Computer Science E-66
Database Systems

Harvard Extension School, Fall 2024

Cody Doucette, Ph.D.

Introduction; Database Design, ER Models, and the Relational Model	2
Relational Algebra and SQL	35
Storage and Indexing	84
Implementing a Logical-to-Physical Mapping	108
Transactions and Schedules	121
Concurrency Control	138
Semistructured Data and XML	164
Distributed Databases and Replication	185
Processing Distributed Data Using MapReduce	201
NoSQL Databases	218
Recovery and Logging	251
Performance Tuning; Wrap-up and Conclusions	282

Computer Science E-66

Introduction
Database Design and ER Models
The Relational Model

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Databases and DBMSs

- A *database* is a collection of related data.
  - refers to the data itself, *not* the program
- Managed by some type of *database management system* (DBMS)

The Conventional Approach

- Use a DBMS that employs the *relational model*
  - use the SQL query language
- Examples: IBM DB2, Oracle, Microsoft SQL Server, MySQL
- Typically follow a client-server model
  - the database server manages the data
  - applications act as clients
- Extremely powerful
  - SQL allows for more or less arbitrary queries
  - support *transactions* and the associated guarantees

Transactions

- A *transaction* is a sequence of operations that is treated as a single logical operation.
- Example: a balance transfer

transaction T1

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

- Other examples:
  - making a flight reservation
    select flight, reserve seat, make payment
  - making an online purchase
- Transactions are *all-or-nothing*: all of a transaction's changes take effect or none of them do.

Why Do We Need Transactions?

- To prevent problems stemming from system failures.
  - example:

transaction

```
read balance1
write(balance1 - 500)
CRASH
read balance2
write(balance2 + 500)
```

- what should happen?

Why Do We Need Transactions? (cont.)

- To ensure that operations performed by different users don't overlap in problematic ways.
  - example: what's wrong with the following?

user 1's transaction

```
read balance1
write(balance1 - 500)



read balance2
write(balance2 + 500)
```

user 2's transaction

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

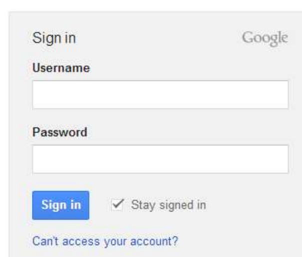
- how could we prevent this?

Limitations of the Conventional Approach

- Can be overkill for applications that don't need all the features
- Can be hard / expensive to setup / maintain / tune
- May not provide the necessary functionality
- Footprint may be too large
  - example: can't put a conventional RDBMS on a small embedded system
- May be unnecessarily slow for some tasks
  - overhead of IPC, query processing, etc.
- Does not scale well to large clusters

Example Problem I: User Accounts

- Database of user information for email, groups, etc.
- Used to authenticate users and manage their preferences
- Needs to be extremely fast and robust
- Don't need SQL. Why?



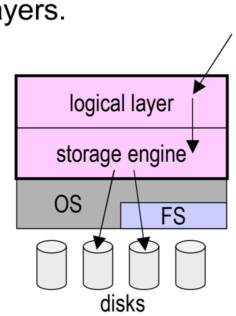
- Possible solution: a key-value store
  - key = user id
  - value = password and other user information
  - less overhead and easier to manage
  - still very powerful: transactions, recovery, replication, etc.

Example Problem II: Web Services

- Services provided or hosted by Google, Amazon, etc.
- Can involve huge amounts of data / traffic
- Scalability is crucial
  - load can increase rapidly and unpredictably
  - use large clusters of commodity machines
- Conventional relational DBMSs don't scale well in this way.
- Solution: some flavor of noSQL

What Other Options Are There?

- View a DBMS as being composed of two layers.
- At the bottom is the *storage layer* or *storage engine*.
  - stores and manages the data
- Above that is the *logical layer.*
  - provides an abstract representation of the data
  - based on some data model
  - includes some query language, tool, or API for accessing and modifying the data
- To get other approaches, choose different options for the layers.



Course Overview

- data models/representations (logical layer), including:
  - entity-relationship (ER): used in database design
  - relational (including SQL)
  - semistructured: XML, JSON
  - noSQL variants
- implementation issues (storage layer), including:
  - storage and index structures
  - transactions
  - concurrency control
  - logging and recovery
  - distributed databases and replication

Course Requirements

- Lectures and weekly sections
  - sections: optional but recommended; start this week
  - also available by streaming and recorded video
- Five problem sets
  - several will involve programming in Java
  - all will include written questions
  - grad-credit students will complete extra problems
  - must be your own work
    - see syllabus or website for the collaboration policy
- Midterm exam
- Final exam

Prerequisites

- A good working knowledge of Java
- A course at the level of CSCI E-22
- Experience with fairly large software systems is helpful.

Course Materials

- Lecture notes will be the primary resource.
- Optional textbook: *Database Systems: The Complete Book* (2nd edition) by Garcia-Molina et al. (Prentice Hall)
- Other options:
  - *Database Management Systems* by Ramakrishnan and Gehrke (McGraw-Hill)
  - *Database System Concepts* by Silberschatz et al. (McGraw-Hill)

Additional Administrivia

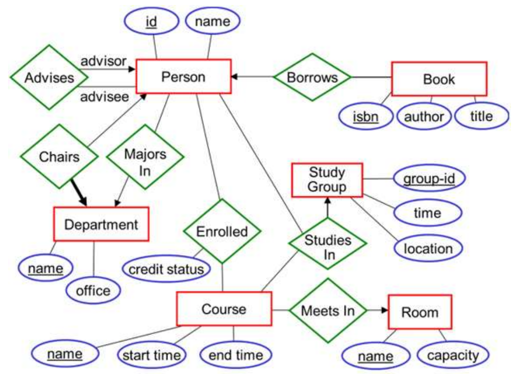
- Instructor: Cody Doucette
- TA: Eli Saracino
- Office hours and contact info. are available on the Web:
  http://cscie66.sites.fas.harvard.edu
- For questions on content, homework, etc.: Ed Discussion

Database Design

- In database design, we determine:
  - which pieces of data to include
  - how they are related
  - how they should be grouped/decomposed
- End result: a *logical schema* for the database
  - describes the contents and structure of the database

ER Models

- An *entity-relationship (ER) model* is a tool for database design.
  - graphical
  - implementation-neutral



- ER models specify:
  - the relevant entities ("things") in a given domain
  - the relationships between them

Sample Domain: A University

- Want to store data about:
  - employees
  - students
  - courses
  - departments
- How many tables do you think we'll need?
  - can be hard to tell before doing the design
  - in particular, hard to determine which tables are needed to encode relationships between data items

Entities: the "Things"

- Represented using rectangles.
- Examples:

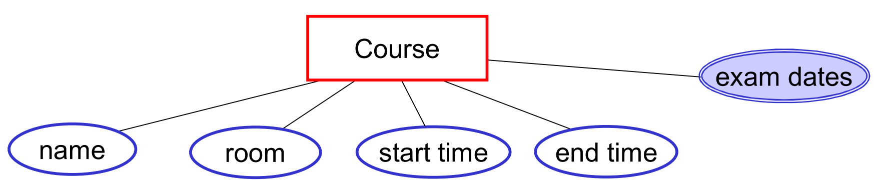
Course | Student | Employee

- Strictly speaking, each rectangle represents an *entity set*, which is a collection of individual entities.

Course	Student	Employee
CSCI E-119	Jill Jones	Drew Faust
English 101	Alan Turing	Dave Sullivan
CSCI E-268	Jose Delgado	Margo Seltzer
…	…	…

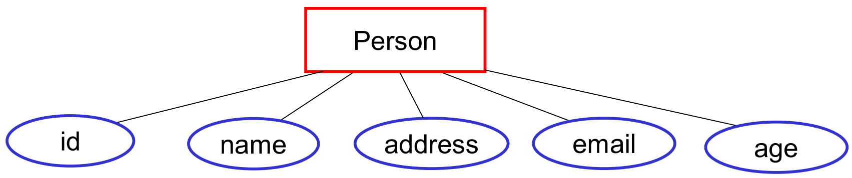
Attributes

- Associated with entities are *attributes* that describe them.
  - represented as ovals connected to the entity by a line
  - double oval = attribute that can have multiple values



Keys

- A *key* is an attribute or collection of attributes that can be used to uniquely identify each entity in an entity set.
- An entity set may have more than one possible key.
  - example:



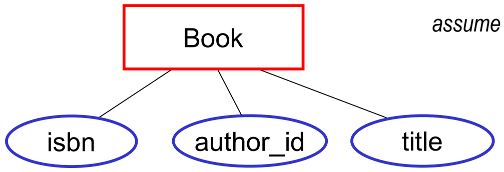
  - possible keys include:

Candidate Key

- A *candidate key* is a *minimal* collection of attributes that is a key.
  - minimal = no unnecessary attributes are included
    - *not* the same as *minimum*
- Example: assume (name, address, age) is a key for Person
  - it is a *minimal* key because we lose uniqueness if we remove any of the three attributes:
    - (name, address) may not be unique
      – e.g., a father and son with the same name and address
    - (name, age) may not be unique
    - (address, age) may not be unique
- Example: (id, email) is a key for Person
  - it is *not* minimal, because just one of these attributes is sufficient for uniqueness
  - therefore, it is *not* a candidate key

Key vs. Candidate Key

- Consider an entity set for books:

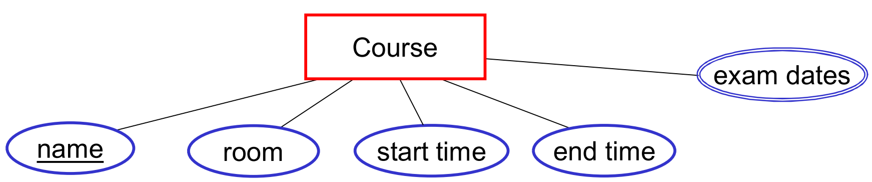


assume that: each book has a unique isbn
an author doesn't write two books with the same title

	key?	candidate key?
isbn		
author_id, title		
author_id, isbn		
author_id		

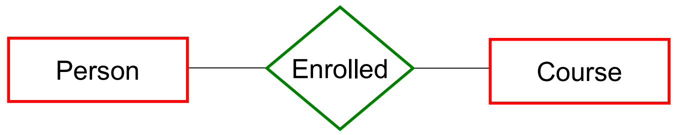
Primary Key

- We typically choose one of the candidate keys as the *primary key.*
- In an ER diagram, we underline the primary key attribute(s).

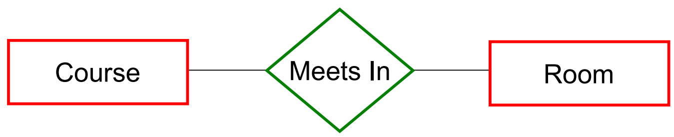


Relationships Between Entities

- Relationships between entities are represented using diamonds that are connected to the relevant entity sets.
- For example: students are enrolled in courses

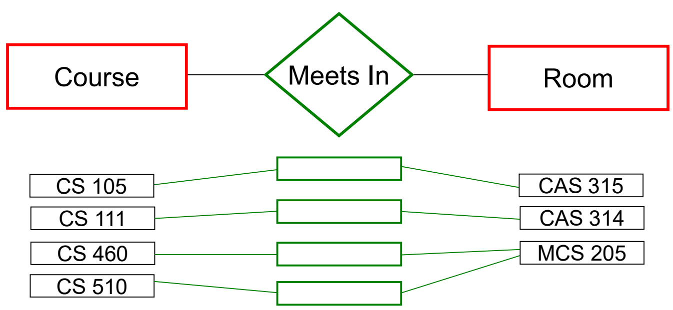


- Another example: courses meet in rooms



Relationships Between Entities (cont.)

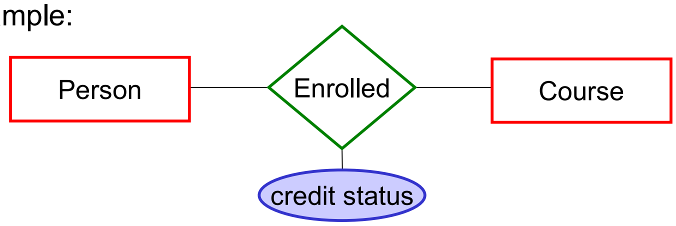
- Strictly speaking, each diamond represents a *relationship set*, which is a collection of relationships between individual entities.



- In a given set of relationships:
  - an individual entity may appear 0, 1, or multiple times
  - a given *combination* of entities may appear at most once
    - example: the combination (CS 105, CAS 315) may appear at most once

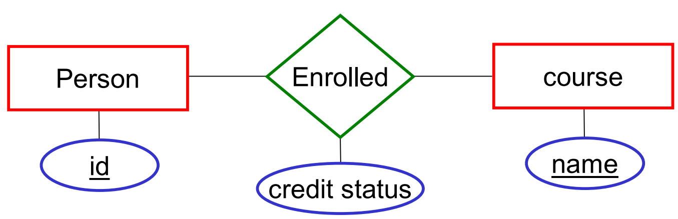
Attributes of Relationships

- A relationship set can also have attributes.
  - they specify info. associated with the relationships in the set
- Example:



Key of a Relationship Set

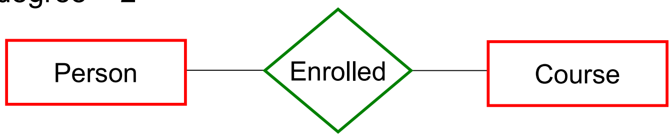
- A key of a relationship set can be formed by taking the union of the primary keys of its participating entities.
  - example: (Person.id, Course.name) is a key of enrolled



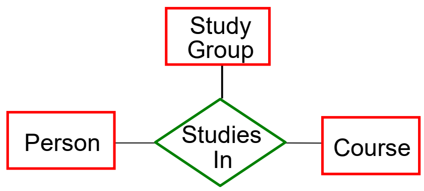
- The resulting key may or may not be a primary key. Why?

Degree of a Relationship Set

- Enrolled is a *binary* relationship set: it connects two entity sets.
  - degree = 2

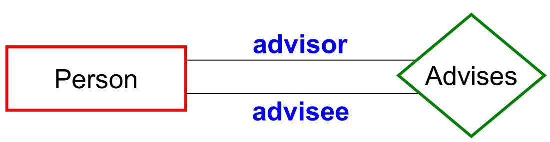


- It's also possible to have higher-degree relationship sets.
- A *ternary* relationship set connects three entity sets.
  - degree = 3



Relationships with Role Indicators

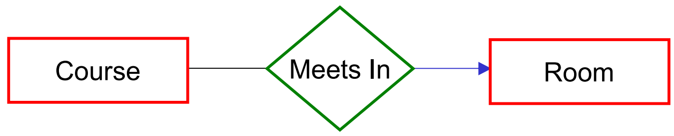
- It's possible for a relationship set to involve more than one entity from the same entity set.
- For example: every student has a faculty advisor, where students and faculty members are both members of the Person entity set.



- In such cases, we use *role indicators* (labels on the lines) to distinguish the roles of the entities in the relationship.

Cardinality (or Key) Constraints

- A *cardinality constraint* (or *key constraint*) limits the number of times that a given entity can appear in a relationship set.
- Example: each course meets in *at most one* (i.e., 0 or 1) room

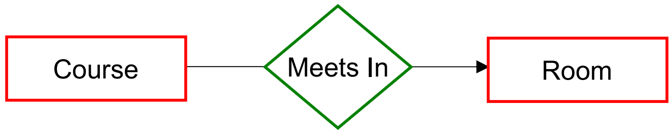


- A key constraint specifies a functional mapping from one entity set to another.
  - each course is mapped to at most one room (course → room)
  - as a result, each course appears in at most one relationship in the *meets in* relationship set
- The arrow in the ER diagram has same direction as the mapping.
  - note: the R&G book uses a different convention for the arrows

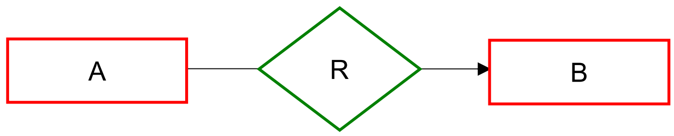
Cardinality Constraints (cont.)

- The presence or absence of cardinality constraints divides relationships into three types:
  - many-to-one
  - one-to-one
  - many-to-many
- We'll now look at each type of relationship.

Many-to-One Relationships

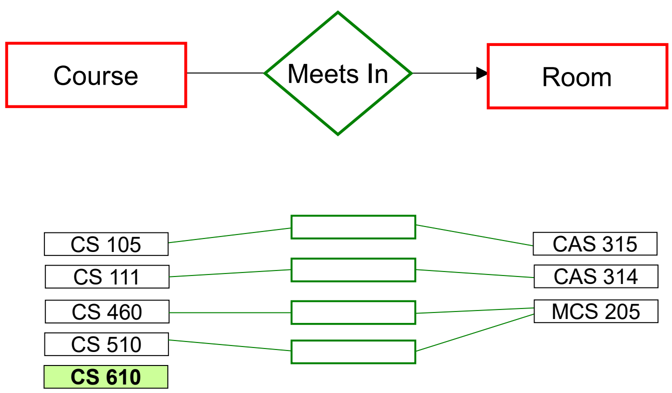


- Meets In is an example of a *many-to-one* relationship.
- We need to specify a *direction* for this type of relationship.
  - example: Meets In is many-to-one <u>from Course to Room</u>
- In general, in a many-to-one relationship from A to B:



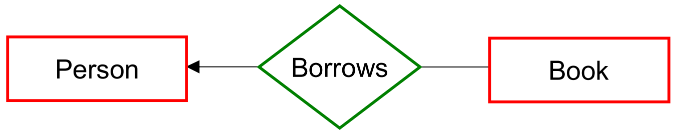
  - an entity in A can be related to *at most one* entity in B
  - an entity in B can be related to an arbitrary number of entities in A (0 or more)

Picturing a Many-to-One Relationship



- Each course participates in at most one relationship, because it can meet in at most one room.
- Because the constraint only specifies a maximum (*at most one*), it's possible for a course to not meet in any room (e.g., CS 610).

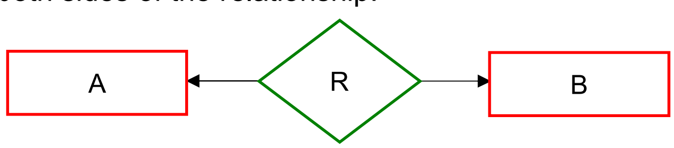
Another Example of a Many-to-One Relationship



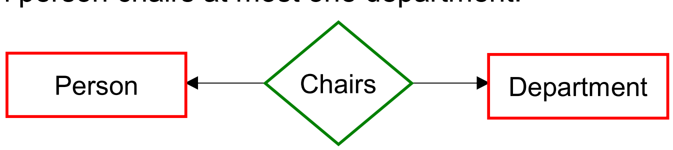
- The diagram above says that:
  - a given book can be borrowed by at most one person
  - a given person can borrow an arbitrary number of books
- Borrows is a many-to-one relationship from Book to Person.
- We could also say that Borrows is a *one-to-many* relationship from Person to Book.
  - one-to-many is the same thing as many-to-one, but the direction is reversed

One-to-One Relationships

- In a *one-to-one relationship* involving A and B: [not *from* A to B]
  - an entity in A can be related to *at most one* entity in B
  - an entity in B can be related to *at most one* entity in A
- We indicate a one-to-one relationship by putting an arrow on both sides of the relationship:



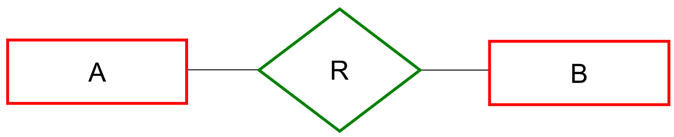
- Example: each department has at most one chairperson, and each person chairs at most one department.



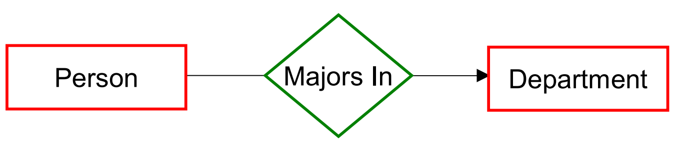
Many-to-Many Relationships

- In a *many-to-many relationship* involving A and B:
  - an entity in A can be related to an arbitrary number of entities in B (0 or more)
  - an entity in B can be related to an arbitrary number of entities in A (0 or more)

- If a relationship has no cardinality constraints specified (i.e., if there are no arrows on the connecting lines), it is assumed to be many-to-many.

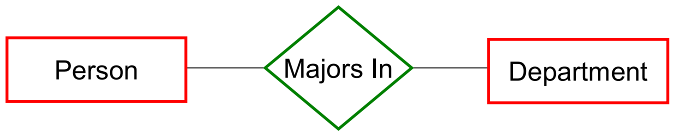


How can we indicate that each student has at most one major?



- *Majors In* is what type of relationship in this case?

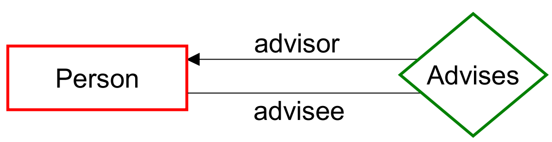
What if each student can have more than one major?



- *Majors In* is what type of relationship in this case?

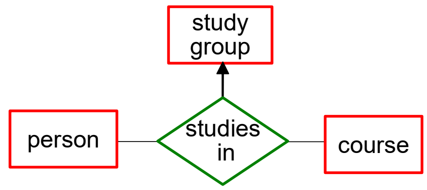
Another Example

- How can we indicate that each student has at most one advisor?



- Advises is what type of relationship?

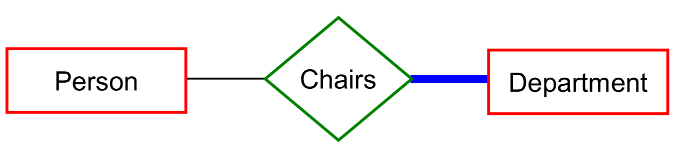
Cardinality Constraints and Ternary Relationship Sets



- The arrow into "study group" encodes the following constraint: "a person studies in at most one study group *for a given course.*"
- In other words, a given (person, course) combination is mapped to at most one study group.
  - a given person or course can itself appear in multiple studies-in relationships
- For relationship sets of degree >= 3, we use at most one arrow, since otherwise the meaning can be ambiguous.

Participation Constraints

- Cardinality constraints allow us to specify that each entity will appear *at most* once in a given relationship set.
- Participation constraints allow us to specify that each entity will appear *at least* once (i.e., 1 or more time).
  - indicate using a thick line (or double line)
- Example: each department must have at least one chairperson.

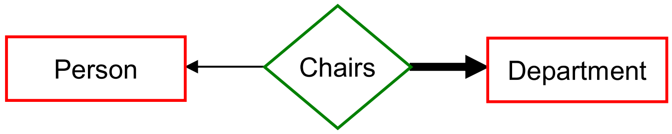


omitting the cardinality constraints for now

- We say Department has *total participation* in Chairs.
  - by contrast, Person has *partial participation*

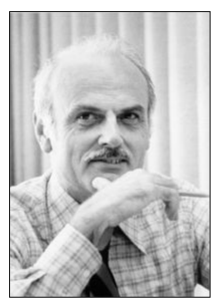
Participation Constraints (cont.)

- We can combine cardinality and participation constraints.



  - a person chairs at most one department
    - specified by which arrow?
  - a department has ____________ person as a chair

The Relational Model: A Brief History



- Defined in a landmark 1970 paper by Edgar 'Ted' Codd.
- Earlier data models were closely tied to the physical representation of the data.
- The relational model was revolutionary because it provided *data independence* – separating the *logical* model of the data from its underlying *physical* representation.
- Allows users to access the data *without* understanding how it is stored on disk.

The Relational Model: Basic Concepts

- A *database* consists of a collection of *tables*.
- Example of a table:

id	name	address	class	dob
12345678	Jill Jones	Canaday C-54	2011	3/10/85
25252525	Alan Turing	Lowell House F-51	2008	2/7/88
33566891	Audrey Chu	Pfoho, Moors 212	2009	10/2/86
45678900	Jose Delgado	Eliot E-21	2009	7/13/88
66666666	Count Dracula	The Dungeon	2007	11/1431
...	...	...	...	...

- Each *row* in a table holds data that describes either:
  - an *entity*
  - a *relationship* between two or more entities
- Each *column* in a table represents one attribute of an entity.
  - each column has a *domain* of possible values

Relational Model: Terminology

- Two sets of terminology:

table	=	relation
row	=	tuple
column	=	attribute

- We'll use both sets of terms.

Requirements of a Relation

- Each column must have a unique name.
- The values in a column must be of the same type (i.e., must come from the same domain).
  - integers, real numbers, dates, strings, etc.
- Each cell must contain a single value.
  - example: we *can't* do something like this:

id	*name*	*...*	*phones*
12345678	Jill Jones	...	123-456-5678, 234-666-7890
25252525	Alan Turing	...	777-777-7777, 111-111-1111
...	...	...	...

- No two rows can be identical.
  - identical rows are known as *duplicates*

Null Values

- By default, the domains of most columns include a special value called *null*.
- Null values can be used to indicate that:
  - the value of an attribute is unknown for a particular tuple
  - the attribute doesn't apply to a particular tuple. example:

Student

id	*name*	*...*	*major*
12345678	Jill Jones	...	computer science
25252525	Alan Turing	...	mathematics
33333333	Dan Dabbler	...	null

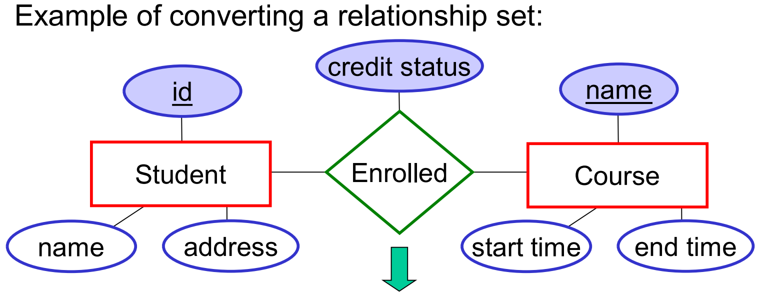
Relational Schema

- The *schema* of a relation consists of:
  - the name of the relation
  - the names of its attributes
  - the attributes' domains (although we'll ignore them for now)
- Example:

  Student(id, name, address, email, phone)
- The schema of a relational database consists of the schema of all of the relations in the database.

ER Diagram to Relational Database Schema

- Basic process:
  - entity set → a relation with the same attributes
  - relationship set → a relation whose attributes are:
    - the primary keys of the connected entity sets
    - the attributes of the relationship set
- Example of converting a relationship set:

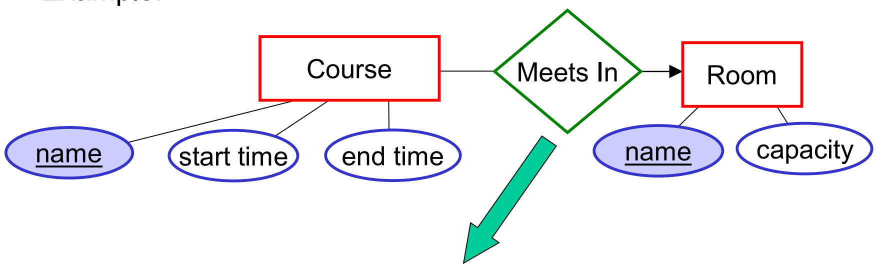


Enrolled(id, name, credit_status)

  - in addition, we would create a relation for each entity set

Renaming Attributes

- When converting a relationship set to a relation, there may be multiple attributes with the same name.
  - need to rename them
- Example:



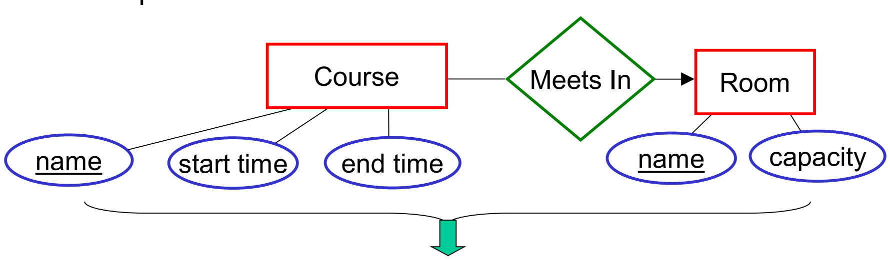
MeetsIn(name, name)

MeetsIn(course_name, room_name)

- We may also choose to rename attributes for the sake of clarity.

Special Case: Many-to-One Relationship Sets

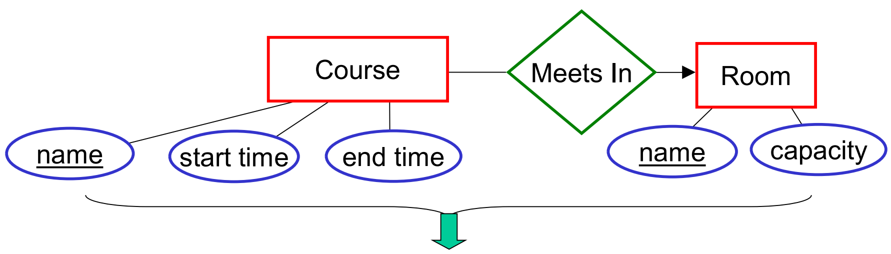
- Ordinarily, a binary relationship set will produce three relations:
  - one for the relationship set
  - one for each of the connected entity sets
- Example:



MeetsIn(course_name, room_name)
Course(name, start_time, end_time)
Room(name, capacity)

Special Case: Many-to-One Relationship Sets (cont.)

- However, if a relationship set is many-to-one, we often:
  - eliminate the relation for the relationship set
  - capture the relationship set in the relation used for the entity set on the *many* side of the relationship



~~*MeetsIn(course_name, room_name)*~~
Course(name, start_time, end_time, room_name)
Room(name, capacity)

Special Case: Many-to-One Relationship Sets (cont.)

- Advantages of this approach:
  - makes some types of queries more efficient to execute
  - uses less space

Course

name	*...*
cscie50b	
cscie119	
cscie160	
cscie268	

MeetsIn

course_name	*room_name*
cscie50b	Sci Ctr B
cscie119	Sever 213
cscie160	Sci Ctr A
cscie268	Sci Ctr A

Course

name	*...*	*room_name*
cscie50b		Sci Ctr B
cscie119		Sever 213
cscie160		Sci Ctr A
cscie268		Sci Ctr A

Special Case: Many-to-One Relationship Sets (cont.)

- If one or more entities don't participate in the relationship, there will be null attributes for the fields that capture the relationship:

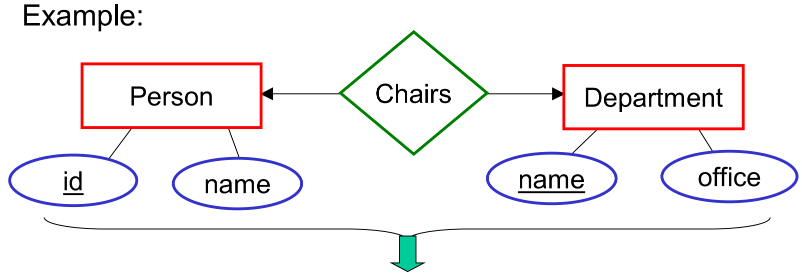
Course

name	...	*room_name*
cscie50b		Sci Ctr B
cscie119		Sever 213
cscie160		Sci Ctr A
cscie268		Sci Ctr A
cscie160		NULL

- If a large number of entities don't participate in the relationship, it may be better to use a separate relation.

Special Case: One-to-One Relationship Sets

- Here again, we're able to have only two relations – one for each of the entity sets.
- In this case, we can capture the relationship set in the relation used for *either of the entity sets.*
- Example:



Person(id, name, chaired_dept)
Department(name, office)

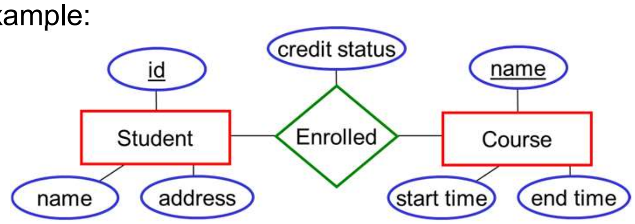
OR

Person(name, id)
Department(name, office, chair_id)

- which of these would probably make more sense?

Many-to-Many Relationship Sets

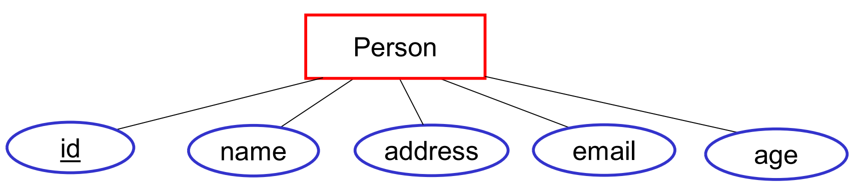
- For many-to-many relationship sets, we need to use a *separate relation* for the relationship set.
  - example:



  - can't capture the relationships in the *Student* table
    - a given student can be enrolled in multiple courses
  - can't capture the relationships in the *Course* table
    - a given course can have multiple students enrolled in it
  - need to use a separate table:
    Enrolled(student_id, course_name, credit_status)

Recall: Primary Key

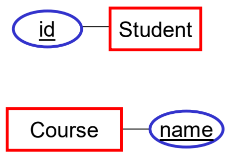
- We typically choose one of the candidate keys as the *primary key.*
- In an ER diagram, we underline the primary key attribute(s).



- In the relational model, we also designate a primary key by underlying it.
  Person(<u>id</u>, name, address, …)
- A relational DBMS will ensure that no two rows have the same value / combination of values for the primary key.
  - known as a *uniqueness constraint*

Primary Keys of Relations for Entity Sets

- When translating an *entity set* to a relation,
  the relation gets the same primary key as the entity set.



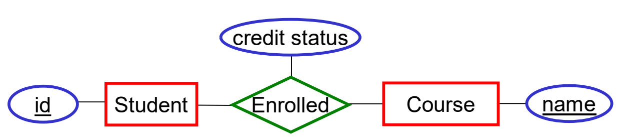
→ *Student(id, …)*

→ *Course(name, …)*

Primary Keys of Relations for Relationship Sets

- When translating a relationship set to a relation,
  the primary key depends on the cardinality constraints.

- For a *many-to-many* relationship set, we take the union
  of the primary keys of the connected entity sets.



→ *Enrolled(student_id, course_name, credit_status)*

  - doing so prevents a given *combination* of entities
    from appearing more than once in the relation
  - it still allows a single entity (e.g., a single student or course)
    to appear multiple times, as part of different combinations

Primary Keys of Relations for Relationship Sets (cont.)

- For a *many-to-one* relationship set,
  if we decide to use a separate relation for it,
  what should that relation's primary key include?

→ *Borrows(person_id, isbn)*

Primary Keys of Relations for Relationship Sets (cont.)

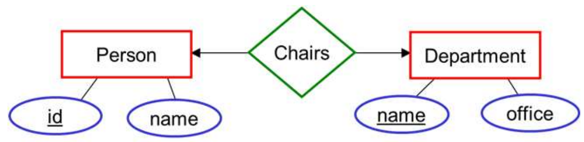
- For a *many-to-one* relationship set,
  if we decide to use a separate relation for it,
  what should that relation's primary key include?

→ *Borrows(person_id, isbn)*

- limiting the primary key enforces the cardinality constraint
  - in this example, the DBMS will ensure that a given book is borrowed by at most once person
- how else could we capture this relationship set?

Primary Keys of Relations for Relationship Sets (cont.)

- For a *one-to-one* relationship set, what should the primary key of the resulting relation be?



→ *Chairs(person_id, department_name)*

Foreign Keys

- A *foreign key* is attribute(s) in one relation that take on values from the primary-key attribute(s) of another relation.
  - example: *MajorsIn* has <u>two</u> foreign keys

MajorsIn

student	***department***
12345678	computer science
12345678	english
...	...

Student

id	***name***	***...***
12345678	Jill Jones	...
25252525	Alan Turing	...
...	...	...

Department

name	***...***
computer science	...
english	...
...	...

- We use foreign keys to capture relationships between entities.

- All values of a foreign key must match the referenced attribute(s) of some tuple in the other relation.
  - known as a *referential integrity* constraint

Enforcing Constraints

- Example: assume that the tables below show *all* of their tuples.

MajorsIn

student	***dept_name***
12345678	computer science
12345678	english

Student

id	***name***	***...***
12345678	Jill Jones	...
25252525	Alan Turing	...

Department

name	***...***
computer science	...
english	...

- Which of the following operations would the DBMS allow?
  - adding (12345678, 'John Smith', …) to *Student*
  - adding (33333333, 'Howdy Doody', …) to *Student*
  - adding (12345678, 'physics') to *MajorsIn*
  - adding (25252525, 'english') to *MajorsIn*

Relational Algebra and SQL

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Example Domain: a University

- Four relations that store info. about a type of entity:
  - *Student(id, name)*
  - *Department(name, office)*
  - *Room(id, name, capacity)*
  - *Course(name, start_time, end_time, room_id)*
- Two relations that capture relationships between entities:
  - *MajorsIn(student_id, dept_name)*
  - *Enrolled(student_id, course_name, credit_status)*
- The *room_id* attribute in the *Course* relation also captures a relationship – the relationship between a course and the room in which it meets.

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Room

id	name	capacity
1000	Sanders Theatre	1000
2000	Sever 111	50
3000	Sever 213	100
4000	Sci Ctr A	300
5000	Sci Ctr B	500
6000	Emerson 105	500
7000	Sci Ctr 110	30

Course

name	start_time	end_time	room_id
cscie119	19:35:00	21:35:00	4000
cscie268	19:35:00	21:35:00	2000
cs165	16:00:00	17:30:00	7000
cscie275	17:30:00	19:30:00	7000

Department

name	office
comp sci	MD 235
mathematics	Sci Ctr 520
the occult	The Dungeon
english	Sever 125

Enrolled

student_id	course_name	credit_status
12345678	cscie268	ugrad
25252525	cs165	ugrad
45678900	cscie119	grad
33566891	cscie268	non-credit
45678900	cscie275	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Relational Algebra

- The query language proposed by Codd.
  - a collection of operations on relations
- Each operation:
  - takes one or more relations
  - produces a relation

one or more relations → *operation* → *a relation*

- Relations are treated as sets.
  - all duplicate tuples are removed from an operation's result

Selection

- What it does: selects tuples from a relation that match a predicate
  - predicate = condition
- Syntax: $\sigma_{predicate}(\text{relation})$
- Example: Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
25252525	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
45678900	cscie119	graduate

$\sigma_{\text{credit\_status = 'graduate'}}(\text{Enrolled})$ =

student_id	course_name	credit_status
45678900	cscie268	graduate
45678900	cscie119	graduate

- Predicates may include: >, <, =, !=, etc., as well as *and*, *or*, *not*

Projection

- What it does: extracts attributes from a relation
- Syntax: $\pi_{attributes}(\text{relation})$
- Example: Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
25252525	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
45678900	cscie119	graduate

$\pi_{\text{student\_id, credit\_status}}(\text{Enrolled})$ =

student_id	credit_status
12345678	undergrad
25252525	undergrad
45678900	graduate
33566891	non-credit
45678900	graduate

duplicates, so we keep only one

student_id	credit_status
12345678	undergrad
25252525	undergrad
45678900	graduate
33566891	non-credit

Combining Operations

- Since each operation produces a relation, we can combine them.
- Example: Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
25252525	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
45678900	cscie119	graduate

$\pi_{\text{student\_id, credit\_status}}(\sigma_{\text{credit\_status = 'graduate'}}(\text{Enrolled}))$ =

student_id	course_name	credit_status
45678900	cscie268	graduate
45678900	cscie119	graduate

↓

student_id	credit_status
45678900	graduate
45678900	graduate

→

student_id	credit_status
45678900	graduate

Mathematical Foundations: Cartesian Product

- Let: A be the set of values { a_1, a_2, … }
  B be the set of values { b_1, b_2, … }

- The *Cartesian product* of A and B (written A x B) is the set of all possible ordered pairs (a_i, b_j), where $a_i \in A$ and $b_j \in B$.

- Example:
  A = { apple, pear, orange }
  B = { cat, dog }

  A x B = { (apple, cat), (apple, dog), (pear, cat), (pear, dog),
  (orange, cat), (orange, dog) }

- Example:
  C = { 5, 10 }
  D = { 2, 4 }

  C x D = ?

Mathematical Foundations: Cartesian Product (cont.)

- We can also take the Cartesian product of three of more sets.
- A x B x C is the set of all possible ordered triples (a_i, b_j, c_k), where $a_i \in A$, $b_j \in B$, and $c_k \in C$.
  - example:
    C = { 5, 10 }
    D = { 2, 4 }
    E = {'hi', 'there'}

    C x D x E = { (5, 2, 'hi'), (5, 2, 'there'),
    (5, 4, 'hi'), (5, 4, 'there'),
    (10, 2, 'hi'), (10, 2, 'there'),
    (10, 4, 'hi'), (10, 4, 'there') }
- $A_1 \times A_2 \times \ldots \times A_n$ is the set of all possible ordered *tuples* $(a_{1i}, a_{2j}, \ldots, a_{nk})$, where $a_{de} \in A_d$.

Cartesian Product in Relational Algebra

- What it does: takes two relations, R_1 and R_2, and forms a new relation containing all possible combinations of tuples from R_1 with tuples from R_2
- Syntax: $R_1 \times R_2$
- Rules:
  - R_1 and R_2 must have different names
  - the resulting relation has a schema that consists of the attributes of R_1 followed by the attributes of R_2
    $(a_{11}, a_{12}, \ldots, a_{1m}) \times (a_{21}, \ldots, a_{2n}) \rightarrow (a_{11}, \ldots, a_{1m}, a_{21}, \ldots, a_{2n})$
  - if there are two attributes with the same name, we prepend the name of the original relation
    - example: the attributes of Enrolled x MajorsIn would be
      (Enrolled.student_id, course_name, credit_status, MajorsIn.student_id, dept_name)

Cartesian Product in Relational Algebra (cont.)

- Example:

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Enrolled x MajorsIn

Enrolled. student_id	course_name	credit_status	MajorsIn. student_id	dept_name
12345678	cscie50b	undergrad	12345678	comp sci
12345678	cscie50b	undergrad	45678900	mathematics
12345678	cscie50b	undergrad	33566891	comp sci
12345678	cscie50b	undergrad	98765432	english
12345678	cscie50b	undergrad	66666666	the occult
45678900	cscie160	undergrad	12345678	comp sci
45678900	cscie160	undergrad	45678900	mathematics
...	...	...	...	...

Rename

- What it does: gives a (possibly new) name to a relation, and optionally to its attributes
- Syntax: ρ_{rel_name}(relation)

  $\rho_{rel\_name(A_1,\ A_2,\ ...,\ A_n)}$(relation)
- Examples:
  - renaming to allow us to take the Cartesian product of a relation with itself:

    ρ_{E1}(Enrolled) x ρ_{E2}(Enrolled)
  - renaming to give a name to the result of an operation:

    $\sigma_{room\ =\ BigRoom.name}$(Course x $\rho_{\mathbf{BigRoom}}(\sigma_{capacity\ >\ 200}$(Room)))

Natural Join

- What it does: performs a "filtered" Cartesian product
  - filters out / removes the tuples in which attributes with the same name have different values
- Syntax: $R_1 \bowtie R_2$
- Example:

R_1

g	h	i
foo	10	4
bar	20	5
baz	30	6

R_2

i	j	g
4	100	foo
4	300	bop
5	400	baz
5	600	bar

$R_1 \bowtie R_2$

g	h	i	j
foo	10	4	100
bar	20	5	600

Performing the Natural Join

- Step 1: take the full Cartesian product
- Example:

R_1

g	h	i
foo	10	4
bar	20	5
baz	30	6

R_2

i	j	g
4	100	foo
4	300	bop
5	400	baz
5	600	bar

$R_1 \times R_2$

$R_1.g$	h	$R_1.i$	$R_2.i$	j	$R_2.g$
foo	10	4	4	100	foo
foo	10	4	4	300	bop
foo	10	4	5	400	baz
foo	10	4	5	600	bar
bar	20	5	4	100	foo
bar	20	5	4	300	bop
bar	20	5	5	400	baz
bar	20	5	5	600	bar
baz	30	6	4	100	foo
baz	30	6	4	300	bop
baz	30	6	5	400	baz
baz	30	6	5	600	bar

Performing the Natural Join

- Step 2: perform a selection that filters out tuples in which attributes with the same name have different values
  - if there are no attributes with the same name, skip this step
- Example:

R_1

g	*h*	*i*
foo	10	4
bar	20	5
baz	30	6

R_2

i	*j*	*g*
4	100	foo
4	300	bop
5	400	baz
5	600	bar

$R_1.g$	*h*	$R_1.i$	$R_2.i$	*j*	$R_2.g$
foo	10	4	4	100	foo
bar	20	5	5	600	bar

Performing the Natural Join

- Step 3: perform a projection that keeps only one copy of each duplicated column.
- Example:

R_1

g	*h*	*i*
foo	10	4
bar	20	5
baz	30	6

R_2

i	*j*	*g*
4	100	foo
4	300	bop
5	400	baz
5	600	bar

$R_1.g$	*h*	$R_1.i$	$R_2.i$	*j*	$R_2.g$
foo	10	4	4	100	foo
bar	20	5	5	600	bar

g	*h*	*i*	*j*
foo	10	4	100
bar	20	5	600

Performing the Natural Join

- Final result: a table with all combinations of "matching" rows from the original tables.

- Example:

R_1

g	h	i
foo	10	4
bar	20	5
baz	30	6

R_2

i	j	g
4	100	foo
4	300	bop
5	400	baz
5	600	bar

$R_1 \bowtie R_2$

g	h	i	j
foo	10	4	100
bar	20	5	600

Natural Join: Summing Up

- The natural join is equivalent to the following:
  - Cartesian product, then selection, then projection

- The resulting relation's schema consists of the attributes of R_1 x R_2, but with common attributes included only once

  (a, b, c) x (a, d, c, f) → (a, b, c, d, f)

- If there are no common attributes, $R_1 \bowtie R_2$ = R_1 x R_2

Condition Joins (aka Theta Joins)

- What it does: performs a "filtered" Cartesian product according to a specified predicate
- Syntax: $R_1 \bowtie_{\theta} R_2$, where θ is a predicate
- Fundamental-operation equivalent: cross, select using θ
- Example: $R_1 \bowtie_{(d > c)} R_2$ = ?

R_1

a	b	c
foo	10	4
bar	20	5
baz	30	6

R_2

d	e
3	100
4	300
5	400
6	600

$R_1 \bowtie_{(d > c)} R_2$

a	b	c	d	e
foo	10	4	5	400
foo	10	4	6	600
bar	20	5	6	600

Which of these queries finds the names of all courses taken by comp sci majors?

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

If there is more than one correct answer, select all answers that apply.

A. $\pi_{course\_name}(\sigma_{dept\_name = 'comp\ sci'}(\text{Enrolled} \times \text{MajorsIn}))$

B. $\pi_{course\_name}(\sigma_{dept\_name = 'comp\ sci'}(\text{Enrolled} \bowtie \text{MajorsIn}))$

C. $\pi_{course\_name}(\text{Enrolled} \bowtie_{dept\_name = 'comp\ sci'} \text{MajorsIn}))$

D. $\pi_{course\_name}(\text{Enrolled} \bowtie (\sigma_{dept\_name = 'comp\ sci'}(\text{MajorsIn})))$

Which of these queries finds the names of all courses taken by comp sci majors?

Enrolled

student_id	*course_name*	*credit_status*
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Which courses should we be finding?

A. $\pi_{\text{course\_name}}(\sigma_{\text{dept\_name = 'comp sci'}}(\text{Enrolled} \times \text{MajorsIn}))$

B. $\pi_{\text{course\_name}}(\sigma_{\text{dept\_name = 'comp sci'}}(\text{Enrolled} \bowtie \text{MajorsIn}))$

C. $\pi_{\text{course\_name}}(\text{Enrolled} \bowtie_{\text{dept\_name = 'comp sci'}} \text{MajorsIn}))$

D. $\pi_{\text{course\_name}}(\text{Enrolled} \bowtie (\sigma_{\text{dept\_name = 'comp sci'}}(\text{MajorsIn})))$

A. $\pi_{\text{course\_name}}(\sigma_{\text{dept\_name = 'comp sci'}}(\text{Enrolled} \times \text{MajorsIn}))$

Enrolled. student_id	*course_name*	*credit_status*	*MajorsIn. student_id*	*dept_name*
12345678	cscie50b	undergrad	12345678	comp sci
12345678	cscie50b	undergrad	33566891	comp sci
45678900	cscie160	undergrad	12345678	comp sci
45678900	cscie160	undergrad	33566891	comp sci
45678900	cscie268	graduate	12345678	comp sci
45678900	cscie268	graduate	33566891	comp sci
33566891	cscie119	non-credit	12345678	comp sci
33566891	cscie119	non-credit	33566891	comp sci
25252525	cscie119	graduate	12345678	comp sci
25252525	cscie119	graduate	33566891	comp sci

course_name
cscie50b
cscie50b
cscie160
cscie160
cscie268
cscie268
cscie119
cscie119
cscie119
cscie119

→

course_name
cscie50b
cscie160 x
cscie268 x
cscie119

In the Cartesian product, the MajorsIn tuples for the comp sci majors are each combined with *every* Enrolled tuple, so we end up getting *every* course in Enrolled, not just the ones taken by comp sci majors.

B. $\pi_{course\_name}(\sigma_{dept\_name = 'comp sci'}(Enrolled \bowtie MajorsIn))$

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Enrolled $\bowtie$ MajorsIn

student_id	course_name	credit_status	dept_name
12345678	cscie50b	undergrad	comp_sci
45678900	cscie160	undergrad	mathematics
45678900	cscie268	graduate	mathematics
33566891	cscie119	non-credit	comp sci

$\sigma_{dept\_name = 'comp sci'}$

student_id	course_name	credit_status	dept_name
12345678	cscie50b	undergrad	comp_sci
33566891	cscie119	non-credit	comp sci

π_{course_name}

course_name
cscie50b
cscie119

This query obtains the correct result.

Which of these queries finds the names of all courses taken by comp sci majors?

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

A. $\pi_{course\_name}(\sigma_{dept\_name = 'comp sci'}(Enrolled \times MajorsIn))$

B. $\pi_{course\_name}(\sigma_{dept\_name = 'comp sci'}(Enrolled \bowtie MajorsIn))$

C. $\pi_{course\_name}(Enrolled \bowtie_{dept\_name = 'comp sci'} MajorsIn))$

D. $\pi_{course\_name}(Enrolled \bowtie (\sigma_{dept\_name = 'comp sci'}(MajorsIn)))$

A and C are equivalent!

D. $\pi_{\text{course\_name}}(\text{Enrolled} \bowtie (\sigma_{\text{dept\_name = 'comp sci'}}(\text{MajorsIn})))$

Enrolled

student_id	*course_name*	*credit_status*
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

$\sigma_{\text{dept\_name = 'comp sci'}}(\text{MajorsIn})$

student_id	*dept_name*
12345678	comp sci
33566891	comp sci

$\text{Enrolled} \bowtie (\sigma_{\text{dept\_name = 'comp sci'}}(\text{MajorsIn}))$

student_id	*course_name*	*credit_status*	*dept_name*
12345678	cscie50b	undergrad	comp_sci
33566891	cscie119	non-credit	comp sci

$\pi_{\text{course\_name}}$

course_name
cscie50b
cscie119

This query obtains the correct result.

Which of these queries finds the names of all courses taken by comp sci majors?

Enrolled

student_id	*course_name*	*credit_status*
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

A. $\pi_{\text{course\_name}}(\sigma_{\text{dept\_name = 'comp sci'}}(\text{Enrolled} \times \text{MajorsIn}))$

B. $\pi_{\text{course\_name}}(\sigma_{\text{dept\_name = 'comp sci'}}(\text{Enrolled} \bowtie \text{MajorsIn}))$

C. $\pi_{\text{course\_name}}(\text{Enrolled} \bowtie_{\text{dept\_name = 'comp sci'}} \text{MajorsIn}))$

D. $\pi_{\text{course\_name}}(\text{Enrolled} \bowtie (\sigma_{\text{dept\_name = 'comp sci'}}(\text{MajorsIn})))$

B and D are equivalent – and correct!

Joins and Unmatched Tuples

- Let's say we want to know the majors of all enrolled students – ***including those with no major***. We begin by trying natural join:

Enrolled

student_id	*course_name*	*credit_status*
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Enrolled $\bowtie$ MajorsIn

student_id	*course_name*	*credit_status*	*dept_name*
12345678	cscie50b	undergrad	comp sci
45678900	cscie160	undergrad	mathematics
45678900	cscie268	graduate	mathematics
33566891	cscie119	non-credit	comp sci

- Why isn't this sufficient?

Outer Joins

- Outer joins allow us to include unmatched tuples in the result.
- Left outer join ($R_1 ⟕ R_2$): in addition to the natural-join tuples, include an extra tuple for each tuple from R_1 with no match in R_2
  - in the extra tuples, give the R_2 attributes values of null

Enrolled

student_id	*course_name*	*credit_status*
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Enrolled ⟕ MajorsIn

student_id	*course_name*	*credit_status*	*dept_name*
12345678	cscie50b	undergrad	comp sci
45678900	cscie160	undergrad	mathematics
45678900	cscie268	graduate	mathematics
33566891	cscie119	non-credit	comp sci
25252525	cscie119	graduate	null

Outer Joins (cont.)

- Right outer join (R_1 ⟖ R_2): include an extra tuple for each tuple from R_2 with no match in R_1

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Enrolled ⟖ MajorsIn

student_id	course_name	credit_status	dept_name
12345678	cscie50b	undergrad	comp sci
45678900	cscie160	undergrad	mathematics
45678900	cscie268	graduate	mathematics
33566891	cscie119	non-credit	comp sci
98765432	null	null	english
66666666	null	null	the occult

Outer Joins (cont.)

- Full outer join (R_1 ⟗ R_2): include an extra tuple for each tuple from either relation with no match in the other relation

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

Enrolled ⟗ MajorsIn

student_id	course_name	credit_status	dept_name
12345678	cscie50b	undergrad	comp sci
45678900	cscie160	undergrad	mathematics
45678900	cscie268	graduate	mathematics
33566891	cscie119	non-credit	comp sci
25252525	cscie119	graduate	null
98765432	null	null	english
66666666	null	null	the occult

Set Difference

- What it does: selects tuples that are in one relation but not in another.
- Syntax: $R_1 - R_2$
- Rules:
  - the relations must have the same number of attributes, and corresponding attributes must have the same domain
  - the resulting relation inherits its attribute names from the first relation
  - duplicates are eliminated, since relational algebra treats relations as sets

Set Difference (cont.)

- Example:

Enrolled

student_id	course_name	credit_status
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
33566891	comp sci
98765432	english
66666666	the occult

$\pi_{student\_id}(\text{MajorsIn}) - \pi_{student\_id}(\text{Enrolled})$

student_id
98765432
66666666

Set Difference (cont.)

- Example of where set difference is required:

  Of the students enrolled in courses, which ones are not enrolled in any courses for graduate credit?

student_id	***course_name***	***credit_status***
12345678	cscie50b	undergrad
45678900	cscie160	undergrad
45678900	cscie268	graduate
33566891	cscie119	non-credit
25252525	cscie119	graduate

  - The following query does *not* work. Why?

    $\pi_{\text{student\_id}} (\sigma_{\text{credit\_status != 'graduate'}}(\text{Enrolled}))$

  - This query *does* work:

    $\pi_{\text{student\_id}} (\text{Enrolled}) - \pi_{\text{student\_id}} (\sigma_{\text{credit\_status = 'graduate'}}(\text{Enrolled}))$

Assignment

- What it does: assigns the result of an operation to a temporary variable, or to an existing relation
- Syntax: relation $\leftarrow$ rel. alg. expression
- Uses:
  - simplying complex expressions
    - example: recall this expression

      $\sigma_{\text{room = BigRoom.name}} (\text{Course} \times \rho_{\text{BigRoom}}(\sigma_{\text{capacity > 200}}(\text{Room}))$

    - simpler version using assignment:

      $\text{BigRoom} \leftarrow \sigma_{\text{capacity > 200}}(\text{Room})$

      $\sigma_{\text{room = BigRoom.name}} (\text{Course} \times \text{BigRoom}))$

SQL

- Structured Query Language
- The query language used by most RDBMSs.
- Originally developed at IBM as part of System R – one of the first RDBMSs.

SELECT

- Used to implement most of the relational-algebra operations
- Basic syntax:

```
SELECT a1, a2, …
FROM R1, R2, …
WHERE selection predicate;
```

- Relational-algebra equivalent: cross, select, project
  1) take the cartesian product $R_1 \times R_2 \times \ldots$
  2) perform a selection that selects tuples from the cross product that satisfy the predicate in the `WHERE` clause
  3) perform a projection of attributes a_1, a_2, … from the tuples selected in step 2, *leaving duplicates alone by default*

  (These steps tell us what tuples will appear in the resulting relation, but the command may be executed differently for the sake of efficiency.)

- *Note:* the `SELECT` clause by itself specifies a projection! The `WHERE` clause specifies a selection.

Example Query

- Given these relations:
  Student(id, name)
  Enrolled(student_id, course_name, credit_status)
  MajorsIn(student_id, dept_name)

  we want find the major of the student Alan Turing.

- Here's a query that will give us the answer:

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing'
  AND id = student_id;
```

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student_id;
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x MajorsIn

id	*name*	*student_id*	*dept_name*
12345678	Jill Jones	12345678	comp sci
12345678	Jill Jones	45678900	mathematics
12345678	Jill Jones	25252525	comp sci
12345678	Jill Jones	45678900	english
12345678	Jill Jones	66666666	the occult
25252525	Alan Turing	12345678	comp sci
25252525	Alan Turing	45678900	mathematics
25252525	Alan Turing	25252525	comp sci
25252525	Alan Turing	45678900	english
...	...	...	...

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student_id;
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Student x MajorsIn WHERE id = student_id

id	*name*	*student_id*	*dept_name*
12345678	Jill Jones	12345678	comp sci
25252525	Alan Turing	25252525	comp sci
45678900	Jose Delgado	45678900	mathematics
45678900	Jose Delgado	45678900	english
66666666	Count Dracula	66666666	the occult

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student_id;
```

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

After selecting only tuples that satisfy the WHERE clause:

id	*name*	*student_id*	*dept_name*
25252525	Alan Turing	25252525	comp sci

After extracting the attribute specified in the SELECT clause:

dept_name
comp sci

Join Conditions

- Here's the query from the previous problem:

```
SELECT dept_name
FROM Student, MajorsIn
WHERE name = 'Alan Turing'
  AND id = student_id;
```

- `id = student_id` is a *join condition* – a condition that is used to match up "related" tuples from the two tables.
  - it selects the tuples in the Cartesian product that "make sense"
  - **for N tables, you typically need N – 1 join conditions**

Student

id	***name***
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

MajorsIn

student_id	***dept_name***
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Selecting Entire Columns

- If there's no WHERE clause, the result will consist of one or more entire columns. No rows will be excluded.

```
SELECT student_id
FROM Enrolled;
```

Enrolled

student_id	***course_name***	***credit_status***
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

SELECT student_id →

student_id
12345678
25252525
45678900
33566891
45678900

Selecting Entire Rows

- If we want the result to include entire rows (i.e., all of the columns), we use a * in the SELECT clause:

```
SELECT *
FROM Enrolled
WHERE credit_status = 'grad';
```

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

`WHERE credit_status = 'grad';`

student_id	course_name	credit_status
45678900	CS 460	grad
45678900	CS 510	grad

The WHERE Clause

SELECT *column1*, *column2*, …
FROM *table1*, *table2*, …
WHERE *selection predicate*;

- The selection predicate in the WHERE clause must consist of an expression that evaluates to either true or false.
- The predicate can include:
  - the name of any column from the table(s) mentioned in the FROM clause
  - literal values (e.g., 'graduate' or 100)
  - the standard comparison operators:
    =, !=, >, <, <=, >=
  - the logical operators AND, OR, and NOT
  - other special operators, including ones for pattern matching and handling null values

Notes:
- use single quotes for string literals
- use = (instead of ==) to test for equality

Comparisons Involving Pattern Matching

- Let's say that we're interested in getting the names and rooms of all CS courses.
  - we know that the names will all begin with 'cs'
  - we need to find courses with names that match this pattern
- Here's how:

```
SELECT name, room_id
FROM Course
WHERE name LIKE 'cs%';
```

- When pattern matching, we use the `LIKE` operator and a string that includes one or more *wildcard characters:*
  - `%` stands for 0 or more arbitrary characters
  - `_` stands for a single arbitrary character
- Can also use `NOT LIKE` to search for the absence of a pattern.

More Examples of Pattern Matching

Student

id	***name***
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

- What are the results of each query?

```
SELECT name
FROM Student
WHERE name LIKE '%u%';

SELECT name
FROM Student
WHERE name LIKE '__u%';
```

2 underscores

```
SELECT name
FROM Student
WHERE name LIKE '%u';
```

Comparisons Involving NULL

- Because NULL is a special value, any comparison involving NULL that uses the standard operators is *always* false.

- For example, all of the following will always be false:

```
room = NULL          NULL != 10
room != NULL         NULL = NULL
```

- This is useful for cases like the following:
  - assume that we add a country column to Student
  - use NULL for students whose country is unknown
  - to get all students from a foreign country:

id	*name*	*country*
12345678	Jill Jones	USA
25252525	Alan Turing	UK
33566891	Audrey Chu	China
45678900	Jose Delgado	USA
66666666	Count Dracula	NULL

```
SELECT name
FROM Student
WHERE country != 'USA';   // won't include NULLs
```

Comparisons Involving NULL (cont.)

- To test for the presence or absence of a NULL value, use special operators:

```
IS NULL
IS NOT NULL
```

- Example: find students whose country is unknown

```
SELECT name
FROM Student
WHERE country IS NULL;
```

Removing Duplicates

- By default, a SELECT command *may* produce duplicates
- To eliminate them, add the `DISTINCT` keyword:

```
SELECT DISTINCT column1, column2, …
```

Avoiding Ambiguous Column Names

- Let's say that we want to find the name and credit status of all students enrolled in cs165 who are majoring in comp sci.
- We need the following tables:
  Student(id, name)
  Enrolled(**student_id**, course_name, credit_status)
  MajorsIn(**student_id**, dept_name)
- Here's the query:

```
SELECT name, credit_status
FROM Student, Enrolled, MajorsIn
WHERE id = Enrolled.student_id
  AND Enrolled.student_id = MajorsIn.student_id
  AND course_name = 'cs165'
  AND dept_name = 'comp sci';
```

- If a column name appears in more than one table in the FROM clause, we must prepend the table name.

Renaming Attributes or Tables

- Use the keyword `AS`:

```
SELECT name AS student, credit_status
FROM Student, Enrolled AS E, MajorsIn AS M
WHERE id = E.student_id
  AND E.student_id = M.student_id
  AND course_name = 'cs165'
  AND dept_name = 'comp sci';
```

student	*credit_status*
Jill Jones	undergrad
Alan Turing	non-credit
…	…

Renaming Attributes or Tables (cont.)

- Renaming allows us to cross a relation with itself:

```
SELECT name
FROM Student, Enrolled AS E1, Enrolled AS E2
WHERE id = E1.student_id AND id = E2.student_id
  AND E1.course_name = 'CS 105'
  AND E2.course_name = 'CS 111';
```

  - what does this find?

- The use of AS is optional when defining an alias.

- I often use an alias even when it's not strictly necessary:

```
SELECT S.name
FROM Student S, Enrolled E1, Enrolled E2
WHERE S.id = E1.student_id AND S.id = E2.student_id
  AND E1.course_name = 'CS 105'
  AND E2.course_name = 'CS 111';
```

Aggregate Functions

- The SELECT clause can include an *aggregate function,* which performs a computation on a collection of values of an attribute.
- Example: find the average capacity of rooms in the Sci Ctr:

```
SELECT AVG(capacity)
FROM Room
WHERE name LIKE 'Sci Ctr%';
```

Room

id	name	capacity
1000	Sanders Theatre	1000
2000	Sever 111	50
3000	Sever 213	100
4000	Sci Ctr A	300
5000	Sci Ctr B	500
6000	Emerson 105	500
7000	Sci Ctr 110	30

WHERE

id	name	capacity
4000	Sci Ctr A	300
5000	Sci Ctr B	500
7000	Sci Ctr 110	30

AVG

AVG(capacity)
276.7

Aggregate Functions (cont.)

- Possible functions include:
  - `MIN, MAX:` find the minimum/maximum of a value
  - `AVG, SUM:` compute the average/sum of numeric values
  - `COUNT:` count the number of values
- For `AVG`, `SUM`, and `COUNT`, we can add the keyword `DISTINCT` to perform the computation on all distinct values.
  - example: find the number of students enrolled for courses:

```
SELECT COUNT(DISTINCT student)
FROM Enrolled;
```

Aggregate Functions (cont.)

- `SELECT COUNT(*)` will count the number of tuples in the result of the select command.
  - example: find the number of CS courses

```
SELECT COUNT(*)
FROM Course
WHERE name LIKE 'cs%';
```

  - `COUNT(`*attribute*`)` counts the number of <u>non-NULL</u> values of *attribute*, so it won't always be equivalent to `COUNT(*)`

- Aggregate functions <u>cannot</u> be used in the `WHERE` clause.

- Another example: write a query to find the largest capacity of any room in the Science Center:

```
SELECT MAX(capacity)
FROM Room
WHERE name LIKE 'Sci Ctr%';
```

Aggregate Functions (cont.)

- What if we wanted the name of the room with the max. capacity?

- The following will *not* work!

```
SELECT name, MAX(capacity)
FROM Room
WHERE name LIKE 'Sci Ctr%';
```

Room

id	*name*	*capacity*
1000	Sanders Theatre	1000
2000	Sever 111	50
3000	Sever 213	100
4000	Sci Ctr A	300
5000	Sci Ctr B	500
6000	Emerson 105	500
7000	Sci Ctr 110	30

WHERE

id	*name*	*capacity*
4000	Sci Ctr A	300
5000	Sci Ctr B	500
7000	Sci Ctr 110	30

name

name
Sci Ctr A
Sci Ctr B
Sci Ctr 110

MAX

MAX(capacity)
500

don't have same number of rows; **error!**

- In general, you can't mix aggregate functions with column names in the SELECT clause.

Subqueries

- We can use a *subquery* to solve the previous problem:

```
SELECT name, capacity
FROM Room
WHERE name LIKE 'Sci Ctr%'
  AND capacity = (SELECT MAX(capacity)
                  FROM Room
                  WHERE name LIKE 'Sci Ctr%');
```

the subquery

```
SELECT name, capacity
FROM Room
WHERE name LIKE 'Sci Ctr%'
  AND capacity = 500;
```

Note Carefully!

- In this case, we need the condition involving the room name in *both* the subquery and the outer query:

```
SELECT name, capacity
FROM Room
WHERE name LIKE 'Sci Ctr%'
  AND capacity = (SELECT MAX(capacity)
                  FROM Room
                  WHERE name LIKE 'Sci Ctr%');
```

the subquery

  - if we remove it from the subquery, might not get the largest capacity in Sci Ctr
  - if we remove it from the outer query, might also get rooms from other buildings
    - ones that have the max capacity found by the subquery, but are not in Sci Ctr

Subqueries and Set Membership

- Subqueries can be used to test for *set membership* in conjunction with the `IN` and `NOT IN` operators.
  - example: find all students who are <u>not</u> enrolled in CSCI E-268

```
SELECT name
FROM Student
WHERE id NOT IN (SELECT student_id
                 FROM Enrolled
                 WHERE course_name = 'cscie268');
```

Enrolled

student_id	*course_name*	*credit_status*
12345678	cscie268	ugrad
25252525	cs165	ugrad
45678900	cscie119	grad
33566891	cscie268	non-credit
45678900	cscie275	grad

Student

id	*name*
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

subquery

student_id
12345678
33566891

name
Alan Turing
Jose Delgado
Count Dracula

Applying an Aggregate Function to Subgroups

- A `GROUP BY` clause allows us to:
  - group together tuples that have a common value
  - apply an aggregate function to the tuples in each subgroup
- Example: find the enrollment of each course:

```
SELECT course_name, COUNT(*)
FROM Enrolled
GROUP BY course_name;
```

- When you group by an attribute, you <u>can</u> include it in the `SELECT` clause with an aggregate function.
  - because we're grouping by that attribute, every tuple in a given group will have the same value for it

Evaluating a query with GROUP BY

```
SELECT course_name, COUNT(*)
FROM Enrolled
GROUP BY course_name;
```

student	course_name	credit_status
12345678	cscie268	ugrad
45678900	cscie119	grad
45678900	cscie275	grad
33566891	cscie268	non-credit
66666666	cscie268	ugrad
25252525	cscie119	ugrad

student	course_name	credit_status
12345678	cscie268	ugrad
33566891	cscie268	non-credit
66666666	cscie268	ugrad
45678900	cscie119	grad
25252525	cscie119	ugrad
45678900	cscie275	grad

course_name	COUNT(*)
cscie268	3
cscie119	2
cscie275	1

Applying a Condition to Subgroups

- A HAVING clause allows us to apply a selection condition to the subgroups produced by a GROUP BY clause.
  - example: find enrollments of courses with at least 2 students

```
SELECT course_name, COUNT(*)
FROM Enrolled
GROUP BY course_name
HAVING COUNT(*) > 1;
```

Enrolled

student	course_name	credit_status
12345678	cscie268	ugrad
33566891	cscie268	non-credit
66666666	cscie268	ugrad
45678900	cscie119	grad
25252525	cscie119	ugrad
45678900	cscie275	grad

result of the query

course_name	COUNT(*)
cscie268	3
cscie119	2

- Important difference:
  - a WHERE clause is applied *before* grouping
  - a HAVING clause is applied *after* grouping

Sorting the Results

- An `ORDER BY` clause sorts the tuples in the result of the query by one or more attributes.
  - ascending order by default
  - example:

```
SELECT name, capacity
FROM Room
WHERE capacity >= 500
ORDER BY capacity;
```

name	*capacity*
Sci Ctr B	500
Emerson 105	500
Sanders Theatre	1000

Sorting the Results (cont.)

- An `ORDER BY` clause sorts the tuples in the result of the query by one or more attributes.
  - ascending order by default, use `DESC` to get descending
  - attributes after the first one are used to break ties
  - example:

```
SELECT name, capacity
FROM Room
WHERE capacity >= 500
ORDER BY capacity DESC, name;
```

name	*capacity*
Sanders Theatre	1000
Emerson 105	*500*
Sci Ctr B	*500*

Finding the Majors of Enrolled Students

- We want the IDs and majors of every student who is enrolled in a course – including those with no major.

Enrolled

student_id	*course_name*	*credit_status*
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

MajorsIn

student_id	*dept_name*
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

- Desired result:

student_id	*dept_name*
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english
33566891	null

- Relational algebra: $\pi_{\text{student\_id, dept\_name}}(\text{Enrolled} ⟕ \text{MajorsIn})$

- SQL:

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
  ON Enrolled.student_id = MajorsIn.student_id;
```

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
SELECT …
FROM T1 LEFT OUTER JOIN T2 ON join condition
WHERE …
```

- The result is equivalent to:
  - *forming the Cartesian product T1 x T2*
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Enrolled. student_id	*course_ name*	*credit_ status*	*MajorsIn. student_id*	*dept_name*
12345678	CS 105	ugrad	12345678	comp sci
12345678	CS 105	ugrad	45678900	math...
12345678	CS 105	ugrad	25252525	comp sci
12345678	CS 105	ugrad	45678900	english
12345678	CS 105	ugrad	66666666	the occult
25252525	CS 111	ugrad	12345678	comp sci
25252525	CS 111	ugrad	45678900	math...
25252525	CS 111	ugrad	25252525	comp sci
25252525	CS 111	ugrad	45678900	english
25252525	CS 111	ugrad	66666666	the occult
45678900	CS 460	grad	12345678	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	25252525	comp sci
45678900	CS 460	grad	45678900	english
45678900	CS 460	grad	66666666	the occult
33566891	CS 105	non-cr	12345678	comp sci
33566891	CS 105	non-cr	45678900	math...
33566891	CS 105	non-cr	25252525	comp sci
33566891	CS 105	non-cr	45678900	english

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
SELECT …
FROM T1 LEFT OUTER JOIN T2 ON join condition
WHERE …
```

Enrolled. student_id	course_ name	credit_ status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - *selecting the rows in T1 x T2 that satisfy the join condition in the ON clause*
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

```
SELECT …
FROM T1 LEFT OUTER JOIN T2 ON join condition
WHERE …
```

Enrolled. student_id	course_ name	credit_ status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english
33566891	CS 105	non-cr		

Enrolled

student_id	course_name	credit_status
12345678	CS 105	ugrad
25252525	CS 111	ugrad
45678900	CS 460	grad
33566891	CS 105	non-credit
45678900	CS 510	grad

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - *including an extra row for each unmatched row from T1 (the "left table")*
  - filling the T2 attributes in the extra rows with nulls
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

SELECT …
FROM *T1* LEFT OUTER JOIN *T2* ON *join condition*
WHERE …

Enrolled. student_id	course_ name	credit_ status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english
33566891	CS 105	non-cr	null	null

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - *filling the T2 attributes in the extra rows with nulls*
  - applying the other clauses as before

Left Outer Joins

```
SELECT DISTINCT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student_id = MajorsIn.student_id;
```

SELECT …
FROM *T1* LEFT OUTER JOIN *T2* ON *join condition*
WHERE …

Enrolled. student_id	course_ name	credit_ status	MajorsIn. student_id	dept_name
12345678	CS 105	ugrad	12345678	comp sci
25252525	CS 111	ugrad	25252525	comp sci
45678900	CS 460	grad	45678900	math...
45678900	CS 460	grad	45678900	english
45678900	CS 510	grad	45678900	math...
45678900	CS 510	grad	45678900	english
33566891	CS 105	non-cr	null	null

- The result is equivalent to:
  - forming the Cartesian product T1 x T2
  - selecting the rows in T1 x T2 that satisfy the join condition in the ON clause
  - including an extra row for each unmatched row from T1 (the "left table")
  - filling the T2 attributes in the extra rows with nulls

  - *applying the other clauses as before*

Enrolled. student_id	dept_name
12345678	comp sci
25252525	comp sci
45678900	mathematics
45678900	english
33566891	null

Outer Joins Can Have a WHERE Clause

- Example: find the IDs and majors of all students enrolled **in CS 105** (including those with no major):

```
SELECT Enrolled.student_id, dept_name
FROM Enrolled LEFT OUTER JOIN MajorsIn
     ON Enrolled.student_id = MajorsIn.student_id
WHERE course_name = 'CS 105';
```

  - this new condition should *not* be in the `ON` clause because it's *not* being used to match up rows from the two tables

Outer Joins Can Have Extra Tables

- Example: find the **names** and majors of all students enrolled in CS 105 (including those with no major):

```
SELECT Student.name, dept_name
FROM Student, Enrolled LEFT OUTER JOIN MajorsIn
     ON Enrolled.student_id = MajorsIn.student_id
WHERE Student.id = Enrolled.student_id
  AND course_name = 'CS 105';
```

  - the extra table requires an additional join condition, which goes in the WHERE clause

Subqueries in FROM clauses

- A subquery can also appear in a FROM clause.
- Useful when you need to perform a computation on values obtained by applying an aggregate.
  - example: find the average enrollment in a CS course

```
SELECT AVG(count)
FROM (SELECT course_name, COUNT(*) AS count
      FROM Enrolled
      GROUP BY course_name)
WHERE course_name LIKE 'cs%';
```

  - the subquery computes the enrollment of each course
  - the outer query selects the enrollments for CS courses and averages them
  - we give the attribute produced by the subquery a name, so we can then refer to it in the outer query.

CREATE TABLE

- What it does: creates a relation with the specified schema
- Basic syntax:

CREATE TABLE *relation_name*(
  attribute1_name attribute1_type,
  attribute2_name attribute2_type,
  …
  attributeN_name attributeN_type
);

- Examples:

```
CREATE TABLE Student(id CHAR(8), name VARCHAR(30));

CREATE TABLE Room(id CHAR(4), name VARCHAR(30),
  capacity INTEGER);
```

Data Types

- The set of possible types depends on the DBMS.
- Standard SQL types include:
  - `INTEGER`: a four-byte integer (-2147483648 to +2147483647)
  - `CHAR(n)`: a fixed-length string of n characters
  - `VARCHAR(n)`: a variable-length string of up to n characters
  - `REAL`: a real number (i.e., one that may have a fractional part)
  - `NUMERIC(n, d)`: a numeric value with at most n digits, exactly d of which are after the decimal point
  - `DATE`: a date of the form yyyy-mm-dd
  - `TIME`: a time of the form hh:mm:ss
- When specifying a non-numeric value, you should surround it with single quotes (e.g., 'Jill Jones' or '2007-01-26').

CHAR vs. VARCHAR

- `CHAR(n)`: a fixed-length string of exactly n characters
  - the DBMS will pad with spaces as needed
  - example: `id CHAR(6)`
    `'12345'` will be stored as `'12345 '`
- `VARCHAR(n)`: a variable-length string of up to n characters
  - the DBMS does *not* pad the value
- In both cases, values will be truncated if they're too long.
- If a string attribute can have a wide range of possible lengths, it's usually better to use VARCHAR.

Types in SQLite

- SQLite has its own types, including:
  - INTEGER
  - REAL
  - TEXT

- It also allows you to use the typical SQL types, but it converts them to one of its own types.

- As a result, the length restrictions indicated for CHAR and VARCHAR are not observed.

String Comparisons

- String comparisons ignore any trailing spaces added for padding.
  - ex: - an attribute named `id` of type `CHAR(5)`
    - insert a tuple with the value `'abc'` for `id`
    - value is stored as `'abc  '` (with 2 spaces of padding)
    - the comparison `id = 'abc'` is true for that tuple

- In standard SQL, string comparisons using both `=` and `LIKE` are case sensitive.
  - some DBMSs provide a case-insensitive version of `LIKE`

- In SQLite:
  - there are no real CHARs, so padding isn't added
  - string comparisons using = are case sensitive
  - string comparisons using `LIKE` are case insensitive

    `'abc' = 'ABC'` is false

    `'abc' LIKE 'ABC'` is true

Terms Used to Express Constraints

- primary key:

```
CREATE TABLE Student(id char(8) primary key,
 name varchar(30));
CREATE TABLE Enrolled(student_id char(8),
 course_name varchar(20), credit_status varchar(15),
 primary key (student_id, course_name));
```

  - no two tuples can have the same combination of values for the primary-key attributes
  - a primary-key attribute can never have a null value

- unique: specifies attribute(s) that form a (non-primary) key

```
CREATE TABLE Course(name varchar(20) primary key,
   start_time time, end_time time, room_id char(4),
   unique (start_time, end_time, room_id));
```

  - a unique attribute *may* have a null value

- not null: specifies that an attribute can never be null

```
CREATE TABLE Student(id char(8) primary key,
   name varchar(30) not null);
```

Terms Used to Express Constraints (cont.)

- foreign key … references:

```
CREATE TABLE MajorsIn(student_id char(8),
 dept_name varchar(30),
 foreign key (student_id) references Student(id),
 foreign key (dept_name) references
    Department(name));
```

MajorsIn

student_id	***dept_name***
12345678	comp sci
45678900	mathematics
...	...

Student

id	***name***
12345678	Jill Jones
25252525	Alan Turing
...	...

Department

name	***office***
comp sci	MD 235
mathematics	Sci Ctr 520
...	...

Terms Used to Express Constraints (cont.)

- foreign key / references (cont.):
  - imposes a *referential integrity* constraint (see last set of slides)
  - a foreign-key attribute *may* have a null value

DROP TABLE

- What it does: removes an entire relation from a database
  - including all of its existing rows
- Syntax:

  `DROP TABLE` *relation_name*`;`
- Example:

```
DROP TABLE MajorsIn;
```

INSERT

- What it does: adds a tuple to a relation
- Syntax:
  `INSERT INTO` *relation* `VALUES (`*val1*`,` *val2*`, …);`
  - the values of the attributes must be given in the order in which the attributes were specified when the table was created
- Example:
  `INSERT INTO MajorsIn VALUES ('10005000', 'math');`

  [Recall the `CREATE TABLE` command:
  ```
  CREATE TABLE MajorsIn(
   student_id char(8), dept_name varchar(30),…); ]
  ```

INSERT (cont.)

- Alternate syntax:
  `INSERT INTO` *relation*`(`*attr1*`,` *attr2*`,…)`
  `VALUES (`*val1*`,` *val2*`, …);`
  - allows you to:
    - specify values of the attributes in a different order
    - specify values for only a subset of the attributes
- Example:
  ```
  INSERT INTO MajorsIn(dept_name, student_id)
        VALUES ('math', '10005000');
  ```
- If the value of a column is not specified, it is assigned a default value.
  - depends on the data type of the column

DELETE

- What it does: remove one or more tuples from a relation
  - basic syntax:

    DELETE FROM *table*
    WHERE *selection condition*;

  - example:

```
DELETE FROM Student
WHERE id = '4567800';
```

UPDATE

- What it does: modify attributes of one or more tuples in a relation
  - basic syntax:

    UPDATE *table*
    SET *list of assignments*
    WHERE *selection condition*;

  - examples:

```
UPDATE MajorsIn
SET dept_name = 'physics'
WHERE student_id = '10005000';

UPDATE Course
SET start_time = '11:00:00', end = '12:30:00'
WHERE name = 'cs165';
```

Writing Queries: Rules of Thumb

- Start with the FROM clause. Which table(s) do you need?
- If you need more than one table, determine the necessary join conditions.
  - for N tables, you typically need N – 1 join conditions
  - is an outer join needed – i.e., do you want unmatched tuples?
- Determine if a GROUP BY clause is needed.
  - are you performing computations involving subgroups?
- Determine any other conditions that are needed.
  - if they rely on aggregate functions, put in a HAVING clause
  - otherwise, add to the WHERE clause
  - is a subquery needed?
- Fill in the rest of the query: SELECT, ORDER BY?
  - is DISTINCT needed?

Which of these problems would require a GROUP BY?

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id) Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

A. finding the Best-Picture winner with the best/smallest earnings rank

B. finding the number of Oscars won by each person that has won an Oscar

C. finding the number of Oscars won by each person, including people who have not won any Oscars

D. both B and C, but not A

E. A, B, and C

Which would require a subquery?

Which would require a LEFT OUTER JOIN?

Now Write the Queries!

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)   Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

1) Find the Best-Picture winner with the best/smallest earnings rank. The result should have the form (name, earnings_rank). Assume no two movies have the same earnings rank.

```
SELECT
FROM
WHERE                   (SELECT
                         FROM
                         WHERE                  );
```

Now Write the Queries!

Person(id, name, dob, pob)
Movie(id, name, year, rating, runtime, genre, earnings_rank)
Actor(actor_id, movie_id)   Director(director_id, movie_id)
Oscar(movie_id, person_id, type, year)

2) Find the number of Oscars won by each person that has won an Oscar. Produce tuples of the form (name, num Oscars).

```
SELECT
FROM
WHERE
```

3) Find the number of Oscars won by each person, including people who have not won an Oscar.

Student

id	name
12345678	Jill Jones
25252525	Alan Turing
33566891	Audrey Chu
45678900	Jose Delgado
66666666	Count Dracula

Room

id	name	capacity
1000	Sanders Theatre	1000
2000	Sever 111	50
3000	Sever 213	100
4000	Sci Ctr A	300
5000	Sci Ctr B	500
6000	Emerson 105	500
7000	Sci Ctr 110	30

Course

name	start_time	end_time	room_id
cscie119	19:35:00	21:35:00	4000
cscie268	19:35:00	21:35:00	2000
cs165	16:00:00	17:30:00	7000
cscie275	17:30:00	19:30:00	7000

Department

name	office
comp sci	MD 235
mathematics	Sci Ctr 520
the occult	The Dungeon
english	Sever 125

Enrolled

student_id	course_name	credit_status
12345678	cscie268	ugrad
25252525	cs165	ugrad
45678900	cscie119	grad
33566891	cscie268	non-credit
45678900	cscie275	grad

MajorsIn

student_id	dept_name
12345678	comp sci
45678900	mathematics
25252525	comp sci
45678900	english
66666666	the occult

Practice Writing Queries

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

1) Find all rooms that can seat at least 100 people.

2) Find the course or courses with the earliest start time.

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

3) Find the number of majors in each department.

4) Find all courses taken by CS ('comp sci') majors.

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

5) Create a list of all Students who are *not* enrolled in a course.

Why won't this work?

```
SELECT name
FROM Student, Enrolled
WHERE Student.id != Enrolled.student_id;
```

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

6) Find the number of CS majors enrolled in cscie268.

6b) Find the number of CS majors enrolled in any course.

Practice Writing Queries (cont.)

Student(id, name) Department(name, office) Room(id, name, capacity)
Course(name, start_time, end_time, room_id) MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

7) Find the number of majors that each student has declared.

Practice Writing Queries (cont.)

Student(id, name)   Department(name, office)   Room(id, name, capacity)
Course(name, start_time, end_time, room_id)   MajorsIn(student_id, dept_name)
Enrolled(student_id, course_name, credit_status)

8) For each department with more than one majoring student, output the department's name and the number of majoring students.

Storage and Indexing

Harvard Extension School

Cody Doucette, Ph.D.

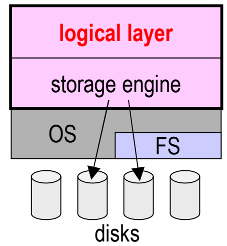
Lecture designed by David G. Sullivan

Accessing the Disk

- Data is arranged on disk in units called *blocks*.
  - typically fairly large (e.g., 4K or 8K)
- Relatively speaking, disk I/O is very expensive.
  - in the time it takes to read a single disk block,
    the processor could be executing millions of instructions!
- The DBMS tries to minimize the number of disk accesses.

Review: DBMS Architecture

- A DBMS can be viewed as a composition of two layers.
- At the bottom is the *storage layer* or *storage engine*, which takes care of storing and retrieving the data.
- Above that is the *logical layer*, which provides an abstract representation of the data.



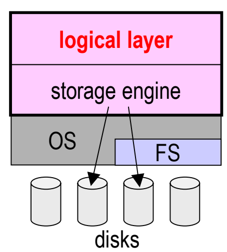
Logical-to-Physical Mapping

- The logical layer implements a mapping between:

  the *logical schema* of a database

  ↓ ↑

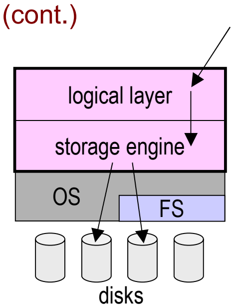
  its *physical representation*



- In the relational model, the schema includes:
  - attributes/columns, including their types
  - tuples/rows
  - relations/tables
- To be model-neutral, we'll use these terms instead:
  - *field* for an individual data value
  - *record* for a group of fields
  - *collection* for a group of records

Logical-to-Physical Mapping (cont.)

- A DBMS may use the filesystem, or it may bypass it and use its own disk manager.



- In either case, a DBMS may use units called *pages* that have a different size than the block size.
  - can be helpful in performance tuning

Logical-to-Physical Mapping (cont.)

- We'll consider:
  - how to map logical records to their physical representation
  - how to organize the records in a given collection
    - including the use of index structures
- Different approaches require different amounts of *metadata* – data about the data.
  - example: the types and lengths of the fields
  - *per-record* metadata – stored within each record
  - *per-collection* metadata – stored once for the entire collection
- Assumptions about data in the rest of this set of slides:
  - each character is stored using 1 byte
  - integer data values are stored using 4 bytes
  - integer metadata (e.g., offsets) are stored using 2 bytes

Fixed- or Variable-Length Records?

- This choice depends on:
  - the types of fields that the records contain
  - the number of fields per record, and whether it can vary
- Simple case: use fixed-length records when
  - all fields are fixed-length (e.g., CHAR or INTEGER),
  - there is a fixed number of fields per record

Fixed- or Variable-Length Records? (cont.)

- The choice is less straightforward when you have either:
  - variable-length fields (e.g., VARCHAR)
  - a variable number of fields per record (e.g., in XML)

Two options:

1. fixed-length records: always allocate the maximum possible length

...	`comp sci`	...
...	`math`	...

   - plusses and minuses:
     - \+ less metadata is needed, because:
       - every record has the same length
       - a given field is in a consistent position within all records
     - \+ changing a field's value doesn't change the record's length
       - thus, changes never necessitate moving the record
     - – we waste space when a record has fields shorter than their max length, or is missing fields

Fixed- or Variable-Length Records? (cont.)

2. variable-length records: only allocate the space that each record actually needs

…	comp sci	…
…	math	…

   - plusses and minuses:
     – more metadata is needed in order to:
       - determine the boundaries between records
       - determine the locations of the fields in a given record

     – changing a field's value can change the record's length
       - thus, we may need to move the record

     + we *don't* waste space when a record has fields shorter than their max length, or is missing fields

Format of Fixed-Length Records

- With fixed-length records, we store the fields one after the other.
- If a fixed-length record contains a variable-length field:
  - allocate the max. length of the field
  - use a delimiter (**#** below) if the value is shorter than the max.
- Example:
  `Dept(id CHAR(7), name VARCHAR(20), num_majors INT)`

id	name	num_majors
1234567	comp sci#	200
9876543	math#	125
4567890	history & literature	175

  - why doesn't `'history & literature'` need a delimiter?

Format of Fixed-Length Records (cont.)

- To find the position of a field, use *per-collection* metadata.
  - typically store the *offset* of each field (O_1 and O_2 below) – how many bytes the field is from the start of the record

id	name	num_majors
1234567	comp sci#	200
9876543	math#	125
4567890	history & literature	175

← O_1 →

← O_2 →

- Notes:
  - the delimiters are the only *per-record* metadata
  - the records are indeed fixed-length – 31 bytes each!
    - 7 bytes for `id`, which is a `CHAR(7)`
    - 20 bytes for `name`, which is a `VARCHAR(20)`
    - 4 bytes for `num_majors`, which is an `INT`

Format of Variable-Length Records

- With variable-length records, we need *per-record* metadata to determine the locations of the fields.

- For simplicity, we'll assume all records in a given collection have the same # of fields.

- We'll look at how the following record would be stored:

```
  CHAR(7)      VARCHAR(20)    INT
('1234567', 'comp sci', 200)
```

- We'll consider two types of operations:
  1. finding/extracting the value of a single field

     ```
     SELECT num_majors
     FROM Dept
     WHERE name = 'comp sci';
     ```

  2. updating the value of a single field
     - its length may become smaller or larger

Format of Variable-Length Records (cont.)

- Option 1: Terminate field values with a special delimiter character.

CHAR(7)		VARCHAR(20)		INT	
1234567	#	comp sci	#	200	#

1. finding/extracting the value of a single field
   this is very inefficient; need to scan byte-by-byte to:
   - find the start of the field we're looking for
   - determine the length of its value (if it is variable-length)
2. updating the value of a single field
   if it changes in size, we need to shift the values after it,
   but we don't need to change their metadata

Format of Variable-Length Records (cont.)

- Option 2: Precede each field by its length.

	CHAR(7)		VARCHAR(20)		INT
7	1234567	8	comp sci	4	200

1. finding/extracting the value of a single field
   this is more efficient
   - can jump over fields, rather than scanning byte-by-byte (but may need to perform multiple jumps)
   - never need to scan to determine the length of a value
2. updating the value of a single field
   same as option 1

Format of Variable-Length Records (cont.)

- Option 3: Put offsets and other metadata in a record header.

# of bytes from the start of the record

0	2	4	6	8	15	23
8	15	23	27	1234567	comp sci	200

record header

computing the offsets

- 3 fields in record → 4 offsets, each of which is a 2-byte int
- thus, the offsets take up 4*2 = 8 bytes
- $offset_0$ = 8, because $field_0$ comes right after the header
- $offset_1$ = 8 + len('1234567') = 8 + 7 = 15
- $offset_2$ = 15 + len('comp sci') = 15 + 8 = 23
- $offset_3$ = *offset of the end of the record*
  = 23 + 4 (since 200 an int) = 27
  We store this offset because it may be needed to compute the length of a field's value!

Format of Variable-Length Records (cont.)

- Option 3 (cont.)

0	2	4	6	8	15	23
8	15	23	27	1234567	comp sci	200

1. finding/extracting the value of a single field
   this representation is the most efficient. it allows us to:
   - jump directly to the field we're interested in
   - compute its length without scanning through its value
2. updating the value of a single field
   less efficient than options 1 and 2 if the length changes. why?

Representing Null Values

- Option 1: add an "out-of-band" value for every data type
  - con: need to increase the size of most data types, or reduce the range of possible values
- Option 2: use per-record metadata
  - example: use a special offset (e.g., -1)

0	2	4	6	8	15
8	15	-1	23	1234567	comp sci

Index Structures

- An index structure stores (key, value) pairs.
  - also known as a *dictionary* or *map*
  - we will sometimes refer to the (key, value) pairs as *items*
- The index allows us to more efficiently access a given record.
  - quickly find it based on a particular field
  - instead of scanning through the entire collection to find it
- A given collection of records may have multiple index structures:
  - one *clustered* or *primary* index
  - some number of *unclustered* or *secondary* indices

Clustered/Primary Index

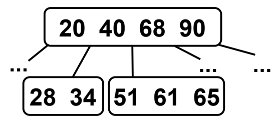
- The *clustered* index is the one that stores the full records.
  - also known as a *primary index,* because it is typically based on the primary key
- If the records are stored outside of an index structure, the resulting file is sometimes called a *heap file*.
  - managed somewhat like the heap memory region

Unclustered/Secondary Indices

- In addition to the clustered/primary index, there can be one or more *unclustered* indices based on other fields.
  - also known as *secondary indices*
- Example: *Customer(id, name, street, city, state, zip)*
  - primary index:
    (key, value) = (*id*, all of the remaining fields in the record)
  - a secondary index to enable quick searches by name
    (key, value) = (*name*, *id*)   *does not include the other fields!*
- We need two lookups when we start with the secondary index.
  - example: looking for Ted Codd's zip code
  - search for `'Ted Codd'` in the secondary index
    ➔ `'123456'` (his id)
  - search for `'123456'` in the primary index
    ➔ his full record, including his zip code

B-Trees

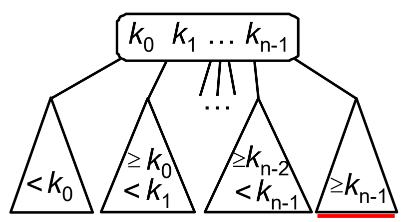
- A B-tree of order *m* is a tree in which each node has:
  - at most 2*m* items (and, for internal nodes, 2*m* + 1 children)
  - at least *m* items (and, for internal nodes, *m* + 1 children)
  - exception: the root node may have as few as 1 item
- Example: a B-tree of order 2



(we're just showing the keys)

- A B-tree has *perfect balance*: all paths from the root node to a leaf node have the same length.

Search in B-Trees

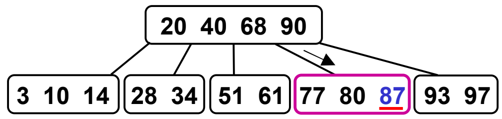


each triangle is a subtree

- A B-tree is a *search tree*.
  - like a binary search tree, but can have more keys per node
- When searching for an item whose key is *k*, we never need to enter more than one of the subtrees of a node.

Search in B-Trees (cont.)

- Example: search for the item whose key is 87



- Here's pseudocode for the algorithm:

```
search(key, node) {
   if (node == null) return null;
   i = 0;
   while (i < node.numkeys  &&  node.key[i] < key)
      i++;
   if (i == node.numkeys || node.key[i] != key)
      return search(key, node.child[i]);
   else        // node.key[i] == key
      return node.data[i];
}
```

Insertion in B-Trees

- Algorithm for inserting an item with a key *k*:

  search for *k* until you reach a leaf node

  if the leaf node has fewer than 2*m* items, add the new item to the leaf node

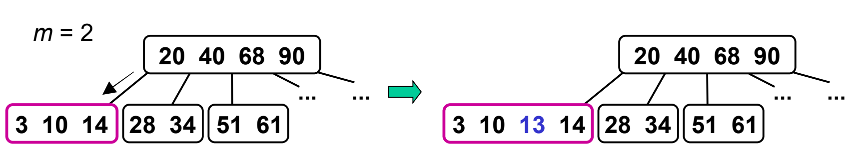
  else *split the node*, dividing up the 2*m* + 1 items:

    the first/smallest *m* items remain in the original node

    the last/largest *m* items go in a new node

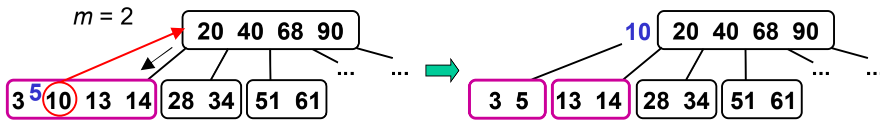
    send the middle item up and insert it (and a pointer to the new node) in the parent

- Example of an insertion without a split: insert 13

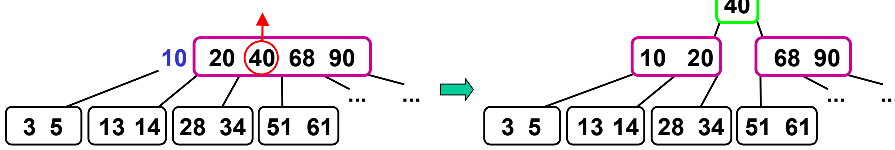


Splits in B-Trees

- Insert 5 into the result of the previous insertion:

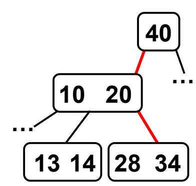


- The middle item (the 10) is sent up to the root.
  The root has no room, so it is also split, and a new root is formed:



- Splitting the root increases the tree's height by 1, but it remains balanced! This is only way the height increases.
- When an internal node is split, its 2*m* + 2 pointers are split evenly between the original node and the new node.

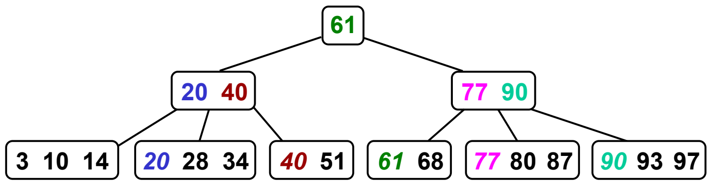
Other Details of B-Trees



- Each node in the tree corresponds to one page in the corresponding index file.
  - child pointers = page numbers
- Efficiency: In the worst case, searching for an item involves traversing a single path from the root to a leaf node.
  - # of nodes accessed <= tree height + 1
  - each internal node has at least m children
    - → tree height <= $\log_m n$, where n = # of items
    - → search and insertion are $O(\log_m n)$
- To minimize disk I/O, make *m* as large as possible.
  - but not too large!
  - if *m* is too large, can end up with items that don't fit on the page and are thus stored in separate *overflow pages*

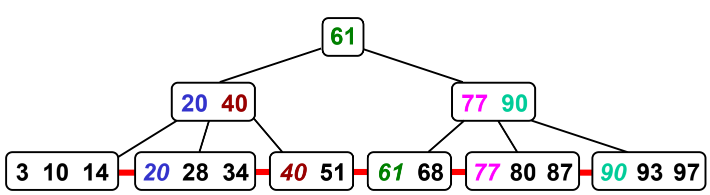
B+Trees

- A B+tree is a B-tree variant in which:
  - data items are only found in the leaf nodes
  - internal nodes contain only keys and child pointers
  - an item's key may appear in a leaf node *and* an internal node
- Example: a B+tree of order 2



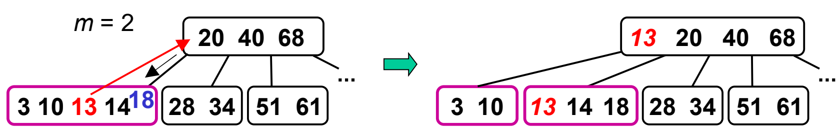
B+Trees (cont.)

- Advantages:
  - there's more room in the internal nodes for child pointers
    - why is this beneficial?
  - because all items are in leaf nodes, we can link the leaves together to improve the efficiency of operations that involve scanning the items in key order (e.g., range searches)



Differences in the Algorithms for B+Trees

- When searching, we keep going until we reach a leaf node, even if we see the key in an internal node.
- When splitting a leaf node with 2*m* + 1 items:
  - the first *m* items remain in the original node as before
  - *all* of the remaining *m* + 1 items are put in the new node, *including the middle item*
  - the *key* of the middle item is *copied* into the parent
    - why can't we move up the entire item as before?
- Example: insert 18



Differences in the Algorithms for B+Trees (cont.)

- Splitting an internal node is the same as before, but with keys only:
  - first *m* keys stay in original node, last *m* keys go to new node
  - middle key is sent up to parent (not copied)

Deletion in B-Trees and B+Trees

- Search for the item and remove it.
- If a node N ends up with fewer than *m* items, do one of the following:
  - if a sibling node has more than *m* items, take items from it and add them to N
  - if the sibling node only has *m* items, *merge* N with the sibling
- If the key of the removed item is in an internal node, *don't* remove it from the internal node.
  - we need the key to navigate to the node's children
  - can remove when the associated child node is merged with a sibling
- Some systems don't worry about nodes with too few items.
  - assume items will be added again eventually

Ideal Case: Searching = Indexing

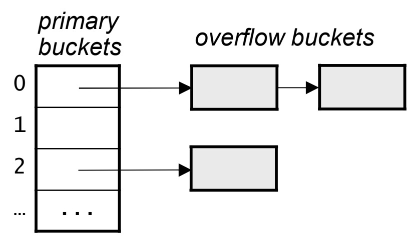
- The ideal index structure would be one in which:
  key of data item = the page number where the item is stored
- In most real-world problems, we can't do this.
  - the key values may not be integers
  - we can't afford to give each key value its own page
- To get something close to the ideal, we perform *hashing*:
  - use a *hash function* to convert the keys to page numbers
    `h('hello')` → `5`
- The resulting index structure is known as a *hash table*.

Hash Tables: In-Memory vs. On-Disk

- In-memory:
  - the hash value is used as an index into an array
  - depending on the approach you're taking, a given array element may only hold one item
  - need to deal with *collisions* = two values hashed to same index
- On-disk:
  - the hash value tells you which *page* the item should be on
  - because pages are large, each page serves as a *bucket* that stores multiple items
  - need to deal with full buckets

Static vs. Dynamic Hashing

- In *static hashing*, the number of buckets never changes.
  - if a bucket becomes full, we use overflow buckets/pages



  - why is this problematic?
- In *dynamic hashing*, the number of buckets can grow over time.
  - can be expensive if you're not careful!

A Simplistic Approach to Dynamic Hashing

- Assume that:
  - we're using keys that are strings
  - h(key) = number of characters in key
  - we use mod (%) to ensure we get a valid bucket number:

    bucket index = h(key) % number of buckets

- When the hash table gets to be too full:
  - double the number of buckets
  - rehash **all** existing items. why?

0	`"if", "case", "continue"`
1	`"class", "for", "extends"`

→

0	`"case", "continue"`
1	`"class"`
2	`"if"`
3	`"for", "extends"`

Linear Hashing

- It does ***not*** use the modulus to determine the bucket index.
- Rather, it treats the hash value as a binary number, and it uses the i *rightmost* bits of that number:

  i = ceil($\log_2 n$) where n is the current number of buckets
  - example: n = 3 → i = ceil($\log_2 3$) = 2

- If there's a bucket with the index given by the i rightmost bits, put the key there.

```
      h("if") = 2 = 00000010
    h("case") = 4 = 00000100
   h("class") = ?
h("continue") = ?
```

00 = 0	`"case"`
01 = 1	
10 = 2	`"if"`

- If not, use the bucket specified by the rightmost i – 1 bits

```
    h("for") = 3 = 00000011   (11 = 3 is too big, so use 1)
h("extends") = ?
```

Linear Hashing: Adding a Bucket

- In linear hashing, we keep track of three values:
  - n, the number of buckets
  - i, the number of bits used to assign keys to buckets
  - f, some measure of how full the buckets are
- When f exceeds some threshold, we:
  - add **only one** new bucket
  - increment n and update i as needed
  - rehash/move keys as needed
- We only need to rehash the keys in **one** of the old buckets!
  - if the new bucket's binary index is 1xyz (xyz = arbitrary bits), rehash the bucket with binary index 0xyz
- Linear hashing has to grow the table more often, but each new addition takes very little work.

Example of Adding a Bucket

- Assume that:
  - our measure of fullness, f = # of items in hash table
  - we add a bucket when f > 2*n
- Continuing with our previous example:
  - n = 3; f = 6 = 2*3, so we're at the threshold
  - adding "switch" exceeds the threshold, so we:
    - add a new bucket whose index = 3 = 11 in binary
    - increment n to 4 → i = ceil($\log_2 4$) = 2 *(unchanged)*

`n = 3, i = 2`

00 = 0	"case", "continue"
01 = 1	"class", "for", "extends"
10 = 2	"if", *"switch"*

`n = 4, i = 2`

00 = 0	"case", "continue"
01 = 1	"class", "for", "extends"
10 = 2	"if", "switch"
11 = 3	

Example of Adding a Bucket (cont.)

- Which previous bucket do we need to rehash?
  - new bucket has a binary index of 11
  - because this bucket wasn't there before,
    items that should now be in 11 were originally put in 01
    (using the rightmost i – 1 bits)

n = 4, i = 2

00 = 0	"case", "continue"
01 = 1	"class", "for", "extends"
10 = 2	"if", "switch"
11 = 3	

Example of Adding a Bucket (cont.)

- Which previous bucket do we need to rehash?
  - new bucket has a binary index of 11
  - because this bucket wasn't there before, items that should now be in 11 were originally put in 01 (using the rightmost i – 1 bits)
  - thus, we rehash bucket 01:
    - `h("class") = 5 = 00000101` (leave where it is)
    - `h("for") = 3 = 00000011` (move to new bucket)
    - `h("extends") = 7 = 00000111`

`n = 4, i = 2`

00 = 0	"case", "continue"
01 = 1	"class", "for", "extends"
10 = 2	"if", "switch"
11 = 3	

→

`n = 4, i = 2`

00 = 0	"case", "continue"
01 = 1	"class"
10 = 2	"if", "switch"
11 = 3	"for"

Additional Details

- If the number of buckets exceeds 2^i, we increment i and begin using one additional bit.

`n = 4, i = 2, f = 9, 9 > 2*4`

00 = 0	"case", "continue"
01 = 1	"class", *"while"*
10 = 2	"if", "switch", *"String"*
11 = 3	"for", "extends"

→

`n = 5, i = 3`

000 = 0	"case", "continue"
001 = 1	"class", "while"
010 = 2	"if", "switch", "String"
011 = 3	"for", "extends"
100 = 4	

which bucket should be rehashed?

Additional Details

- If the number of buckets exceeds 2^i, we increment i and begin using one additional bit.

`n = 4, i = 2, f = 9, 9 > 2*4`

00 = 0	"case", "continue"
01 = 1	"class", *"while"*
10 = 2	"if", "switch", *"String"*
11 = 3	"for", "extends"

⇨

`n = 5, i = 3`

000 = 0	"continue"
001 = 1	"class", "while"
010 = 2	"if", "switch", "String"
011 = 3	"for", "extends"
100 = 4	"case"

- The process of adding a bucket is sometimes referred to as *splitting* a bucket.
  - example: adding bucket 4 <==> splitting bucket 0
    because some of 0's items may get moved to bucket 4

- The split bucket:
  - may retain all, some, or none of its items
  - may not be as full as other buckets
    - thus, linear hashing still allows for overflow buckets as needed

More Examples

- Assume again that we add a bucket whenever the # of items exceeds 2n.
- What will the table below look like after inserting the following sequence of keys? (assume no overflow buckets are needed)

`"toString":   h("toString")  = ?`

`n = 5, i = 3`

000 = 0	"continue"
001 = 1	"class", "while"
010 = 2	"if", "switch", "String"
011 = 3	"for", "extends"
100 = 4	"case"

Hash Table Efficiency

- In the best case, search and insertion require at most one disk access.
- In the worst case, search and insertion require *k* accesses, where *k* is the length of the largest bucket chain.
- Dynamic hashing can keep the worst case from being too bad.

Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can't easily:
  - access the records in sorted order
  - perform a range search
  - perform a rank search – get the kth largest value of some field

  We *can* do all of these things with a B-tree / B+tree.

Which Index Structure Should You Choose?

- Recently accessed pages are stored in a *cache* in memory.
- Working set = collection of frequently accessed pages
- If the working set fits in the cache, use a B-tree / B+tree.
  - efficiently supports a wider range of queries (see last slide)
- If the working set can't fit in memory:
  - choose a B-tree/B+tree if the workload exhibits *locality*
    - locality = a query for a key is often followed by a query for a key that is nearby in the space of keys
    - because the items are sorted by key, the neighbor will be in the cache
  - choose a hash table if the working set is very large
    - uses less space for "bookkeeping" (pointers, etc.), and can thus fit more of the working set in the cache
    - fewer operations are needed before going to disk

Implementing a Logical-to-Physical Mapping

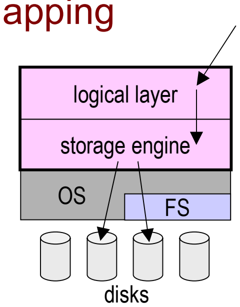
Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Recall: Logical-to-Physical Mapping

- Recall our earlier diagram of a DBMS, which divides it into two layers:
  - the *logical layer*
  - the *storage layer* or *storage engine*

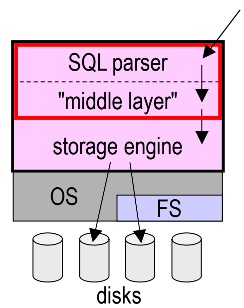


- The logical layer implements a mapping from the logical schema of a collection of data to its physical representation.
  - example: for the relational model, it maps:

attributes		fields
tuples	to	records
relations		files and index structures
selects, projects, etc.		scans, searches, field extractions

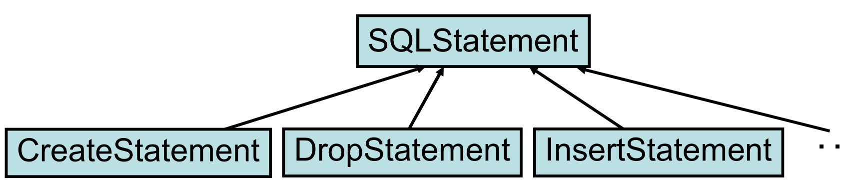
Your Task

- On the homework, you will implement portions of the logical-to-physical mapping for a simple relational DBMS.
- We're giving you:
  - a SQL parser
  - a storage engine: Berkeley DB
  - portions of the code needed for the mapping, and a framework for the code that you will write
- In a sense, we've divided the logical layer into two layers:
  - a SQL parser
  - everything else – the "middle layer"
    - you'll implement parts of this



The Parser

- Takes a string containing a SQL statement
- Creates an instance of a subclass of the class `SQLStatement`:



- `SQLStatement` is an *abstract class*.
  - contains fields and methods inherited by the subclasses
  - includes an *abstract* `execute()` method
    - just the method header, not the body
- Each subclass implements its own version of `execute()`
  - you'll do this for some of the subclasses

SQLStatement Class

- Looks something like this:

```
public abstract class SQLStatement {
    private ArrayList<Table> tables;
    private ArrayList<Column> columns;
    private ArrayList<Object> columnVals;
    private ConditionalExpression where;
    private ArrayList<Column> whereColumns;

    public abstract void execute();
    …
```

Other Aspects of the Code Framework

- `DBMS`: the "main" class
  - methods to initialize, shutdown, or abort the system
  - methods to maintain and access the state of the system
  - to allow access to the `DBMS` methods from other classes, we make its methods static
    - this means the class name can be used to invoke them
- Classes that represent relational constructs, including:
  - `Table`
  - `Column`
  - `InsertRow`: a row that is being prepared for insertion in a table
- `Catalog`: a class that maintains the per-table metadata
  - here again, the methods are static

The Storage Engine: Berkeley DB (BDB)

- An embedded database library for managing key/value pairs
  - fast: runs in the application’s address space, no IPC
  - reliable: transactions, recovery, etc.
- One example of a type of noSQL database known as a key-value store.
- We're using Berkeley DB Java Edition (JE)
- Note: We're *not* using the Berkeley DB SQL interface.
  - we're writing our own!

Berkeley DB Terminology

- A *database* in BDB is a collection of key/value pairs that are stored in the same index structure.
  - BDB docs say "key/data pairs" instead of "key/value pairs"
- BDB Java Edition always uses a B+tree.
  - other versions of BDB provide other index-structure options
- A database is operated on by making method calls using a *database handle* – an instance of the `Database` class.
- We will use one BDB database for each table/relation.

Berkeley DB Terminology (cont.)

- An *environment* in BDB encapsulates:
  - a set of one or more related BDB databases
  - the state associated with the BDB subsystems for those databases
- RDBMS: related tables are grouped together into a database.
  BDB: related databases are grouped together into an environment.
- Files for a given environment are put in the same folder.
  - known as the environment's *home directory*

Opening/Creating a BDB Database

- We give you the code for this in the DBMS framework:
  - `CreateStatement.execute()` creates a database for a new table
  - `Table.open()` opens the database for an existing table
- Use the table's primary key for the keys in the key/value pairs.
  - if one wasn't specified when the table was created, we use the first column
  - can assume no multi-attribute primary keys

Key/Value Pairs

- When manipulating keys and values within a program, we represent them using a `DatabaseEntry` object.
- For a given key/value pair, we need *two* `DatabaseEntrys`.
  - one for the key
  - one for the value
- Each `DatabaseEntry` encapsulates:
  - a reference to the collection of bytes (the *data*)
  - the *size* of the data (i.e., its length in bytes)
  - some additional fields
  - methods: `getData`, `getSize`, …
  - consult the Berkeley DB API for info on the methods!

Byte Arrays

- In Berkeley DB, the on-disk keys and values are *byte arrays* – i.e., arbitrary collections of bytes.
- Berkeley DB does *not* attempt to interpret them.
- Your code will need to impose structure on these byte arrays.

Marshalling the Data

- When inserting a row, we need to turn a collection of fields into a key/value pair.
  - example:

('9876543', 'psych', 125) →

key

9876543

value

-2	8	13	17	pysch	125

- In BDB, the key and value are each:
  - represented by a `DatabaseEntry` object
  - based on a byte array that we need to create
- This process is referred to as *marshalling* the data.
- The reverse process is known as *unmarshalling*.

The Required Record Format

- Here's what option 3 did:

('1234567', 'comp sci', 200) →

8	15	23	27	1234567	comp sci	200

- We'll do something a bit different:

('1234567', 'comp sci', 200) →

key

1234567

value

-2	?	?	?	comp sci	200

  - the primary-key value becomes the key in the key/value pair
  - the value is the other fields with a header of offsets
  - we use a special offset for the primary-key in the header (note: it won't always be the first column!)
  - what should the remaining offsets be in this case? (assume 2-byte offsets and 4-byte integer values)

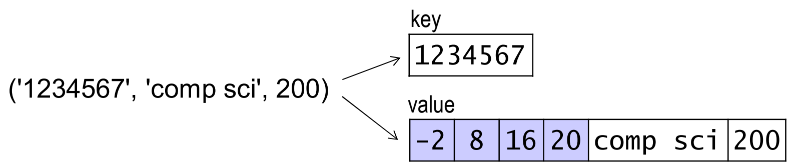
Classes for Manipulating Byte Arrays

- `RowOutput`: an output stream that writes into a byte array
  - inherits from Java's `DataOutputStream`:
    - `writeBytes(String val)`
    - `writeShort(int val)  // can use for offsets!`
    - `writeInt(int val)`
    - `writeDouble(double val)`
  - methods for obtaining the results of the writes:
    - `getBufferBytes()`
    - `getBufferLength()`
  - includes a `toString()` method that shows the current contents of the byte array

Classes for Manipulating Byte Arrays (cont.)

- `RowInput`: an input stream that reads from a byte array
  - methods that take an offset from the start of the byte array
    - `readBytesAtOffset(int offset, int length)`
    - `readIntAtOffset(int offset)`
    - etc.
  - methods that read from the current offset (i.e., from where the last read left off)
    - `readNextBytes(int length)`
    - `readNextInt()`
    - etc.
  - includes a `toString()` method that shows the contents of the byte array and the current offset

Example of Marshalling



- Marshalling this row could be done as follows:

```
RowOutput keyBuffer = new RowOutput();
keyBuffer.writeBytes("1234567");

RowOutput valuebuffer = new RowOutput();
valueBuffer.writeShort(-2);
valueBuffer.writeShort(8);
valueBuffer.writeShort(16);
valueBuffer.writeShort(20);
valueBuffer.writeBytes("comp sci");
valueBuffer.writeInt(200);
```

Inserting Data into a BDB Database

- Create the `DatabaseEntry` objects for the key and value:

```
// see previous slide for marshalling code
byte[] bytes = keyBuffer.getBufferBytes();
int numBytes = keyBuffer.getBufferLength();
DatabaseEntry key = new DatabaseEntry(bytes, 0, numBytes);

bytes = valueBuffer.getBufferBytes();
numBytes = valueBuffer.getBufferLength();
DatabaseEntry value = new DatabaseEntry(bytes, 0, numBytes);
```

- Use the `Database` `putNoOverwrite` method:

```
Database db;  // assume it has been opened
OperationStatus ret = db.putNoOverwrite(null, key, value);
```

  - `null` because we are not using transactions
  - if there is an existing key/value pair with the specified key:
    - the insertion fails
    - the method returns `OperationStatus.KEYEXIST`
  - if the insertion succeeds, returns `OperationStatus.SUCCESS`

Cursors in Berkeley DB

- In general, a *cursor* is a construct used to iterate over records in a database file.
  - similar to an iterator for a collection class
- In BDB, cursors iterate over key/value pairs in a BDB database.
  - based on method calls using an instance of the `Cursor` class
- The key/value pairs are returned in "empty" `DatabaseEntrys` that are passed as parameters to the cursor's `getNext` method:

```
DatabaseEntry key = new DatabaseEntry();
DatabaseEntry value = new DatabaseEntry();
OperationStatus ret = curs.getNext(key, value, null);
```

Table Iterators

- In PS 2, a cursor is used to implement a `TableIterator` class.
- It can be used to iterate over the tuples in either:
  - an entire single table:

```
SELECT *
FROM Movie;
```

  - *or* the relation that is produced by applying a selection operator to the tuples of single table:

```
SELECT *
FROM Movie
WHERE rating = 'PG-13' and year > 2010;
```

- A `TableIterator` has:
  - fields for the current key/value pair accessed by the cursor
  - methods for advancing/resetting the cursor
  - a method you'll implement for getting a column's value

Unmarshalling a Single Field's Value

- You will write a `TableIterator` method that unmarshalls the value of *a* single column from the current key/value pair.

```
public Object getColumnVal(int colIndex)
```

- First, you'll need to create the necessary `RowInput` objects:

```
RowInput keyIn = new RowInput(this.key.getData());
RowInput valueIn = new RowInput(this.value.getData());
```

- Then you'll use `RowInput` methods to access the necessary offset(s) and value.
- You should *not* unmarshall the entire record – only the portions that are needed to get the value of the specified column.
- Thus, you should mostly use the "at offset" versions of the `RowInput` methods.
  - `readBytesAtOffset`, `readIntAtOffset`, etc.

Examples of Unmarshalling: Assumptions

- We have a simplified version of the Movie table from PS 1:

```
Movie(id CHAR(7), name VARCHAR(64), runtime INT,
      rating VARCHAR(5), earnings_rank INT)
```

- We didn't specify a primary key when we created the table.
  - thus, id is the primary key – and the key in the key/value pair
  - the rest of the row is in the value portion of the key/value pair
- We're using 2-byte offsets.
  - `-2` indicates the primary key
  - `-1` indicates a NULL value
- The cursor/iterator is currently positioned on this key/value pair:

key: `4975722`

	0	2	4	6	8	10	12	21	25
value	-2	12	21	25	-1	26	Moonlight	111	R

Example 1

	0	2	4	6	8	10	12	21	25
value	-2	12	21	25	-1	26	Moonlight	111	R

```
Movie(id CHAR(7), name VARCHAR(64), runtime INT,
      rating VARCHAR(5), earnings_rank INT)
```

- To retrieve the movie's name ($field_1$ – the second field):
  - determine that $offset_1$ is 1*2 = 2 bytes from the start
  - perform a read at an offset of 2 to obtain $offset_1$ → 12
  - because name is a VARCHAR, read $offset_2$ → 21
    and compute this name's length = 21 – 12 = 9
  - read 9 bytes at an offset of 12 bytes → 'Moonlight'

Example 2

	0	2	4	6	8	10	12	21	25
value	-2	12	21	25	-1	26	Moonlight	111	R

```
Movie(id CHAR(7), name VARCHAR(64), runtime INT,
      rating VARCHAR(5), earnings_rank INT)
```

- To retrieve the earnings_rank ($field_4$)
  - determine that $offset_4$ is 4*2 = 8 bytes from the start
  - perform a read at an offset of 8 to obtain $offset_4$ → -1
  - conclude that the value is NULL

Example 3

	0	2	4	6	8	10	12	21	25
value	-2	12	21	25	-1	26	Moonlight	111	R

```
Movie(id CHAR(7), name VARCHAR(64), runtime INT,
      rating VARCHAR(5), earnings_rank INT)
```

- To retrieve the rating ($field_3$):
  - determine that $offset_3$ is 3*2 = 6 bytes from the start
  - perform a read at an offset of 6 to obtain $offset_3$ → 25
  - because rating is a VARCHAR:
    - read $offset_4$ → -1, *so we need to keep going!*
    - read $offset_5$ → 26
    - compute this rating's length = 26 – 25 = 1
  - read 1 byte at an offset of 25 → 'R'

Transactions and Schedules

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Transactions: An Overview

- A *transaction* is a sequence of operations that is treated as a single logical operation. (abbreviation = txn)
- Example: a balance transfer

transaction T1

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

- Transactions are *all-or-nothing*: all of a transaction's changes take effect or none of them do.

Executing a Transaction

1. Issue a command indicating the start of the transaction.

2. Perform the operations in the transaction.
   - in SQL: `SELECT`, `UPDATE`, etc.

3. End the transaction in one of two ways:
   - *commit it:* make all of its results visible and persistent
     - *all* of the changes happen

   - *roll it back / abort it:* undo all of its changes, returning to the state before the transaction began
     - *none* of the changes happen

Why Do We Need Transactions?

- To prevent problems stemming from system failures.
  - example: a balance transfer

```
read balance1
write(balance1 - 500)
CRASH
read balance2
write(balance2 + 500)
```

Why Do We Need Transactions? (cont.)

- To ensure that operations performed by different users don't overlap in problematic ways.
  - example: this should *not* be allowed

user 1

```
read balance1
write(balance1 - 500)



read balance2
write(balance2 + 500)
```

user 2

```
read balance1
read balance2
if (balance1+balance2 < min)
    write(balance1 - fee)
```

ACID Properties

- A transaction has the following "ACID" properties:

  Atomicity: either all of its changes take effect or none do

  Consistency preservation: its operations take the database from one consistent state to another

  - consistent = satisfies the constraints from the schema, and any other expectations about the values in the database

  Isolation: it is not affected by and does not affect other concurrent transactions

  Durability: once it commits, its changes survive failures

- The user plays a role in consistency preservation.
  - ex: add to balance2 the same amnt subtracted from balance1
  - the DBMS helps by rejecting changes that violate constraints
  - guaranteeing the other properties also preserves consistency

Atomicity and Durability

- These properties are guaranteed by the part of the system that performs logging and recovery.
- After a crash, the recovery subsystem:
  - *redoes* as needed all changes by committed txns
  - *undoes* as needed all changes by uncommitted txns
    - restoring the old values of the changed data items
- We'll look more at logging and recovery later in the semester.

Isolation

- To guarantee isolation, the DBMS has to prevent problematic interleavings like the one we saw earlier:

transaction T1

```
read balance1
write(balance1 - 500)



read balance2
write(balance2 + 500)
```

transaction T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

- One possibility: enforce a *serial schedule* (no interleaving).

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

or

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

  - doesn't make sense for performance reasons. why?

Serializability

- A *serializable schedule* is one whose effects are equivalent to the effects of some serial schedule. For example:

schedule 1

transaction T1
```
read X
X = X + 5
write X

read Y
Y = Y + 6
write Y
```

transaction T2
```
read Y
Y = Y + 2
write Y

read X
X = X + 10
write X
```

- X is increased by 15
- Y is increased by 8

schedule 2 (a serial schedule)

transaction T1
```
read X
X = X + 5
write X
read Y
Y = Y + 6
write Y
```

transaction T2
```
read Y
Y = Y + 2
write Y
read X
X = X + 10
write X
```

- X is increased by 15
- Y is increased by 8

- Because the effects schedule 1 are equivalent to the effects of a serial schedule (schedule 2), schedule 1 is *serializable*.

Not All Schedules Are Serializable!

- Schedule 1 is a special case.

schedule 1

transaction T1
```
read X
X = X + 5
write X

read Y
Y = Y + 6
write Y
```

transaction T2
```
read Y
Y = Y + 2
write Y

read X
X = X + 10
write X
```

schedule 2 (a serial schedule)

transaction T1
```
read X
X = X + 5
write X
read Y
Y = Y + 6
write Y
```

transaction T2
```
read Y
Y = Y + 2
write Y
read X
X = X + 10
write X
```

- both T1 and T2 use addition to change the values of X and Y
- addition is commutative
- thus, the order in which T1 and T2 make their changes doesn't matter!

Not All Schedules Are Serializable! (cont.)

- If we change T2 so that it uses multiplication, the original interleaving is no longer serializable.

schedule 1B

transaction T1	transaction T2B
read X X = X + 5 write X	
	read Y Y = Y * 2 write Y
read Y Y = Y + 6 write Y	
	read X X = X * 10 write X

- X → **10(X + 5)**
- Y → **2Y + 6**

schedule 2B

transaction T1	transaction T2B
read X X = X + 5 write X read Y Y = Y + 6 write Y	
	read Y Y = Y * 2 write Y read X X = X * 10 write X

- X → 10(X + 5)
- Y → **2(Y + 6)**

schedule 3

transaction T1	transaction T2B
	read Y Y = Y * 2 write Y read X X = X * 10 write X
read X X = X + 5 write X read Y Y = Y + 6 write Y	

- X → **10X + 5**
- Y → 2Y + 6

- Because the effects schedule 1B are ***not*** equivalent to the effects of *any* serial schedule of T1 + T2B, schedule 1B is ***not*** serializable.

Conventions for Schedules

- We abstract all transactions into sequences of reads and writes.
  - example:

T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

→

T2

```
read(A)
read(B)
write(A)
```

  - we use a different variable for each data item that is read or written
  - we ignore:
    - the actual meaning and values of the data items
    - the nature of the changes that are made to them
    - things like comparisons that a transaction does in its own address space

Conventions for Schedules (cont.)

- We can represent a schedule using a table.
  - one column for each transaction
  - operations are performed in the order given by reading from top to bottom

T_1	T_2
r(A)	
	r(B)
w(A)	
	r(A)
	w(A)

- We can also write a schedule on a single line using this notation:

  $r_i(A)$ = transaction T_i reads A
  $w_i(A)$ = transaction T_i writes A

  - example for the table above:

    $r_1(A)$; $r_2(B)$; $w_1(A)$; $r_2(A)$; $\mathbf{w_2(A)}$

Serializability of Abstract Schedules

- How can we determine if an abstract schedule is serializable?
  - given that we don't know the exact nature of the changes made to the data
- We'll focus on the following:
  - which transaction is the last one to write each data item
    - that's the version that will be seen after the schedule
  - which version of a data item is read by each transaction
    - assume that if a transaction reads a different version, its subsequent behavior might be different

Conflicts in Schedules

- A *conflict* is a pair of actions that can't be swapped without potentially changing the behavior of one or more transactions.
- Examples in the schedule at right:
  - $w_1(A)$ and $r_2(A)$
    - swapping them leads T2 to read a different value of A
    - this may cause T2 to behave differently
  - $w_2(B)$ and $w_1(B)$
    - swapping them means later readers of B will see a different value of B
    - this may cause them to behave differently
- $r_1(B)$ and $r_2(B)$ do *not* conflict. why?

T_1	T_2
r(B)	
	r(B)
w(A)	
	r(A)
	w(A)
	w(B)
w(B)	
…	…

Which Actions Conflict?

- Actions in different transactions conflict if:
  1) they involve the same data item
  and 2) at least one of them is a write
- Pairs of actions that *do* conflict (assume i != j):
  - $w_i(A)$; $r_j(A)$ the value read by T_j may change if we swap them
  - $r_i(A)$; $w_j(A)$ the value read by T_i may change if we swap them
  - $w_i(A)$; $w_j(A)$ subsequent reads may change if we swap them
  - two actions from the same txn (their order is fixed by the client)
- Pairs of actions that *don't* conflict:
  - $r_i(A)$; $r_j(A)$ – two reads of the same item by different txns
  - $r_i(A)$; $r_j(B)$
  - $r_i(A)$; $w_j(B)$
  - $w_i(A)$; $r_j(B)$
  - $w_i(A)$; $w_j(B)$

  operations on two *different* items by different txns

Conflict Serializability

- Rather than ensuring serializability, it's easier to ensure a stricter condition known as *conflict serializability*.
- A schedule is *conflict serializable* if we can turn it into a serial schedule by swapping pairs of *consecutive* actions that *don't* conflict.

Example of a Conflict Serializable Schedule

$r_2(A)$; $r_1(A)$; $r_2(B)$; $w_1(A)$; $w_2(B)$; $r_1(B)$; $w_1(B)$

$r_2(A)$; $r_2(B)$; $r_1(A)$; $w_1(A)$; $w_2(B)$; $r_1(B)$; $w_1(B)$

$r_2(A)$; $r_2(B)$; $r_1(A)$; $w_2(B)$; $w_1(A)$; $r_1(B)$; $w_1(B)$

$r_2(A)$; $r_2(B)$; $w_2(B)$; $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$

T_1	T_2
	r(A)
r(A)	
	r(B)
w(A)	
	w(B)
r(B)	
w(B)	

T_1	T_2
	r(A)
	r(B)
	w(B)
r(A)	
w(A)	
r(B)	
w(B)	

- The final schedule is referred to as an *equivalent serial schedule.*
  - *serial* – all of T2, followed by all of T1
  - *equivalent* – it produces the same results as the original schedule

Testing for Conflict Serializability

- Because conflicting pairs of actions can't be swapped, they impose constraints on the order of the txns in an equivalent serial schedule.
  - example: if a schedule includes $w_1(A)$ … $r_2(A)$, T1 must come before T2 in any equivalent serial schedule
- To test for conflict serializability:
  - determine all such constraints
  - make sure they aren't contradictory
- Example: $r_2(A); r_1(A); r_2(B); w_1(A); w_2(B); r_1(B); w_1(B)$

  $r_2(A)$ … $w_1(A)$ means T2 must come before T1
  $r_2(B)$ … $w_1(B)$ means T2 must come before T1
  $w_2(B)$ … $r_1(B)$ means T2 must come before T1
  $w_2(B)$ … $w_1(B)$ means T2 must come before T1

  no contradictions, so this schedule is equivalent to the serial ordering T2;T1

  Thus, this schedule *is* conflict serializable.

Testing for Conflict Serializability (cont.)

- What about this schedule? $r_1(B); w_1(B); r_2(B); r_2(A); w_2(A); r_1(A)$
- Which of the following pairs of actions from this schedule conflict? (choose all that apply)

A. $r_1(B); r_2(B)$

B. $r_1(B); w_2(A)$

C. $w_1(B); r_2(B)$

D. $r_2(B); r_2(A)$

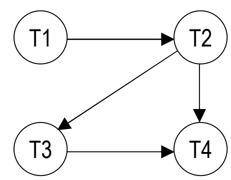
E. $w_2(A); r_1(A)$

Using a Precedence Graph

- Tests for conflict serializability can use a *precedence graph*.
  - the vertices/nodes are the transactions
  - add an edge for each precedence constraint: T1 → T2 means T1 must come before T2 in an equivalent serial schedule

- Example: $r_2(A); r_3(A); r_1(B); w_4(A); w_2(B); r_3(B)$

  $r_2(A) \ldots w_4(A)$ means T2 → T4
  $r_3(A) \ldots w_4(A)$ means T3 → T4
  $r_1(B) \ldots w_2(B)$ means T1 → T2
  $w_2(B) \ldots r_3(B)$ means T2 → T3

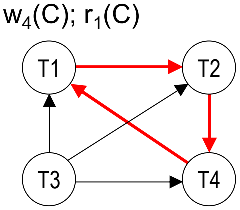


- After the graph is constructed, we test for *cycles* (i.e., paths of the form A → … → A).
  - if the graph is acyclic, the schedule is conflict serializable
    - use the constraints to determine an equivalent serial schedule (in this case: T1;T2;T3;T4)
  - if there's a cycle, the schedule is *not* conflict serializable

More Examples

- Determine if the following are conflict serializable:
  - $r_1(A); r_3(A); r_1(B); w_2(A); r_4(A); w_2(B); w_3(C); w_4(C); r_1(C)$

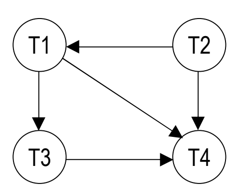
    $r_1(A) \ldots w_2(A)$ means T1 → T2
    $r_3(A) \ldots w_2(A)$ means T3 → T2
    $r_1(B) \ldots w_2(B)$ means T1 → T2
    $w_2(A) \ldots r_4(A)$ means T2 → T4
    $w_3(C) \ldots w_4(C)$ means T3 → T4
    $w_3(C) \ldots r_1(C)$ means T3 → T1
    $w_4(C) \ldots r_1(C)$ means T4 → T1



cycle: T1 → T2 → T4 → T1
***not** conflict serializable*

  - $r_1(A); w_3(A); w_4(A); w_2(B); r_2(B); r_1(B); r_4(B)$

    $r_1(A) \ldots w_3(A)$ means T1 → T3
    $r_1(A) \ldots w_4(A)$ means T1 → T4
    $w_3(A) \ldots w_4(A)$ means T3 → T4
    $w_2(B) \ldots r_1(B)$ means T2 → T1
    $w_2(B) \ldots r_4(B)$ means T2 → T4



no cycles, so conflict serializable.
equivalent to T2; T1; T3; T4

Conflict Serializability vs. Serializability

- Conflict serializability is a *sufficient* condition for serializability, but it's not a *necessary* condition.
  - all conflict serializable schedules are serializable
  - not all serializable schedules are conflict serializable
- Consider the following schedule involving three txns:
- It is *not* conflict serializable, because:
  $r_2(A)$ … $w_1(A)$ means $T_2 \rightarrow T_1$
  $w_1(A)$ … $w_2(A)$ means $T_1 \rightarrow T_2$
- It *is* serializable because its effects are equivalent to either
  $T_1; T_2; T_3$ *or* $T_2; T_1; T_3$ why?

T_1	T_2	T_3
	r(A)	
r(A)		
	r(B)	
w(A)		
		r(B)
	w(A)	
		w(A)

Recoverability

- While serializability is important, it isn't enough for full isolation.
- Consider the serializable schedule at right.
  - includes "c" actions that indicate when the transactions commit
- Imagine that the system crashes:
  - *after* T1's commit
  - *before* T2's commit
- During recovery from the crash, the system:
  - *keeps* all of T1's changes, because it committed before the crash
  - *undoes* all of T2's changes, because it didn't commit before the crash

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
CRASH	
	c

Recoverability (cont.)

- This is problematic!
  - T1 reads T2's write of B
  - it then performs actions that may be based on the new value of B
  - during recovery from the crash, T2 is rolled back
    ➔ B's old value is restored
  - it's possible T1 would have behaved differently if it had read B's old value
  - it's too late to roll back T1, because it has already committed!
- We say that this schedule is *unrecoverable.*
  - if a crash occurs between the two commits, the process of recovering from the crash could lead to problematic results

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
CRASH	
	c

Recoverability (cont.)

- In a *recoverable* schedule, if T1 reads a value written by T2, T1 must commit *after* T2 commits.
- This allows us to safely recover from a crash at any point:

unrecoverable

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
	c

➔

recoverable

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
	c
c	

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
	c
c	
CRASH	

the reader of the changed value survives the crash, but so does the writer

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
CRASH	
	c
c	

the writer of the changed value is rolled back, but so is the reader

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
	c
CRASH	
c	

the reader is rolled back and the writer isn't, but that's okay since the writer didn't base its actions on what the reader did

Dirty Reads and Cascading Rollbacks

- *Dirty data* is data written by an uncommitted txn.
  - it remains dirty until the txn is either:
    - committed: in which case the data is no longer dirty and it is safe for other txns to read it
    - rolled back (either voluntarily or by the DBMS): in which case the write of the dirty data is undone
- A *dirty read* is a read of dirty data.
- Dirty reads can lead to *cascading rollbacks*.
  - if the writer of the dirty data is rolled back, the reader must be, too

Dirty Reads and Cascading Rollbacks (cont.)

- We made our earlier schedule recoverable by switching the order of the commits:

T_1	T_2
	r(A)
	w(B)
r(B)	
w(B)	
c	
	c

T_1	T_2
	r(A)
	w(B)
r(B)	
w(B)	
	c
c	

- Could the revised schedule lead to a cascading rollback?

- To get a *casecadeless* schedule, don't allow dirty reads.

Goals for Schedules

- We want to ensure that schedules of concurrent txns are:
  - *serializable*: equivalent to some serial schedule
  - *recoverable*: ordered so that the system can safely recover from a crash or undo an aborted transaction
  - *cascadeless*: ensure that rolling back one transaction does not produce a series of *cascading rollbacks*
- To achieve these goals, we use some type of *concurrency control mechanism*.
  - *controls* the actions of *concurrent* transactions
  - prevents problematic interleavings

Extra Practice

- Is the schedule at right:
  - conflict serializable?
  - serializable?
  - recoverable?
  - cascadeless?

T_1	T_2
	r(B)
r(B)	
	w(B)
w(A)	
	r(A)
c	
	c

Extra Practice

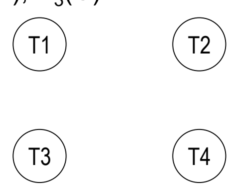
- What scenarios involving the schedule at right could produce cascading rollbacks?

T_1	T_2	T_3
		w(C)
	r(C)	
	w(B)	
r(B)		
w(A)		
		r(A)
…	…	…

Is This Schedule Conflict Serializable?

- Draw the precedence graph to find out!

$w_1(A); r_2(B); r_2(A); r_4(A); w_2(B); r_4(B); w_4(C); r_3(D); w_3(C)$



A. Yes. It is equivalent to the serial schedule T1;T2;T3;T4

B. Yes. It is equivalent to the serial schedule T1;T2;T4;T3

C. No. The graph includes the cycle T1 → T4 → T2 → T1

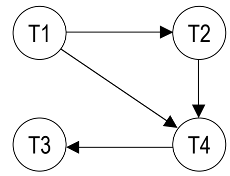
D. No. The graph includes the cycle T1 → T2 → T4 → T1

What If We Add This Write?

- Draw the precedence graph to find out!

$w_1(A)$; $r_2(B)$; $r_2(A)$; $r_4(A)$; $w_2(B)$; $r_4(B)$; $w_4(C)$; $r_3(D)$; $w_3(C)$; **$w_1(D)$**

$w_1(A)$ … $r_3(A)$ means T1 → T3
$w_1(A)$ … $r_2(A)$ means T1 → T2
$w_1(A)$ … $r_4(A)$ means T1 → T4
$w_2(B)$ … $r_4(B)$ means T2 → T4
$w_4(C)$ … $w_3(C)$ means T4 → T3



Concurrency Control

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Goals for Schedules

- We want to ensure that schedules of concurrent txns are:
  - *serializable*: equivalent to some serial schedule
  - *recoverable*: ordered so that the system can safely recover from a crash or undo an aborted transaction
  - *cascadeless*: ensure that an abort of one transaction does not produce a series of *cascading rollbacks*
- To achieve these goals, we use some type of *concurrency control mechanism*.
  - *controls* the actions of *concurrent* transactions
  - prevents problematic interleavings

Locking

- Locking is one way to provide concurrency control.
- Involves associating one or more *locks* with each *database element*.
  - each page
  - each record
  - possibly even each collection

Locking Basics

- A transaction must ***request and acquire a lock*** for a data element before it can access it.
- In our initial scheme, every lock can be held by only one txn at a time.

T_1	T_2
l(X)	
r(X)	
	l(X) ***denied; wait for T1***
w(X)	
u(X)	
	l(X) ***granted***
	r(X)
	u(X)

- As necessary, the DBMS:
  - ***denies*** lock requests for elements that are currently locked
  - makes the requesting transaction wait
- A transaction ***unlocks an element*** when it's done with it.
- After the unlock, the DBMS can grant the lock to a waiting txn.
  - we'll show a ***second lock request*** when the lock is granted

Locking and Serializability

- Just having locks isn't enough to guarantee serializability.
- Example: our problematic schedule can still be carried out.

T1

```
read balance1
write(balance1 - 500)


read balance2
write(balance2 + 500)
```

T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

T_1	T_2
l(bal1);r(bal1) w(bal1); *u(bal1)*	
	l(bal1);r(bal1) *l(bal2)*;r(bal2) w(bal1) *u(bal1);u(bal2)*
l(bal2);r(bal2) w(bal2); *u(bal2)*	

Two-Phase Locking (2PL)

- One way to ensure serializability is *two-phase locking (2PL)*.
- 2PL requires that *all* of a txn's lock actions come before *all* its unlock actions.
- Two phases:
  1. lock-acquisition phase:
     a txn acquires locks, but it doesn't release any
  2. lock-release phase:
     once a txn releases a lock, it can't acquire any new ones
- Reads and writes can occur in both phases.
  - provided that a txn holds the necessary locks
- 2PL is *per-transaction*.
  - one txn could be in its lock-release phase
    while another txn is still in its lock-acquisition phase

Two-Phase Locking (2PL) (cont.)

- In our earlier example, T1 does *not* follow the 2PL rule.

T_1	T_2
l(bal1);r(bal1) w(bal1); ***u(bal1)***	
	l(bal1);r(bal1) *l(bal2)*;r(bal2) w(bal1) *u(bal1);u(bal2)*
l(bal2);r(bal2) w(bal2); *u(bal2)*	

2PL would prevent this interleaving.

- More generally, 2PL produces conflict serializable schedules.

An Informal Argument for 2PL's Correctness

- Consider schedules involving only two transactions. To get one that is ***not*** conflict serializable, we need:
  1) at least one conflict that requires T1 → T2
     - T1 operates first on the data item in this conflict
     - T1 must unlock it before T2 can lock it: $u_1(A)$ .. $l_2(A)$
  2) at least one conflict that requires T2 → T1
     - T2 operates first on the data item in this conflict
     - T2 must unlock it before T1 can lock it: $u_2(B)$ .. $l_1(B)$
- Consider all of the ways these pairs of actions could be ordered:

.. $u_1(A)$ .. $l_2(A)$ .. $u_2(B)$ .. $\boldsymbol{l_1(B)}$ ..
.. $u_2(B)$ .. $l_1(B)$ .. $u_1(A)$ .. $\boldsymbol{l_2(A)}$ ..
.. $u_1(A)$ .. $u_2(B)$ .. $\boldsymbol{l_2(A)}$ .. $\boldsymbol{l_1(B)}$ ..
.. $u_2(B)$ .. $u_1(A)$ .. $\boldsymbol{l_1(B)}$ .. $\boldsymbol{l_2(A)}$ ..
.. $u_1(A)$ .. $u_2(B)$ .. $\boldsymbol{l_1(B)}$ .. $\boldsymbol{l_2(A)}$ ..
.. $u_2(B)$ .. $u_1(A)$ .. $\boldsymbol{l_2(A)}$ .. $\boldsymbol{l_1(B)}$ ..

- none of these are possible under 2PL, because they require at least one txn to lock after unlocking.

The Need for Different Types of Locks

- With only one type of lock, overlapping transactions can't read the same data item, even though two reads don't conflict.
- To get around this, use more than one *mode* of lock.

Exclusive vs. Shared Locks

- An *exclusive lock* allows a transaction to write or read an item.
  - gives the txn exclusive access to that item
  - only one txn can hold it at a given time
  - $xl_i(A)$ = transaction T_i requests an exclusive lock for A
    - if another txn holds *any* lock for A, T_i must wait until that lock is released
- A *shared lock* only allows a transaction to read an item.
  - multiple txns can hold a shared lock for the same data item at the same time
  - $sl_i(A)$ = transaction T_i requests a shared lock for A
    - if another txn holds an *exclusive* lock for A, T_i must wait until that lock is released

Lock Compatibility Matrix

- Used to specify when a lock request for a currently locked item should be granted.

mode of lock requested for item

mode of existing lock for that item (held by a *different* txn)

	shared	**exclusive**
shared	yes	no
exclusive	no	no

Examples of Using Shared and Exclusive Locks

$sl_i(A)$ = transaction T_i requests a shared lock for A
$xl_i(A)$ = transaction T_i requests an exclusive lock for A

- Examples:

T_1	T_2
	xl(A); w(A)
sl(B); r(B)	sl(B);r(B)
xl(C); r(C) w(C) u(B); u(C)	u(A); u(B)

without shared locks,T2 would need to wait until T1 unlocked B

Note: T1 acquires an exclusive lock before reading C. ***Why?***

What About Recoverability / Cascadelessness?

- 2PL alone does *not* guarantee either of them.

- Example:

T_1	T_2
	xl(A); w(A) sl(C) u(A)
xl(A); r(A)	
	r(C); u(C)
w(A); u(A) commit	
	commit

2PL?

not recoverable. why not?

not cascadeless. why not?

Strict Locking

- *Strict locking* makes txns hold all *exclusive* locks until they commit or abort.
  - doing so prevents dirty reads, which means schedules will be recoverable and cascadeless

T_1	T_2
	xl(A); w(A) **sl(C)** ***u(A)***
xl(A); r(A)	
	r(C); **u(C)**
w(A); ***u(A)*** commit	
	commit

→

T_1	T_2
	xl(A); w(A) **sl(C)**
xl(A); r(A)	
	r(C); **u(C)**
w(A) commit ***u(A)***	
	commit ***u(A)***

What else needs to change?

Strict Locking

- *Strict locking* makes txns hold all *exclusive* locks until they commit or abort.
  - doing so prevents dirty reads, which means schedules will be recoverable and cascadeless

T_1	T_2
	xl(A); w(A)
	sl(C)
	u(A)
xl(A); r(A)	
	r(C); **u(C)**
w(A); ***u(A)***	
commit	
	commit

T_1	T_2
	xl(A); w(A)
	sl(C)
xl(A); wait	
	r(C); **u(C)**
	commit
	u(A)
xl(A); r(A)	
w(A)	
commit	
u(A)	

T1 can't acquire the lock for A until after T2 commits.
Thus, its read of A is not dirty!

- strict + 2PL = strict 2PL

Rigorous Locking

- Under strict locking, it's possible to get something like this:

T_1	T_2	T_3
…		
sl(A); r(A)		
u(A)		
	xl(A); w(A)	
…	commit	
	u(A)	
		sl(A); r(A)
		commit
		u(A)
		print A
commit		
print A		

- T3 reports A's new value.
- T1 reports A's old value, even though it commits after T3.
- the ordering of commits (T2,T3,T1) is not same as the equivalent serial ordering (T1,T2,T3)

- *Rigorous locking* requires txns to hold *all* locks until commit/abort.
- It guarantees that transactions commit in the same order as they would in the equivalent serial schedule.
- rigorous + 2PL = rigorous 2PL

Deadlock

- Consider the following schedule:

T_1	T_2
	xl(A);w(A)
sl(B);r(B)	
	xl(B) ***denied; wait for T1***
sl(A) ***denied; wait for T2***	

- This schedule produces *deadlock.*
  - T1 is waiting for T2 to unlock A
  - T2 is waiting for T1 to unlock B
  - neither can make progress!
- We'll see later how to deal with this.

Lock Upgrades

- It can be problematic to acquire an exclusive lock earlier than necessary.
- Instead:
  - acquire a shared lock to read the item
  - *upgrade* to an exclusive lock when you need to write
  - may need to wait to upgrade if others hold shared locks
- Note: we're ***not*** releasing the shared lock before acquiring the exclusive one. why not?

T_1	T_2
xl(A)	
r(A)	
	sl(A) *waits a long time for T1!*
VERY LONG computation	
w(A)	
u(A)	
	r(A) *finally!*

T_1	T_2
sl(A)	
r(A)	
	sl(A)
	r(A) *right away!*
	u(A)
VERY LONG computation	
xl(A)	
w(A)	
u(A)	

A Problem with Lock Upgrades

- Upgrades can lead to deadlock:
  - two txns each hold a shared lock for an item
  - both txns attempt to upgrade their locks
  - each txn is waiting for the other to release its shared lock
  - *deadlock!*
- Example:

T_1	T_2
	sl(A) r(A)
sl(A) r(A) **xl(A)** ***denied; wait for T2***	
	xl(A) ***denied; wait for T1***

Update Locks

- To avoid deadlocks from lock upgrades, some systems provide two different lock modes for reading:
  - shared locks – used if you *only* want to read an item
  - *update locks* – used if you want to read an item *and later update it*

	shared lock	**update lock**
what does holding this type of lock let you do?	read the locked item	read the locked item (in anticipation of updating it later)
can it be upgraded to an exclusive lock?	**no** (not in this locking scheme)	**yes**
how many txns can hold this type of lock for a given item?	an arbitrary number	**only one** *(and thus there can't be a deadlock from two txns trying to upgrade!)*

Different Locks for Different Purposes

- If you only need to read an item, acquire a shared lock.
- If you only need to write an item, acquire an exclusive lock.
- If you need to read and then write an item:
  - acquire an update lock for the read
  - upgrade it to an exclusive lock for the write
  - this sequence of operations is sometimes called *read-modify-write* (RMW)

Compatibility Matrix with Update Locks

mode of existing lock for that item (held by a *different* txn)	mode of lock requested for item: **shared**	**exclusive**	**update**
shared	yes	no	yes
exclusive	no	no	no
update	no	no	no

- When there are one or more shared locks on an item, a txn can still acquire an update lock for that item.
  - allows for concurrency on the read portion of RMW txns
- There can't be more than one update lock on an item.
  - prevents deadlocks when upgrading from update to exclusive
- If a txn holds an update lock on an item, other txns *can't* acquire any *new* locks on that item.
  - prevents the RMW txn from waiting indefinitely to upgrade

Examples of Using Update Locks

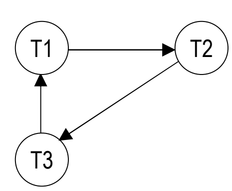
$ul_i(A)$ = T_i requests an update lock for A

T_1	T_2	T_3	
		sl(A) r(A)	
	sl(A) r(A)		
sl(B) r(B) ul(C) r(C)			
	ul(B) r(B)		← **request A?**
		ul(C) r(C)	← **request B**
	xl(A) w(A)		← **request C**
xl(C) w(C) …			← **request D**

Detecting and Handling Deadlocks

- When DBMS detects a deadlock, it roll backs one of the deadlocked transactions.
- Can use a *waits-for graph* to detect the deadlock.
  - the vertices are the transactions
  - an edge from T1 → T2 means
T1 is waiting for T2 to release a lock
  - a cycle indicates a deadlock
- Example:

T_1	T_2	T_3
		xl(C)
xl(A)		
		sl(A) ***denied; wait for T1***
	sl(B) sl(C) ***denied; wait for T3***	
xl(B) ***denied; wait for T2***		



cycle – deadlock!

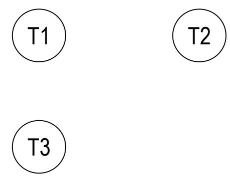
Another Example

- Would the following schedule produce deadlock?

  $r_1(B); w_1(B); r_3(A); r_2(C); r_2(B); r_1(A); w_1(A); w_3(C); w_2(A); r_1(C); w_3(A)$

  - assume: no update locks;
    a lock for an item is acquired just before it is first needed

T_1	T_2	T_3
sl(B); r(B)		
xl(B); w(B)		
		sl(A); r(A)
	sl(C); r(C)	



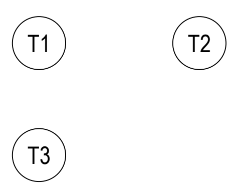
Extra Practice *(try this later on your own!)*

- Would the following schedule produce deadlock?

  $w_1(A); w_3(B); r_3(C); r_2(D); r_1(D); w_1(D); w_2(C); r_3(A); w_2(A)$

  - assume: no update locks;
    a lock for an item is acquired just before it is first needed

T_1	T_2	T_3



Optimistic Concurrency Control

- Locking is *pessimistic.*
  - assumes serializability will be violated
  - prevents transactions from performing actions that *might* violate serializability
    - example:

T_1	T_2
	xl(A); w(A)
sl(B); r(B)	
	xl(B)
…	

denied, because T1 *might* read B again

- There are other approaches that are *optimistic*.
  - assume serializability will be maintained
  - only interfere with a transaction if it actually does something that violates serializability
- We'll look at one such approach – one that uses timestamps.

Timestamp-Based Concurrency Control

- In this approach, the DBMS assigns timestamps to txns.
  - TS(T) = the timestamp of transaction T
  - the timestamps must be unique
  - TS(T1) < TS(T2) if and only if T1 started *before* T2
- The system ensures that all operations are consistent with a *serial* ordering based on the timestamps.
  - if TS(T1) < TS(T2), the DBMS only allows actions that are consistent with the serial schedule T1; T2

Timestamp-Based Concurrency Control (cont.)

- Examples of actions that are not allowed:
  - example 1:

actual schedule

T_1	T_2
	TS = 100 r(C)
TS = 102 w(A)	
	r(A)

not allowed

- T2 starts before T1
- thus, T2 comes before T1 in the equivalent serial schedule (see left)
- in the serial schedule, T2 would *not* see see T1's write
- thus, T2's read should have come *before* T1's write, and we can't allow the read
- we say that ***T2's read is too late***

equivalent serial schedule

T_1	T_2
	TS = 100 r(C) **r(A)** ...
TS = 102 **w(A)** ...	

Timestamp-Based Concurrency Control (cont.)

- Examples of actions that are not allowed:
  - example 2:

actual schedule

T_1	T_2
TS = 205 r(A)	
	TS = 209 r(B)
w(B)	

not allowed

- T1 starts before T2
- thus, T1 comes before T2 in the equivalent serial schedule (see left)
- in the serial schedule, T2 *would* see T1's write
- thus, T1's write should have come *before* T2's read, and we can't allow the write
- we say that ***T1's write is too late***

equivalent serial schedule

T_1	T_2
TS = 205 r(A) **w(B)** ...	
	TS = 209 **r(B)** ...

Timestamp-Based Concurrency Control (cont.)

- When a txn attempts to perform an action that is inconsistent with a timestamp ordering:
  - the offending txn is rolled back
  - it is restarted with a new, larger timestamp
- With a larger timestamp, the txn comes later in the equivalent serial ordering.
  - allows it to perform the offending operation
- Rolling back the txn ensures that all of its actions correspond to the new timestamp.

Timestamps on Data Elements

- To determine if an action should be allowed, the DBMS associates two timestamps with each data element:
  - a *read timestamp*:
    RTS(A) = the largest timestamp of any txn that has read A
    - the timestamp of the reader that comes latest in the equivalent serial ordering
  - a *write timestamp*:
    WTS(A) = the largest timestamp of any txn that has written A
    - the timestamp of the writer that comes latest in the equivalent serial ordering
    - *the timestamp of the txn that wrote A's current value*

Timestamp Rules for Reads

- When T tries to read A:
  - if TS(T) < WTS(A), roll back T and restart it
    - T comes *before* the txn that wrote A, so T shouldn't be able to see A's current value
    - T's read is too late (see our earlier example 1)
  - else allow the read
    - T comes *after* the txn that wrote A, so the read is OK
    - the system also updates RTS(A):
      RTS(A) = max(TS(T), RTS(A))
      - why can't we just set RTS(A) to T's timestamp?

Timestamp Rules for Reads (cont.)

- Example: assume that T1 wants to read A, and we have the following timestamps:

TS(T1) = 30	WTS(A) = 10
TS(T2) = 50	RTS(A) = 50

- T1 started before T2 (30 < 50)
  - thus T1 comes before T2 in the equivalent serial ordering
- T2 has already read A. How do we know? RTS(A) = TS(T2)
- Despite that, it's okay for T1 to read A.
  - reads don't conflict, so we don't care about the equivalent serial ordering of *two readers* of an item
  - what matters is that T1 comes after the *writer* of A's current value (30 > 10)

Timestamp Rules for Writes

- When T tries to write A:
  - if TS(T) < RTS(A), roll back T and restart it
    - T comes *before* the txn that read A, so that other txn should have read the value T wants to write
    - T's write is too late (see our earlier example 2)
  - else if TS(T) < WTS(A), ignore the write and let T continue
    - T comes *before* the txn that wrote A's current value
    - thus, in the equivalent serial schedule, T's write would have been overwritten by A's current value
  - else allow the write
    - how should the system update WTS(A)?

Thomas Write Rule

- The policy of ignoring out-of-date writes is known as the *Thomas Write Rule:*

  …else if TS(T) < WTS(A), ignore the write and let T continue

- What if there is a txn that should have read A *between* the two writes? It's still okay to ignore T's write of A.
  - example:
    - TS(T) = 80, WTS(A) = 100 ➔ we ignore T's write of A what if txn U with TS(U) = 90 is supposed to read A?
    - if U had already read A, Thomas write rule wouldn't apply:
      - RTS(A) = 90
      - T would be rolled back because TS(T) < RTS(A)
    - if U tries to read A after we ignore T's write:
      - U will be rolled back because TS(U) < WTS(A)

Example of Using Timestamps

- They prevent our problematic balance-transfer example.

T1

```
read balance1
write(balance1 – 500)


read balance2
write(balance2 + 500)
```

T2

```
read balance1
read balance2
if (balance1+balance2 < min)
    write(balance1 – fee)
```

T1	T2	bal1	bal2
		RTS = WTS = 0	RTS = WTS = 0
TS = 350			
r(bal1)		RTS = 350	
w(bal1)		WTS = 350	
	TS = 375		
	r(bal1); r(bal2)	RTS = 375	RTS = 375
	w(bal1)	WTS = 375	
r(bal2)			RTS: no change
w(bal2)			
denied:rollback			

what's the problem here?

Preventing Dirty Reads Using a Commit Bit

- We associate a ***commit bit* c(A)** with each data element A.
  - tells us whether the writer of A's value has committed
  - initially, c(A) is true
- When a txn is allowed to write A:
  - set c(A) to false
  - update WTS(A) as before
- If the timestamps would allow a txn to read A but c(A) is false, the txn is made to wait.
  - preventing a dirty read!
- When A's writer commits, we:
  - set c(A) to true
  - allow waiting txns try again

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
TS = 200		
r(A)		RTS = 200
w(A)		**c = false** WTS = 200
	TS = 210 r(A) ***denied: wait***	
commit		**c = true**
	r(A)?	

Preventing Dirty Reads Using a Commit Bit (cont.)

- If a txn is allowed to write A and c(A) is already false:
  - c(A) remains false
  - update WTS(A) as before
- If the timestamps would cause a txn's write of A to *be ignored* but c(A) is false, the txn must wait.
  - we'll need its write if the writer of A's current value is rolled back

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
	TS = 400 w(A)	**c = false** WTS = 400
TS = 450 w(A)		**c stays false** WTS = 450
	w(A) ***denied: wait***	
commit		**c = true**
	w(A) ***ignored*** …	

Preventing Dirty Reads Using a Commit Bit (cont.)

- Note: c(A) remains false until the writer of the *current value* commits.
- Example: what if T2 had committed after T1's write?

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
	TS = 400 w(A)	**c = false** WTS = 400
TS = 450 w(A)		**c stays false** WTS = 450
	commit	

Preventing Dirty Reads Using a Commit Bit (cont.)

- What happens when a txn T is rolled back?
  - restore the prior state (value and timestamps) of all data elements of which T is the most recent writer
  - set the commit bits of those elements based on whether the writer of the prior value has committed
  - make waiting txns try again
  - in addition, if there were a data element B for which RTS(B) == TS(T), we would restore its old RTS value

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
	TS = 400 w(A)	**c = false** **WTS = 400**
TS = 450 w(A)		**c stays false** WTS = 450
	w(A) ***denied: wait***	
roll back		**WTS = 400** **c = false**
	w(A) ***allowed!***	no changes

Example of Using Timestamps and Commit Bits

- The balance-transfer example would now proceed differently.

T1
```
read balance1
write(balance1 - 500)


read balance2
write(balance2 + 500)
```

T2
```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

T1	T2	bal1	bal2
		RTS = WTS = 0 c = true	RTS = WTS = 0 c = true
TS = 350 r(bal1) w(bal1)		RTS = 350 WTS = 350; c = false	
	TS = 375 r(bal1) ***denied: wait***		
r(bal2) w(bal2) commit		c = true	RTS = 350 WTS = 350; c = false c = true
	r(bal1) ***and completes***	RTS = 375	

Multiversion Timestamp Protocol

- To reduce the number of rollbacks, the DBMS can keep old versions of data elements, along with the associated timestamps.
- When a txn T tries to read A, it's given the version of A that it should read, based on the timestamps.
  - *the DBMS never needs to roll back a read-only transaction!*

two different versions of A

T1	T2	T3	A(0)	A(105)
	TS = 101		RTS = WTS = 0	
TS = 105			c = true; val = "foo"	
r(A)			RTS = 105	
w(A)				*created*
				RTS = 0; WTS = 105
				c = false; val = "bar"
	r(A): *get A(0)*		no change	
commit				c = true
		TS = 112		
		r(A)		
		get A(105)		RTS = 112

Multiversion Timestamp Protocol (cont.)

- Because each write creates a new version, the WTS of a given version never changes.
- The DBMS maintains RTSs and commit bits for each version, and it updates them using the same rules as before.
- If txn T attempts to write A:
  - find the version of A that T should be overwriting (the one with the largest WTS < TS(T))
  - compare TS(T) with the RTS of that version
  - example: txn T (TS = 50) wants to write A
    - it should be overwriting A(0)
    - show we allow its write and create A(50)?

A(0)	A(105)
RTS = 75	RTS = 0

Multiversion Timestamp Protocol (cont.)

- If T's write of A is *not* too late:
  - create a new version of A with WTS = TS(T)
- Writes are never ignored.
  - there may be active txns that should read that version
- Versions can be discarded as soon as there are no active transactions that could read them.
  - can discard A(t1) if:
    - there is another, later version, A(t2), with t2 > t1
      and
    - there is no active transaction with a TS < t2
  - example: we can discard A(0) as soon as …?

A(0)	**A(105)**
RTS = 75	RTS = 0

Locking vs. Timestamps

- Advantages of timestamps:
  - txns spend less time waiting
  - no deadlocks
- Disadvantages of timestamps:
  - can get more rollbacks, which are expensive
  - may use somewhat more space to keep track of timestamps
- Advantages of locks:
  - only deadlocked txns are rolled back
- Disadvantages of locks:
  - unnecessary waits may occur

The Best of Both Worlds

- Combine 2PL and multiversion timestamping!
- Transactions that perform writes use 2PL.
  - their actions are governed by locks, not timestamps
  - thus, only deadlocked txns are rolled back
- Multiple versions of data elements are maintained.
  - each write creates a new version
  - the WTS of a version is based on when the writer *commits,* not when it started
- Read-only transactions do *not* use 2PL.
  - they are assigned timestamps when they start
  - when T reads A, it gets the version from right before T started
    - will only get a version whose writer has committed
  - read-only txns never need to wait or be rolled back!

Summary: Timestamp Rules for Reads and Writes

when not using commit bits

- When T tries to read A:
  - if TS(T) < WTS(A), roll back T and restart it
    - T's read is too late
  - else allow the read
    - set RTS(A) = max(TS(T), RTS(A))
- When T tries to write A:
  - if TS(T) < RTS(A), roll back T and restart it
    - T's write is too late
  - else if TS(T) < WTS(A), ignore the write and let T continue
    - in the equiv serial sched, T's write would be overwritten
  - else allow the write
    - set WTS(A) = TS(T)

Summary: Timestamp Rules for Reads and Writes

when using commit bits

- When T tries to read A:
  - if TS(T) < WTS(A), roll back T and restart it
    - T's read is too late
  - else allow the read **(but if c(A) == false, make it wait)**
    - set RTS(A) = max(TS(T), RTS(A))
- When T tries to write A:
  - if TS(T) < RTS(A), roll back T and restart it
    - T's write is too late
  - else if TS(T) < WTS(A), ignore the write and let T continue **(but if c(A) == false, make it wait)**
    - in the equiv serial sched, T's write would be overwritten
  - else allow the write
    - set WTS(A) = TS(T) **(and set c(A) to false)**

Summary: Other Details for Commit Bits

- When the writer of *the current value* of data item A commits, we:
  - set c(A) to true
  - allow waiting txns try again
- When a txn T is rolled back, we process:
  - all data elements A for which WTS(A) == TS(T)
    - restore their prior state (value and timestamps)
    - set their commit bits based on whether the writer of the prior value has committed
    - make waiting txns try again
  - all data elements A for which RTS(A) == TS(T)
    - restore their prior RTS

Extra Practice Problem 1

- How will this schedule be executed?

  $w_1(A); w_2(A); r_3(B); w_3(B); r_3(A); r_2(B); w_1(B); r_2(A)$

T1	T2	T3	A	B
			RTS = WTS = 0 c = true	RTS = WTS = 0 c = true

Extra Practice Problem 2

- How will this schedule be executed?

  $r_1(B); r_2(B); w_1(B); w_3(A); w_2(A); w_3(B); commit_3; r_2(A)$

T1	T2	T3	A	B
			RTS = WTS = 0 c = true	RTS = WTS = 0 c = true

Semistructured Data and XML

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Structured Data

- We've covered two logical data models thus far:
  - ER diagrams
  - relational schemas
- Both use a *schema* to define the structure of the data.
- The schema in these models is:
  - *separate* from the data itself
  - *rigid*: all data items of a particular type must have the same set of fields/attributes

Semistructured Data

- In semistructured data:
  - there may or may not be a *separate* schema
  - the schema is *not* rigid
    - example: capturing people's addresses
      - some records may have 4 separate fields:
        - `street`, `city`, `state`, `zip`
      - other records may use a single `address` field
- Semistructured data is *self-documenting*.
  - information describing the data is embedded with the data

```
<course>
    <name>CS 460</name>
    <begin>1:25</begin>
    …
</course>
```

Semistructured Data (cont.)

- Its features facilitate:
  - the integration of information from different sources
  - the exchange of information between applications
- Example: company A receives data from company B
  - A only cares about certain fields in certain types of records
  - B's data includes:
    - other types of records
    - other fields within the records that company A cares about
  - with semistructured data, A can easily recognize and ignore unexpected elements
  - the exchange is more complicated with structured data

XML (Extensible Markup Language)

- One way of representing semistructured data.
- Like HTML, XML is a *markup language*.
  - it annotates ("marks up") documents with tags
  - tags generally come in pairs:
    - begin tag: <*tagname*>
    - end tag: </*tagname*>
  - example:

```
<li>Like HTML, XML is a markup language.</li>
```

HTML begin tag for a list item *HTML end tag for a list item*

- Unlike HTML, XML is *extensible*.
  - the set of possible tags – and their meaning – is not fixed

XML Elements

- An XML *element* is:
  - a begin tag
  - an end tag (in some cases, this is merged into the begin tag)
  - all info. between them.
  - example:

```
<name>CS 460</name>
```

- An element can include other nested *child elements.*

```
<course>
    <name>CS 460</name>
    <begin>1:25</begin>
    …
</course>
```

- Related XML elements are grouped together into *documents*.
  - may or may not be stored as an actual text document

XML Attributes

- An element may also include *attributes* that describe it.
- Specified within the element's begin tag.
  - syntax: *name*="*value*"
- Example:

```
<course catalog_number="12345" exam_group="16">
   <name>CS 460</name>
   <begin>1:25</begin>
   …
</course>
```

Attributes vs. Child Elements

	attribute	***child element***
number of occurrences	at most once in a given element	an arbitrary number of times
value	always a string	can have its own children

- The string values used for attributes can serve special purposes (more on this later)

Well-Formed XML

- In a *well-formed* XML document:
  - there is a single root element that contains all other elements
    - may optionally be preceded by an XML declaration (more on this in a moment)
  - each child element is completely nested within its parent
    - this would *not* be allowed:

```
<course><name>CS 460</name>
   <time>
      <begin>1:25</begin>
      <end>2:15</end>
</course>
</time>
```

- The elements need not correspond to any predefined standard.
  - a separate schema is not required

Example of an XML Document

```
<?xml version="1.0" standalone="yes"?>      ← optional declaration
<university-data>  ← single root element
   <course>
       <name>CS 111</name>
       <start>10:10</start>
       <end>11:00</end>
   </course>
   <room>
       <bldg>CAS</bldg>
       <num>B12</num>
   </room>
   <course>
       <name>CS 460</name>
       <time>
           <begin>1:25</begin>
           <end>2:15</end>
       </time>
   </course>
   ...
</university-data>
```

Specifying a Separate Schema

- XML doesn't require a separate schema.
- However, we still need one if we want programs to:
  - easily process XML documents
  - validate the contents of a given document
- The resulting schema can still be semistructured.
  - for example, can include optional components
  - more flexible than ER models and relational schema

Special Types of Attributes

- `ID` an identifier that must be unique within the document (among *all* `ID` attributes – not just this attribute)
- `IDREF` a single value that is the value of an `ID` attribute elsewhere in the document
- `IDREFS` a *list* of `ID` values from elsewhere in the document

Capturing Relationships in XML

- Two options:

1. store references from one element to other elements using `ID`, `IDREF` and `IDREFS` attributes:

```
<course cid="C20119" teacher="P123456">
  <cname>CS 111</cname>
  …
</course>

<course cid="C20268" teacher="P123456">
  <cname>CS 460</cname>
  …
</course>

<person pid="P123456" teaches="C20119 C20268">
  <pname>
     <last>Sullivan</last>
     <first>David</first>
  </pname>
</person>
```

   - where have we seen something similar?

Capturing Relationships in XML (cont.)

2. use child elements:

```
<course cid="C20119">
  <cname>CS 111</cname>
  <teacher id="P123456">David Sullivan</teacher>
</course>

 …
<person pid="P123456">
  <pname>
     <last>Sullivan</last>
     <first>David</first>
  </pname>
  <courses-taught>
     <course-taught>CS 111</course-taught>
     <course-taught>CS 460</course-taught>
  </courses-taught>
</person>
```

- There are pluses and minuses to each approach.
  - we'll revisit this design issue later in the course

Summary: Features of an XML Document

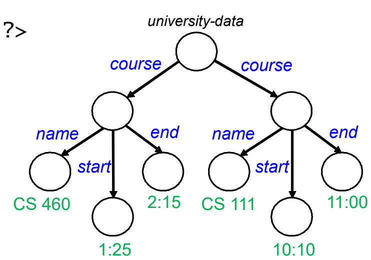
```
<?xml version="1.0" standalone="yes"?>     <- optional declaration

<university-data>    <- single root element
  <course cid="C20268" teacher="P123456">
      <name>CS 460</name>
      <start>1:25</start>
      <end>2:15</end>
  </course>
  <course cid="C20119" teacher="P123456" room="CAS 522">
      <name>CS 111</name><start>10:10</start><end>11:00</end>
  </course>
  <person pid="P123456"
          teaches="C20119 C20268">
    <name>
        <last>Sullivan</last>
        <first>David</first>
    </name>
  </person>
  <holiday date="04/15/2019" />
    ...
</university-data>
```

- *Elements* can have other *child elements* nested inside them.
- *Attributes* are found in the start tag of an element.
- *Simple elements* have no children or attributes.
- *Empty elements* only have a start tag (and possibly attributes)
    - use a / at end of start tag

XML Documents as Trees

```
<?xml version="1.0" standalone="yes"?>
<university-data>
    <course><name>CS 460</name>
      <start>1:25</start>
      <end>2:15</end>
    </course>
      ...
    <course><name>CS 111</name>
      <start>10:10</start>
      <end>11:00</end>
    </course>
      ...
</university-data>
```



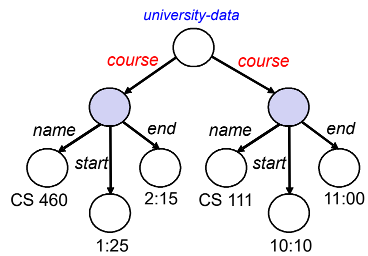
- Elements correspond to nodes in the tree.
    - root element == root node of the entire tree
    - child element == child of a node
    - leaf nodes == empty elements or ones without child elements
- Start tags are edge labels.
- Attributes and text values are data stored in the node.

XPath Expressions

- Used to specify one or more elements or attributes by providing a path to the relevant nodes in the document tree.
  - like a pathname in a hierarchical filesystem
- Expressions that begin with `/` specify a path that begins at the root of the document.

  `/university-data/course`

  - selects all `course` elements that are children of the `university-data` root element



XPath Expressions

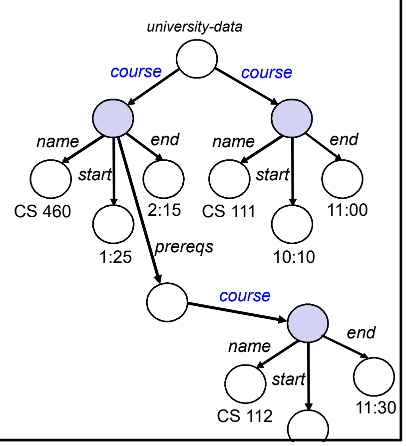
- Used to specify one or more elements or attributes by providing a path to the relevant nodes in the document tree.
  - like a pathname in a hierarchical filesystem
- Expressions that begin with `/` specify a path that begins at the root of the document.

  `/university-data/course`

  - selects all `course` elements that are children of the `university-data` root element
- Expressions that begin with `//` select elements from *anywhere* in the document.

  `//course`

  - selects *all* `course` elements, regardless of where they appear



XPath Expressions (cont.)

- Attribute names are preceded by an `@` symbol:
  - example: `//person/@pid`
    - selects all `pid` attributes of all `person` elements
- We can specify a particular document as follows:

  `document("`*`doc-name`*`")`*`path-expression`*

  - example:

  `document("university.xml")//course/start`

Predicates in XPath Expressions

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

- Example:

  `//course[@teacher="P123456"]`

  - selects all `course` elements with a `teacher` **attribute** of "P123456"
- In general, predicates are:
  - surrounded by square brackets
  - applied to elements selected by the preceding path expression

Predicates in XPath Expressions (cont.)

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

`//course[name="CS 460"]`

- selects all `course` elements with a `name` child element whose value is "CS 460"

➔
```
<course cid="C20268" teacher="P123456">
  <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
```

`//course[start="1:25"]/name`

Predicates in XPath Expressions (cont.)

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

`//course[name="CS 112"]/@room`

Predicates in XPath Expressions (cont.)

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

- We can test for the presence of an element or attribute:
  - example: `//course[@room]`
    - selects all `course` elements that have a specified `room` attribute
- We can use the `contains()` function for substring matching:
  - example: `//course[contains(name, "CS")]`

Predicates in XPath Expressions (cont.)

```
<room>
   <building>CAS</building><room_num>212</room_num>
</room>
<room>
   <building>CAS</building><room_num>100</room_num>
</room>
<room>
   <building>KCB</building><room_num>101</room_num>
</room>
<room>
   <building>PSY</building><room_num>228D</room_num>
</room>
```

- Use `.` to represent nodes selected by the preceding path.

  `//room/room_num[. < 200]`
  - selects all `room_num` elements with values < 200

  `//room[room_num < 200]`
  - selects all `room` elements with `room_num` child values < 200

Predicates in XPath Expressions (cont.)

```
<room>
   <building>CAS</building><room_num>212</room_num>
</room>
<room>
   <building>CAS</building><room_num>100</room_num>
</room>
<room>
   <building>KCB</building><room_num>101</room_num>
</room>
<room>
   <building>PSY</building><room_num>228D</room_num>
</room>
```

- Use `..` to represent the *parents* of the nodes selected by the preceding path.

  `//room_num[../building="CAS"]` ➔ `<room_num>212</room_num>` `<room_num>100</room_num>`

  - selects all `room_num` elements for parent elements that also have a `building` child whose value is "CAS"
  - this is similar: `//room[building="CAS"]/room_num`

Predicates in XPath Expressions (cont.)

```
<room>
   <building>CAS</building><room_num>212</room_num>
</room>
<office>
   <building>CAS</building><room_num>100</room_num>
</office>
<room>
   <building>KCB</building><room_num>101</room_num>
</room>
<office>
   <building>PSY</building><room_num>228D</room_num>
</office>
```

- If there are *other* elements that also have nested `room_num` and `building` elements (like `office` elements above)
  - `//room_num[../building="CAS"]` will get `room_num` children from *all* such elements with a `building` child = "CAS"
  - `//room[building="CAS"]/room_num` will only get `room_num` children from *room* elements with a `building` child = "CAS"

What would this expression select?

```
<course cid="C20119" teacher="P123456" room="CAS 522">
  <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
  <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
  <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

```
//end[../@teacher="P778787"]
```

A.
```
<course cid="C20757" teacher="P778787" room="COM 101">
  <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

B. `<course teacher="P778787"><end>12:45</end></course>`

C. `<end>12:45</end>`

D. none of these

Which of these would select the highlighted element?

```
<course id="C20119" teacher="P123456" room="011">
  <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course id="C20268" teacher="P123456">
  <name>CS 460</name><start>13:25</start><end>14:15</end>
</course>
<course id="C20757" teacher="P778787" room="789">
  <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

A. `//course[start = "10:10"]`

B. `//course/start[. = "10:10"]`

C. `/course/start[. = "10:10"]`

D. `/course[start = "10:10"]`

E. `//start[../end = "11:00"]`

XQuery and FLWOR Expressions

- XQuery is to XML documents what SQL is to relational tables.
- XPath is a subset of XQuery.
  - every XPath expression is a valid XQuery query
- In addition, XQuery provides FLWOR expressions.
  - similar to SQL SELECT commands
  - syntax:

```
for $fvar1 in xpath_f1,
    $fvar2 in xpath_f2,…
let $lvar1 := xpath_l1, …
where condition
order by xpath_o1, …
return result-format
```

FLWOR Expressions

```
for $r in //room[contains(name, "CAS")],
    $c in //course
let $e := //person[contains(@enrolled, $c/@id)]
where $c/@room = $r/@id and count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

- The `for` clause is like the `FROM` clause in SQL.
  - the query iterates over all combinations of values from its XPath expressions (like Cartesian product!)
  - query above looks at combos of CAS `rooms` and `courses`
- The `let` clause is applied to each combo. from the `for` clause.
  - each variable gets the *full set* produced by its XPath expr.
  - unlike a `for` clause, which assigns the results of the XPath expression one value at a time

FLWOR Expressions (cont.)

```
for $r in //room[contains(name, "CAS")],
    $c in //course
let $e := //person[contains(@enrolled, $c/@id)]
where $c/@room = $r/@id and count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

- The `where` clause is applied to the results of `for` and `let`.
- If the `where` clause is true, the `return` clause is applied.
- The `order by` clause can be used to sort the results.

Note: The Location of Predicates

```
for $r in //room[contains(name, "CAS")],
    $c in //course
let $e := //person[contains(@enrolled, $c/@id)]
where $c/@room = $r/@id and count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

- It's sometimes possible to move components of the `where` clause up into the `for` clause as predicates.
- In the above query, we could move the first condition up:

```
for $r in //room[contains(name, "CAS")],
    $c in //course[@room = $r/@id]
let $e := //person[contains(@enrolled, $c/@id)]
where count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

return Clause

```
     <course cid="C20119" teacher="P123456" room="CAS 522">
       <name>CS 111</name><start>10:10</start><end>11:00</end>
     </course>
     <course cid="C20268" teacher="P123456">
       <name>CS 460</name><start>13:25</start><end>14:15</end>
     </course>
$c = <course cid="C20757" teacher="P778787" room="COM 101">
       <name>CS 112</name><start>11:30</start><end>12:45</end>
     </course>
```

- Like the SELECT clause in SQL.
- Can be used to perform something like a projection.

```
for $c in //course
where $c/start > "11:00"
return $c/name

➔ <name>CS 460</name>
  <name>CS 112</name>
```

return Clause (cont.)

- Another example:

```
for $c in //course
where $c/start > "11:00"
return ($c/name, $c/start, " ")
```

- To return multiple elements/attributes for each item:
  - separate them using a comma
  - surround them with parentheses, because the comma operator has higher precedence and would end the FLWOR
  - you can also include string literals
    - above, we specify a blank line after the start time
    - full elements already appear on separate lines, so we don't need spaces for that

Reshaping the Output

- We can reshape the output by constructing new elements:

```
for $c in //course
where $c/start > "11:00"
return <after11-course>
       {$c/name/text(), " - ", $c/start/text()}
       </after11-course>
```

  - the `text()` function gives just the value of a simple element
    - without its start and end tags
  - when constructing a new element, need curly braces around expressions that should be evaluated
    - otherwise, they'll be treated as literal text that is the value of the new element
  - here again, use commas to separate items
    - because we're using `text()`, there are no newlines after the name and start time
    - we use a string literal to put something between them

Reshaping the Output (cont.)

```
<course id="C20119" teacher="P123456" room="011">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course id="C20268" teacher="P123456">
   <name>CS 460</name><start>13:25</start><end>14:15</end>
</course>
<course id="C20757" teacher="P778787" room="789">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

```
for $c in //course
where $c/start > "11:00"
return <after11-course>
       {$c/name/text(), " - ", $c/start/text()}
       </after11-course>
```

- The result will look something like this:

```
<after11-course>CS 460 - 13:25</after11-course>
<after11-course>CS 112 - 11:30</after11-course>
```

for vs. let

- Here's an example that illustrates how they differ:

```
for $d in document("depts.xml")/depts/dept/deptno
let $e := document("emps.xml")/emps/emp[deptno = $d]
where count($e) >= 10
return <big-dept>
       {
           $d,
           <headcount>{count($e)}</headcount>,
           <avgsal>{avg($e/salary)}</avgsal>
       }
       </big-dept>
```

  - the `for` clause assigns to `$d` one `deptno` element at a time
  - for each value of `$d`, the `let` clause assigns to `$e` the *full set* of `emp` elements from that department
  - the `where` clause limits us to depts with >= 10 employees
  - we create a new element for each such dept.
  - we use functions on the set `$e` and on values derived from it

Nested Queries

- We can nest FLWOR expressions:
  - example: group together each instructor's person info. with the courses taught by him/her

```
for $p in //person[@teaches]
return <instructor-courses>
    { $p,
      for $c in //course
      where contains($p/@teaches, $c/@id)
      return $c
    }
    </instructor-courses>
```

  - result:

```
<instructor-courses>
   <person id="P123456" teaches="C20119 C20268">
      <name><last>Sullivan</last>…</name>
   </person>
   <course id="C20119" teacher="P123456">
      <name>CS 111</name> …
   </course>
   …
</instructor-courses>
...
```

Reformatting the Results of the Previous Query

```
for $p in //person[@teaches]
return
  <instructor>
  {<name>{$p/pname/first/text(), " ", $p/pname/last/text()}
   </name>,
   for $c in //course
   where contains($p/@teaches, $c/@id)
   return <course>{$c/name/text()}</course>
  }
  </instructor>
```

- result:

```
<instructor>
   <name>David Sullivan</name>
   <course>CS 111</course>
   <course>CS 460</course>
   …
</instructor>
…
```

Implementing an XML DBMS

- Two possible approaches:
  1) build it on top of a DBMS that uses another model
     - use a logical-to-***logical*** mapping that can accommodate any XML document
     - example: define an XML-to-relational mapping

       XML document → one or more tuples

  2) build it directly on top of a storage engine (or file system!)
     - use an appropriate logical-to-***physical*** mapping
     - similar to what you did in PS 2, Part II!

Approach 1: Logical-to-*Logical* Mappings

- Possible XML-to-relational mappings:
  1) use a schema that stores an entire XML document as the value of a single attribute:
     document(id, contents)
     - useful if you need to preserve the exact bytes of the original document (ex: for legal purposes)
     - may also be useful if you have small documents that are typically retrieved in their entirety
  2) use a schema that encodes the tree structure of the document
     - example: a table for elements that looks something like
       element(id, parent_id, name, value)

Approach 2: Logical-to-*Physical* Mappings

- Option 1: Store each document in a flat file.
  - advantages:
    - the mapping is very simple!
    - there are many tools that allow you to manipulate XML that is stored in this way
    - it makes the data easily readable
  - disadvantages?
    -
    -
- Option 2: make direct use of a traditional storage engine
  - get the benefits of a DBMS (indexing, transactions, etc.) without the overhead of a logical-to-logical mapping
  - the logical-to-physical mapping is less straightforward

Distributed Databases and Replication

Harvard Extension School

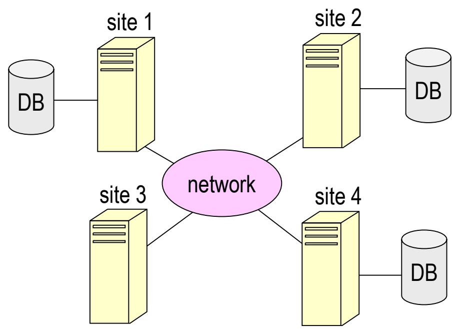
Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

What Is a Distributed Database?

- One in which data is:
  - *partitioned / fragmented* among multiple machines
    and/or
  - *replicated* – copies of the same data are made available on multiple machines
- It is managed by a *distributed DBMS (DDBMS)* – processes on two or more machines that jointly provide access to a single logical database.
- The machines in question may be:
  - at different locations (e.g., different branches of a bank)
  - at the same location (e.g., a cluster of machines)
- In the remaining slides, we will use the term *site* to mean one of the machines involved in a DDBMS.
  - may or may not be at the same location

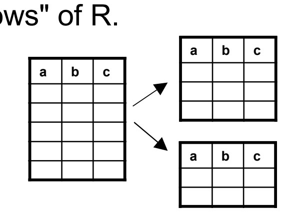
What Is a Distributed Database? (cont.)



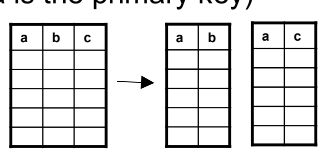
- A given site may have a local copy of all, part, or none of a particular database.
  - makes requests of other sites as needed

Fragmentation / Sharding

- Divides up a database's records among several sites
  - the resulting "pieces" are known as *fragments/shards*
- Let R be a collection of records of the same type (e.g., a relation).
- *Horizontal fragmentation* divides up the "rows" of R.
  - R(a, b, c) → R1(a, b, c), R2(a, b, c), …
  - R = R1 ∪ R2 ∪ …



- *Vertical fragmentation* divides up the "columns" of R.
  - R(a, b, c) → R1(a, b), R2(a, c), … (a is the primary key)
  - R = R1 ⋈ R2 ⋈ …



Fragmentation / Sharding (cont.)

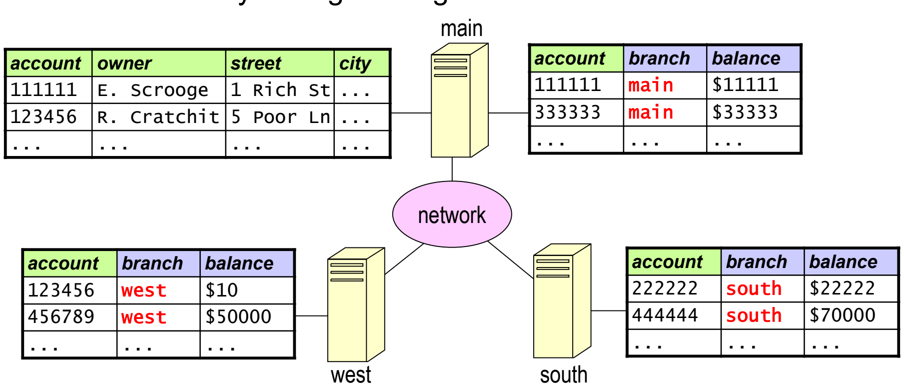
- Another version of vertical fragmentation:
  divide up the tables (or other collections of records).
  - e.g., site 1 gets tables A and B
    site 2 gets tables C and D

Example of Fragmentation

- Here's a relation from a centralized bank database:

account	owner	street	city	branch	balance
111111	E. Scrooge	1 Rich St	...	main	$11111
123456	R. Cratchit	5 Poor Ln	...	west	$10
...	...	...	...	...	...

- Here's one way of fragmenting it:



account	owner	street	city
111111	E. Scrooge	1 Rich St	...
123456	R. Cratchit	5 Poor Ln	...
...	...	...	...

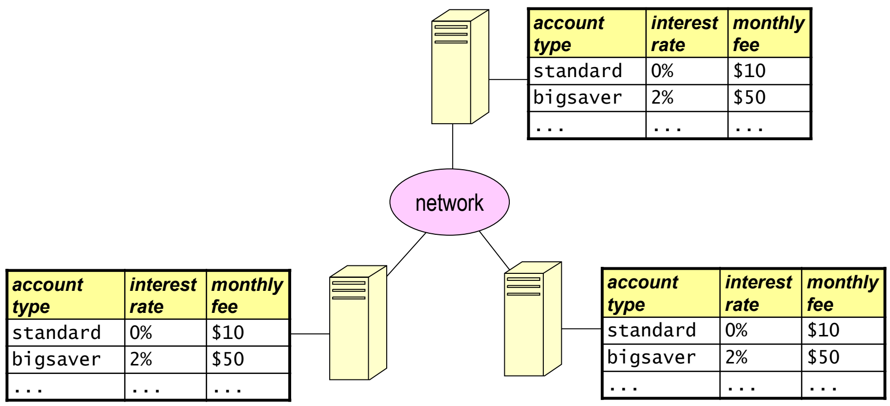
account	branch	balance
111111	main	$11111
333333	main	$33333
...	...	...

account	branch	balance
123456	west	$10
456789	west	$50000
...	...	...

account	branch	balance
222222	south	$22222
444444	south	$70000
...	...	...

Replication

- Replication involves putting copies of the same collection of records at different sites.



Reasons for Using a DDBMS

- to improve performance
  - how does distribution do this?
- to provide *high availability*
  - replication allows a database to remain available in the event of a failure at one site
- to allow for modular growth
  - add sites as demand increases
  - adapt to changes in organizational structure
- to integrate data from two or more existing systems
  - without needing to combine them
  - allows for the continued use of legacy systems
  - gives users a unified view of data maintained by different organizations

Challenges of Using a DDBMS (partial list)

- determining the best way to distribute the data
  - when should we use vertical/horizontal fragmentation?
  - what should be replicated, and how many copies do we need?
- determining the best way to execute a query
  - need to factor in communication costs
- maintaining integrity constraints (primary key, foreign key, etc.)
- ensuring that copies of replicated data remain consistent
- managing *distributed txns*: ones that involve data at multiple sites
  - atomicity and isolation can be harder to guarantee

Failures in a DDBMS

- In addition to the failures that can occur in a centralized system, there are additional types of failures for a DDBMS.
- These include:
  - the loss or corruption of messages
    - TCP/IP handles this type of error
  - the failure of a site
  - the failure of a communication link
    - can often be dealt with by rerouting the messages
  - *network partition:* failures prevent communication between two subgroups of the sites

Distributed Transactions

- A *distributed transaction* involves data stored at multiple sites.
- One of the sites serves as the *coordinator* of the transaction.
  - one option: the site on which the txn originated
- The coordinator divides a distributed transaction into *subtransactions*, each of which executes on one of the sites.

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

subtxn 1

```
read balance1
write(balance1 - 500)
```

subtxn 2

```
read balance2
write(balance2 + 500)
```

Types of Replication

- In *synchronous replication*, transactions are guaranteed to see the most up-to-date value of an item.
- In *asynchronous replication*, transactions are *not* guaranteed to see the most up-to-date value.

Synchronous Replication I: Read-Any, Write-All

- *Read-Any:* when reading an item, access *any* of the replicas.
- *Write-All:* when writing an item, must update *all* of the replicas.
- Works well when reads are much more frequent than writes.
- Drawback: writes are very expensive.

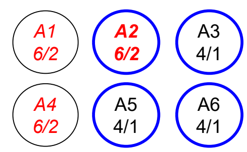
Synchronous Replication II: Voting

- When writing, update some fraction of the replicas.
  - each value has a version number that is increased when the value is updated
- When reading, read enough copies to ensure you get at least one copy of the most recent value (see next slide).
  - the copies "vote" on the value of the item
  - the copy with the highest version number is the most recent
- Drawback: reads are now more expensive

Synchronous Replication II: Voting (cont.)

- How many copies must be read?
  - let: n = the number of copies
    w = the number of copies that are written
    r = the number of copies that are read
  - need: r > n – w (i.e., at least n – w + 1)
  - example: n = 6 copies
    update w = 3 copies
    must read at least 4 copies

- Example: 6 copies of data item A, each with value = 4, version = 1.
  - txn 2 updates A1, A2, and A4 to be 6 (and their version number becomes 2)
  - txn 1 reads A2, A3, A5, and A6
  - A2 has the highest version number (2), so its value (6) is the most recent.



Which of these allow us to ensure that clients always get the most up-to-date value?

- 10 replicas – i.e., 10 copies of each item

- voting-based approach with the following requirements:

	number of copies <u>accessed when reading</u>	number of copies <u>accessed when writing</u>
A.	7	3
B.	5	5
C.	9	2
D.	4	8

(select all that work)

Distributed Concurrency Control

- To ensure the isolation of distributed transactions, need some form of distributed concurrency control.
- Extend the concurrency control schemes that we studied earlier.
  - we'll focus on extending strict 2PL
- If we just used strict 2PL at each site, we would ensure that the schedule of subtxns *at each site* is serializable.
  - why isn't this sufficient?

Distributed Concurrency Control (cont.)

- Example of why special steps are needed:
  - voting-based synchronous replication with 6 replicas
  - let's say that we configure the voting as before:
    - each write updates 3 copies
    - each read accesses 4 copies
  - can end up with schedules that are not conflict serializable
  - example:

T_1	T_2
xl(A1); xl(A2); xl(A3) w(A1); w(A2); w(A3)	
	xl(A4); xl(A5); xl(A6) w(A4); w(A5); w(A6)
	xl(B4); xl(B5); xl(B6) w(B4); w(B5); w(B6)
xl(B1); xl(B2); xl(B3) w(B1); w(B2); w(B3)	

Xi = the copy of item X at site i

T1 should come *before* T2 based on the order in which they write A.

T1 should come *after* T2 based on the order in which they write B.

What Do We Need?

- We need shared and exclusive locks for a *logical item*, not just for individual copies of that item.
  - referred to as *global locks*
  - doesn't necessarily mean locking every copy
- Requirements for global locks:
  - no two txns can hold a global exclusive lock for the same item
  - any number of txns can hold a global shared lock for an item
  - a txn cannot acquire a global exclusive lock on an item if another txn holds a global shared lock on that item, and vice versa

What Do We Need? (cont.)

- In addition, we need to ensure the correct ordering of operations *within* each distributed transaction.
  - don't want a subtxn to get ahead of where it should be in the context of the txn as a whole
  - relevant even in the absence of replication
  - one option: have the coordinator of the txn acquire the necessary locks before sending operations to a site

Option 1: Centralized Locking

- One site manages the lock requests for *all* items in the distributed database.
  - even items that don't have copies stored at that site
  - since there's only one place to acquire locks, these locks are obviously global locks!
- Problems with this approach?
  - the lock site can become a bottleneck
  - if the lock site crashes, operations at all sites are blocked

Option 2: Primary-Copy Locking

- One copy of an item is designated the *primary copy*.
- The site holding the primary copy handles all lock requests for that item.
  - acquiring a shared lock for the primary copy gives you a global shared lock for the item
  - acquiring an exclusive lock for the primary copy gives you a global exclusive lock for the item
- To prevent one site from becoming a bottleneck, distribute the primary copies among the sites.
- Problem: If a site goes down, operations are blocked on all items for which it holds the primary copy.

Option 3: Fully Distributed Locking

- No one site is responsible for managing lock requests for a given item.
- A transaction acquires a global lock for an item by locking a sufficient number of the item's copies.
  - these local locks combine to form the global lock
- To acquire a global shared lock, acquire local shared locks for a sufficient number of copies (see next slide).
- To acquire a global exclusive lock, acquire local exclusive locks for a sufficient number of copies (see next slide).

Option 3: Fully Distributed Locking (cont.)

- How many copies must be locked?
  - let: n = the total number of copies
    x = the number of copies that must be locked to acquire a global exclusive lock
    s = the number of copies that must be locked to acquire a global shared lock
  - we need $x > n/2$
    - guarantees that no two txns can both acquire a global exclusive lock at the same time
  - we need $s > n - x$ (i.e., $s + x > n$)
    - if there's a global exclusive lock on an item, there aren't enough unlocked copies for a global shared lock
    - if there's a global shared lock on an item, there aren't enough unlocked copies for a global excl. lock

Option 3: Fully Distributed Locking (cont.)

- Our earlier example would no longer be possible:

T_1	T_2
xl(A1); xl(A2); xl(A3)	
w(A1); w(A2); w(A3)	
	xl(A4); xl(A5); xl(A6) w(A4); w(A5); w(A6)
	xl(B4); xl(B5); xl(B6) w(B4); w(B5); w(B6)
xl(B1); xl(B2); xl(B3) w(B1); w(B2); w(B3)	

T_1	T_2
xl(A1); xl(A2); xl(A3); xl(A6)	
w(A1); w(A2); w(A3); w(A6)	
	xl(A4); xl(A5); xl(A6) – ***denied*** must wait for T1

- n = 6
- need x > 6/2
- must acquire at least 4 local exclusive locks before writing

Synchronous Replication and Fully Distributed Locking

- Read-any write-all:
  - when writing an item, a txn must update all of the replicas
    - this gives it x = n exclusive locks, so x > n/2
  - when reading an item, a txn can access any of the replicas
    - this gives it s = 1 shared lock, and 1 > n – n
- Voting:
  - when writing, a txn updates *a majority of the copies* – i.e., w copies, where w > n/2.
    - this gives it x > n/2 exclusive locks as required
  - when reading, a txn reads r > n – w copies
    - this gives it s > n – x shared locks as required

Which of these would work?

- 9 replicas – i.e., 9 copies of each item
- *fully distributed* locking
- voting-based approach with the following requirements:

	number of copies read	number of copies written
A.	5	5
B.	6	4
C.	7	3
D.	4	5

(select all that work)

Which of these would work?

- 9 replicas – i.e., 9 copies of each item
- *primary-copy* locking
- voting-based approach with the following requirements:

	number of copies read	number of copies written
A.	5	5
B.	6	4
C.	7	3
D.	4	5

(select all that work)

Distributed Deadlock Handling

- Under centralized locking, we can just use one of the schemes that we studied earlier in the semester.
- Under the other two locking schemes, *deadlock detection* becomes more difficult.
  - local waits-for graphs alone will not necessarily detect a deadlock
    - example:

  - one option: periodically send local waits-for graphs to one site that checks for deadlocks
- Instead of using deadlock detection, it's often easier to use a timeout-based scheme.
  - if a txn waits too long, presume deadlock and roll it back!

Recall: Types of Replication

- In *synchronous replication*, transactions are guaranteed to see the most up-to-date value of an item.
- In *asynchronous replication*, transactions are *not* guaranteed to see the most up-to-date value.

Asynchronous Replication I: Primary Site

- In *primary-site replication*, one replica is designated the *primary* or *master* replica.
- All writes go to the primary.
  - propagated asynchronously to the other replicas (the *secondaries*)
- The secondaries can only be read.
  - no locks are acquired when accessing them
  - thus, we only use them when performing read-only txns
- Drawbacks of this approach?

Asynchronous Replication II: Peer-to-Peer

- In *peer-to-peer replication*, more than one replica can be updated.
- Problem: need to somehow resolve conflicting updates!

Processing Distributed Data Using MapReduce

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

MapReduce

- A framework for computation on large data sets that are fragmented and replicated across a cluster of machines.
  - spreads the computation across the machines, letting them work in parallel
  - tries to minimize the amount of data that is transferred between machines
- The original version was Google's MapReduce system.
- An open-source version is part of the Hadoop project.
  - we'll use it as part of PS 4

Sample Problem: Totalling Customer Orders

- Acme Widgets is a company that sells only one type of product.
- *Data set:* a large collection of records about customer orders
  - fragmented and replicated across a cluster of machines
  - sample record:

```
('U123', 500, '03/22/17', 'active')
```

customer id   amount ordered   date ordered   order status

- *Desired computation:* For each customer, compute the total amount in that customer's active orders.
- Inefficient approach: Ship all of the data to one machine and compute the totals there.

Sample Problem: Totalling Customer Orders (cont.)

- MapReduce does better using "divide-and-conquer" approach.
  - splits the collection of records into subcollections that are processed in parallel
- For each subcollection, a *mapper task* maps the records to smaller key-value pairs – in this case, (cust_id, amount active).

```
('U123', 500, '03/22/17', 'active')   → ('U123', 500)
('U456',  50, '02/10/17', 'done')     → ('U456', 0)
('U123', 150, '03/23/17', 'active')   → ('U123', 150)
('U456',  75, '03/28/17', 'active')   → ('U456', 75)
```

- These smaller pairs are distributed by cust_id to other tasks that again work in parallel.
- These *reducer tasks* combine the pairs for a given cust_id to compute the per-customer totals:

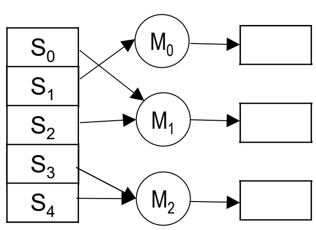
```
('U123', 500)                   ('U456', 0)
('U123', 150) → ('U123', 650)   ('U456', 75) → ('U456', 75)
```

Benefits of MapReduce

- Parallel processing reduces overall computation time.
- Less data is sent between machines.
  - the mappers often operate on local data
  - the key-value pairs sent to the reducers are smaller than the original records
  - an initial reduction can sometimes be done locally
    - example: compute local subtotals for each customer, then send those subtotals to the reducers
- It provides fault tolerance.
  - if a given task fails or is too slow, re-execute it
- The framework handles all of the hard/messy parts.
- The user can just focus on the problem being solved!

MapReduce In General: Mapping

- The system divides up the collection of input records, and assigns each subcollection S_i to a mapper task M_j.



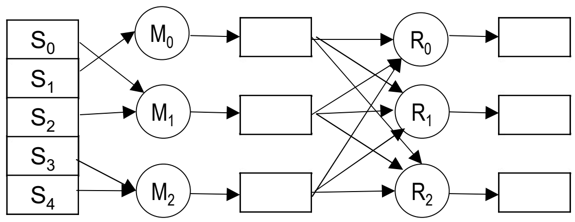
- The mappers apply a map function to each record:

```
map(k, v):    # treat record as a key-value pair
   emit 0 or more new key-value pairs (k', v')
```

  - the resulting keys and values (the *intermediate results)* can have different types than the original ones
  - the input and intermediate keys do *not* have to be unique

MapReduce In General: Reducing

- The system partitions the intermediate results by key, and assigns each range of keys to a reducer task R_k.



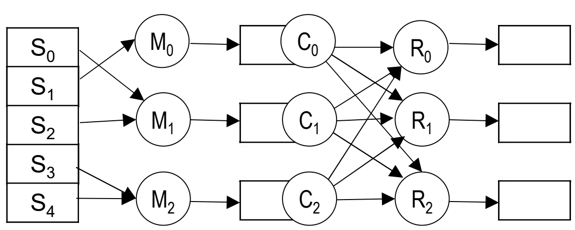
- Key-value pairs with the same key are grouped together:
  (k', v'$_0$), (k', v'$_1$), (k', v'$_2$) → (k', [v'$_0$, v'$_1$, v'$_2$, ...])
  - so that *all* values for a given key are processed together
- The reducers apply a reduce function to each (key, value-list):

```
reduce(k', [v'0, v'1, v'2, ...]):
    emit 0 or more key-value pairs (k", v")
```

  - the types of the (k", v") can be different from the (k', v')

MapReduce In General: Combining (Optional)

- In some cases, the intermediate results can be aggregated locally using *combiner* tasks C_n.



- Often, the combiners use the same reduce function as the reducers.
  - produces partial results that can then be combined
- This cuts down on the data transferred to the reducers.

Hadoop MapReduce Framework

- Implemented in Java
- It also includes other, non-Java options for writing MapReduce applications.
- In PS 4, you'll write simple MapReduce applications in Java.
- To do so, you need to become familiar with some key classes from the MapReduce API.
- We'll also review some relevant Java concepts.

Classes and Interfaces for Keys and Values

- Found in the `org.apache.hadoop.io` package
- Types used for values must implement the `Writable` interface.
  - includes methods for efficiently serializing/writing the value
- Types used for keys must implement `WritableComparable`.
  - in addition to the `Writeable` methods, must also have a `compareTo()` method that allows values to be compared
  - needed to sort the keys and create key subranges
- The following classes implement both interfaces:
  - `IntWritable` – for 4-byte integers
  - `LongWritable` – for long integers
  - `DoubleWritable` – for floating-point numbers
  - `Text` – for strings/text (encoded using UTF8)

Recall: Generic Classes

```
public class ArrayList<T> {
    private T[] items;
    …
    public boolean add(T item) {
        …
    }
    …
}
```

- The header of a generic class includes one or more *type variables*.
  - in the above example: the variable `T`
- The type variables serve as placeholders for actual data types.
- They can be used as the types of:
  - fields
  - method parameters
  - method return types

Recall: Generic Classes (cont.)

```
public class ArrayList<T> {
    private T[] items;
    …
    public boolean add(T item) {
        …
    }
    …
}
```

- When we create an instance of a generic class, we specify types for the type variables:

  ```
  ArrayList<Integer> vals = new ArrayList<Integer>();
  ```

  - `vals` will have an `items` field of type `Integer[]`
  - `vals` will have an `add` method that takes an `Integer`
- We can also do this when we create a subclass of a generic class:

  ```
  public class IntList extends ArrayList<Integer> {
      ...
  ```

Mapper Class

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
```

KEYIN, VALUEIN: type variables for the (key, value) pairs given to the mapper

KEYOUT, VALUEOUT: type variables for the (key, value) pairs produced by the mapper

- the principal method:

  ```
  void map(KEYIN key, VALUEIN value, Context context)
  ```

- To implement a mapper:
  - extend this class with appropriate replacements for the type variables; for example:

    ```
    class MyMapper
      extends Mapper<Object, Text, Text, IntWritable>
    ```

  - override `map()`

Reducer Class

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
```

KEYIN, VALUEIN: type variables for the (key, value) pairs given to the reducer

KEYOUT, VALUEOUT: type variables for the (key, value) pairs produced by the reducer

- the principal method:

  ```
  void reduce(KEYIN key, Iterable<VALUEIN> values,
              Context context)
  ```

- To implement a reducer:
  - extend this class with appropriate replacements for the type variables
  - override `reduce()`

`Context` Objects

- Both `map()` and `reduce()` are passed a `Context` object:

```
void map(KEYIN key, VALUEIN value, Context context)
void reduce(KEYIN key, Iterable<VALUEIN> values,
            Context context)
```

- Allows `Mappers` and `Reducers` to communicate with the `MapReduce` framework.

- Includes a `write()` method used to output (key, value) pairs:

```
void write(KEYOUT key, VALUEOUT value)
```

Example

```
class MyMapper extends Mapper<Object, Text,
                      LongWriteable, IntWritable>
```

Which of these is the correct header for the `map` method?

A. `void map(LongWriteable key, IntWritable value, Context context)`

B. `void map(Text key, LongWriteable value, Context context)`

C. `void map(Object key, IntWriteable value, Context context)`

D. `void map(Object key, Text value, Context context)`

Example 1: Birth-Month Counter

- ***The data:*** text file(s) containing person records that look like this

  `id,name,dob,email`

  where dob is in the form `yyyy-mm-dd`

- ***The problem:*** Find the number of people born in each month.

Example 1: Birth-Month Counter (cont.)

- `map` should:
  - extract the month from the person's dob
  - emit a single key-value pair of the form (month string, `1`)

```
111,Alan Turing,1912-06-23,al@aol.com           → ("06", 1)
234,Grace Hopper,1906-12-09,gmh@harvard.edu     → ("12", 1)
444,Ada Lovelace,1815-12-10,ada@1800s.org       → ("12", 1)
567,Howard Aiken,1900-03-08,aiken@harvard.edu   → ("03", 1)
777,Joan Clarke,1917-06-24,joan@bletchley.org   → ("06", 1)
999,J. von Neumann,1903-12-28,jvn@princeton.edu → ("12", 1)
```

- The intermediate results are distributed by key to the reducers.

- `reduce` should:
  - add up the 1s for a given month
  - emit a single key-value pair of the form (month string, total)

```
("06", [1, 1])      → ("06", 2)
("12", [1, 1, 1])   → ("12", 3)
("03", [1])         → ("03", 1)
```

Mapper for Example 1

```
public class BirthMonthCounter {
    public static class MyMapper
        extends Mapper<Object, Text, Text, IntWritable>
```

- For data obtained from text files, the Mapper's inputs will be key-values pairs in which:
  - value = a single line from one of the files (a `Text` value)
  - key = the location of the line in the file (a `LongWritable`)
    - however, we use the `Object` type for the key because we ignore it, and thus we don't need any `LongWritable` methods
- The `map` method will output pairs in which:
  - key = a month string (use `Text` for it)
  - value = 1 (use `IntWritable`)

Mapper for Example 1 (cont.)

```
public class BirthMonthCounter {
    public static class MyMapper
        extends Mapper<Object, Text, Text, IntWritable>
    {
        public void map(Object key, Text value,
                        Context context)
        {
            String record = value.toString();
            // code to extract month string goes here
            context.write(new Text(month),
                          new IntWritable(1));
        }
    }
    ...
}
```

Splitting a String

- The `String` class includes a method named `split()`.
  - breaks a string into component strings
  - takes a parameter indicating what delimiter should be used when performing the split
  - returns a `String` array containing the components
- Example:

```
String sentence = "How now brown cow?";
String[] words = sentence.split(" ");
System.out.println(words[0]);
System.out.println(words[3]);
System.out.println(words.length);
```

would output:

Processing an Input Record in `map`

```
void map(Object key, Text value, Context context)
```

- Recall: `value` is a `Text` object representing one record.
  - for Example 1, it looks like:

    ```
    111,Alan Turing,1912-06-23,al@aol.com
    ```

- To extract the month string:
  - use the `toString()` method to convert `Text` to `String`:

    ```
    String line = value.toString();
    ```

  - split `line` on the commas to get the fields:

    ```
    String[] fields = line.split(",");
    ```

  - similarly, split the date field on the hyphens to get its components
  - could we just split `line` on the hyphens?

Reducer for Example 1

```
public static class MyMapper
    extends Mapper<Object, Text, Text, IntWritable>
{
    …
}

public static class MyReducer
    extends Reducer<Text, IntWritable,
                    Text, LongWritable>
{
   public void reduce(Text key,
        Iterable<IntWritable> values, Context context)
   {
       // code to add up the list of 1s goes here
       context.write(key, new LongWritable(total));
   }
   ...
```

- Use `LongWritable` to avoid overflow with large totals.

Processing the List of Values in `reduce`

```
void reduce(Text key, Iterable<IntWritable> values,
            Context context)
```

- Use a *for-each* loop. In this case:

```
for (IntWritable val : values)
```

- More generally, if `values` is of type `Iterable<T>` :

```
for (T val : values)
```

- To extract the underlying value from most `Writable` objects, use the `get()` method:

```
int count = val.get();    // val is IntWritable
```

- However, `Text` doesn't have a get() method.
  - use `toString()` instead (see earlier notes)

Reducer for Birth-Month Counter

```
public class BirthMonthCounter {
 ...
 public static class MyReducer
     extends Reducer<Text, IntWritable,
                     Text, LongWritable>
 {
    public void reduce(Text key,
         Iterable<IntWritable> values, Context context)
    {
        long total = 0;
        for (IntWritable val : values) {
            total += val.get()
        }

        context.write(key, new LongWritable(total));
    }
    ...
```

- Use `long` and `LongWritable` to avoid overflow.

`Job` Objects

- We use a `Job` object to:
    - provide information about our MapReduce job, such as:
        - the name of the Mapper class
        - the name of the Reducer class
        - the types of values produced by the job
        - the format of the input to the job
    - execute the job
- We'll give you a template for the necessary method calls.

Configuring and Running the Job

```
public class BirthMonthCounter {
    public static class MyMapper extends... {
       ...
    public static class MyReducer extends... {
       ...
    public static void main(String args)
      throws Exception {
         // code to configure and run the job
    }
}
```

Configuring and Running the Job

```
public static void main(String[] args)
  throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "birth month");
    job.setJarByClass(BirthMonthCounter.class);

    job.setMapperClass(MyMapper.class);
    job.setReducerClass(MyReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);

    // type for mapper's output value,
    // because its not the same as the reducer's
    job.setMapOutputValueClass(IntWritable.class);

    job.setInputFormatClass(TextInputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.waitForCompletion(true);
}
```

Example 2: Month with the Most Birthdays

- ***The data:*** same as Example 1. Records of the form

  `id,name,dob,email`

  where dob is in the form `yyyy-mm-dd`

- ***The problem:*** Find the month with the most birthdays.

Example 2: Month with the Most Birthdays (cont.)

- `map` should behave as before:

```
111,Alan Turing,1912-06-23,al@aol.com          → ("06", 1)
234,Grace Hopper,1906-12-09,gmh@harvard.edu    → ("12", 1)
444,Ada Lovelace,1815-12-10,ada@1800s.org      → ("12", 1)
```

- `reduce` needs to:
  - add up the 1s for a given month

    ```
    ("06", [1, 1])      → ("06", 2)
    ("12", [1, 1, 1])   → ("12", 3)
    ("03", [1])         → ("03", 1)
    ```
  - determine which month has the largest total
  - ***but...***
    - there can be multiple reducer tasks, each of which handles one subset of the months
    - each reducer can only determine the largest month in its subset
  - ***the solution:*** a *chain* of two MapReduce jobs

Example 2: Chaining Jobs

- First job = count birth months as we did in Example 1
  - map1: person record → (birth month, 1)
  - reduce1: (birth month, [1, 1, ...]) → (birth month, total)

- The second job processes the results of the first job!
  - map2: (birth month, total) → (**c**, **(birth month, total)**)
    - output **key c** = an arbitrary ***constant***, used for *all* k-v pairs
    - output **value** = a pairing of a birth month and its total

```
("06", 2) → ("month sum", "06,2")
("12", 3) → ("month sum", "12,3")
("03", 1) → ("month sum", "03,1")
```

    - because there is only one output key, there is only one reducer task!
  - reduce2: find the month with the most birthdays

```
("month sum", ["06,2", "12,3", "03,1"]) → ("12", 3)
```

Example 2: Chaining Jobs (cont.)

```
public class MostBirthdaysMonth {
    public static class MyMapper1 extends... {
        ...
    }
    public static class MyReducer1 extends... {
        ...
    }
    public static class MyMapper2 extends... {
        ...
    }
    public static class MyReducer2 extends... {
        ...
    }

    public static void main(String[] args) throws... {
        ...
}
```

Configuring and Running a Chain of Jobs

```
public static void main(String args)
  throws Exception {
    Configuration conf = new Configuration();
    Job job1 = Job.getInstance(conf, "birth month");
    job1.setJarByClass(MostBirthdaysMonth.class);
    job1.setMapperClass(MyMapper1.class);
    job1.setReducerClass(MyReducer1.class);
    ...
    FileInputFormat.addInputPath(job1, new Path(args[0]));
    FileOutputFormat.setOutputPath(job1, new Path(args[1]));
    job1.waitForCompletion(true);

    Job job2 = Job.getInstance(conf, "max month");
    job2.setJarByClass(MostBirthdaysMonth.class);
    job2.setMapperClass(MyMapper2.class);
    job2.setReducerClass(MyReducer2.class);
    ...
    FileInputFormat.addInputPath(job2, new Path(args[1]));
    FileOutputFormat.setOutputPath(job2, new Path(args[2]));
    job2.waitForCompletion(true);
}
```

Structure of the Java Files

- In theory, we could use multiple Java files for each problem:
  - one file for the program as a whole
  - one file for the Mapper class, one for the Reducer class, etc.
- Instead, we'll put all of the classes in the same file by using *static nested classes:*

```
public class MyProblem {
    public static class MyMapper extends ... {
          ...
    }
    public static class MyReducer extends ... {
          ...
    }
```

- Unlike an inner class (aka a *non*-static nested class), static nested classes do *not* depend on their outer class.
  - they are equivalent to an outer class from another file
  - allows the MapReduce system to instantiate them

NoSQL Databases

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

The Rise of NoSQL

- Beginning in the early 2000s, web-based applications increasingly needed to deal with massive amounts of:
  - data
  - traffic / queries
- Scalability is crucial.
  - load can increase rapidly and unpredictably
- Large servers are expensive and can only grow so large.
- Solution: use clusters of small commodity machines
  - use both fragmentation/sharding and replication
  - cheaper
  - greater overall reliability
  - can take advantage of cloud-based storage

The Rise of NoSQL (cont.)

- Problem: Relational DBMSs do not scale well to large clusters.
- Google and Amazon each developed their own alternative approaches to data management on clusters.
  - Google: BigTable
  - Amazon: DynamoDB
- The papers that Google and Amazon published about their efforts got others interested in developing similar DBMSs.

  ➔ noSQL

What Does NoSQL Mean?

- Not well defined.
- Typical characteristics of NoSQL DBMSs:
  - don't use SQL / the relational model
  - open-source
  - designed for use on clusters
    - support for sharding/fragmentation and replication
  - schema-less or flexible schema
- One good overview:

  Sadalage and Fowler, *NoSQL Distilled*
  (Addison-Wesley, 2013).

Flavors of NoSQL

- Various taxonomies have been proposed
- Three of the main classes of NoSQL databases are:
  - key-value stores
  - document databases
  - column-family (aka big-table) stores
- Some people also include graph databases.
  - very different than the others
  - example: they are *not* designed for clusters

Key-Value Stores

- We've already worked with one of these: Berkeley DB
- Simple data model: key/value pairs
  - the DBMS does *not* attempt to interpret the value
- Queries are limited to query by key.
  - get/put/update/delete a key/value pair
  - iterate over key/value pairs

Document Databases

- Also store key/value pairs
- Unlike key-value stores, the value is *not* opaque.
  - it is a *document* containing semistructured data
  - it *can* be examined and used by the DBMS
- Queries:
  - can be based on the key (as in key/value stores)
  - more often, are based on the contents of the document
- Here again, there is support for sharding and replication.
  - the sharding can be based on values within the document

Column-Family Databases

- Google's BigTable and systems based on it
- To understand the motivation behind their design, consider one type of problem BigTable was designed to solve:
  - You want to store info about web pages!
  - For each URL, you want to store:
    - its contents
    - its language
    - for each other page that links to it, the *anchor text* associated with the link (i.e., the text that you click on)

Storing Web-Page Data in a Traditional Table

page URL	**language**	**contents**	**anchor text from www.cnn.com**	**anchor from www.bu.edu**	***one col per page***								
www.cnn.com	English	<html>…											…
www.bu.edu	English	<html>…											…
www.nytimes.com	English	<html>…		"news story"									…
www.lemonde.fr	French	<html>…	"French elections"										…
…													…
													…

- One row per web page
- Single columns for its language and contents
- *One column for the anchor text from each possible page,* since in theory any page could link to any other page!
- Leads to a huge *sparse* table – most cells are empty/unused.

Storing Web-Page Data in BigTable

- Rather than defining all possible columns, define a set of *column families* that each row should have.
  - example: a column family called *anchor* that replaces all of the separate anchor columns on the last slide
  - can also have column families that are like typical columns
- In a given row, only store columns with an actual value, representing them as (column key, value) pairs
  - column key = column family:qualifier
  - ex: `("anchor:www.bu.edu", "news story")`

column family — *qualifier* — *value*

column key

Data Model for Column-Family Databases

- Different rows can have different schema.
  - i.e., different sets of column *keys*
  - (column key, value) pairs can be added or removed from a given row over time
- The set of column *families* in a given table rarely change.

Advantages of Column Families

- Gives an additional unit of data, beyond just a single row.
- Can be used for access controls.
  - restrict an application to only certain column families
- Column families can be divided up into *locality groups* that are stored together.
  - based on which column families are typically accessed together
  - advantage?

Aggregate Orientation

- Key-value, document, and column-family stores all lend themselves to an *aggregate-oriented* approach.
  - group together data that "belongs" together
    - i.e., that will tend to be accessed together

type of database	unit of aggregation
key-value store	the value part of the key/value pair
document database	a document
column-family store	a row (plus column-family sub-aggregates)

- Relational databases can't fully support aggregation.
  - no multi-valued attributes; focus on avoiding duplicated data
  - give each type of entity its own table, rather than grouping together entities/attributes that are accessed together

Aggregate Orientation (cont.)

- Example: data about customers
  - RDBMS: store a customer's address in only one table
    - use foreign keys in other tables that refer to the address
  - aggregate-oriented system: store the full customer address in several places:
    - customer aggregates
    - order aggregates
    - etc.
- Benefits of an aggregate-based approach in a NoSQL store:
  - provides a unit for sharding across the cluster
  - allows us to get related data without needing to access many different nodes

Schemalessness

- NoSQL systems are completely or mostly schemaless.
- Key-value stores: put whatever you like in the value
- Document databases: no restrictions on the schema used by the semistructured data inside each document.
  - although some do allow a schema, as with XML
- Column-family databases:
  - we do specify the column families in a given table
  - but no restrictions on the columns in a given column family and different rows can have different columns

Schemalessness (cont.)

- Advantages:
  - allows the types of data that are stored to evolve over time
  - makes it easier to handle nonuniform data
    - e.g., sparse tables
- Despite the fact that a schema is not required,
  programs that use the data need at least an *implicit* schema.
- Disadvantages of an implicit schema:
  - the DBMS can't enforce it
  - the DBMS can't use it to try to make accesses more efficient
  - different programs that access the same database can have conflicting notions of the schema

Example Document Database: MongoDB

- Mongo (from hu<u>mongo</u>us)
- Key features include:
  - replication for high availability
  - auto-sharding for scalability
  - documents are expressed using JSON/BSON
  - queries can be based on the contents of the documents
- Related documents are grouped together into *collections*.
  - what does this remind you of?

mongoDB

JSON

- JSON is an alternative data model for semistructured data.
  - <u>J</u>ava<u>S</u>cript <u>O</u>bject <u>N</u>otation
- Built on two key structures:
  - an *object*, which is a sequence of *fields* (name:value pairs)

```
{ id: "1000",
  name: "Sanders Theatre",
  capacity: 1000 }
```

  - an *array* of values

```
["123-456-7890", "222-222-2222", "333-333-3333"]
```

- A value can be:
  - an atomic value: string, number, true, false, null
  - an object
  - an array

Example: JSON Object for a Person

```
{   firstName: "John",
    lastName: "Smith",
    age: 25,
    address: {
        streetAddress: "21 2nd Street",
        city: "New York",
        state: "NY",
        postalCode: "10021"
    },
    phoneNumbers: [
        {   type: "home",
            number: "212-555-1234"
        },
        {   type: "mobile",
            number: "646-555-4567"
        }
    ]
}
```

BSON

- MongoDB actually uses BSON.
  - a binary representation of JSON
  - BSON = marshalled JSON!
- BSON includes some additional types that are not part of JSON.
  - in particular, a type called ObjectID for unique id values.
- Each MongoDB document is a BSON object.

The `_id` Field

- Every MongoDB document must have an `_id` field.
  - its value must be unique within the collection
  - acts as the primary key of the collection
  - it is the key in the key/value pair
- If you create a document without an `_id` field:
  - MongoDB adds the field for you
  - assigns it a unique BSON ObjectID

MongoDB Terminology

relational term	**MongoDB equivalent**
database	database
table	collection
row	document
attributes	fields (name:value pairs)
primary key	the _id field, which is the key associated with the document

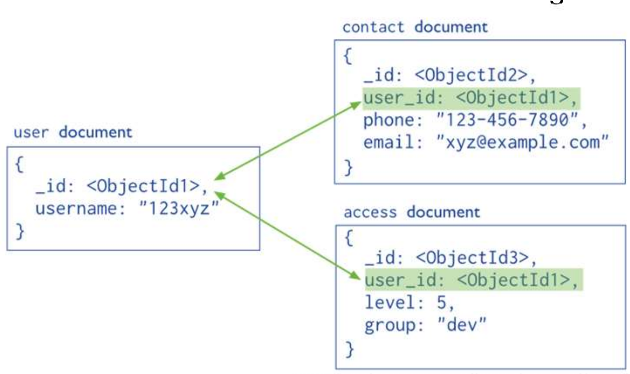
- Documents in a given collection typically have a similar purpose.
- However, no schema is enforced.
  - different documents in the same collection can have different fields

Data Modeling in MongoDB

- Need to determine how to map
  entities and relationships → collections of documents
- Could in theory give each type of entity:
  - its own (flexibly formatted) type of document
  - those documents would be stored in the same collection
- However, recall that NoSQL models allow for *aggregates* in which different types of entities are grouped together.
- Determining what the aggregates should look like involves deciding how we want to represent relationships.

Capturing Relationships in MongoDB

- Two options:
  1. store references to other documents using their `_id` values



source: docs.mongodb.org/manual/core/ data-model-design

  - where have we seen this before?

Capturing Relationships in MongoDB (cont.)

- Two options (cont.):

  2. embed documents within other documents

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
          }
}
```

Embedded sub-document (contact)

Embedded sub-document (access)

source: docs.mongodb.org/manual/core/ data-model-design

  - where have we seen this before?

Factors Relevant to Data Modeling

- A given MongoDB query can only access a single collection.
  - joins of documents are *not* supported
  - need to issue multiple requests

  → group together data that would otherwise need to be joined

- Atomicity is only provided for operations on a single document (and its embedded subdocuments).

  → group together data that needs to be updated as part of single logical operation (e.g., a balance transfer!)

  → group together data items A and B if A's current value affects whether/how you update B

Factors Relevant to Data Modeling (cont.)

- If an update makes a document bigger than the space allocated for it on disk, it may need to be relocated.
  - slows down the update, and can cause disk fragmentation
  - MongoDB adds padding to documents to reduce the need for relocation
  - → use references if embedded documents could lead to significant growth in the size of the document over time

Factors Relevant to Data Modeling

- Pluses and minuses of embedding (a partial list):
  - \+ need to make fewer requests for a given logical operation
  - \+ less network/disk I/O
  - \+ enables atomic updates
  - – duplication of data
  - – possibility for inconsistencies between different copies of duplicated data
  - – can lead documents to become very large, and to document relocation
- Pluses and minuses of using references:
  - take the opposite of the pluses and minuses of the above!
  - \+ allow you to capture more complicated relationships
    - ones that would be modelled using *graphs*

Data Model for the Movie Database

- Recall our movie database from PS 1.
  Person(id, name, dob, pob)
  Movie(id, name, year, rating, runtime, genre, earnings_rank)
  Oscar(movie_id, person_id, type, year)
  Actor(actor_id, movie_id)
  Director(director_id, movie_id)

- Three types of entities: movies, people, oscars

- Need to decide how we should capture the relationships
  - between movies and actors
  - between movies and directors
  - between Oscars and the associated people and movies

Data Model for the Movie Database (cont.)

- Assumptions about the relationships:
  - there are only one or two directors per movie
  - there are approx. five actors associated with each movie
  - the number of people associated with a given movie is fixed
  - each Oscar has exactly one associated movie
    and at most one associated person

- Assumptions about the queries:
  - Queries that involve both movies and people usually involve only the *names* of the people, not their other info.

    common: *Who directed Avatar?*
    common: *Which movies did Tom Hanks act in?*
    less common: *Which movies have actors from Boston?*
  - Queries that involve both Oscars and other entities usually involve only the *name(s)* of the person/movie.

Data Model for the Movie Database (cont.)

- Given our assumptions, we can take a hybrid approach that includes both references and embedding.
- Use three collections: movies, people, oscars
- Use references as follows:
  - in movie documents, include ids of the actors and directors
  - in oscar documents, include ids of the person and movie
- Whenever we refer to a person or movie, we also embed the associated entity's name.
  - allows us to satisfy common queries like *Who acted in…?*
- For less common queries that involve info. from multiple entities, use the references.

Data Model for the Movie Database (cont.)

- In addition, add two boolean fields to person documents:
  - hasActed, hasDirected
  - only include when true
  - allows us to find all actors/directors that meet criteria involving their pob/dob
- Note that most per-entity state appears only once, in the main document for that entity.
- The only duplication is of people/movie names and ids.

Sample Movie Document

```
{ _id: "0499549",
  name: "Avatar",
  year: 2009,
  rating: "PG-13",
  runtime: 162,
  genre: "AVYS",
  earnings_rank: 1,
  actors: [ { id: "0000244",
              name: "Sigourney Weaver" },
            { id: "0002332",
              name: "Stephen Lang" },
            { id: "0735442",
              name: "Michelle Rodriguez" },
            { id: "0757855",
              name: "Zoe Saldana" },
            { id: "0941777",
              name: "Sam Worthington" } ],
  directors: [ { id: "0000116",
                 name: "James Cameron" } ] }
```

Sample Person and Oscar Documents

```
{ _id: "0000059",
  name: "Laurence Olivier",
  dob: "1907-5-22",
  pob: "Dorking, Surrey, England, UK",
  hasActed: true,
  hasDirected: true
}

{ _id: ObjectId("528bf38ce6d3df97b49a0569"),
  year: 2013,
  type: "BEST-ACTOR",
  person: { id: "0000358",
            name: "Daniel Day-Lewis" },
  movie: { id: "0443272",
           name: "Lincoln" }
}
```

Queries in MongoDB

- Each query can only access a single collection of documents.
- Use a method called `db.collection.find()`

  `db.collection.find(<selection>, <projection>)`

  - *`collection`* is the name of the collection
  - *`<selection>`* is an optional document that specifies one or more selection criteria
    - omitting it (i.e., using an empty document `{}`) selects all documents in the collection
  - *`<projection>`* is an optional document that specifies which fields should be returned
    - omitting it gets all fields in the document

- Example: find the names of all R-rated movies:

```
db.movies.find({ rating: "R" }, { name: 1 })
```

Comparison with SQL

- Example: find the names and runtimes of all R-rated movies that were released in 2000.
- SQL:

```
SELECT name, runtime
FROM Movie
WHERE rating = 'R' and year = 2000;
```

- MongoDB:

```
db.movies.find({ rating: "R", year: 2000 },
               { name: 1, runtime: 1 })
```

Query Selection Criteria

```
db.collection.find(<selection>, <projection>)
```

- To find documents that match a set of field values, use a selection document consisting of those name/value pairs (see previous example).
- Operators for other types of comparisons:

MongoDB	SQL equivalent
`$gt, $gte`	`>, >=`
`$lt, $lte`	`<, <=`
`$ne`	`!=`

- Example: find all movies with an earnings rank <= 200

```
db.movies.find({ earnings_rank: { $lte: 200 }})
```

- Note that the operator is the field name of a subdocument.

Query Selection Criteria (cont.)

- Logical operators: `$and, $or, $not, $nor`
  - take an *array* of selection subdocuments
  - example: find all movies rated R or PG-13:

```
db.movies.find({ $or: [ { rating: "R" },
                        { rating: "PG-13" }
                      ]
               })
```

  - example: find all movies *except* those rated R or PG-13 :

```
db.movies.find({ $nor: [ { rating: "R" },
                         { rating: "PG-13" }
                       ]
               })
```

Query Selection Criteria (cont.)

- To test for set-membership or lack thereof: `$in`, `$nin`
  - example: find all movies rated R or PG-13:

```
db.movies.find({ rating: { $in: ["R", "PG-13"] }
               })
```

  - example: find all movies *except* those rated R or PG-13 :

```
db.movies.find({ rating: { $nin: ["R", "PG-13"] }
               })
```

  - ***note:*** `$in/$nin` is generally more efficient than `$or/$nor`

- To test for the presence/absence of a field: `$exists`
  - example: find all movies with an earnings rank:

```
db.movies.find({ earnings_rank: { $exists: true }})
```

  - example: find all movies *without* an earnings rank:

```
db.movies.find({ earnings_rank: { $exists: false }})
```

Logical AND

- You get an implicit logical AND by simply specifying a list of fields.
  - recall our previous example:

```
db.movies.find({ rating: "R", year: 2000 })
```

  - example: find all R-rated movies shorter than 90 minutes:

```
db.movies.find({ rating: "R",
                 runtime: { $lt: 90 }
               })
```

Logical AND (cont.)

- `$and` is needed if the subconditions involve the same field
  - can't have duplicate field names in a given document
- Example: find all Oscars given in the 1990s.
  - the following would ***not*** work:

```
db.oscars.find({ year: { $gte: 1990 },
                 year: { $lte: 1999 }
               })
```

  - one option that would work:

```
db.oscars.find({ $and: [ { year: { $gte: 1990 } },
                         { year: { $lte: 1999 } } ]
               })
```

  - another option: use an implicit AND on the operator subdocs:

```
db.oscars.find({ year: { $gte: 1990, $lte: 1999 }
               })
```

Pattern Matching

- Use a regular expression surrounded with //
  - example: find all people born in Boston

```
db.people.find({ pob: /*Boston,*/ })
```

  - `*` is a wildcard character that acts like `%` in SQL
- We get a `*` by default on either end of the expression, so we can do this instead:

```
db.people.find({ pob: /Boston,/ })
```

- To override the default * characters, use:
  ^ to require a match with the beginning of the value
  $ to require a match with the end of the value
  - `/Boston,/` would match "`South Boston, Mass`"
  - `/^Boston,/` would not, because the ^ indicates "`Boston`" must be at the start of the value
  - `/USA$/` requires "`USA`" to be at the end of the value

Query Practice Problem

- Recall our sample person document:

```
{ _id: "0000059",
  name: "Laurence Olivier",
  dob: "1907-5-22",
  pob: "Dorking, Surrey, England, UK",
  hasActed: true,
  hasDirected: true
}
```

- How could we find all directors born in the UK? (Select all that apply.)

A. `db.people.find({ pob: /UK$/, hasDirected: true })`

B.
```
db.people.find({ pob: /UK$/,
                 hasDirected: { $exists: true }})
```

C.
```
db.people.find({ pob: /UK/,
                 hasDirected: { $exists: true }})
```

D. `db.people.find({ $pob: /UK/, $hasDirected: true })`

Queries on Arrays/Subdocuments

- If a field has an array type

    `db.`*`collection`*`.find( { arrayField: val } )`

  finds all documents in which `val` is at least one of the elements in the array associated with `arrayField`

- Example: suppose that we stored a movie's genres as an array:

```
{ _id: "0317219", name: "Cars", year: 2006,
  rating: "G", runtime: 124, earnings_rank: 80,
  genre: ["N", "C", "F"], ...}
```

  - to find all animated movies – ones with a genre of "N":

    `db.movies.find( { genre: "N"} )`

- Given that we actually store the genres as a single string (e.g., "NCF"), how would we find animated movies?

Queries on Arrays/Subdocuments (cont.)

- Use dot notation to access fields within a subdocument, or within an array of subdocuments:
  - example: find all Oscars won by the movie *Gladiator:*

```
> db.oscars.find( { "movie.name": "Gladiator" } )

{ _id: <ObjectID1>, year: 2001,
  type: "BEST-PICTURE",
  movie: { id: "0172495",
           name: "Gladiator" }}
{ _id: <ObjectID2>, year: 2001,
  type: "BEST-ACTOR",
  movie: { id: "0172495",
           name: "Gladiator" },
  person: { id: "0000128",
            name: "Russell Crowe" }}
```

- ***Note***: When using dot notation, the field name must be surrounded by quotes.

Queries on Arrays/Subdocuments (cont.)

- example: find all movies in which Tom Hanks has acted:

```
> db.movies.find( { "actors.name": "Tom Hanks"} )

{ _id: "0107818", name: "Philadelphia", year: 1993,
  rating: "PG-13", runtime: 125, genre: "D"
  actors: [ { id: "0000158",
              name: "Tom Hanks" },
            { id: "0000243",
              name: "Denzel Washington" },
            ...
          ],
  directors: [ { id: "0001129",
                 name: "Jonathan Demme" } ]
}
{ _id: "0109830", name: "Forrest Gump", year: 1994,
  rating: "PG-13", runtime: 142, genre: "CD"
  actors: [ { id: "0000158",
              name: "Tom Hanks" },
            ...
```

Projections

`db.collection.find(<selection>, <projection>)`

- The projection document is a list of *fieldname:value* pairs:
  - a value of 1 indicates the field should be included
  - a value of 0 indicates the field should be excluded
- Recall our previous example:

```
db.movies.find({ rating: "R", year: 2000 },
               { name: 1, runtime: 1 })
```

- Example: find all info. about R-rated movies except their genres:

```
db.movies.find({ rating: "R" }, { genre: 0 })
```

Projections (cont.)

- The _id field is returned unless you explicitly exclude it.

```
> db.movies.find({ rating: "R", year: 2011 },
                 { name: 1 })
{ "_id" : "1411697", "name" : "The Hangover Part II" }
{ "_id" : "1478338", "name" : "Bridesmaids" }
{ "_id" : "1532503", "name" : "Beginners" }

> db.movies.find({ rating: "R", year: 2011 },
                 { name: 1, _id: 0 })
{ "name" : "The Hangover Part II" }
{ "name" : "Bridesmaids" }
{ "name" : "Beginners" }
```

- A given projection should either have:
  - all values of 1: specifying the fields to include
  - all values of 0: specifying the fields to exclude
  - one exception: specify fields to include, and exclude `_id`

Iterating Over the Results of a Query

- `db.`*`collection`*`.find()` returns a cursor that can be used to iterate over the results of a query
- In the MongoDB shell, if you don't assign the cursor to a variable, it will automatically be used to print up to 20 results.
  - if more than 20, use the command `it` to continue the iteration
- Another way to view all of the result documents:
  - assign the cursor to a variable:

    ```
    var cursor = db.movies.find({ year: 2000 })
    ```

  - use the following method call to print each result document in JSON:

    ```
    cursor.forEach(printjson)
    ```

Aggregation

- Recall the aggregate operators in SQL: `AVG()`, `SUM()`, etc.
- More generally, *aggregation* involves computing a result from a collection of data.
- MongoDB supports two approaches to aggregation:
  - single-purpose aggregation methods
  - an aggregation pipeline

Single-Purpose Aggregation Methods

- `db.collection.count(<selection>)`

  `countDocuments` is now the preferred name

  - returns the number of documents in the collection that satisfy the specified selection document
  - ex: how may R-rated movies are shorter than 90 minutes?

```
db.movies.count({ rating: "R",
                  runtime: { $lt: 90 }})
```

- `db.collection.distinct(<field>, <selection>)`
  - returns an array with the distinct values of the specified field in documents that satisfy the specified selection document
  - if omit the selection, get all distinct values of that field
  - ex: which actors have been in one or more of the top 10 grossing movies?

```
db.movies.distinct("actors.name",
                   { earnings_rank: { $lte: 10 }}
                  )
```

Aggregation Pipeline

- A more general-purpose and flexible approach to aggregation is to use a *pipeline* of aggregation operations.
- Each stage of the pipeline:
  - takes a set of documents as input
  - applies a *pipeline operator* to those documents, which transforms / filters / aggregates them in some way
  - produces a new set of documents as output
- `db.collection.aggregate(`

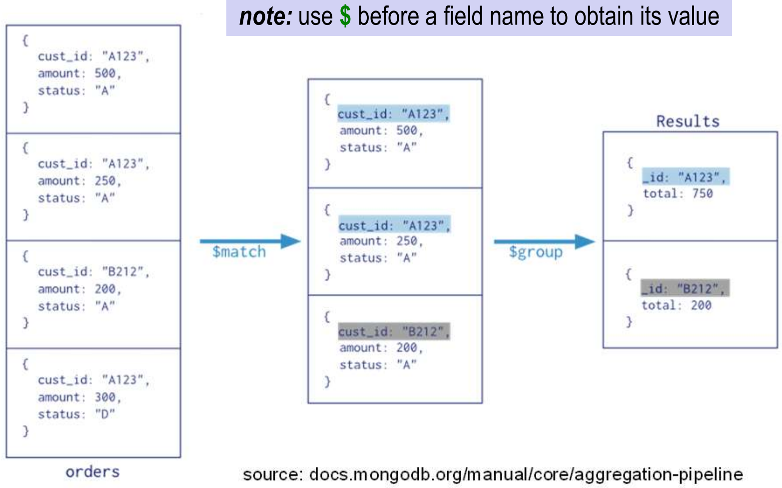
```
  { <pipeline-op1>: <pipeline-expression1> },
  { <pipeline-op2>: <pipeline-expression2> },
  ...,
  { <pipeline-opN>: <pipeline-expressionN> })
```

full collection $\xrightarrow{op1}$ op1 results $\xrightarrow{op2}$ op2 results ... $\xrightarrow{opN}$ final results

Aggregation Pipeline Example

```
db.orders.aggregate(
  { $match: { status: "A" } },
  { $group: { _id: "$cust_id", total: { $sum: "$amount"} } }
)
```

note: use **$** before a field name to obtain its value



source: docs.mongodb.org/manual/core/aggregation-pipeline

Pipeline Operators

- `$project` – include, exclude, rename, or create fields
  - Example of a single-stage pipeline using $project:

```
db.people.aggregate(
  { $project: {
        name: 1,
        whereBorn: "$pob",
        yearBorn: { $substr: ["$dob", 0, 4] }
    }
  })
```

  - for each document in the people collection, extracts:
    - `name` (1 = include, as in earlier projection documents)
    - pob, which is renamed `whereBorn`
    - a new field called `yearBorn`, which is derived from the existing dob values (yyyy-m-d → yyyy)
    - the _id field, because we didn't exclude it
  - ***note:*** use **$** before a field name to obtain its value

Pipeline Operators (cont.)

- `$group` – like GROUP BY in SQL

  ```
  $group: { _id: <field or fields to group by>,
            <computed-field-1>,
            ..., <computed-field-N> }
  ```

  - example: compute the number of movies with each rating

    ```
    db.movies.aggregate(
      { $group: { _id: "$rating",
                  numMovies: { $sum: 1 }
                } } )
    ```

    - `{ $sum: 1 }` is equivalent to COUNT(*) in SQL
      - for each document in a given subgroup,
        adds 1 to that subgroup's value of the computed field
    - can also sum values of a specific field (see earlier slide)
    - `$sum` is one example of an *accumulator*
    - others include: `$min`, `$max`, `$avg`, `$addToSet`

Pipeline Operators (cont.)

- `$match` – selects documents according to some criteria

  `$match: <selection>`

  where *<selection>* has identical syntax to the selection documents used by `db.`*`collection`*`.find()`

- `$unwind` – takes a document with an array of values and creates a separate document for each value in the array.
  - see the next example

Example of a Three-Stage Pipeline

```
db.movies.aggregate(
    { $match: { year: 2013 }},
    { $project: { _id: 0,
                  movie: "$name",
                  actor: "$actors.name" } },
    { $unwind: "$actor" }
)
```

- What does each stage do?
  - `$match`: select movies released in 2013
  - `$project`: for each such movie, create a document with:
    - no `_id` field
    - the `name` field of the movie, but renamed `movie`
    - the names of the actors (an array), as a field named `actor`
  - `$unwind`: turn each movie's document into a set of documents, one for each actor in the array of actors

Another Example: What does each stage do?

```
db.oscars.aggregate(
    { $match: { year: { $gte: 1980 } } },
    { $group: { _id: "$year", count: { $sum: 1 } } },
    { $match: { count: { $gt: 6 } } },
    { $project: { _id: 0, year: "$_id",
                  num_awards: "$count" } }  )
```

- first `$match`: select Oscars awarded in 1980 or later
- `$group`: take the Oscar docs selected by `$match` and:
  - create subgroups based on year (as specified by `_id` field)
  - for each subgroup, create a new doc with year as `_id` and a `count` field with the num. of Oscars from that year
- second `$match`: select docs for years with more than 6 Oscars
- `$project`: for each such year, create a document with:
  - no `_id` field
  - the `_id` field produced by `$group`, but renamed `year`
  - the `count` field produced by `$group`, renamed `num_awards`

More on Computing Aggregates

```
db.oscars.aggregate(
   { $match: { year: { $gte: 1980 } } },
   { $group: { _id: "$year", count: { $sum: 1 } } },
   { $match: { count: { $gt: 6 } } },
   { $project: { _id: 0, year: "$_id",
                 num_awards: "$count" } }  )
```

- The `$group` stage in the prior query computed a separate count for each of several subgroups.
- This is comparable to using an aggregate function with GROUP BY in SQL.

More on Computing Aggregates (cont.)

- What if we just want to compute a single count, average, etc.?
  - example: find the average runtime of all R-rated movies.
- In SQL, we would do something like this (with no GROUP BY):

```
SELECT AVG(runtime)
FROM Movie
WHERE rating = 'R';
```

- In MongoDB, we still need a `$group` stage, but we group on `null` in order to create a single group:

```
db.movies.aggregate(
    { $match: { rating: "R" } },
    { $group: { _id: null,
                avg_runtime: { $avg: "$runtime" }} },
    { $project: { _id: 0, avg_runtime: 1 } }
)
```

Two Additional Pipeline Operators

- `$sort` – sorts documents according to one of the fields

```
{ $sort: { field1_to_sort_on: sort_order1,
           field2_to_sort_on: sort_order2, …} }
```

  - for sort_order, use 1 for ascending
    -1 for descending

- `$limit` – include only the first n documents in a set of results

```
{ $limit: n }
```

- Example: Find the name and runtime of the movie with the longest runtime:

```
db.movies.aggregate( { $sort: { runtime: -1 } },
                     { $limit: 1 },
                     { $project: { _id: 0,
                                   name: 1,
                                   runtime: 1 } } )
```

  - note: if 2 or more movies are tied, will only get one of them

Recall: Sample Movie Document

```
{ _id: "0499549",
  name: "Avatar",
  year: 2009,
  rating: "PG-13",
  runtime: 162,
  genre: "AVYS",
  earnings_rank: 1,
  actors: [ { id: "0000244",
              name: "Sigourney Weaver" },
            { id: "0002332",
              name: "Stephen Lang" },
            { id: "0735442",
              name: "Michelle Rodriguez" },
            { id: "0757855",
              name: "Zoe Saldana" },
            { id: "0941777",
              name: "Sam Worthington" } ],
  directors: [ { id: "0000116",
                 name: "James Cameron" } ] }
```

Recall: Sample Person and Oscar Documents

```
{ _id: "0000059",
  name: "Laurence Olivier",
  dob: "1907-5-22",
  pob: "Dorking, Surrey, England, UK",
  hasActed: true,
  hasDirected: true
}
```

```
{ _id: ObjectId("528bf38ce6d3df97b49a0569"),
  year: 2013,
  type: "BEST-ACTOR",
  person: { id: "0000358",
            name: "Daniel Day-Lewis" },
  movie: { id: "0443272",
           name: "Lincoln" }
}
```

Extra Practice Writing Queries

1) Find the names of all people in the database who acted in *Avatar.*
   - SQL:

     ```
     SELECT P.name
     FROM Person P, Actor A, Movie M
     WHERE P.id = A.actor_id
     AND M.id = A.movie_id
     AND M.name = 'Avatar';
     ```

   - MongoDB:

Extra Practice Writing Queries (cont.)

2) How many people in the database who were born in California have won an Oscar?

- SQL:

```
SELECT COUNT(DISTINCT P.id)
FROM Person P, Oscar O
WHERE P.id = O.person_id
AND P.pob LIKE '%,%California%';
```

- Can't easily answer this question using our MongoDB version of the database. Why not?

Recovery and Logging

Harvard Extension School

Cody Doucette, Ph.D.

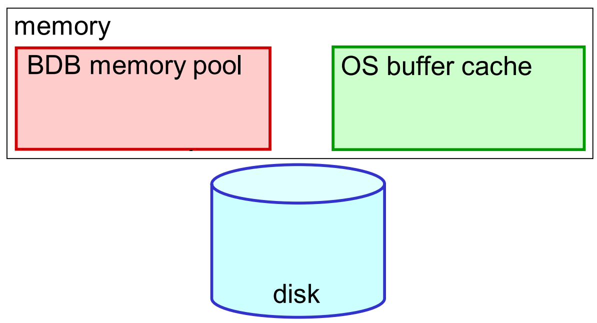
Lecture designed by David G. Sullivan

Review: ACID Properties

- A transaction has the following "ACID" properties:
  - Atomicity: either all of its changes take effect or none do
  - Consistency preservation: its operations take the database from one consistent state to another
  - Isolation: it is not affected by and does not affect other concurrent transactions
  - Durability: once it completes, its changes survive failures
- We'll now look at how the DBMS guarantees atomicity and durability.
  - ensured by the subsystem responsible for *recovery*

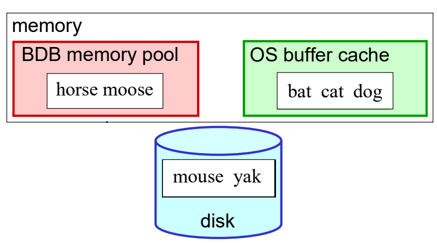
A Quick Look at Caching

- Recently accessed database pages are *cached* in memory so that subsequent accesses to them don't require disk I/O.
- There may be more than one cache:
  - the DBMS's own cache (called the memory pool in BDB)
  - the operating system's buffer cache



Caching Example 1

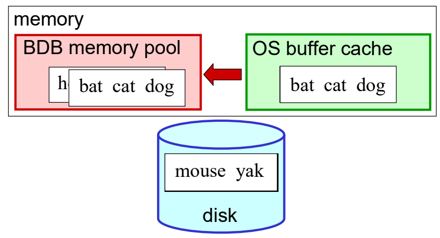
- The user requests the item with the key "horse."



- The page containing "horse" is already in the database's own cache, so no disk I/O is needed.

Caching Example 2

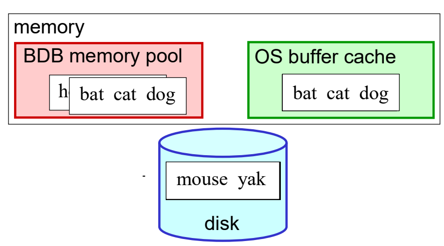
- The user requests the item with the key "cat."



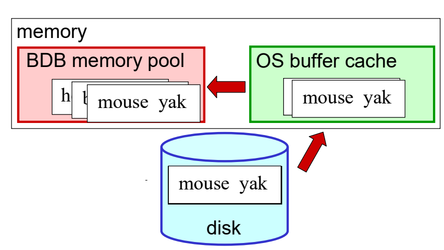
- The page containing "cat" is in the OS buffer cache, so it just needs to be brought into the database's cache. No disk I/O.
- This produces *double buffering* – two copies of the same page in memory.
  - one reason that some DBMSs bypass the filesystem

Caching Example 3

- The user requests the item with the key "yak."

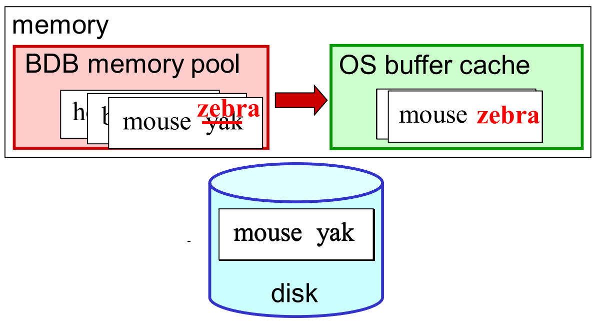


- The page with "yak" is in neither cache, so it is:
  - read from disk into the buffer cache
  - read into the database's own cache



Caching and Disk Writes

- Updates to a page may not make it to disk until the page is evicted from all of the caches.
  - initially, only the page in the DBMS's cache is updated
  - when evicted from the DBMS's cache, it is written to the backing file, but it may not go to disk right away



- This complicates recovery, because changes may not be on disk.

What Is Recovery?

- Recovery is performed after:
  - a crash of the DBMS
  - other non-catastrophic failures (e.g., a reboot)
  - (for catastrophic failures, need an archive or replication)
- It makes everything right again.
  - allows the rest of the DBMS to be built as if failures don't occur
- "the scariest code you'll ever write" (Margo Seltzer)
  - it has to work
  - it's rarely executed
  - it can be difficult to test

What Is Recovery? (cont.)

- During recovery, the DBMS takes the steps needed to:
  - *redo* changes made by any committed txn, if there's a chance the changes didn't make it to disk
    - ➔ durability: the txn's changes are still there after the crash
    - ➔ atomicity: *all* of its changes take effect
  - *undo* changes made by any txn that didn't commit, if there's a chance the changes made it to disk
    - ➔ atomicity: *none* of its changes take effect
    - also used when a transaction is rolled back
- In order for recovery to work, need to maintain enough state about txns to be able to redo or undo them.

Logging

- The *log* is a file that stores the info. needed for recovery.
- It contains:
  - update records, each of which summarizes a write
  - records for transaction begin and commit
- It does *not* record reads.
  - don't affect the state of the database
  - aren't relevant to recovery

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; old: 3000; new: 2500
225	txn: 1; item: D2; old: 1000; new: 1500
350	txn: 2; BEGIN
400	txn: 2; item: D3; old: 7200; new: 6780
470	txn: 1; item: D1; old: 2500; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D2; old: 1500; new: 1300
675	txn: 2; item: D3; old: 6780; new: 6760

- The log is append-only: records are added at the end, and blocks of the log file are written to disk sequentially.
  - more efficient than non-sequential writes to the database files

Write-Ahead Logging (WAL)

- Both updated database pages and log records are cached.
- It's important that they go to disk in a specific order.
- Example of what can go wrong:

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
CRASH
```

  - assume that:
    - `write(balance1 - 500)` made it to disk
    - `write(balance2 + 500)` didn't make it to disk
    - neither of the corresponding log records made it to disk
  - the database is in an inconsistent state
  - without the log records, the recovery system can't restore it

Write-Ahead Logging (WAL) (cont.)

- The write-ahead logging (WAL) policy:

  before a modified database page is written to disk,
  all update log records describing changes on that page
  must be forced to disk

    - the log records are "written ahead" of the database page
- This ensures that the recovery system can restore the database to a consistent state.

Undo-Redo Logging

- Update log records must include both the old *and* new values of the changed data element.
- Example log after a crash:
  - the database could be in an inconsistent state
  - why?
    - some of T1's changes may not have made it to disk.
      need to redo
    - some of T2's changes may have made it to disk.
      need to undo

LSN	*record contents*
100	**txn: 1; BEGIN**
150	**txn: 1; item: D1; old: 3000; new: 2500**
225	**txn: 1; item: D2; old: 1000; new: 1500**
350	**txn: 2; BEGIN**
400	**txn: 2; item: D3; old: 7200; new: 6780**
470	**txn: 1; item: D1; old: 2500; new: 2750**
500	**txn: 1; item: D2; old: 1500; new: 2100**
550	**txn: 1; COMMIT**
585	**txn: 2; item: D2; old: 1500; new: 1300**
675	**txn: 2; item: D3; old: 6780; new: 6760**

Undo-Redo Logging (cont.)

- To ensure that it can undo/redo txns as needed, undo-redo logging follows the WAL policy.
- In addition, it does the following when a transaction commits:
  1. writes the commit log record to the in-memory log buffer
  2. forces to disk all dirty log records
     (dirty = not yet written to disk)
- It does *not* force the dirty database pages to disk.
- At recovery, it performs two passes:
  - first, a *backward pass* to undo uncommitted transactions
  - then, a *forward pass* to redo committed transactions

Recovery Using Undo-Redo Logging

- Backward pass: begin at the last log record and scan backward
  - for each commit record, add the txn to a *commit list*
  - for each update by a txn *not* on the commit list, undo the update (restoring the old value)
  - for now, we skip:
    - updates by txns that *are* on the commit list
    - all begin records
- Forward pass:
  - for each update by a txn that *is* on the commit list, redo the update (writing the new value)
  - skip updates by txns that are *not* on the commit list, because they were handled on the backward pass
  - skip other records as well

Recovery Using Undo-Redo Logging (cont.)

- Here's how it would work on our earlier example:

LSN	*record contents*	*backward pass*	*forward pass*
100	txn: 1; BEGIN	skip	skip
150	txn: 1; item: D1; old: 3000; new: 2500	skip	redo: D1 = 2500
225	txn: 1; item: D2; old: 1000; new: 1500	skip	redo: D2 = 1500
350	txn: 2; BEGIN	skip	skip
400	txn: 2; item: D3; old: 7200; new: 6780	undo: D3 = 7200	skip
470	txn: 1; item: D1; old: 2500; new: 2750	skip	redo: D1 = 2750
500	txn: 1; item: D2; old: 1500; new: 2100	skip	redo: D2 = 2100
550	txn: 1; COMMIT	add to commit list	skip
585	txn: 2; item: D2; old: 1500; new: 1300	undo: D2 = 1500	skip
675	txn: 2; item: D3; old: 6780; new: 6760	undo: D3 = 6780	skip

- Recovery restores the database to a consistent state that reflects:
  - *all* of the updates by txn 1 (which committed before the crash)
  - *none* of the updates by txn 2 (which did *not* commit)

The Details Matter!

LSN	record contents	backward pass	forward pass
100	txn: 1; BEGIN	skip	skip
150	txn: 1; item: D1; old: 3000; new: 2500	skip	redo: D1 = 2500
225	txn: 1; item: D2; old: 1000; new: 1500	skip	redo: D2 = 1500
350	txn: 2; BEGIN	skip	skip
400	txn: 2; item: D3; old: 7200; new: 6780	undo: D3 = 7200	skip
470	txn: 1; item: D1; old: 2500; new: 2750	skip	redo: D1 = 2750
500	txn: 1; item: D2; old: 1500; new: 2100	skip	redo: D2 = 2100
550	txn: 1; COMMIT	add to commit list	skip
585	txn: 2; item: D2; old: 1500; new: 1300	undo: D2 = 1500	skip
675	txn: 2; item: D3; old: 6780; new: 6760	undo: D3 = 6780	skip

1) Scanning backward at the start of recovery provides the info needed for undo / redo decisions.
   - when we see an update, we already know whether the txn has committed!

The Details Matter!

LSN	record contents	backward pass	forward pass
100	txn: 1; BEGIN	skip	skip
150	txn: 1; item: D1; old: 3000; new: 2500	skip	**redo: D1 = 2500**
225	txn: 1; item: D2; old: 1000; new: 1500	skip	redo: D2 = 1500
350	txn: 2; BEGIN	skip	skip
400	txn: 2; item: D3; old: 7200; new: 6780	undo: D3 = 7200	skip
470	txn: 1; item: D1; old: 2500; new: 2750	skip	**redo: D1 = 2750**
500	txn: 1; item: D2; old: 1500; new: 2100	skip	redo: D2 = 2100
550	txn: 1; COMMIT	add to commit list	skip
585	txn: 2; item: D2; old: 1500; new: 1300	undo: D2 = 1500	skip
675	txn: 2; item: D3; old: 6780; new: 6760	undo: D3 = 6780	skip

2) To ensure the correct values are on disk after recovery, we:
   - put all redos after all undos (consider D2 above)
   - perform the undos in reverse order (consider D3 above)
   - perform the redos in the same order as the original updates (consider D1 above)

Logical Logging

- We've assumed that update records store the old + new values of the changed data element.
- It's also possible to use *logical logging*, which stores a logical description of the update operation.
  - example: increment D1 by 1
- Logical logging is especially useful when we use pages or blocks as data elements, rather than records.
  - storing the old and new contents of a page or block would take up a lot of space
  - instead, store a logical description
    - for example: "add record r somewhere on D1"

Logical Logging (cont.)

- When we store old and new data values, the associated undo/redo operations are *idempotent* .
  - can be performed multiple times without changing the result
- Problem: logical update operations *may not be* idempotent.
  - example: if "increment D1 by 1" has already been performed, we don't want to redo it
  - example: if "increment D1 by 1" has <u>not</u> been performed, we don't want to undo it
  - example: if "add record r to page D1" has already been performed, we don't want to redo it
- To ensure that only the necessary undo/redos are made, the DBMS makes use of the *log sequence numbers* (*LSNs*) associated with the update log records.

Storing LSNs with Data Elements

- When a data element is updated, the DBMS:
  - stores the LSN of the update log record with the data element
    - known as the *datum LSN*
  - stores the old LSN of the data element in the log record

log file

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; **new: "bar"**; **old: "foo"; olsn: 0**

data elements (value / datum LSN)

D1	*D2*	*D3*
"foo" / 0	"oh" / 0	"moo" / 0
"bar"/ 150		

Recovery Using LSNs

- During recovery, there are three LSNs to consider for each update record:
  1) the ***record LSN***: the one for the update record itself

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900

Recovery Using LSNs

- During recovery, there are three LSNs to consider for each update record:
  1) the ***record LSN***: the one for the update record itself
  2) the on-disk ***datum LSN*** for the data item
     - the one associated with it in the database file

on-disk datum LSNs:
D4: 0, D5: 0, D6: **1100**, D7: 930

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900

Recovery Using LSNs

- During recovery, there are three LSNs to consider for each update record:
  1) the ***record LSN***: the one for the update record itself
  2) the on-disk ***datum LSN*** for the data item
     - the one associated with it in the database file
  3) the ***olsn***: the old datum LSN for the data item
     - the one associated with it when the update was originally requested

on-disk datum LSNs:
D4: 0, D5: 0, D6: **1100**, D7: 930

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: **900**

The Backward Pass Using LSNs

- During the backward pass, we undo an update if:
  - the txn did *not* commit
  - **datum LSN** == **record LSN**
- When we undo, we also set:
  datum LSN = olsn

on-disk datum LSNs:
D4: 0, D5: 0, D6: **1100**, D7: 930

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: **900**

Which updates will be undone?

- datum LSNs: D4: 0 D5: 0 D6: 1100 D7: 930

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN		
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0		
825	txn: 4; BEGIN		
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0		
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0		
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0		
980	txn: 4; COMMIT		
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850		
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900		

Which updates will be undone?

- datum LSNs: D4: 0 D5: 0 D6: ~~1100~~, 900 D7: ~~930~~, 0

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN	skip	
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0	0 != 770 don't undo	
825	txn: 4; BEGIN	skip	
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0	skip	
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0	skip	
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0	930 == 930 undo: D7 = "zoo" datum LSN = 0	
980	txn: 4; COMMIT	add to commit list	
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850	0 != 1000 don't undo	
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900	1100 == 1100 undo: D6 = 8.9 datum LSN = 900	

The Forward Pass Using LSNs

- During the forward pass, we redo an update if:
  - the txn *did* commit
  - **datum LSN** == **olsn**
- When we redo, we also set:
  datum LSN = record LSN

on-disk datum LSNs:
D4: **0**, D5: 0, D6: 900, D7: 0

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: **0**
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900

Which updates will be redone?

- datum LSNs: D4: 0 D5: 0 D6: ~~1100~~, 900 D7: ~~930~~, 0

LSN	record contents	backward pass	forward pass
700	txn: 3; BEGIN	skip	
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0	0 != 770 don't undo	
825	txn: 4; BEGIN	skip	
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0	skip	
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0	skip	
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0	930 == 930 undo: D7 = "zoo" datum LSN = 0	
980	txn: 4; COMMIT	add to commit list	
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850	0 != 1000 don't undo	
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900	1100 == 1100 undo: D6 = 8.9 datum LSN = 900	

Which updates will be redone?

- datum LSNs: D4: ~~0~~, 850   D5: 0   D6: ~~1100~~, 900   D7: ~~930~~, 0

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN	skip	skip
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0	0 != 770 don't undo	skip
825	txn: 4; BEGIN	skip	skip
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0	skip	0 == 0 redo: D4 = 8500 datum LSN = 850
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0	skip	900 != 0 don't redo
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0	930 == 930 undo: D7 = "zoo" datum LSN = 0	skip
980	txn: 4; COMMIT	add to commit list	skip
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850	0 != 1000 don't undo	skip
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900	1100 == 1100 undo: D6 = 8.9 datum LSN = 900	skip

Checkpoints

- As a DBMS runs, the log gets longer and longer.
  - thus, recovery could end up taking a very long time!
- To avoid long recoveries, periodically perform a *checkpoint*.
  - force data and log records to disk to create a consistent on-disk database state
  - during recovery, don't need to consider operations that preceded this consistent state

Static Checkpoints

- Stop activity and wait for a consistent state.
  1) prohibit new transactions from starting and wait until all current transactions have aborted or committed.
- Once there is a consistent state:
  2) force all dirty log records to disk
     (dirty = not yet written to disk)
  3) force all dirty database pages to disk
  4) write a *checkpoint record* to the log
  - these steps must be performed in the specified order!
- When performing recovery, go back to the most recent checkpoint record.
- Problem with this approach?

Dynamic Checkpoints

- Don't stop and wait for a consistent state.
  Steps:
  1) prevent all update operations
  2) force all dirty log records to disk
  3) force all dirty database pages to disk
  4) write a *checkpoint record* to the log
     - include a list of all active txns
- When performing recovery:
  - backward pass: go back until you've seen the start records of all uncommitted txns in the most recent checkpoint record
  - forward pass: begin from the log record that comes after the most recent checkpoint record. why?
  - note: if all txns in the checkpoint record are on the commit list, we stop the backward pass at the checkpoint record

Example of Recovery with Dynamic Checkpoints

- Initial datum LSNs: D4: 110 D5: ~~140~~,0 D6: 80

LSN	*record contents*	*backward pass*	*forward pass*
100	txn: 1; BEGIN		
110	txn: 1; item: **D4**; old: 20; new: 15; olsn: 0		
120	txn: 2; BEGIN	***stop here***	
130	txn: 1; COMMIT	add to commit list	
140	txn: 2; item: D5; old: 12; new: 13; olsn: 0	undo: D5 = 12 datum LSN = 0	
150	**CHECKPOINT (active txns = 2)**	note active txns	
160	txn: 2; item: **D4**; old: 15; new: 50; olsn: 110	don't undo	***start here*** skip
170	txn: 3; BEGIN	skip	skip
180	txn: 3; item: D6; old: 6; new: 8; olsn: 80	don't undo	skip

Could D4 have a datum LSN of less than 110?

Undo-Only Logging

- Only store the info. needed to undo txns.
  - update records include only the old value
- Like undo-redo logging, undo-only logging follows WAL.
- In addition, all database pages changed by a transaction must be forced to disk before allowing the transaction to commit. Why?
- At transaction commit:
  1. force all dirty log records to disk
  2. force database pages changed by the txn to disk
  3. write the commit log record
  4. force the commit log record to disk
- During recovery, the system only performs the backward pass.

Redo-Only Logging

- Only store the info. needed to redo txns.
  - update records include only the new value
- Like the other two schemes, redo-only logging follows WAL.
- In addition, all database pages changed by a txn are held in memory until it commits and its commit record is forced to disk.
- At transaction commit:
  1. write the commit log record
  2. force all dirty log records to disk

(changed database pages are allowed to go to disk anytime after this)

- If a transaction aborts, none of its changes can be on disk.
- During recovery, perform the backward pass to build the commit list (no undos). Then perform the forward pass as in undo-redo.

Comparing the Three Logging Schemes

- Factors to consider in the comparison:
  - complexity/efficiency of recovery
  - size of the log files
  - what needs to happen when a txn commits
  - other restrictions that a logging scheme imposes on the system
- We'll list advantages and disadvantages of each scheme.
- Undo-only:

  \+ smaller logs than undo-redo

  \+ simple and quick recovery procedure (only one pass)

  – forces log and data to disk at commit;
  have to wait for the I/Os

Comparing the Three Logging Schemes (cont.)

- Redo-only:
  + smaller logs than undo-redo
  +/ – recovery: more complex than undo-only, less than undo-redo
  – must be able to cache all changes until the txn commits
    - limits the size of transactions
    - constrains the replacement policy of the cache
  + forces only log records to disk at commit

- Undo-redo:
  – larger logs
  – more complex recovery
  + forces only log records to disk at commit
  + don't need to retain all data in the cache until commit

Reviewing the Log Record Types

- Why is each type needed?
  - assume undo-redo logging
- update records: hold the info. needed to undo/redo changes
- commit records: allow us to determine which changes should be undone and which should be redone
- begin records: allow us to determine the extent of the backward pass in the presence of dynamic checkpoints
- checkpoint records: limit the amount of the log that is processed during recovery

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- *What type of logging is being used to create the log at right?*

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80
500	txn: 2; item: D3; old: 30
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- *What type of logging is being used to create the log at right?*
  undo-only
- To make the rest of the problem easier, add the new values to the log…

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; **new: 75**
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; **new: 25**
500	txn: 2; item: D3; old: 30; **new: 60**
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; **new: 90**

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- At the start of recovery, what are the possible on-disk values under ***undo-only***?
  - does *not* pin values in memory
    ➔ *may* go to disk at any time
  - at commit, forces dirty data values to disk
    ➔ older values are no longer possible

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; new: 75
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; new: 25
500	txn: 2; item: D3; old: 30; new: 60
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; new: 90

	in-memory	possible on-disk
D1:		
D2:		
D3:		

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- At the start of recovery, what are the possible on-disk values under ***redo-only***?
  - *does* pin values in memory
    ➔ *can't* go to disk until commit
  - at commit, unpins values but does *not* force them to disk
    ➔ older values are still possible

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; new: 75
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; new: 25
500	txn: 2; item: D3; old: 30; new: 60
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; new: 90

	in-memory	possible on-disk
D1:		
D2:		
D3:		

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- At the start of recovery, what are the possible on-disk values under ***undo-redo***?
  - does *not* pin values in memory
    ➔ *may* go to disk at any time
  - at commit, does *not* force dirty data to disk
    ➔ older values are still possible

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; new: 75
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; new: 25
500	txn: 2; item: D3; old: 30; new: 60
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; new: 90

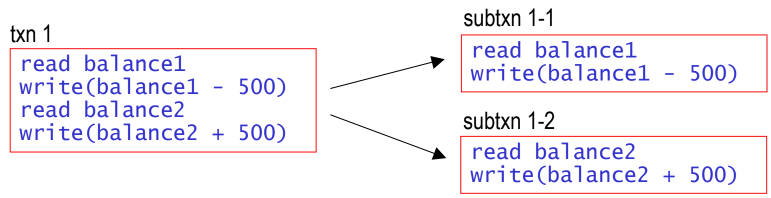
	in-memory	possible on-disk
D1:		
D2:		
D3:		

Atomicity

- In a centralized database, logging and recovery are enough to ensure atomicity.
  - if a txn's commit record makes it to the log, *all* of its changes will eventually take effect
  - if a txn's commit record isn't in the log when a crash occurs, *none* of its changes will remain after recovery
- What about atomicity in a distributed database?

Recall: Distributed Transactions

- A *distributed transaction* involves data stored at multiple sites.
- One of the sites serves as the *coordinator* of the transaction.
- The coordinator divides a distributed transaction into *subtransactions*, each of which executes on one of the sites.



Distributed Atomicity

- In a distributed database:
  - each site performs local logging and recovery of its subtxns
  - that alone is *not* enough to ensure atomicity
- The sites must coordinate to ensure that either:
  - *all* of the subtxns are committed

    or
  - *none* of them are

Distributed Atomicity (cont.)

- Example of what could go wrong:
  - a subtxn at one of the sites deadlocks and is aborted
  - before the coordinator of the txn finds out about this, it tells the other sites to commit, and they do so
- Another example:
  - the coordinator notifies the other sites that it's time to commit
  - most of the sites commit their subtxns
  - one of the sites crashes before committing

Two-Phase Commit (2PC)

- A protocol for deciding whether to commit a distributed txn.
- Basic idea:
  - coordinator asks sites if they're ready to commit
  - if a site is ready, it:
    1. *prepares* its subtxn – putting it in the *ready state*
    2. tells the coordinator it's ready
  - if all sites say they're ready, all subtxns are committed
  - otherwise, all subtxns are aborted (i.e., rolled back)
- Preparing a subtxn means ensuring it can be *either* committed or rolled back – even after a failure.
  - need to at least…
  - some logging schemes need additional steps
- After saying it's ready, a site *must wait* to be told what to do next.

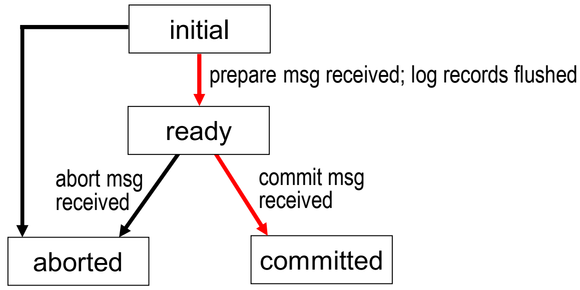
2PC Phase I: Prepare

- When it's time to commit a distributed txn T, the coordinator:
  - force-writes a *prepare record* for T to its own log
  - sends a *prepare message* to each participating site
- If a site can commit its subtxn, it:
  - takes the steps needed to put its txn in the ready state
  - force-writes a *ready record* for T to its log
  - sends a *ready message* for T to the coordinator and waits
- If a site needs to abort its subtxn, it:
  - force-writes a *do-not-commit record* for T to its log
  - sends a *do-not-commit message* for T to the coordinator
  - can it abort the subtxn now?
- Note: we always log a message before sending it to others.
  - allows the decision to send the message to survive a crash

2PC Phase II: Commit or Abort

- The coordinator reviews the messages from the sites.
  - if it doesn't hear from a site within some time interval, it assumes a do-not-commit message
- If all sites sent ready messages for T, the coordinator:
  - force-writes a commit record for T to its log
    - T is now officially committed
  - sends *commit messages* for T to the participating sites
- Otherwise, the coordinator:
  - force-writes an abort record for T to its log
  - sends *abort messages* for T to the participating sites
- Each site:
  - force-writes the appropriate record (commit or abort) to its log
  - commits or aborts its subtxn as instructed

2PC State Transitions



- A subtxn can enter the aborted state from the initial state at any time.
- After entering the ready state, it can only enter the aborted state after receiving an abort message.
- A subtxn can only enter the committed state from the ready state, and only after receiving a commit message.

Recovery When Using 2PC

- When a site recovers, its decides whether to undo or redo its subtxn for a txn T based on the last record for T in its log.
- Case 1: the last log record for T is a commit record.
  - redo the subtxn's updates as needed
- Case 2: the last log record for T is an abort record.
  - undo the subtxn's updates as needed
- Case 3: the last log record for T is a do-not-commit record.
  - undo the subtxn's updates as needed
  - why is this correct?

Recovery When Using 2PC (cont.)

- Case 4: the last log record for T is <u>from before 2PC began</u> (e.g., an update record).
  - undo the subtxn's updates as needed
  - this works in both of the possible situations:
    - 2PC has already completed without hearing from this site
      why?
    - 2PC is still be going on
      why?

- Case 5: the last log record for T is a <u>ready</u> record.
  - contact the coordinator (or another site) to determine T's fate
  - why can the site still commit or abort T as needed?
  - if it can't reach another site, it must block until it can reach one!

What if the Coordinator Fails?

- The other sites can either:
  - wait for the coordinator to recover
  - elect a new coordinator
- In the meantime, each site can determine the fate of any current distributed transactions.
- Case 1: a site has *not* received a prepare message for txn T
  - can abort its subtxn for T
  - preferable to waiting for the coordinator to recover, because it allows the T's fate to be decided
- Case 2: a site *has* received a prepare message for T, but has *not* yet sent ready message
  - can also abort its subtxn for T now. why?

What if the Coordinator Fails? (cont.)

- Case 3: a site sent a ready message for T but didn't hear back
  - poll the other sites to determine T's fate

evidence	conclusion/action
at least one site has a commit record for T	???
at least one site has an abort record for T	???
no commit/abort records for T; at least one site does *not* have a ready record for T	???
no commit/abort records for T; *all* surviving sites have ready records for T	can't know T's fate unless coordinator recovers. why?

Extra Practice

- What type of logging is being used to create the log at right?

- At the start of recovery, what are the possible on-disk values?

original values:
D1=1000, D2=3000

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; new: 2500
350	txn: 2; BEGIN
400	txn: 2; item: D2; new: 6780
470	txn: 1; item: D1; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D1; new: 1300

Extra Practice

- What if the DBMS were using ***undo-only*** logging instead?
- At the start of recovery, what are the possible on-disk values?

original values:
D1=1000, D2=3000

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; new: 2500
350	txn: 2; BEGIN
400	txn: 2; item: D2; new: 6780
470	txn: 1; item: D1; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D1; new: 1300

	in-memory	possible on-disk
D1:		1000
D2:		3000

Extra Practice

- What if the DBMS were using ***undo-redo*** logging instead?

- At the start of recovery, what are the possible on-disk values?

original values:
D1=1000, D2=3000

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; new: 2500
350	txn: 2; BEGIN
400	txn: 2; item: D2; new: 6780
470	txn: 1; item: D1; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D1; new: 1300

	in-memory	possible on-disk
D1:		1000
D2:		3000

Performance Tuning

Wrap-up and Conclusions

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Reference

Database Tuning: A Principled Approach, Dennis E. Shasha

Goals of Performance Tuning

- Increase *throughput* – work completed per time
  - in a DBMS, typically transactions per second (txns/sec)
  - other options: reads/sec, writes/sec, operations/sec
  - measure over some interval (time-based or work-based)
- Decrease *response time* or *latency*
  – the time spent waiting for an operation to complete
  - overall throughput may be good,
    but some txns may spend a long time waiting
- Secondary goals (ways of achieving the other two):
  - reduce lock contention
  - reduce disk I/Os
  - etc.

Challenges of Tuning

- Often need to balance conflicting goals
  - example: tuning the *checkpoint interval*
    – the amount of time between checkpoints of the log.
    - goals?
      -
      -
- It's typically difficult to:
  - determine what to tune
  - predict the impact of a potential tuning decision
- The optimal tuning is workload-dependent.
  - can vary over time

What Can Be Tuned?

- Three levels of tuning:
  1. low level: hardware
     - disks, memory, CPU, etc.
  2. middle level: DBMS parameters
     - page size, checkpoint interval, etc.
  3. high level
     - schema, indices, transactions, queries, etc.

- These levels interact with each other.
  - tuning on one level may change the tuning needs on another level
  - need to consider together

1. Hardware-Level Tuning (Low Level)

- Disk subsystem
  - limiting factor = rate at which data can be accessed
  - based on:
    - disk characteristics (seek time, transfer time, etc.)
    - number of disks
    - layout of data on the disk
  - adding disks increases parallelism
    - may thus increase throughput
  - adjusting on-disk layout may also improve performance
    - sequential accesses are more efficient than random ones

- Memory
  - adding memory allows more pages to fit in the cache
  - can thereby reduce the number of I/Os
  - however, memory is more expensive than disk

Other Details of Hardware Tuning

- Can also add:
  - processing power
  - network bandwidth (in the case of a distributed system)
- Rules of thumb for adding hardware (Shasha)
  - start by adding memory
    - based on some measure of your *working set*
  - then add disks if disks are still overloaded
  - then add processing power if CPU utilization >= 85%
  - then consider adding network bandwidth
- Consider other options before adding hardware!
  - tune software: e.g., add an index to facilitate a common query
  - use current hardware more effectively:
    - example: give the log its own disk

2. Parameter Tuning (Middle Level)

- DBMSs—like most complex software systems—include parameters (“knobs”) that can be tuned by the user.
- Example knobs:
  - checkpoint interval
  - deadlock-detection interval
  - several more we'll look at in a moment
- Optimal knob settings depend on the workload.

Example: Tuning Lock Granularity

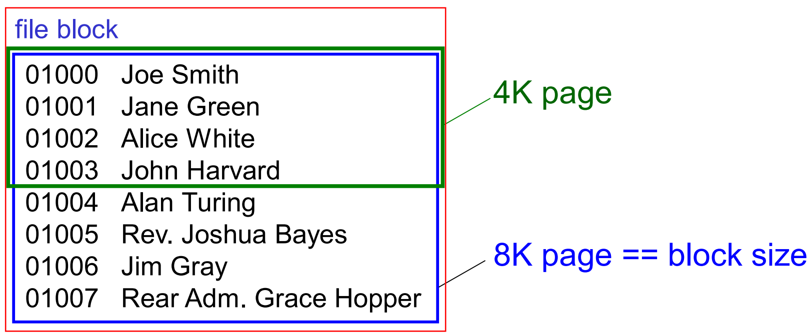
- possibilities include: page, record, entire table
- How could finer-grained locking improve performance?
  -
- How could finer-grained locking degrade performance?
  -
  -
- Rule of thumb (Shasha):
  - measure the "length" of a txn in terms of the percentage of the table that it accesses
  - "long" txns should use table-level locking
  - "medium" txns that are based on a clustered/internal index should use page-level locking
  - "short" txns should use record-level locking

Example: Tuning the MPL

- MPL = maximum number of txns that can operate concurrently
- How could increasing the MPL improve performance?
  -
- How could increasing the MPL degrade performance?
  -
- Shasha: no rule of thumb works in all cases.
  Instead, use an incremental approach:
  - start with a small MPL value
  - increase MPL by one and measure performance
  - keep increasing MPL until performance no longer improves

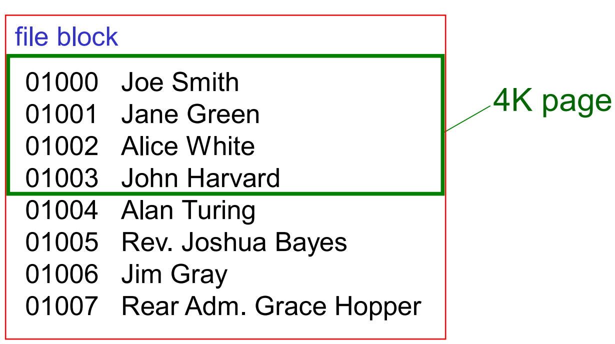
Example: Tuning Page Size

- Recall:
  - the filesystem transfers data in units called *blocks*
  - the DBMS groups data into *pages*
    - may or may not correspond to a block



Tuning Page Size (cont.)

- How could a smaller page size improve performance?



Tuning Page Size (cont.)

- How could a smaller page size *degrade* performance?

file block

01000 Joe Smith
01001 Jane Green
01002 Alice White
01003 John Harvard
01004 Alan Turing
01005 Rev. Joshua Bayes
01006 Jim Gray
01007 Rear Adm. Grace Hopper

4K page

Tuning Page Size (cont.)

- What if we select a page size > block size?
  - can reduce your ability to keep useful data in the cache (when accesses are more or less random)
  - if page-level locking, can increase contention for locks
  - can lead to unnecessary I/O due to OS prefetching

file block

01000 Joe Smith
01001 Jane Green
01002 Alice White
01003 John Harvard
01004 Alan Turing
01005 Rev. Joshua Bayes
01006 Jim Gray
01007 Rear Adm. Grace Hopper

01008 Ted Codd
01009 Margo Seltzer

16K page

+ can reduce the number of overflow pages
+ reduces I/O for workloads with locality (e.g., range searches)

Tuning Page Size (cont.)

- Rule of thumb?
  - page size = block size is usually best
  - if lots of lock contention, reduce the page size
  - if lots of large items, increase the page size

3. High-Level Tuning

- Tune aspects of the schema and workload:
  - relations
  - indices/views
  - transactions/queries
- Tuning at this level:
  - is more system-independent than tuning at the other levels
  - may eliminate the need for tuning at the lower levels

Tuning a Relational Schema

- Example schema: *account(account-num, branch, balance)*
  customer(customer-num, name, address)
  owner(account-num, customer-num)
  (One account may have multiple owners.)

- Vertical fragmentation: divide one relation into two or more
  - e.g., what if most queries involving account are only interested in the account-num and balance?

- Combining relations:
  - e.g., store the join of account and owner:
    account2(account-num, branch, balance, customer-num)
  - what's one drawback of this approach?

Recall: Primary vs. Secondary Indices

- Data records are stored inside a *clustered* index structure.
  - also known as the *primary* index

- We can also have *unclustered* indices based on other fields.
  - also known as *secondary indices*

- Example: *Customer(id, name, street, city, state, zip)*
  - primary index:
    (key, value) = (*id*, all of the remaining fields)
  - a secondary index to enable quick searches by name
    (key, value) = (*name*, *id*) *does not include the other fields!*

Tuning Indices

- If SELECTs are slow, add one or more secondary index.
- If modifications are slow, remove one or more index. Why?
- Other index-tuning decisions:
  - what type of index?
    - hash or B-tree; see lecture on storage structures
  - which index should be the clustered/primary?
- Complication: the optimal set of indices may depend on the query-evaluation plans selected by the query optimizer!

Tuning Transactions/Queries

- Banking database example:
  - lots of short transactions that update balances
  - long, read-only transactions that scan the entire account relation to compute summary statistics for each branch
  - what happens if these two types of transactions run concurrently? (assume rigorous 2PL)
- Possible options:
  - execute the long txns during a quiet period
  - multiversion concurrency control
    - make the long, read-only txns operate on an earlier version, so they don't conflict with the short update txns
  - use a weaker isolation level
    - ex: allow read-only txn to execute without acquiring locks

Deciding What to Tune

- Your system is slow. What should you do?
- Not a simple process
  - many factors may contribute to a given bottleneck
  - fixing one problem may not eliminate the bottleneck
  - eliminating one bottleneck may expose others

Deciding What to Tune (cont.)

- Iterative approach (Shasha):

```
repeat
   monitor the system
   tune important queries
   tune global parameters (includes DBMS params,
     OS params, relations, indices, views, etc.)
until satisfied or can do no more

if still unsatisfied
   add appropriate hardware (see rules of thumb from earlier)
   start over from the beginning!
```

Example Tuning Scenarios

- From Shasha's book
- All scenarios start with the complaint that an application is running too slowly.
- Scenario 1:
  - workload:
    - data-mining application for a chain of department stores
    - queries the following relation during the day:
      oldsales(cust-num, cust-city, item, quantity, date, price)
    - indices on cust-num, cust-city, item to speed up the queries
    - at night:
      - updates performed as a bulk load
      - bulk delete to eliminate records more than 3 weeks old
  - specific problems:
    - bulk load times are very slow
    - daytime queries are also degenerating

Example Tuning Scenarios (cont.)

- Scenario 2:
  - workload:
    - an application that is essentially read-only
    - performs many scans of a relation
  - relevant info:
    - disks show high access utilization but low space utilization
    - the log is on a disk by itself
    - each scan currently requires many disk seeks
    - management refuses to buy more disks

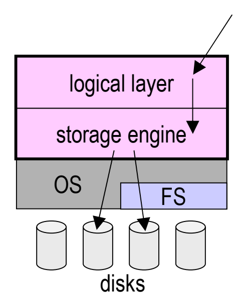
Example Tuning Scenarios (cont.)

- Scenario 3:
  - workload:
    - an airline manages 100 flights per day
    - two tables:

      passenger(passenger-name, flight-num, seat-num)
      occupancy(flight-num, total-passengers)
    - every reservation txn updates both tables
  - relevant info:
    - there is a high degree of lock contention

Looking Back

- Recall our two-layer view of a DBMS:



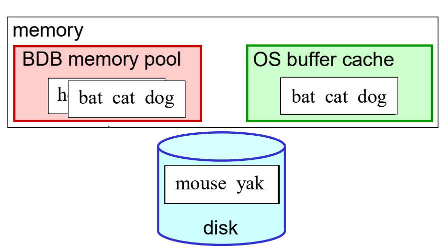
- When choosing an approach to information management, choose an option for each layer.
- We've seen several options for the storage layer:
  - transactional storage engine
  - plain-text files (e.g., for XML or JSON)
  - native XML DBMS
  - NoSQL DBMS (with support for sharding and replication)
- We've also looked at several options for the logical layer:
  - relational model
  - semistructured: XML, JSON
  - other NoSQL models: key/value pairs, column-families

One Size Does *Not* Fit All

- An RDBMS is an extremely powerful tool for managing data.
- However, it may not always be the best choice.
  - see the first lecture for a reminder of the reasons why!
- Need to learn to choose the right tool for a given job.
- In some cases, may need to develop new tools!

Implementing a Storage Engine

- We looked at ways that data is stored on disk.
- We considered *index structures.*
  - B-trees and hash tables
  - provide efficient search and insertion according to one or more key fields
- We also spoke briefly about the use of *caching* to reduce disk I/Os.



Implementing a *Transactional* Storage Engine

- We looked at how the "ACID" properties are guaranteed:

  Atomicity: either all of a txn's changes take effect or none do

  Consistency preservation: a txn's operations take the database from one consistent state to another

  Isolation: a txn is not affected by other concurrent txns

  Durability: once a txn completes, its changes survive failures

Distributed Databases and NoSQL Stores

- We looked at how databases can be:
  - fragmented/sharded
  - replicated
- We also looked at NoSQL data stores:
  - designed for use on clusters of machines
  - can handle massive amounts of data / queries

Logical-to-Physical Mapping

- The topics related to storage engines are potentially relevant to *any* database system.
  - not just RDBMSs
  - any logical layer can be built on top of any storage layer
- Regardless of the model, you need a *logical-to-physical mapping.*
- In PS 2, you implemented part of a logical-to-physical mapping for the relational model using Berkeley DB.
- We also looked at options for how to perform this mapping for XML.