard
  - We might still have to *force* some pages to disk, but minimal.

STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

	No Steal	Steal
No Force		Desired
Force	Trivial	

STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

- How to implement A and D when No Steal and Force ?
  - Only updates from committed transaction are written to disk (since no steal)
  - Updates from a transaction are forced to disk before commit (since force)
    - A minor problem: how do you guarantee that all updates from a transaction make it to the disk atomically ?
      - Remember we are only guaranteed an atomic *block write*
      - What if some updates make it to disk, and other don't ?
    - Can use something like shadow copying/shadow paging
  - No atomicity/durability problem arise.

Terminology

- Deferred Database Modification:
  - Similar to NO STEAL, NO FORCE
    - Not identical
  - Only need *redos, no undos*
  - We won't cover this in detail
- Immediate Database Modification:
  - Similar to STEAL, NO FORCE
  - Need both *redos, and undos*

CMSC424: Database Design

Module: Transactions and ACID Properties

Recovery: Basics of Logging and UNDO

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Recovery

- Book Chapters
  - 16.3.1, 16.3.5
- Key topics:
  - Generating log records
  - Using log records to support UNDO/Rollback

Log-based Recovery

- Most commonly used recovery method
- Intuitively, a log is a record of everything the database system does
- For every operation done by the database, a *log record* is generated and stored *typically on a different (log) disk*
- *<T1, START>*
- *<T2, COMMIT>*
- *<T2, ABORT>*
- *<T1, A, 100, 200>*
  - T1 modified A; old value = 100, new value = 200

Log

- Example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (*A*)
A: - A - 50
write (*A*)
read (*B*)
B:- B + 50
write (*B*)

T_1 : **read** (*C*)
C:- C- 100
write (*C*)

- Log:

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

Log-based Recovery

- Assumptions:
  1. Log records are immediately pushed to the disk as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated *before* the actual data value is updated
  4. *Strict two-phase locking*
  - ★ The first assumption can be relaxed
  - ★ As a special case, a transaction is considered *committed* only after the *<T1, COMMIT> has been pushed to the disk*
- But, this seems like exactly what we are trying to avoid ??
  - ★ Log writes are *sequential*
  - ★ They are also typically on a different disk
- Aside: LFS == log-structured file system

Log-based Recovery

- Assumptions:
  1. Log records are immediately pushed to the disk as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated *before* the actual data value is updated
  4. *Strict two-phase locking*
  - The first assumption can be relaxed
  - As a special case, a transaction is considered *committed* only after the *<T1, COMMIT> has been pushed to the disk*
- NOTE: As a result of assumptions 1 and 2, if *data item A* is updated, the log record corresponding to the update is always forced to the disk before *data item A* is written to the disk
  - This is actually the only property we need; assumption 1 can be relaxed to just guarantee this (called *write-ahead logging)*

Using the log to *abort/rollback*

- STEAL is allowed, so changes of a transaction may have made it to the disk

- UNDO(T1):
  - Procedure executed to *rollback/undo* the effects of a transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - T1 decides to abort

  - Any of the changes might have made it to the disk

Using the log to *abort/rollback*

- UDO(T1):
  - Go *backwards* in the *log* looking for log records belonging to T1
  - Restore the values to the old values
  - NOTE: Going backwards is important.
    - *A* was updated twice
  - In the example, we simply:
    - Restore A to 300
    - Restore B to 400
    - Restore A to 200
  - Write a log record $<T_i , X_j, V_1>$
    - such log records are called **compensation log records**
    - **<T1, A, 300>, <T1, B, 400>, <T1, A, 200>**
  - Note: No other transaction better have changed A or B in the meantime
    - *Strict two-phase locking*

CMSC424: Database Design

Module: Transactions and ACID Properties

Recovery: Log-based Restart Recovery

Instructor: Amol Deshpande
amol@umd.edu

Using Logs for Recovery

- Book Chapters
  - 16.4
- Key topics:
  - How to use logs for REDO
  - Idempotency of log records
  - Restart recovery after a failure

Using the log to *recover*

- We don't require FORCE, so a change made by the committed transaction may not have made it to the disk before the system crashed
  - BUT, the log record did (recall our assumptions)
- REDO(T1):
  - Procedure executed to recover a committed transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - *<T1, COMMIT>*
  - By our assumptions, all the log records made it to the disk (since the transaction committed)
  - But any or none of the changes to A or B might have made it to disk

Using the log to *recover*

- REDO(T1):
  - Go *forwards* in the *log* looking for log records belonging to T1
  - Set the values to the new values
  - NOTE: Going forwards is important.
  - In the example, we simply:
    - Set A to 300
    - Set B to 300
    - Set A to 200

Idempotency

- Both redo and undo are required to *idempotent*
  - *F is idempotent, if F(x) = F(F(x)) = F(F(F(F(…F(x)))))*
- Multiple applications shouldn't change the effect
  - This is important because we don't know exactly what made it to the disk, and we can't keep track of that
  - E.g. consider a log record of the type
    - <T1, A, *incremented by 100*>
    - Old value was 200, and so new value was 300
  - But the on disk value might be 200 or 300 (since we have no control over the buffer manager)
  - So we have no idea whether to apply this log record or not
  - Hence, *value based logging* is used (also called *physical),* not operation based (also called *logical)*

Log-based recovery

- Log is maintained
- If during the normal processing, a transaction needs to abort
  - UNDO() is used for that purpose
- If the system crashes, then we need to do recovery using both UNDO() and REDO()
  - Some transactions that were going on at the time of crash may not have completed, and must be *aborted/undone*
  - Some transaction may have committed, but their changes didn't make it to disk, so they must be *redone*
  - Called *restart recovery*

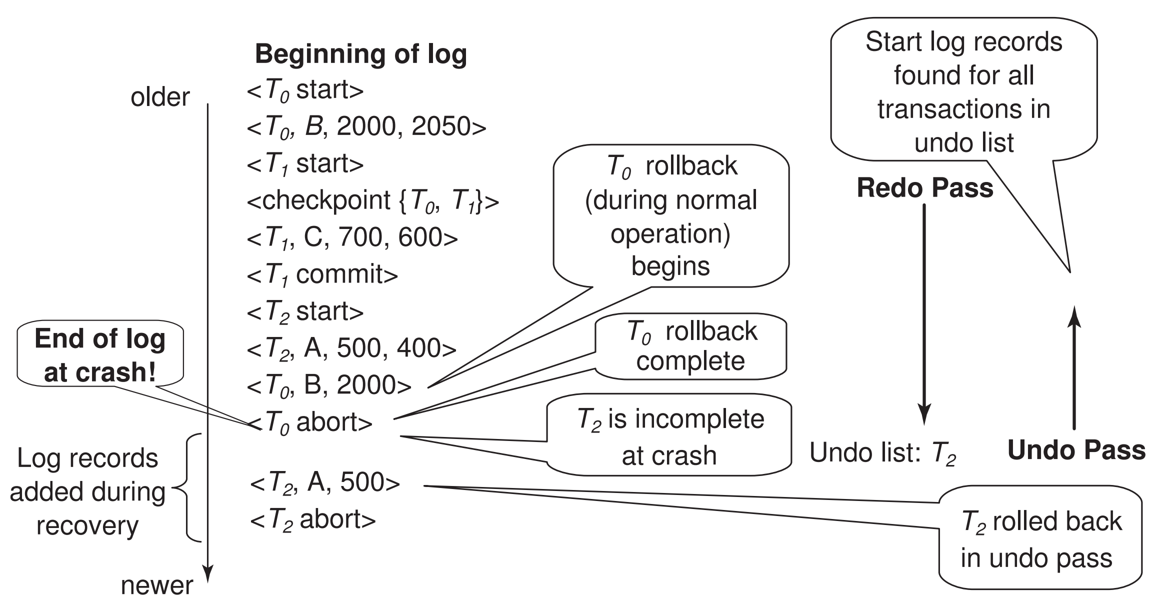
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Set undo-list to *{} (empty)*.
  2. Scan forward from first log record
     1. Whenever a record <T_i, X_j, V_1, V_2> is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record $<T_i, X_j, V_1>$
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. <T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery



CMSC424: Database Design

Module: Transactions and ACID Properties

Checkpointing; Write-ahead Logging; Recap

Instructor: Amol Deshpande
amol@umd.edu

Recovery: Recap

- Book Chapters
  - 16.3.6, 16.5
- Key topics:
  - Checkpointing
  - Write-ahead logging
  - Recap

Checkpointing

- How far should we go back in the log while constructing redo and undo lists ??
  - It is possible that a transaction made an update at the very beginning of the system, and that update never made it to disk
    - very very unlikely, but possible (because we don't do force)
  - For correctness, we have to go back all the way to the beginning of the log
  - Bad idea !!

- Checkpointing is a mechanism to reduce this

Checkpointing

- Periodically, the database system writes out everything in the memory to disk
  - Goal is to get the database in a state that we know (not necessarily consistent state)
- Steps:
  - Stop all other activity in the database system
  - Write out the entire contents of the memory to the disk
    - Only need to write updated pages, so not so bad
    - Entire === all updates, whether committed or not
  - Write out all the log records to the disk
  - Write out a special log record to disk
    - *<CHECKPOINT LIST_OF_ACTIVE_TRANSACTIONS>*
    - The second component is the list of all active transactions in the system right now
  - Continue with the transactions again

Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions

- **Redo phase** (No difference for Undo phase):
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.

     - If no checkpoint record, start at the beginning
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recap so far …

- Log-based recovery
  - Uses a *log* to aid during recovery

- UNDO()
  - Used for normal transaction abort/rollback, as well as during restart recovery

- REDO()
  - Used during restart recovery

- Checkpoints
  - Used to reduce the restart recovery time

Write-ahead logging

- We assumed that log records are written to disk as soon as generated
  - Too restrictive
- Write-ahead logging:
  - Before an update on a data item (say A) makes it to disk, the log records referring to the update must be forced to disk
  - How ?
    - Each log record has a log sequence number (LSN)
      - Monotonically increasing
    - For each page in the memory, we maintain the LSN of the *last log record* that updated a record on this page
      - *pageLSN*
    - If a page *P* is to be written to disk, all the log records till *pageLSN(P)* are forced to disk

Write-ahead logging

- Write-ahead logging (WAL) is sufficient for all our purposes
  - All the algorithms discussed before work
- Note the special case:
  - A transaction is not considered committed, unless the <T, commit> record is on disk

Other issues

- The system halts during checkpointing
  - Not acceptable
  - Advanced recovery techniques allow the system to continue processing while checkpointing is going on
- System may crash during recovery
  - Our simple protocol is actually fine
  - In general, this can be painful to handle
- B+-Tree and other indexing techniques
  - Strict 2PL is typically not followed (we didn't cover this)
  - So physical logging is not sufficient; must have logical logging

Other issues

- ARIES: Considered *the canonical description of log-based recovery*
  - Used in most systems
  - Has many other types of log records that simplify recovery significantly
- Loss of disk:
  - Can use a scheme similar to checkpoining to periodically dump the database onto *tapes* or *optical storage*
  - Techniques exist for doing this while the transactions are executing (called *fuzzy dumps)*
- Shadow paging:
  - Read up

Recap

- STEAL vs NO STEAL, FORCE vs NO FORCE
  - We studied how to do STEAL and NO FORCE through log-based recovery scheme

	No Steal	Steal
No Force		Desired
Force	Trivial	

	No Steal	Steal
No Force	REDO NO UNDO	REDO UNDO
Force	NO REDO NO UNDO	NO REDO UNDO

Recap

- ACID Properties
  - Atomicity and Durability :
    - Logs, undo(), redo(), WAL etc
  - Consistency and Isolation:
    - Concurrency schemes
  - Strong interactions:
    - We had to assume Strict 2PL for proving correctness of recovery

CMSC424: Database Design

Module: Transactions and ACID Properties

Distributed Transactions

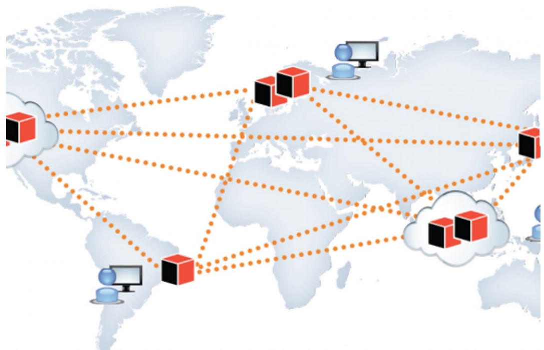
Instructor: Amol Deshpande
amol@umd.edu

Distributed Transactions

- Book Chapters
  - 19.1-19.4, 19.6: at a fairly high level
- Key topics:
  - Distributed databases and replication
  - Transaction processing in distributed databases
  - 2-Phase Commit
  - Brief discussion of other protocols including Paxos

Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
  - Or not – lot of variations here
- Transactions may access data at one or more sites
  - Because of replication, even updating a single data item involves a "distributed transaction" (to keep all replicas up to date)

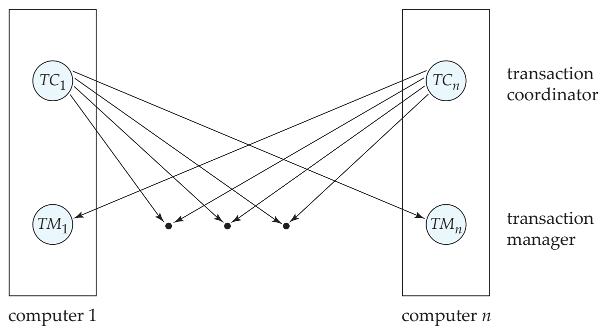


Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites
- Advantages:
  - **Availability**: failures can be handled through replicas
  - **Parallelism**: queries can be run on any replica
  - **Reduced data transfer**: queries can go to the "closest" replica
- Disadvantages:
  - **Increased cost of updates:** both computation as well as latency
  - **Increased complexity of concurrency control:** need to update all copies of a data item/tuple
- **Typically we use the term "data items", which may be tuples or relations or relation partitions**

Distributed Transactions

- Transaction may access data at several sites
  - As noted, single data item update is also a distributed transaction
- **Each site has a local transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the concurrent execution of the transactions
- **Each site has a transaction coordinator,** which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing sub-transactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site -- transaction may commit at all sites or abort at all sites.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- **Two-phase *commit* (2PC)** protocol is widely used
- **Three-phase commit (3PC)** protocol
  - Handles some situations that 2PC doesn't
  - Not widely used
- **Paxos**
  - Robust alternative to 2PC that handles more situations as well
  - Was considered too expensive at one point, but widely used today
- **RAFT**: Alternative to Paxos

Two Phase Commit Protocol (2PC)

- Assumes **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
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Two Phase Commit Protocol (2PC)

Coordinator Log	Messages	Subordinate Log
	PREPARE →	
		prepare*/abort*
	← VOTE YES/NO	
commit*/abort*		
	COMMIT/ABORT→	
		commit*/abort*
	← ACK	
end		

Goal: Make sure all "sites" commit or abort

Assumption: Some log records can be "forced" (denote * above)

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <**ready** *T*> to the log
  - force *all records* for *T* to stable storage
  - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed of C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: txn had completed, nothing to be done
- Log contains <**abort** *T*> record: txn had completed, nothing to be done
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*); write <**commit** *T*> record
  - If *T* aborted, **undo** (*T*)
- The log contains no log records concerning *T*:
  - Implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response, coordinator C_1 must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*.
     - Can therefore abort *T;* however, such a site must reject any subsequent <**prepare** *T*> message from C_i
  4. 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 C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- 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

More…

- Three-phase Commit
  - 2PC can't handle failure of a coordinator well – everything halts waiting for the coordinator to come back up
  - Three-phase commit handles that through another phase
- Paxos and RAFT
  - Solutions for the "consensus problem": get a collection of distributed entities to "choose" a single value
    - In case of transaction, you are choosing abort/commit
  - Fairly complex, but well-understood today
  - Widely used in most distributed systems today
  - See the Wikipedia pages
  - A nice recent paper: **Paxos vs Raft: Have we reached consensus on distributed consensus? – Heidi Howard, 2020**

More…

- Bitcoin (and other cryptocurrencies)
  - Fundamental problem is the same one, of obtaining "consensus"
    - But need to support a large number of entities, 1000s or more
    - Can't assume full one-to-one communication
  - Instead:
    - Choose a "leader" based on "proof of work"
      - Whoever solves a hard puzzle first becomes the "leader"
    - The "leader" chooses the next "block" in the blockchain
      - A block is basically a list of transactions to accept
    - Reward the puzzle solvers with money ("bitcoins")
      - So they have an incentive to keep solving puzzles
  - Blockchain?
    - Blockchain is a small part of bitcoin
    - A cryptographically designed chain of blocks that are immutable